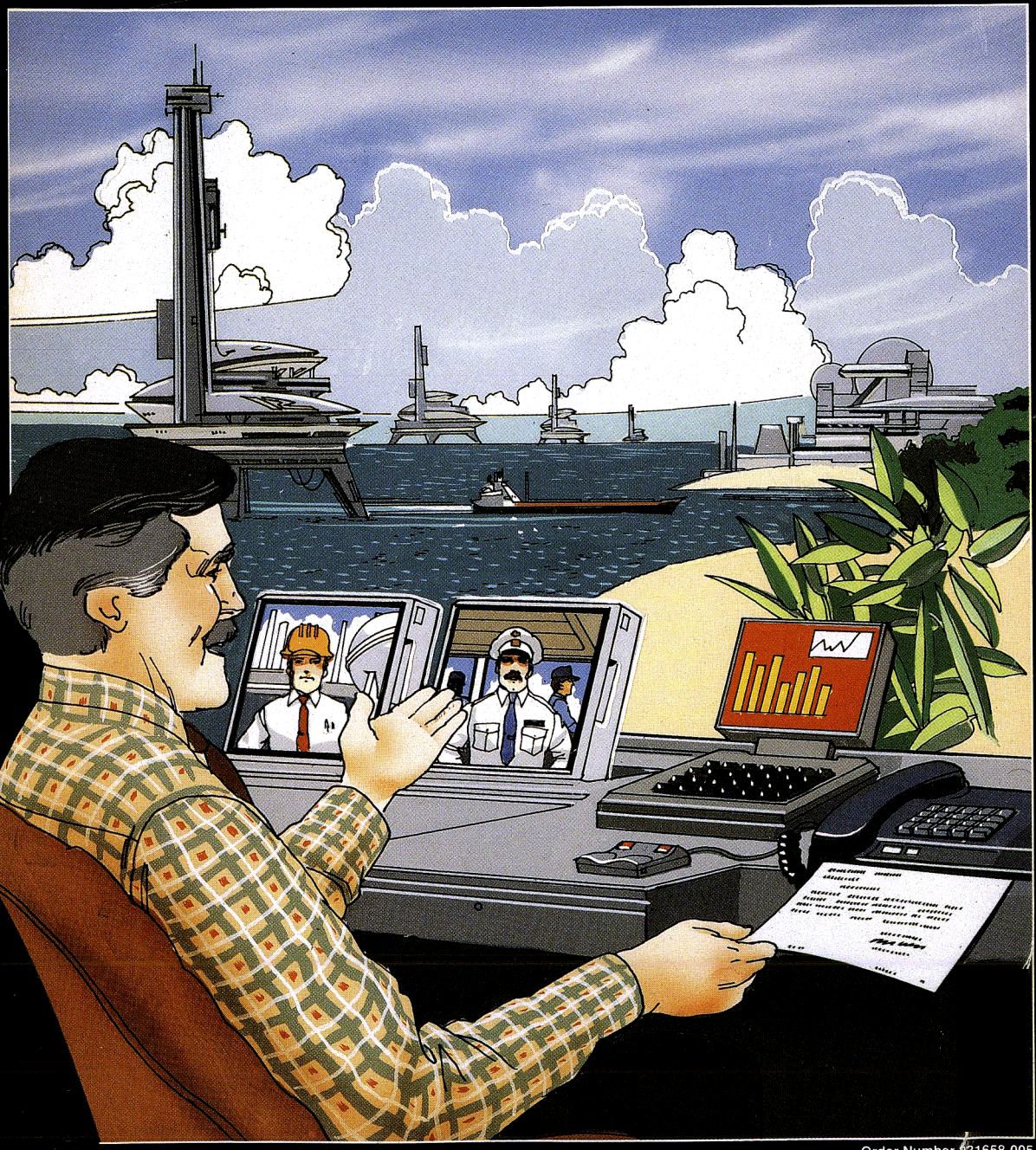
Microcommunications

Volume II - Applications



Ahearn/Ericksen

Order Number 231658-005

LITERATURE

To order Intel Literature or obtain literature pricing information in the U.S. and Canada call or write Intel Literature Sales. In Europe and other international locations, please contact your local sales office or distributor.

INTEL LITERATURE SALES
P.O. BOX 7641
Mt. Prospect, IL 60056-7641

In the U.S. and Canada
call toll free
(800) 548-4725

CURRENT HANDBOOKS

Product line handbooks contain data sheets,application notes, article reprints and other design information.

TITLE	LITERATURE ORDER NUMBER
SET OF 11 HANDBOOKS (Available in U.S. and Canada only)	**231003**
EMBEDDED APPLICATIONS	270648
8-BIT EMBEDDED CONTROLLERS	270645
16-BIT EMBEDDED CONTROLLERS	270646
16/32-BIT EMBEDDED PROCESSORS	270647
MEMORY	210830
MICROCOMMUNICATIONS (2 volume set)	231658
MICROCOMPUTER SYSTEMS	280407
MICROPROCESSORS	230843
PERIPHERALS	296467
PRODUCT GUIDE (Overview of Intel's complete product lines)	210846
PROGRAMMABLE LOGIC	296083
ADDITIONAL LITERATURE (Not included in handbook set)	
AUTOMOTIVE SUPPLEMENT	231792
COMPONENTS QUALITY/RELIABILITY HANDBOOK	210997
INTEL PACKAGING OUTLINES AND DIMENSIONS (Packaging types, number of leads, etc.)	231369
INTERNATIONAL LITERATURE GUIDE	E00029
LITERATURE PRICE LIST (U.S. and Canada) (Comprehensive list of current Intel Literature)	210620
MILITARY (2 volume set)	210461
SYSTEMS QUALITY/RELIABILITY	231762

Intel the Microcomputer Company:

When Intel invented the microprocessor in 1971, it created the era of microcomputers. Whether used in embedded applications such as automobiles or microwave ovens, or as the CPU in personal computers or supercomputers, Intel's microcomputers have always offered leading-edge technology. Intel continues to strive for the highest standards in memory, microcomputer components, modules and systems to give its customers the best possible competitive advantages.

MICROCOMMUNICATIONS APPLICATIONS

1990

intel

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

376, 386, 387, 486, 4-SITE, Above, ACE51, ACE96, ACE186, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, DVI, ETOX, FaxBACK, Genius, i, i, i486, i750, i860, ICE, iCEL, ICEVIEW, iCS, iDBP, iDIS, I²ICE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, intel, Intel386, intelBOS, Intel Certified, Intelevision, intelligent Identifier, intelligent Programming, Intellec, Intellink, iOSP, iPAT, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PRO750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, ToolTALK, UPI, Visual Edge, VLSiCEL, and ZapCode, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

*MULTIBUS is a patented Intel bus.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

CUSTOMER SUPPORT

INTEL'S COMPLETE SUPPORT SOLUTION WORLDWIDE

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, consulting services and network management services. For detailed information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer's expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel's customer support is quite extensive. It can start with assistance during your development effort to network management. 100 Intel sales and service offices are located worldwide – in the U.S., Canada, Europe and the Far East. So wherever you're using Intel technology, our professional staff is within close reach.

HARDWARE SUPPORT SERVICES

Intel's hardware maintenance service, starting with complete on-site installation will boost your productivity from the start and keep you running at maximum efficiency. Support for system or board level products can be tailored to match your needs, from complete on-site repair and maintenance support to economical carry-in or mail-in factory service.

Intel can provide support service for not only Intel systems and emulators, but also support for equipment in your development lab or provide service on your product to your end-user/customer.

SOFTWARE SUPPORT SERVICES

Software products are supported by our Technical Information Service (TIPS) that has a special toll free number to provide you with direct, ready information on known, documented problems and deficiencies, as well as work-arounds, patches and other solutions.

Intel's software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and; *COMMENTS Magazine*). Basic support consists of updates and the subscription service. Contracts are sold in environments which represent product groupings (e.g., iRMX® environment).

CONSULTING SERVICES

Intel provides field system engineering consulting services for any phase of your development or application effort. You can use our system engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training and customizing an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

CUSTOMER TRAINING

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers worldwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel's major course categories include: architecture and assembly language, programming and operating systems, BITBUS™ and LAN applications.

NETWORK MANAGEMENT SERVICES

Today's networking products are powerful and extremely flexible. The return they can provide on your investment via increased productivity and reduced costs can be very substantial.

Intel offers complete network support, from definition of your network's physical and functional design, to implementation, installation and maintenance. Whether installing your first network or adding to an existing one, Intel's Networking Specialists can optimize network performance for you.

Table of Contents

Alphanumeric Index

OVERVIEW

Imagine for a moment a world where all electronic communications were instantaneous. A world where voice, data, and graphics could all be transported via telephone lines to a variety of computers and receiving systems. A world where the touch of a finger could summon information ranging from stock reports to classical literature and bring it into environments as diverse as offices and labs, factories and living rooms.

Unfortunately, these promises of the Information Age still remain largely unfulfilled. While computer technology has accelerated rapidly over the last twenty years, the communications methods used to tie the wide variety of electronic systems in the world together have, by comparison, failed to keep pace. Faced with a tangle of proprietary offerings, high costs, evolving standards, and incomplete technologies, the world is still waiting for networks that are truly all-encompassing, the missing links to today's communications puzzle.

Enter microcommunications—microchip-based digital communications products and services. A migration of the key electronics communications functions into silicon is now taking place, providing the vital interfaces that have been lacking among the various networks now employed throughout the world. Through the evolution of VLSI (Very Large Scale Integration) technology, microcommunications now can offer the performance required to effect these communications interfaces at affordable costs, spanning the globe with silicon to eradicate the troublesome bottleneck that has plagued information transfer during recent years.

"There are three parts to the communications puzzle," says Gordon Moore, Intel Chairman and CEO. "The first incorporates the actual systems that communicate with each other, and the second is the physical means to connect them—such as cables, microwave technology, or fiber optics. It is the third area, the interfaces between the systems and the physical links, where silicon will act as the linchpin. That, in essence, is what microcommunications is all about."

THE COMMUNICATIONS BOTTLENECK

Visions of global networks are not new. Perhaps one of the most noteworthy of these has been espoused by Dr. Koji Kobayashi, chairman of NEC Corporation. His view of the future, developed over the nearly fifty years of his association with NEC, is known as C&C (Computers and Communications). It defines the marriage of passive communications systems and computers as processors and manipulators of information, providing the foundation for a discipline that is changing the basic character of modern society.

Kobayashi's macro vision hints at the obstacles confronting the future of C&C. When taken to the micro level, to silicon itself, one begins to understand the complexities that are involved. When Intel invented the microprocessor fifteen years ago, the first seeds of the personal computer revolution were sown , marking an era that over the last decade has dramatically influenced the way people work and live. PCs now proliferate in the office, in factories, and throughout laboratory environments. And their "intimidation" factor has lessened to where they are also becoming more and more prevalent in the home, beginning to penetrate a market that to date has remained relatively untapped.

Thanks to semiconductor technology, the personal computer has raised the level of productivity in our society. But most of that productivity has been gained by individuals at isolated workstations. Group productivity, meanwhile, still leaves much to be desired. The collective productivity of organizations can only be enhanced through more sophisticated networking technology. We are now faced with isolated "islands of automation" that must somehow be developed into networks of productivity.

But no amount of computing can meet these challenges if the corresponding communications technology is not sufficiently in step. The Information Age can only grow as fast as the lowest common denominator—which in this case is the aggregate communications bandwidth that continues to lag behind our increased computing power. Such is the nature of the communications bottleneck, where the growing amounts of information we are capable of generating can only flow as fast as the limited and incompatible communications capabilities now in place. Clearly, a crisis is at hand.

BREAKING UP THE BOTTLENECK

Three factors have contributed to this logjam: lack of industry standards, an insufficient cost/performance ratio, and the incomplete status of available communications technology to date.

- Standards—One look at the tangle of proprietary systems now populating office, factory, and laboratory environments gives a good indication of the inherent difficulty in hooking these diverse systems together. And these systems do not merely feature different architectures—they also represent completely different levels of computing, ranging from giant mainframes at one end of the scale down to individual microcontrollers on the other.

  The market has simply grown too fast to effectively accommodate the changes that have occurred. Suppliers face the dilemma of meshing product differentiation issues with industry-wide compatibility as

they develop their strategies; opting for one in the past often meant forsaking the other. And while some standards have coalesced, the industry still faces a technological Tower of Babel, with many proprietary solutions vying to be recognized in leadership positions.

- Cost/Performance Ratio—While various communications technologies struggle toward maturity, the industry has had to cope with tremendous costs associated with interconnectivity and interoperation. Before the shift to microelectronic interfaces began to occur, these connections often were prohibitively expensive.

  Says Ron Whittier, Intel Vice President and Director of Marketing: "Mainframes offer significant computing and communications power, but at a price that limits the number of users. What is needed is cost-effective communications solutions to hook together the roughly 16 million installed PCs in the market, as well as the soon-to-exist voice/data terminals. That's the role of microcommunications—bringing cost-effective communications solutions to the microcomputer world."
- Incomplete Technology—Different suppliers have developed many networking schemes, but virtually all have been fragmented and unable to meet the wide range of needs in the marketplace. Some of these approaches have only served to create additional problems, making OEMs and systems houses loathe to commit to suppliers who they fear cannot provide answers at all of the levels of communications that are now funneled into the bottleneck.

THE NETWORK TRINITY

Three principal types of networks now comprise the electronic communications marketplace: Wide Area Networks (WANs), Local Area Networks (LANs), and Small Area Networks (SANs). Each in its own fashion is turning to microcommunications for answers to its networking problems.

WANs—known by some as Global Area Networks (GANs)—are most commonly associated with the worldwide analog telephone system. The category also includes a number of other segments, such as satellite and microwave communications, traditional networks (like mainframe-to-mainframe connections), modems, statistical multiplexers, and front-end communications processors. The lion's share of nodes—electronic network connections—in the WAN arena, however, resides in the telecommunications segment. This is where the emerging ISDN (Integrated Services Digital Network) standard comes into focus as the most visible portion of the WAN marketplace.

The distances over which information may be transmitted via a WAN are essentially unlimited. The goal of ISDN is to take what is largely an analog global system and transform it into a digital network by defining the standard interfaces that will provide connections at each node.

These interfaces will allow basic digital communications to occur via the existing twisted pair of wires that comprise the telephone lines in place today. This would bypass the unfeasible alternative of installing completely new lines, which would be at cross purposes with the charter of ISDN: to reduce costs and boost performance through realization of an all-digital network.

The second category, Local Area Networks, represents the most talked-about link provided by microcommunications. In their most common form, LANs are comprised of—but not limited to—PC-to-PC connections. They incorporate information exchange over limited distances, usually not exceeding five kilometers, which often takes place within the same building or between adjacent work areas. The whole phenomenon surrounding LAN development, personal computing, and distributed processing essentially owes its existence to microcomputer technology, so it is not surprising that this segment of networking has garnered the attention it has in microelectronic circles.

Because of that, progress is being made in this area. The most prominent standard—which also applies to WANs and SANs—is the seven-layer Open Systems Interconnection (OSI) Model, established by the International Standards Organization (ISO). The model provides the foundation to which all LAN configurations must adhere if they hope to have any success in the marketplace. Interconnection protocols determining how systems are tied together are defined in the first five layers. Interoperation concepts are covered in the upper two layers, defining how systems can communicate with each other once they are tied together.

In the LAN marketplace, a large number of networking products and philosophies are available today, offering solutions at various price/performance points. Diverse approaches such as StarLAN, Token Bus and Token Ring, Ethernet, and PC-NET, to name a few of the more popular office LAN architectures, point to many choices for OEMs and end users.

A similar situation exists in the factory. While the Manufacturing Automation Protocol (MAP) standard is coalescing around the leadership of General Motors,

Boeing, and others, a variety of proprietary solutions also abound. The challenge is for a complete set of interfaces to emerge that can potentially tie all of these networks together in—and among—the office, factory, and lab environments.

The final third of the network trinity is the Small Area Network (SAN). This category is concerned with communications over very short distances, usually not exceeding 100 meters. SANs most often deal with chip-to-chip or chip-to-system transfer of information; they are optimized to deal with real-time applications generally managed by microcontrollers, such as those that take place on the factory floor among robots at various workstations.

SANs incorporate communications functions that are undertaken via serial backplanes in microelectronic equipment. While they represent a relatively small market in 1986 when compared to WANs and LANs, a tenfold increase is expected through 1990. SANs will have the greatest number of nodes among network applications by the next decade, thanks to their preponderance in many consumer products.

While factory applications will make up a large part of the SAN marketplace probably the greatest contributor to growth will be in automotive applications. Microcontrollers are now used in many dashboards to control a variety of engine tasks electronically, but they do not yet work together in organized and efficient networks. As Intel's Gordon Moore commented earlier this year to the New York Society of Security Analysts, when this technology shifts into full gear during the next decade, the total automobile electronics market will be larger than the entire semiconductor market was in 1985.

MARKET OPPORTUNITIES

Such growth is also mirrored in the projections for the WAN and LAN segments, which, when combined with SANs, make up the microcommunications market pie. According to Intel analysts, the total silicon microcommunications market in 1985 amounted to $522 million. By 1989, Intel predicts this figure will have expanded to $1290 million, representing a compounded annual growth rate of 25%.

And although the WAN market will continue to grow at a comfortable rate, the SAN and LAN pieces of the pie will increase the most dramatically. Whereas SANs represented only about 12.5% ($65 million) in 1985, they could explode to 22.5% ($290 million) of the larger pie by 1989. This growth is paralleled by increases in the LAN segment, which should grow from 34.5% of the total silicon microcommunications market in 1985 to 44.5% of the expanded pie in 1989.

Opportunities abound for microcommunications suppliers as the migration to silicon continues. And perhaps no VLSI supplier is as well-positioned in this marketplace as Intel, which predicts that 50% of its products will be microcommunications-related by 1990. The key here is the corporation's ability to bridge the three issues that contribute to the communications bottleneck: standards, cost-performance considerations, and the completeness of microcomputer and microcommunications product offerings.

INTEL AND VLSI: THE MICROCOMMUNICATIONS MATCH

Intel innovations helped make the microcomputer revolution possible. Such industry "firsts" include the microprocessor, the EPROM, the E2PROM, the microcontroller, development systems, and single board computers. Given this legacy, it is not surprising that the corporation should come to the microcommunications marketplace already equipped with a potent arsenal of tools and capabilities.

The first area centers on industry standards. As a VLSI microelectronic leader, Intel has been responsible for driving many of the standards that are accepted by the industry today. And when not actually initiating these standards, Intel has supported other existing and emerging standards through its longtime "open systems" philosophy. This approach protects substantial customer investments and ensures easy upgradability by observing compatibility with previous architectures and industry-leading standards.

Such a position is accentuated by Intel's technology relationships and alliances with many significant names in the microcommunications field. Giants like AT&T in the ISDN arena, General Motors in factory networking, and IBM in office automation all are working closely with Intel to further the standardization of the communications interfaces that are so vital to the world's networking future.

Cost/performance considerations also point to Intel's strengths. As a pioneer in VLSI technology, Intel has been at the forefront of achieving greater circuit densities and performance on single pieces of silicon: witness the 275,000 transistors housed on the 32-bit 80386, the highest performance commercial microprocessor ever built. As integration has increased, cost-per-bit has decreased steadily, marking a trend that remains consistent in the semiconductor industry. And one thing is

certain: microcommunications has a healthy appetite for transistors, placing it squarely in the center of the VLSI explosion.

But it is in the final area—completeness of technology and products—where Intel is perhaps the strongest. No other microelectronic vendor can point to as wide an array of products positioned across the various segments that comprise the microelectronic marketplace. Whether it be leadership in the WAN marketplace as the number one supplier of merchant telecommunications components, strength in SANs with world leadership in microcontrollers, or overall presence in the LAN arena with complete solutions in components, boards, software, and systems, Intel is a vital presence in the growing microcommunications arena.

That leadership extends beyond products. Along with its own application software, Intel is promoting expansion through partnerships with many different independent software vendors (ISVs), ensuring that the necessary application programs will be in place to fuel the gains provided by the silicon "engines" residing at the interface level. And finally, the corporation's commitment to technical support training, service, and its strong force of field applications engineers guarantees that it will back up its position and serve the needs that will continue to spring up as the microcommunications evolution becomes a reality.

Together, all the market segment alluded to in this article comprise the world of microcommunications, a world coming closer together every day as the web of networking solutions expands—all thanks to the technological ties that bind, reaching out to span the globe with silicon.

Local Area Networks 1

intel®

APPLICATION NOTE

AP-235

November 1986

An 82586 Data Link Driver

CHARLES YAGER

Order Number: 231421-002

INTRODUCTION

This application note describes a design example of an IEEE 802.2/802.3 compatible Data Link Driver using the 82586 LAN Coprocessor. The design example is based on the "Design Model" illustrated in "Programming the 82586". It is recommended that before reading this application note, the reader clearly understands the 82586 data structures and the Design Model given in "Programming the 82586".

"Programming the 82586" discusses two basic issues in the design of the 82586 data link driver. The first is how the 82586 handler fits into the operating system. One approach is that the 82586 handler is treated as a "special kind of interface" rather than a standard I/O interface. The special interface means a special driver that has the advantage of utilizing the 82586 features to enhance performance. However the performance enhancement is at the expense of device dependent upper layer software which precludes the use of a standard I/O interface.

The second issue "Programming the 82586" discusses is which algorithms to choose for the CPU to control the 82586. The algorithms used in this data link design are taken directly from "Programming the 82586". Command processing uses a linear static list, while receive processing uses a linear dynamic list.

The application example is written in C and uses the Intel C compiler. The target hardware for the Data Link Driver is the iSBC 186/51 COMMputer, however a version of the software is also available to run on the LANHIB Demo board.

1.0 FITTING THE SOFTWARE INTO THE OSI MODEL

The application example consists of four software modules:

- Data Link Driver (DLD): drives the 82586, also known as the 82586 Handler.
- Logical Link Control (LLC): implements the IEEE 802.2 standard.
- User Application (UAP): exercises the other software modules and runs a specific application.
- C hardware support: written in assembly language, supports the Intel C compiler for I/O, interrupts, and run time initialization for target hardware.

Figure 1 illustrates how these software modules combined with the 82586, 82501 and 82502 complete the first two layers of the OSI model. The 82502 implements an IEEE 802.3 compatible transceiver, while the 82501 completes the Physical layer by performing the serial interface encode/decode function.

The Data Link Layer, as defined in the IEEE 802 standard documents, is divided into two sublayers: the Logical Link Control (LLC) and the Medium Access Control (MAC) sublayers. The Medium Access Control sublayer is further divided into the 82586 Coprocessor plus the 82586 Handler. On top of the MAC is the LLC software module which provides IEEE 802.2 compatibility. The LLC software module implements the Station Component responses, dynamic addition and deletion of Service Access Points (SAPs), and a class 1 level of service. (For more information on the LLC sublayer, refer to IEEE 802.2 Logical Link Control Draft Standard.) The class 1 level of service provides a connectionless datagram interface as opposed to the class 2 level of service which provides a connection oriented level of service similar to HDLC Asynchronous Balanced Mode.

On top of the Data Link Layer is the Upper Layer Communications Software (ULCS). This contains the Network, Transport, Session, and Presentation Layers. These layers are not included in the design example, therefore the application layer of this ap note interfaces directly to the Data Link layer.

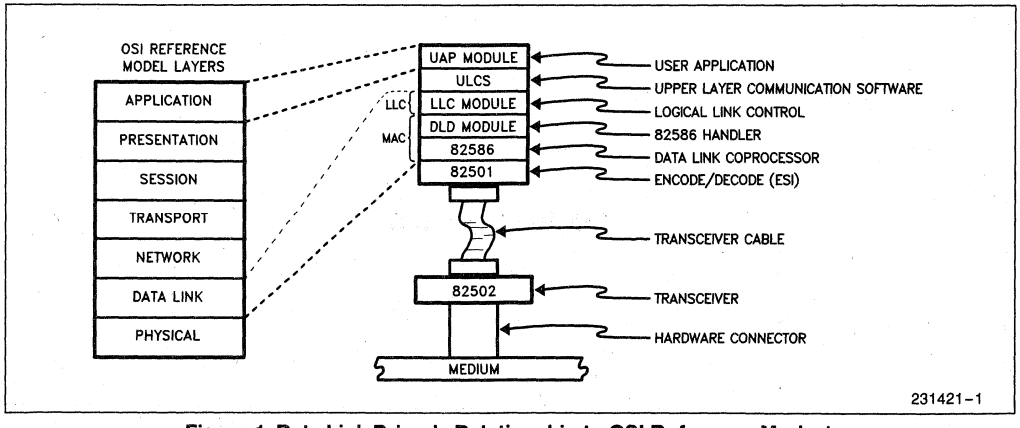


Figure 1. Data Link Driver's Relationship to OSI Reference Mode 1

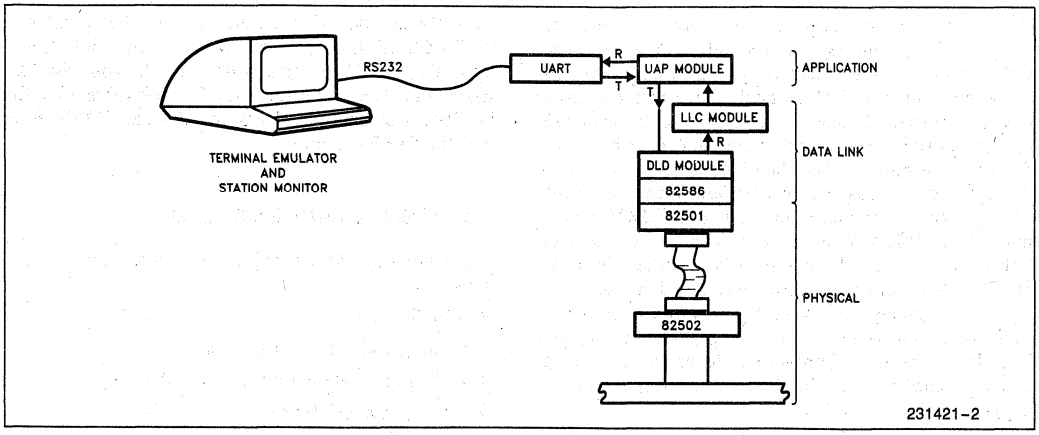


Figure 2. Block Diagram of the Hardware and Software

The application layer is implemented in the User Application (UAP) software module. The UAP module operates in one of three modes: Terminal Mode, Monitor Mode, and High Speed Transmit Mode. The software initially enters a menu driven interface which allows the program to modify several network parameters or enter one of the three modes.

The Terminal Mode implements a virtual terminal with datagram capability (connectionless "class 1" service). This mode can also be thought of as an async to IEEE 802.3/802.2 protocol converter.

The Monitor Mode provides a dynamic update on the terminal of 6 station related parameters. While in the monitor mode, any size frame can be repeatedly transmitted to the cable in a software loop.

High Speed Transmit Mode transmits frames to the cable as fast as the software possibly can. This mode demonstrates the throughput performance of the Data Link Driver.

The UAP gathers network statistics in all three modes as well as when it is in the menu. In addition, the UAP module provides the capability to alter MAC and LLC addresses and re-initialize the data link. (Figure 2 shows a combined software and hardware block diagram.)

2.0 LARGE MODEL COMPILATION

All the modules in this design example are compiled under the Large Model option. This has the advantages of using the entire 1 Mbyte address space, and allowing the string constants to be stored in ROM. In the Large Model it is important to consider that the 82586's data structures, SCB, CB, TBD, FD, and RBD, must reside within the same data segment. This data segment is determined at locate time.

The C_Assy_Support module has a run time start off function which loads the DLD data segment into a global variable SEGMT_. This data segment is used by the 82586 Handler for address translation purposes. The 82586 uses a flat address while the 80186 uses a segmented address. Any time a conversion between 82586 and 80186 addresses are needed the SEGMT_ variable is used.

Pointers for the 80186 in the large model are 32 bits, segment and offset. All the 82586 link pointers are 16 bit offsets. Therefore when trading pointers between the 82586 and the 80186, two functions are called: Offset (ptr), and Build_Ptr (offset). Offset (ptr) takes a 32 bit 80186 pointer and returns just the offset portion for the 82586 link pointer. While Build_Ptr (offset) takes an 82586 link pointer and returns a 32 bit 80186 pointer, with the segment part being the SEGMT_ variable. Offset ( ) and Build_Ptr( ) are simple functions written in assembly language included in the C_Assy_Support module.

In the small model, Offset ( ) and Build_Ptr( ) are not needed, but the variable SEGMT_ is still needed for determining the SCB pointer in the ISCP, and in the Transmit and Receive Buffer Descriptors.

3.0 THE 82586 HANDLER

3.1 The Buffer Model

The buffer model chosen for the 82586 Handler is the "Design Model" as described in "Programming the 82586". This is based on the 82586 driver as a special driver rather than as a standard driver. Using this approach the ULCS directly accesses the 82586's Transmit and Receive Buffers, Buffer Descriptors and Frame Descriptors. This eliminates buffer copying. Transmit and receiver buffer passing is done entirely through pointers.

The only hardware dependencies between the Data Link and ULCS interface are the buffer structures. The ULCS does not handle the 82586's CBs, SCB or initialization structures. To isolate the data link interface from any hardware dependencies while still using the design model, another level of buffer copying must be introduced. For example, when the ULCS transmits a frame it would have to pass its own buffers to the data link. The data link then copies the data from ULCS buffers into 82586 buffers. When a frame is received, the data link copies the data from the 82586's buffers into the ULCS buffers. The more copying that is done the slower the throughput. However, this may be the only way to fit the data link into the operating system. The 82586 Handler can be made hardware independent by adding a receive and transmit function to perform the buffer copying.

The 82586 Handler allocates buffers from two pools of memory: the Transmit pool, and the Receive pool as illustrated in Figure 3. The Transmit pool contains Transmit Buffer Descriptors (TBDs) and Transmit Buffers (TBs). The Receive pool contains Frame Descriptors (FDs), Receive Buffer Descriptors (RBDs), and Receive Buffers (RBs).

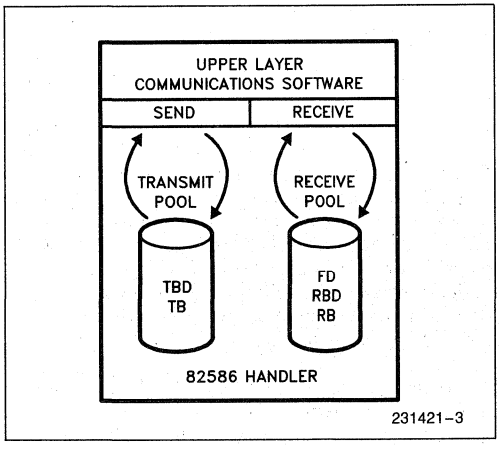


Figure 3. 82586 Handler Memory Management Model

When the ULCS wants to transmit, it requests a TBD from the handler. The handler returns a pointer to a free TBD. Each TBD has a TB attached to it. The ULCS fills the buffer, sets the appropriate fields in the TBD, and passes the TBD pointer back to the handler for transmission. After the frame is transmitted, the handler places the TBD back into the free TBD pool. If the ULCS needs more than one buffer per frame, it simply requests another TBD from the handler and performs the necessary linkage to the previous TBD.

On the receive side, the RFA pool is managed by the 82586 itself. When a frame is received, the 82586 interrupts the handler. The handler passes a FD pointer to the ULCS. Linked to the FD is one or more RBDs and RBs. The ULCS extracts what it needs from the FD, RBDs and RBs, and returns the FD pointer back to the handler. The handler places the FD and RBDs back into the free RFA pool.

3.2 The Handler Interface

The handler interface provides the following basic functions:

- initialization
- sending and receiving frames
- adding and deleting multicast addresses
- getting transmit buffers
- returning receive buffers

Figure 4 lists the Handler Interface functions.

On power up, the initialization function is called. This function initializes the 82586, and performs diagnostics. After initialization, the handler is ready to transmit and receive frames, and add and delete multicast addresses.

To send a frame, the ULCS gets one or more transmit buffers from the handler, fills them with data, and calls the send function. When a frame is received, the handler calls a receive function in the ULCS. The ULCS receive function removes the information it needs and returns the receive buffers to the handler. The addition and deletion of multicast addresses can be done "on the fly" any time after initialization. The **receiver doesn't have to be disabled** when this is done.

The command interface to the handler is totally asynchronous—the ULCS can issue transmit commands or multicast address commands whenever it wants. The commands are queued by the handler for the 82586 to execute. If the command queue is full, the send frame procedure returns a false status rather than true. The size of the command queue can be set at compile time by setting the CB—CNT constant. Typically the command queue never has more than a few commands on it because the 82586 can execute commands faster than the ULCS can issue them. This is not the case in a heavily loaded network when deferrals, collisions, and retries occur.

The command interface to the 82586 handler is hardware independent; the only hardware dependence is the buffering. A hardware independent command interface doesn't have any performance penalty, but some 82586 programmability is lost. This shouldn't be of concern since most data links do not change configuration parameters during operation. One can simply modify a few constants and recompile to change frame and network parameters to support other data links.

Handler Interface Functions	Description
Init_586( )	Initialize the Handler
Send_Frame (ptbd, padd)	Sends a frame to the cable. ptbd—Transmit Buffer Descriptor pointer padd—Destination Address pointer
Recv_Frame (pfd)	Handler calls this function which resides in the ULCS. pfd_Frame Descriptor pointer
Add_Multicast_Address (pma)	Adds one multicast address pma—Multicast Address pointer
Delete_Multicast_Address (pma)	Deletes one multicast address
Get_Tbd( )	Get a Transmit Buffer Descriptor pointer
Put_Free_Rfa (pfd)	Returns a Frame Descriptor and Receive Buffer Descriptors to the 82586.

Figure 4. List of Handler Interface Functions

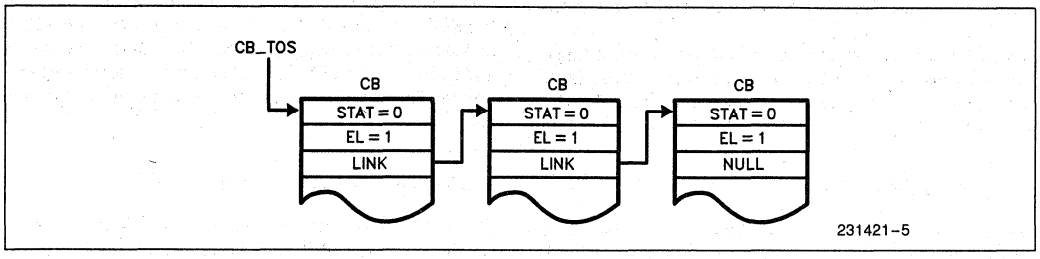


Figure 5. Free CB Pool

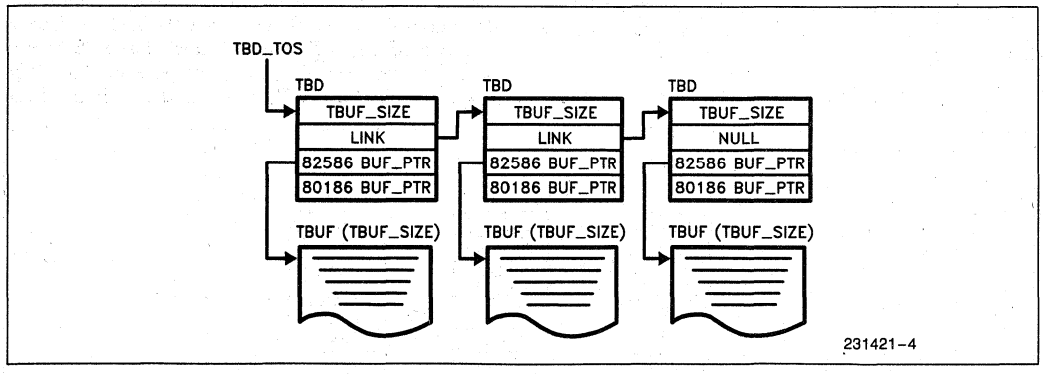


Figure 6. Free Transmit Buffer Descriptor Pool

3.3 Initialization

The function which initializes the 82586 handler, Init_586( ), is called by the ULCS on power up or reinitialization. **Before this function is called, an 82586 hardware or software reset should occur.** The Initialization occurs in three phases. The first phase is to initialize the memory. This includes flags, vectors, counters, and data structures. The second phase is to initialize the 82586. The third phase is to perform self test diagnostics. Init_586( ) returns a status byte indicating the results of the diagnostics.

Init_586( ) begins by toggling the 82501 loopback pin. If the 82501 is powered up in loopback, the $\overline{CRS}$ and $\overline{CDT}$ pin may be active. To reset this condition, the loopback pin is toggled. The 82501 should remain in loopback for the first part of the initialization function.

Phase 1 executes initialization of all the handlers flags, interrupt vectors, counters, and 82586 data structures. There are two separate functions which initialize the CB and RFA pools: Build_CB( ) and Build_Rfa( ).

3.3.1 BUILDING THE CB AND RFA POOLS

Build_CB( ) builds a stack of free linked Command Blocks, and another stack of free linked Transmit Buffer Descriptors. (See Figures 5 and 6.) Each stack has a Top of Stack pointer, which points to the next free structure. The last structure on the list has a NULL link pointer.

The CBs within the list are initialized with 0 status, EL bit set, and a link to the next CB. The TBD structures are initialized with the buffer size, which is set at compile time with the TBUF_SIZE constant, a link to the next TBD, and an 82586 pointer to the transmit buffer. This pointer is a 24 bit flat/physical address. The address is built by taking the transmit buffer's data segment address, shifting it to the left by 4 and adding it to the transmit buffer offset. An 80186 pointer to the transmit buffer is added to the TBD structure so that the 80186 does not have to translate the address each time it accesses the transmit buffer.

Build_Rfa( ) builds a linear linked Frame Descriptor list and a Receive Buffer Descriptor list as shown in Figure 7. The status and EL bits for all the free FDs are 0. The last FD's EL bit is 1 and link pointer is NULL. The first FD on the FD list points to the first RBD on the RBD list. The RBDs are initialized with both 82586 and 80186 buffer pointers. The 80186 buffer pointer is added to the end of the RBD structure. Begin and end pointers are used to mark the boundaries of the free lists.

3.3.2 82586 INITIALIZATION

The 82586 initialization data structure SCP is already set since it resides in ROM, however, the ISCP must be loaded with information. Within the SCP ROM is the pointer to the ISCP; the ISCP is the only absolute address needed in the software. Once the ISCP address is determined, the ISCP can be loaded. The SCB base is obtained from the C_Assy_Support module. The global variable SEGMT_contains the address of the

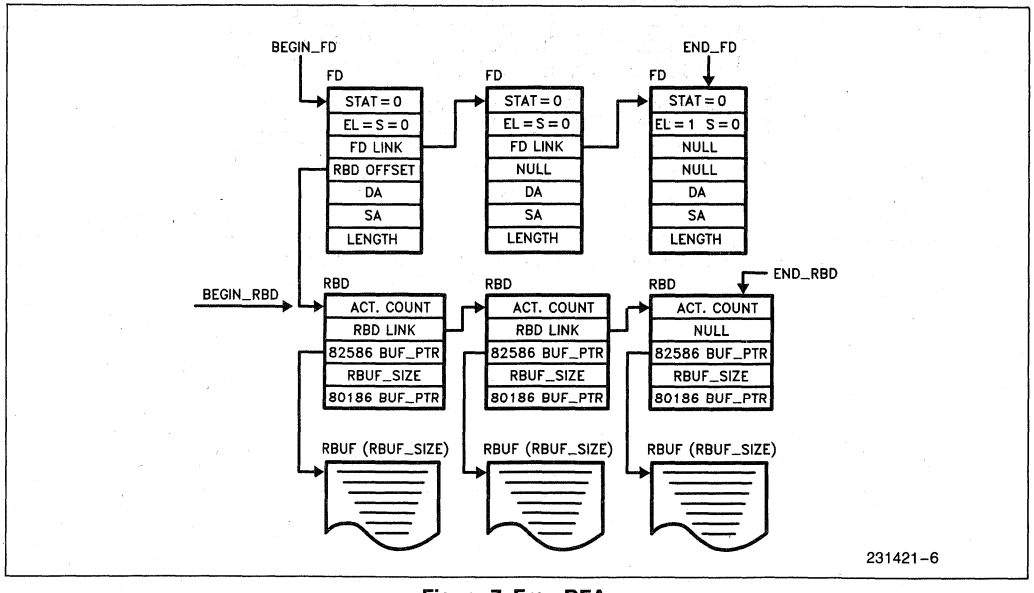


Figure 7. Free RFA

data segment of the handler. The 80186 shifts this value to the left by 4 and loads it into the SCB base. The SCB offset is now determined by taking the 32 bit SCB pointer and passing it to the Offset( ) function.

The 82586 interrupt is disabled during initialization because the interrupt function is not designed to handle 82586 reset interrupts. To determime when the 82586 is finished with its reset/initialization, the SCB status is polled for both the CX and CNA bits to be set. After the 82586 is initialized, both the CX and CNA interrupts are acknowledged.

The 82586 is now ready to execute commands. The Configuration is executed first to place the 82586 in internal loopback mode, followed by the IA command. The address for the IA command is read off of a prom on the PC board.

3.3.3 SELF TEST DIAGNOSTICS

The final phase of the handler initialization is to run the self test diagnostics. Four tests are executed: Diagnose command, Internal loopback, External loopback through the 82501, and External loopback through the transceiver. If these four tests pass, the data link is ready to go on line.

The function that executes these diagnostics is called Test_Link( ). If any of the tests fail, Test_Link( ) returns immediately with the Self_Test global variable set to the type of failure. This Self_Test global variable is then returned to the function which originally called Init_586( ). Therefore Init_586( ) can return one of five results: FAILED_DIAGNOSE, FAILED_LPBK_INTERNAL, FAILED_LPBK_EXTERNAL, FAILED_LPBK_TRANSCEIVER or PASSED.

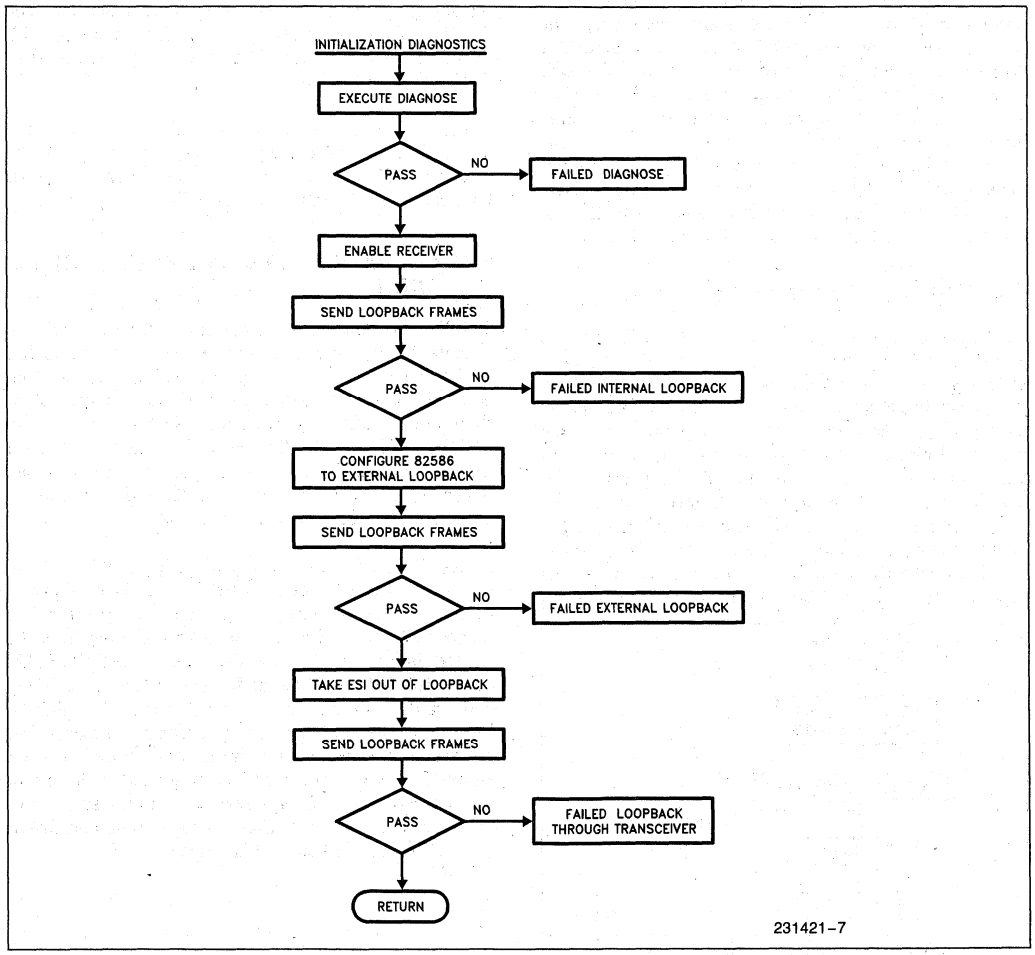


Figure 8. Initialization Diagnostics: Test_Link ( )

The Diagnose( ) function, called by Test_Link( ), does not return until the diagnose command is completed. If the interrupt service routine detects that a Diagnose command was completed then it sets a flag to allow the Diagnose( ) function to return, and it also sets the Self_Test variable to FAIL if the Diagnose command failed. If the Diagnose command completed successfully, the loopback tests are performed.

Before any loopback tests are executed, the Receive Unit is enabled by calling Ru_Start( ). Loopback tests begin by calling Send_Lpbk_Frame( ), which sends 8 frames with known loopback data and its own destination address. More than one loopback frame is sent in case one or more of them are lost. Also several of the frames will have been received by the time flags.lpbk_test is checked.

Two flag bits are used for the loopback tests: flags.lpbk_mode, and flags.lpbk_test. flags.lpbk_mode is used to indicate to the receive section that the frames received are potentially loopback frames. The receive section will pass receive frames to the Loopback Check( ) function if the flags.lpbk_mode bit is set. The Loopback_Check( ) function first compares the source address of the frame with its station address. If this matches then the data is checked with the known loopback data. If the data matches, then the flags.lpbk_test bit is set, indicating a successful loopback. The flow of the Test_Link( ) function is displayed in Figure 8.

3.4 Command Processing

Command blocks are queued up on a static list for the 82586 to execute. The flow of a command block is given in Figure 9. When the handler executes a command it first has to get a free command block. It does this by calling Get_CB( ) which returns a pointer to a free command block. The CB structure is a generic one in which all commands except the MC-Setup can fit in. The handler then loads into the CB structure the type of command and associated parameters. To issue the command to the 82586 the Issue_CU_Cmd( ) function is called with the pointer to the CB passed to this function. Issue_CU_Cmd( ) places the command on the 82586's static command block list. After the 82586 executes the command, it generates an interrupt. The interrupt routine, Isr_586( ), processes the command and returns the Command Block to the free command block list by calling Put_Cb( ).

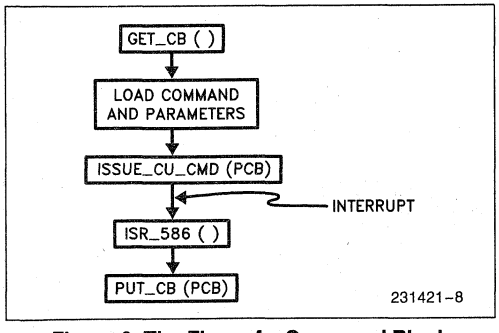


Figure 9. The Flow of a Command Block

3.4.1 ACCESSING COMMAND BLOCKS-GET_CB() and PUT_CB()

Get_Cb( ) returns a pointer to a free command block. The free command blocks are in a linear linked list structure which is treated as a stack. The pointer cb_tos points to the next available CB. Each time a CB is requested, Get_Cb( ) pops a CB off the stack. It does this by returning the pointer of cb_tos. cb_tos is then updated with the CB's link pointer. When the CB list is empty, Get_Cb( ) returns NULL.

There are two types of nulls, the 82586 'NULL' is a 16 bit offset, OFFFFH, in the 82586 data structures. The 80186 null pointer, 'pNULL', is a 32 bit pointer; with OFFFFH offset and the 82586 handler's data segment, SEGMT_, as the base.

Put_Cb( ) pushes a free command block back on the list. It does this by placing the cb_tos variable in the returned CB's link pointer field, then updates cb_tos with the pointer to the returned CB.

3.4.2 ISSUING CU COMMANDS-ISSUE_CU_CMD()

This function queues up a command for the 82586 to execute. Since static lists are used, each command has its EL bit set. There is a begin_cbl pointer and an end_cbl pointer to delineate the 82586's static list. If there are no CBs on the list, then begin_cbl is set to pNULL. (Figure 10 illustrates the static list.) Each time a command is issued, a deadman timer is set. When the 82586 interrupts the CPU with a command completed, the deadman timer is reset.

Issue_Cu_Cmd( ) begins by disabling the 82586's interrupt. It then determines whether the list is empty or not. If the list is empty, begin and end pointers are loaded with the CB's address. The CU must then be started. Before a CU_START can be issued, the SCB's cbl_offset field must be loaded with the address of the command, the Wait_Scb( ) function must be called to insure that the SCB is ready to accept a command, and the deadman timer must be initialized. If the list is not empty, then the command block is queued at the end of the list, and the interrupt service routine Isr_586( ), will continue generating CAs for each command linked on the CB list until the list is empty.

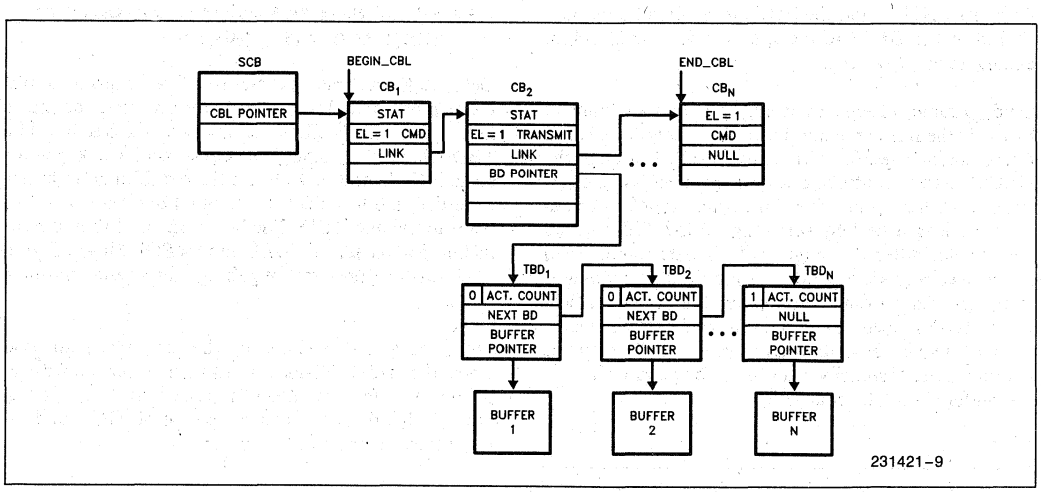


Figure 10. The Static Command Block List

3.4.3 INTERRUPT SERVICE ROUTINE-ISR__586()

Isr__586( ) starts off by saving the interrupts that were generated by the 82586 and acknowledging them. Acknowledgment must be done immediately because if a second interrupt were generated before the acknowledgment, the second interrupt would be missed. The interrupt status is then checked for a receive interrupt and if one occurred the Recv__Int__Processing( ) function is called. After receive processing is check the CPU checks whether a command interrupt occurred. If one did, then the deadman timer is reset and the results of the command are checked. There are only two particular commands which the interrupt results are checked for: Transmit and Diagnose. The Diagnose command needs to be tested to see if it passed, plus the diagnose status flag needs to be set so that the initialization process can continue.

The transmit command status provides network management and station diagnostic information which is useful for the "Network Management" function of the ISO model. The following statistics are gathered in the interrupt routine: good__transmit__cnt, sqe__err__cnt, defer__cnt, no__crs__cnt, underrun__cnt, max__col__cnt. To speed up transmit interrupt processing a flag is tested to determine whether these statistics are desired, if not this section of code is skipped.

The sqe error requires special considerations when used for statistic gathering or diagnostics. The sqe status bit indicates whether the transceiver passed its self test or not. The transceiver executes a self test after each transmission. If the transceiver's self test passed, it will activate the collision signal during the IFS time.

The sqe status bit will be set if the transceiver's self test passed. However if the sqe status bit is not set, the transceiver may still have passed its self test. Several events can prevent the sqe bit from being set. For example, the first transmit command status after power up will not have the sqe bit set because the sqe is always from the previous command. Also if any collisions occur, the sqe bit might not be set. This has to do with the timing of when the sqe signal comes from the transceiver. It is possible that a JAM signal from a remote station can overlap the sqe signal in which case the 82586 will not set the sqe status bit. Therefore the sqe error count should only be recorded when no collisions occur.

One other situation can occur which will prevent the SQE status bit from being set. If transmit command reaches the maximum retry count, the next transmit command's SQE bit will not be set.

The final phase of interrupt command processing determines if another command is linked, and returns the CB to the free command block list. Another command being linked is indicated by the CB link field not being NULL. In this case the deadman timer and the 82586's CU are re-started. If the CB link is NULL, there are no further commands to execute, and begin__cbl is set to pNULL.

3.4.4 SENDING FRAMES-SEND__FRAME (PTBD, PADD)

Send__Frame( ) receives two parameters, a pointer to the first Transmit Buffer Descriptor, and a pointer to the destination address. There may be one or more TBDs attached. The last TBD is indicated by its link

field being NULL and the EOF bit set. It is the responsibility of the ULCS to make sure this is done before calling Send__Frame( ).

Send__Frame( ) begins by trying to obtain a command block. If the free command block list is empty, the send frame function returns with a false result. It is up to the ULCS to either continue attempting transmission or attempt at a later time. The send frame function calculates the length field by summing up the TBDs actual count field. After the length field is determined, send frame checks to see if padding is required. If padding is necessary, Send Frame will change the act count field in the TBD to meet the minimum frame requirements. This technique transmits what ever was in the buffer as padding data. If security is an issue, the padding data in the buffer should be changed.

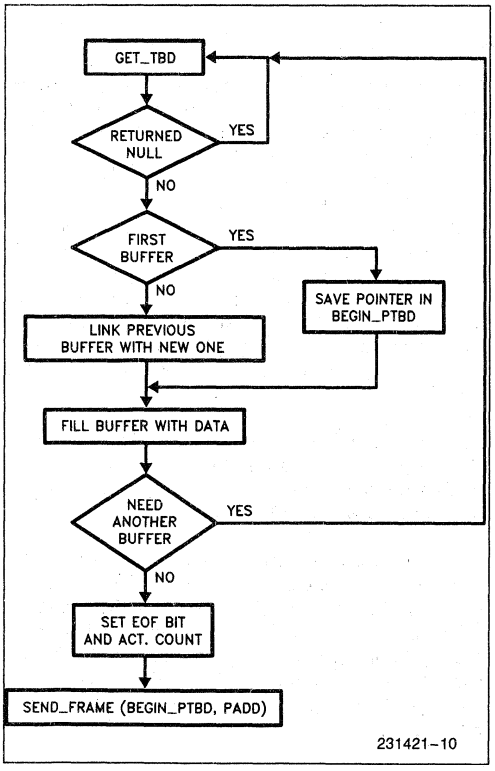


Figure 11. Flow Chart for Sending a Frame

3.4.5 ACCESSING TRANSMIT BUFFERS-GET__TBD() AND PUT__TBD()

Get__Tbd( ) returns a pointer to a free Transmit Buffer Descriptor, and Put__Tbd( ) returns one or more linked Transmit Buffer Descriptors to the free list. The TBD which Get__Tbd( ) allocates has its link pointer set to NULL, and its EOF bit cleared. If another buffer is needed, the link field in the old TBD must be set to point to the new TBD. The last TBD used should have its link pointer set to NULL and its EOF bit set. Figure 11 shows the flow chart of getting buffers and sending a frame.

Put__Tbd (ptbd) is called by the Isr__586( ) function when the 82586 is done transmitting the buffers. A pointer to the first TBD is passed to Put__Tbd( ). Put__Tbd( ) finds the end of the list of TBDs and returns them to the free buffer list.

3.4.6 MULTICAST ADDRESSES

The 82586 handler maintains a table of multicast addresses. Initially this table is empty. To enable a multicast address the Add__Multicast__Address(pma) function is called; to disable a multicast address, Delete__Multicast__Address(pma) function is called. Both functions accept a parameter which points to the multicast address. Add and Delete functions perform linear searches through the Multicast Address Table (MAT).

Add scans the entire MAT once to check if the address being added is a duplicate of one already loaded. Add will not enter a duplicate muilticast address. If there are no duplicates Add goes to the beginning of the MAT and looks for a free location. If it finds one, it loads the new address into the free location and sets the location status to INUSE. If no free locations are available, Add returns a false result.

Delete looks for a used location in the MAT. When it finds one, it compares the address in the table with the address passed to it. If they match, the location status is set to FREE and a TRUE result is returned. If no match occurs, the result returned is FALSE.

If Add or Delete change the MAT, they update the 82586 by calling Set__Multicast__Address( ). This function executes an 82586 MC Setup command. Set__Mulitcast__Address( ) uses the addresses in the MAT to build the MC Setup command. The MC Setup command is too big to be built from the free CBs. Free CB

command blocks are 18 bytes long, while the MC Setup command can be up to 16,392 bytes. Therefore a separate Multicast Address Command Block (ma_cb) must be allocated and used. The size of the ma_cb and MAT are determined at compile time based on the MULTI_ADDR_CNT constant. The design example allows up to 16 multicast addresses.

Since there is only one ma_cb, and it is not compatible with the other CBs, it must be treated differently. Only one ma_cb can be on the 82586 command list. The ma_cb command word is used as a semaphore. If it is zero, the command is available. If not, Set_Multicast_Address( ) must wait until the ma_cb is free. Also the interrupt routine can't return the ma_cb to the free CB list. It just clears the cmd field, to indicate that ma_cb is available.

The 82586's receiver does not have to be disabled to execute the MC Setup command. If the 82586 is receiving while this command is accessed, the 82586 will finish reception before executing the MC Setup comand. If the MC Setup command is executing, the 82586 automatically ignores incoming frames until the MC Setup is completed. Therefore multicast addresses can be added and deleted on the fly.

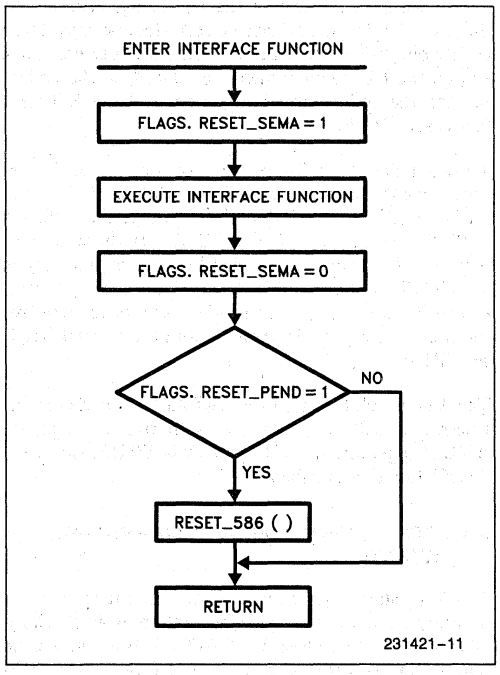


Figure 12. Reset Semaphore

3.4.7 RESETTING THE 82586-RESET_586()

The 82586 rarely if ever locks up in a well behaved network; (i.e. one that obeys IEEE 802.3 specifications). The lock-ups identified were artificially created and would normally not occur. This data link driver has been tested in an 8 station network under various loading conditions. No lock-ups occurred under any of the data link drivers test conditions. However the reset software has been tested by simulating a lockup. This can be done by having the 82586 transmit, and disabling the CTS pin for a time longer than the deadman timer.

An 82586 deadlock is not a fatal error. The handler is designed to recover from this problem. As mentioned before, each time the 82586 is given a CA to begin executing a command, a deadman timer is set. The deadman timer is reset when a CNR interrupt is generated. If the CNR interrupt is not generated before the deadman timer expires, the 82586 must be reset.

Resetting of the 82586 should not be done while the handler software is executing. This could create a software deadlock by interrupting a critical section of code in the handler. To insure that the Reset_586( ) function is not executed while the handler is executing, all of the entry points to the handler (i.e. interface functions) set a semaphore flag bit called flags.reset_sema. This flag is cleared when the interface functions are exited.

If the Deadman timer interrupt occurs while flags.reset_sema is set, another flag is set (flag.reset_pend) indicating that the Reset_586( ) function should be called when the interface functions are exited. However if the deadman timer interrupt occurs when flags.reset_sema is clear, Reset_586( ) is called immediately. Figure 12 shows the logic for entering and exiting interface functions.

Reset_586( ) begins by disabling the 82586 interrupt, placing the ESI in loopback, and resetting the 82586. The reset can be a software or a hardware reset. However, there are certain lockups in the 82586 where only a hardware reset will suffice. (The 82586 errata sheet explicitly indicates which deadlocks require a hardware reset.) After the reset, Reset_586( ) executes a Configure, IA-Setup, and a MC-Setup command; the MC_Setup command is built from the multicast address table (MAT). The 82586 Command Queues and Receive Frame Queues are left untouched so that the 82586 can continue executing where it left off before the deadlock. This way no frames or commands are lost. This requires that a separate reset CB and reset Multicast CB is used, because other CBs already in use cannot be disturbed.

3.5 Receive Frame Processing

The following functions are used for Receive Frame Processing:

Recv_Int_Processing( )	Called by Isr_586( ) to remove FDs and RBDs from the 82586's RFA
Recv_Frame (pfd)	Called by Recv_Int_Processing( ). This function resides in the ULCS
Check_Multicast (pfd)	Used for perfect Multicast filtering
Put_Free_Rfa (pfd)	Returns FDs and RBDs to the 82586's RFA
Ru_Start( )	Restarts the RU when in the IDLE or No Resources state.

3.5.1 RECEIVE INTERRUPT PROCESSING-RECV_INT_PROCESSING()

The Recv_Int_Processing( ) function is called by Isr_586( ) when the FR bit in the SCB is set. The Recv_Int_Processing( ) function checks whether any FDs and RBDs on the free list have been used by the 82586. If they have, Recv_Int_Processing( ) removes the used FDs and RBDs from the free list, and passes them to the ULCS.

The Recv_Int_Processing( ) function is a loop where each pass removes a frame from the 82586's RFA. When there are no more used FDs and RBDs on the RFA, the function calls RU_Start( ), then returns to Isr_586( ). The first part of the loop checks to see if the C bit in the first FD of the free FD list is set. If the C bit is set, the function determines if one or more RBDs are attached. If there are RBDs attached, the end of the RBD list is found. The last RBD's link field is used to update begin_rbd pointer, and then it's set to NULL.

After the receive frame has been delineated from the RFA, some information about the frame is needed to determine which function to pass it to. Since the save bad frame configure bit is not set, the only bad frame on the list could be an out of resource frame. An out of resource frame is returned to the RFA by calling Put_Free_RFA (pfd). If the flags.lpbk_mode bit is set, the frame is given to the loopback check function. If the destination address of the frame indicates a multicast, the check multicast function is called. If the frame has passed all of the above tests and still has not been returned, it is passed to the Recv_Frame( ) function which resides in the ULCS.

Check_Multicast (pfd) determines whether the multicast address received is in the multicast address table. This is necessary because the 82586 does not have perfect multicast address filtering. Check_Multicast does a byte by byte comparison of the destination address with the addresses in the multicast address table. If no match occurs, it returns false, and Recv_Int_Processing calls Put_Free_RFA( ) to return the frame to the RFA. If there is a match, Check_Multicast( ) returns TRUE and Recv_Int_Processing( ) calls Recv_Frame( ), passing the pointer to the FD of the frame received.

3.5.2 RETURNING FDs AND RBDs-PUT_FREE_RFA (pfd)

Put_Free_RFA combines Supply_FD and Supply_RBD algorithms described in "Programming the 82586" into one function. The begin and end pointers delineate what the CPU believes is the beginning and end of the free list. The decision of whether to restart the RU is made when examining both the free FD list and the free RBD list. This is why two ru_start_flags are used, one for the FD list and one for the RBD list. Both flags are initialized to FALSE.

The function starts off by initializing the FD so that the EL bit is set, the status is 0, and the FD link field is NULL. The rbd pointer is saved before the rbd pointer field in the FD is set to NULL. The free FD list is examined and if it's empty, begin—fd and end—fd are loaded with the address of the FD being returned. In this case the RU should not be restarted, because there is only one FD on the free list. If the free FD list is not empty, the FD being returned is placed on the end of the list, the end pointer is updated, and the RU start flag is set TRUE.

To begin the RBD list processing the end of the returned RBD list is determined, and this last RBD's EL bit is set. If the free RBD list is empty, the returned RBD list becomes the free RBD list. If there is more than one RBD on the returned list, the ru start flag is set TRUE. If the free RBD list is not empty, the returned RBD list is appended on the end of the free list, the end—rbd pointer is updated, and the ru start flag is set TRUE.

The last part of Put_Free_RFA( ) is to determine whether to call RU_Start( ). Both ru start flags are ANDed together, and if the result is TRUE, the Ru_Start( ) function is called.

3.5.3 RESTARTING THE RECEIVE UNIT-RU_START()

The Ru_Start( ) function checks two things before it decides to restart the RU. The first thing it checks is whether the RU is already READY. If it is, there is no reason to restart it. If the RU is IDLE or in NO_RESOURCES, then the second thing to check is whether the first free FD on the free FD list has its C bit set. If it does, then the RU should not be restarted. The reason is that the free FD list should only contain free FDs

when the RU is started. If the C bit is set in the FD, then not all the used FD have been removed yet. If the RU is started when used FDs are still in the RFA, the 82586 will write over the used FDs and frames will be lost. Therefore Ru__Start( ) is exited if the first FD in the RFA has its C bit set. If the RU is not READY, and begin__fd doesn't point to a used FD, then the RU is restarted.

Note that in "Programming the 82586" there are two more conditions to be met before the RU is started: two or more FD on the RFA, and two or more RBD on the RFA. These conditions are checked in Put__Free__RFA( ), and Ru__Start( ) isn't called unless they are met.

4.0 LOGICAL LINK CONTROL

The IEEE 802.2 LLC function completes the Data Link Layer of the OSI model. The LLC module in this design example implements a class 1 level of service which provides a connectionless datagram interface. Several data link users or processes can run on top of the data link layer. Each user is identified by a link service access point (LSAP). Communication between data link users is via LSAPs. An LSAP is an address that identifies a specific user process or another layer (see Figure 13). The LSAP addresses are defined as follows:

Data Link Layer (Station Component)	00H
Transport Layer	FEH
Network Management Layer	08H
User Processes	multiples of 4 in the range OCH < LSAP ≤ FCH

Each receiving process is identified by a destination LSAP (DSAP) and each sending process is identified by a source LSAP (SSAP). Before a destination process can receive a packet, its DSAP must be included in a list of active DSAPs for the data link.

Figure 14 illustrates the relationship between the Station Component and the SAP components. (The SAP components are user processes.) The Station Component receives all of the good frames from the Handler and checks the DSAP address. If the DSAP address is 0, then the frame is addressed to the Station Component and a Station Component Response is generated. If the DSAP address is on the active DSAP list, then the Station Component passes the frame to the addressed SAP. If the DSAP address is unknown, the frame is returned to the handler.

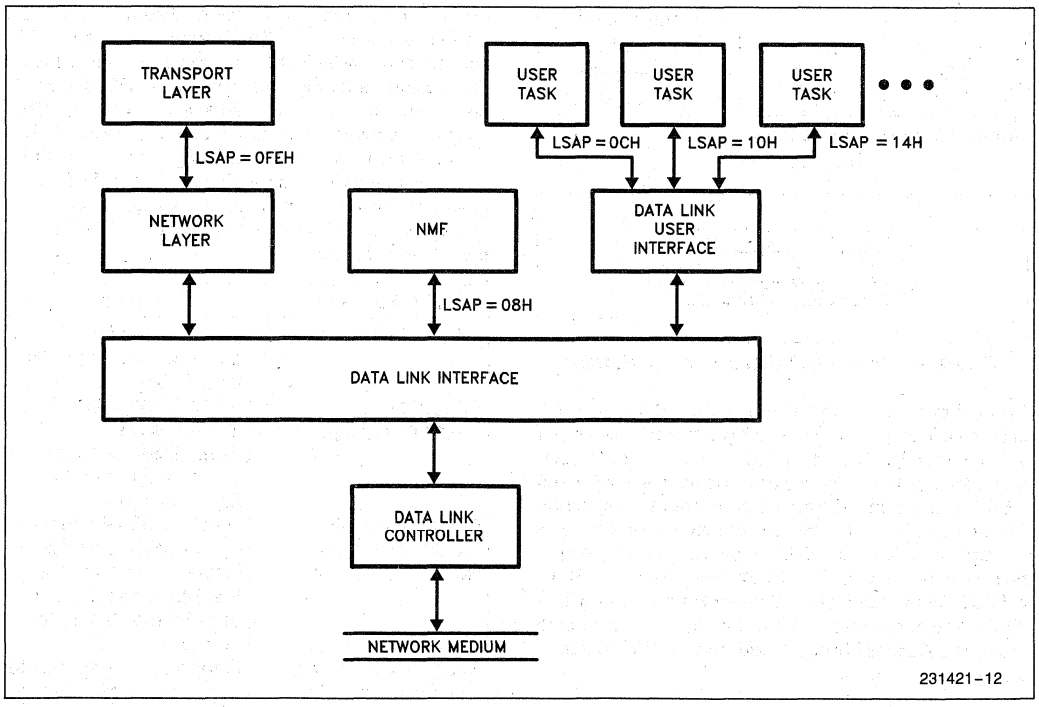


Figure 13. Data Link Interface

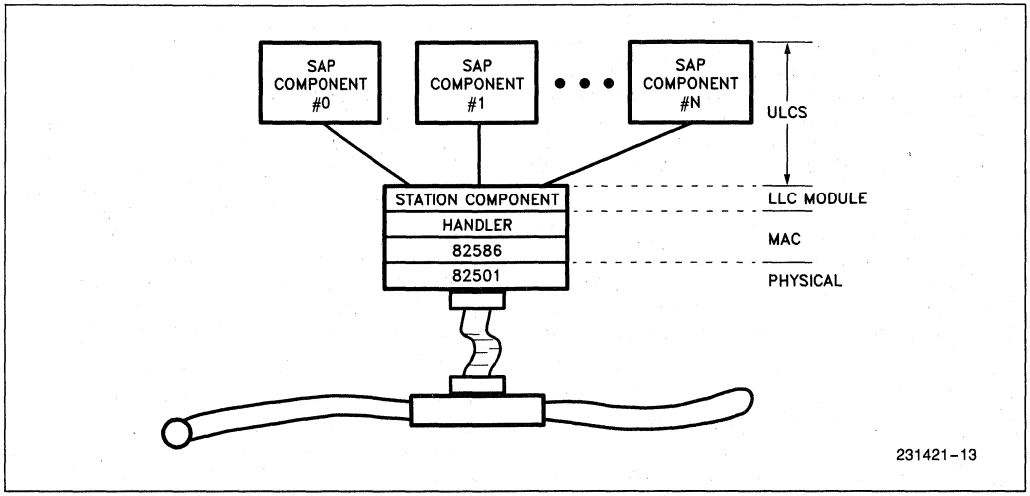


Figure 14. Station Component Relationship

There are 3 commands and 2 responses which the class 1 LLC layer must implement. Figure 15 shows IEEE 802.2 Class 1 commands and responses and Figure 16 shows the IEEE 802.2 Class 1 frame format.

Commands	Responses	Description
UI		Unnumbered Information
XID	XID	Exchange ID
TEST	TEST	Remote Loopback

Figure 15. IEEE 802.2 Class 1, Type 1 Commands and Responses

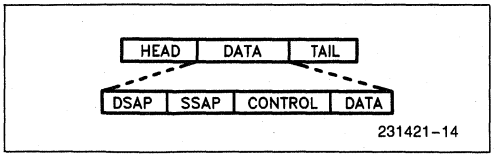


Figure 16. IEEE 802.2 Class 1 Frame Format

From Figure 15 it can be seen that there are no LLC class 1 UI responses because information frames are not acknowledged at the data link level. The only command frames that may require responses are XID and TEST. If a command frame is addressed to the Station Component, it checks the control field to see what type of frame it is. If it's an XID frame, the Station Component responds with a class 1 XID response frame. If it's a TEST frame, the Station Component responds with a TEST frame, echoing back the data it received. In both cases, the response frame is addressed to the source of the command frame.

Any frames addressed to active SAPs are passed directly to them. The Station Component will not respond to SAP addressed frames. Therefore it is the responsibility of the SAPs to recognize and respond to frames addressed to them. When a SAP transmits a frame, it builds the IEEE 802.2 frame itself and calls the Handler's Send_Frame( ) function directly. The LLC module is not used for SAP frame transmission. The only functions which the LLC module implement are the dynamic addition and deletion of DSAPs, multiplexing the frames to user SAPs, and the Station Component command recognition and responses. This is one implementation of the IEEE 802.2 standard. Other implementations may have the LLC module do more functions, such as SAP command recognitions and responses. A list of the functions included in the LLC module is as follows:

LLC Functions	Description
Init_Llc( )	Initializes the DSAP address table and calls Init_586( )
Add_Dsap_ Address (dsap, pfunc)	Add a DSAP address to the active list dsap - DSAP address pfunc - pointer to the SAP function
Delete—Dsap— Address (dsap)	Delete a DSAP address dsap - DSAP address
Recv—Frame (pfd)	Receives a frame from the 82586 Handler pfd - Frame Descriptor Pointer
Station—Component— Response (pfd)	Generates a response to a frame addressed to the Station Component pfd - Frame Descriptor Pointer

4.1 Adding and Deleting LSAPs

When a user process wants to add a LSAP to the active list, the process calls Add_Dsap_Address(dsap, pfunc). The dsap parameter is the actual DSAP address, and the pfunc parameter is the address of the function to be called when a frame with the associated DSAP address is received.

The LLC module maintains a table of active dsaps which consists of an array of structures. Each structure contains two members: stat - indicates whether the address is free or inuse, and (*p_sap_func)( ) contains the address of the function to call. The index into the array of structures is the DSAP address. This speeds up processing by eliminating a linear search. Delete_Dsap_Address (dsap) simply uses the DSAP index to mark the stat field FREE.

5.0 APPLICATION LAYER

For most networks the application layer resides on top of several other layers referred to here as ULCS. These other layers in the OSI model run from the network layer through the presentation layer. The implementation of the ULCS layers is beyond the scope of this application note, however Intel provides these layers as well as the data link layer with the OpenNET product line. For the purpose of this application note the application layer resides on top of the data link layer and its use is to demonstrate, exercise and test the data link layer design example.

There can be several processes sitting on top of the data link layer. Each process appears as a SAP to the data link. The UAP module, which implements the application layer, is the only SAP residing on top of the data link layer in this application example. Other SAPs could certainly be added such as additional "connectionless" terminals, a networking gateway, or a transport layer, however in the interest of time this was not done.

5.1 Application Layer Human Interface

The UAP provides a menu driven human interface via an async terminal connected to port B on the iSBC 186/51 board. The menu of the commands is listed in Figure 17 along with a description that follows:

Terminal Mode - implements a virtual terminal with datagram capability (connectionless "class 1" service). This mode can also be thought of as an async to IEEE 802.2/802.3 protocol converter.

Monitor Mode - allows the station to repeatedly transmit any size frame to the cable. While in the Monitor Mode, the terminal provides a dynamic update of 6 station related parameters.

High Speed Transmit Mode - sends frames to the cable as fast as the software possibly can. This mode demonstrates the throughput performance of the Data Link Driver.

Change Transmit Statistics - When Transmit Statistics is on several transmit statistics are gathered during transmission. If Transmit Statistics is off, statistics are not gathered and the program jumps over the section of code in the interrupt routine which gathers these statistics. The transmission rate is slightly increase when Transmit Statistics is off.

Print All Counters - Provides current information on the following counters.

Good frames transmitted:
Good frames received:

CRC errors received:
Alignment errors received:

Out of Resource frames:
Receiver overrun frames:

Each time a frame has been successfully transmitted the Good frames transmitted count is incremented. The same holds true for reception. CRC, Alignment, Out of Resources, and Overrun Errors are all obtained from the SCB. Underrun, lost CRS, SQE error, Max retry, and Frames that deferred are all transmit statistics that are obtained from the Transmit command status word. 82586 Reset is a count which is incremented each time the 82586 locks up. This count has never normally been incremented.

T - Terminal Mode	M - Monitor Mode
X - High Speed Transmit Mode	V - Change Transmit Statistics
P - Print All Counters	C - Clear All Counters
A - Add a Multicast Address	Z - Delete a Multicast Address
S - Change the SSAP Address	D - Change the DSAP Address
N - Change Destination Node Address	L - Print All Addresses
R - Re-Initialize the Data Link	B - Change the Number Base

Figure 17. Menu of Data Link Driver Commands

Clear All Counters - Resets all of the counters.

Add/Delete Multicast Address - Adds and Deletes Multicast Addresses.

Change SSAP Address - Deletes the previous SSAP and adds a new one to the active list. The SSAP in this case is this stations LSAP. When a frame is received, the DSAP address in the frame received is compared with any active LSAPs on the list. The SSAP is also used in the SSAP field of all transmitted frames.

Change DSAP Address - Delete the old DSAP and add a new one. The DSAP is the address of the LSAP which all transmit frames are sent to.

Change Destination Node Address - Address a new node.

Print All Addresses - Display on the terminal the station address, destination address, SSAP, DSAP, and all multicast addresses.

Re-initialize Data Link - This causes the Data Link to completely reinitialize itself. The 82586 is reset and reinitialized, and the selftest diagnostic and loopback tests are executed. The results of the diagnostics are printed on the terminal. The possible output messages from the 82586 selftest diagnostics are:

Passed Diagnostic Self Tests

Failed: Self Test Diagnose Command

Failed: Internal Loopback Self Test

Failed: External Loopback Self Test

Failed: External Loopback Through Transceiver Self Test

Change Base - Allows all numbers to be displayed in Hex or Decimal.

5.2 A Sample Session

The following text was taken directly from running the Data Link software on a 186/51 board. It begins with the iSDM monitor signing on and continues into executing the Data Link Driver software.

```
iSDM 86 Monitor, V1.0

Copyright 1983 Intel Corporation

.G D000:6

            ************************************************************
            *                                                          *
            * 82586 IEEE 802.2/802.3 Compatible Data Link Driver       *
            *                                                          *
            ************************************************************

Passed Diagnostic Self Tests

Enter the Address of the Destination Node in Hex -> 00AA0000179E

Enter this Station's LSAP in Hex -> 20

Enter the Destination Node's LSAP in Hex -> 20

Do you want to Load any Multicast Addresses? (Y or N) -> Y

Enter the Multicast Address in Hex -> 00AA00111111

Would you like to add another Multicast Address? (Y or N) -> N

This Station's Host Address is: 00AA00001868

The Address of the Destination Node is: 00AA0000179E

This Station's LSAP Address is: 20

The Address of the Destination LSAP is: 20

The following Multicast Addresses are enabled: 00AA00111111
```

```
Commands are:
T - Terminal Mode                          M - Monitor Mode
X - High Speed Transmit Mode               V - Change Transmit Statistics
P - Print All Counters                     C - Clear All Counters
A - Add a Multicast Address                Z - Delete a Multicast Address
S - Change the SSAP Address                D - Change the DSAP Address
N - Change Destination Node Address        L - Print All Addresses
R - Re-Initialize the Data Link            B - Change the number Base
Enter a command, type H for Help -> P
Good frames transmitted:     24      Good frames received:       1
CRC errors received:          0      Alignment errors received:  0
Out of Resource frames:       0      Receiver overrun frames:    0
82586 Reset:                  0      Transmit underrun frames:   0
Lost CRS:                     0      SQE errors:                 9
Maximum retry:                0      Frames that deferred:       4
Enter a command, type H for Help --> T
Would you like the local echo on? (Y or N) --> Y
This program will now enter the terminal mode.
Press ^C then CR to return back to the menu
Hello this is a test.
/*^C CR */
Enter a command, type H for Help --> M
Do you want this station to transmit? (Y or N) --> Y
Enter the number of data bytes in the frame --> 1500
Hit any key to exit Monitor Mode.
 # of Good      # of Good       CRC       Alignment      No        Receive
   Frames        Frames        Errors      Errors      Resource  Overrun
Transmitted     Received                               Errors    Errors
        32            0        00000       00000       00000     00000
/* CR */
Enter a command, type H for Help --> X
Hit any key to exit High Speed Transmit Mode.
/* CR */
Enter a command, type H for Help --> R
Passed Diagnostic Self Tests
```

5.3 Terminal Mode

The Terminal mode buffers characters received from the terminal and sends them in a frame to the cable. When a frame is received from the cable, data is extracted and sent to the terminal. One of three events initiate the UAP to send a frame providing there is data to send: buffering more than 1500 bytes, receiving a Carriage Return from the terminal, or receiving an interrupt from the virtual terminal timer.

The virtual terminal timer employs timer 1 in the 80130 to cause an interrupt every .125 seconds. Each time the interrupt occurs the software checks to see if it received one or more characters from the terminal. If it did, then it sends the characters in a frame.

The interface to the async terminal is a 256 byte software FIFO. Since the terminal communication is full duplex, there are two half duplex FIFOs: a Transmit FIFO and a Receive FIFO. Each FIFO uses two functions for I/O: Fifo_In( ) and Fifo_Out( ). A block diagram is displayed in Figure 18.

The serial I/O for the async terminal interface is always polled except in the Terminal mode where it is interrupt driven. The Terminal mode begins by enabling the 8274 receive interrupt but leaves the 8274 transmit interrupt disabled. This way any characters received from the terminal will cause an interrupt. These characters are then placed in the Transmit FIFO. The only time the 8274 transmit interrupt is enabled is when the Receive FIFO has data in it. The receive FIFO is filled from frames being received from the cable. Each time a transmit interrupt occurs a byte is removed from the Receive FIFO and written to the 8274. When the Receive FIFO empties, the 8274 transmit interrupt is disabled.

The flow control implemented for the terminal interface is via RTS and CTS. When the Transmit FIFO is full, RTS goes inactive preventing further reception of characters (see Table 1). If the Receive FIFO is full, receive frames are lost because there is no way for the data link using class 1 service to communicate to the remote station that the buffers are full. Lost receive frames are accounted for by the Out of Resources Frame counter.

The Async Terminal bit rate sets the throughput capability of the station in the terminal mode because the bottle neck for this network is the RS232 interface. Using this fact a simple test was conducted to verify the data link driver's capability of switching between the receiver's No Resource state and the Ready State. For example if station B is sending frames in the High Speed Transmit mode to station A which is in the Terminal mode, frames will be lost in station A. Under these circumstances station A's receiver will be switching from Ready state to Out of Resources state. The sum of Good frames received plus Out of Resource frames from station A should equal Good frames transmitted from station B; unless there were any underruns or overruns.

Table 1. FIFO State Table

Function	Present State	Next State	Action
FIFO_T_IN( )	EMPTY	IN USE	Start Filling Transmit Buffer
	IN USE	FULL	Shut Off RTS
FIFO_T_OUT( )	FULL	IN USE	Enable RTS
	IN USE	EMPTY	Stop Filling Transmit Buffer
FIFO_R_IN( )	EMPTY	IN USE	Turn on TxInt
	IN USE	FULL	Stop Filling FIFO from Receive Buffer
FIFO_R_OUT( )	FULL	IN USE	Start Filling FIFO from Receive Buffer
	IN USE	EMPTY	Turn Off TxInt

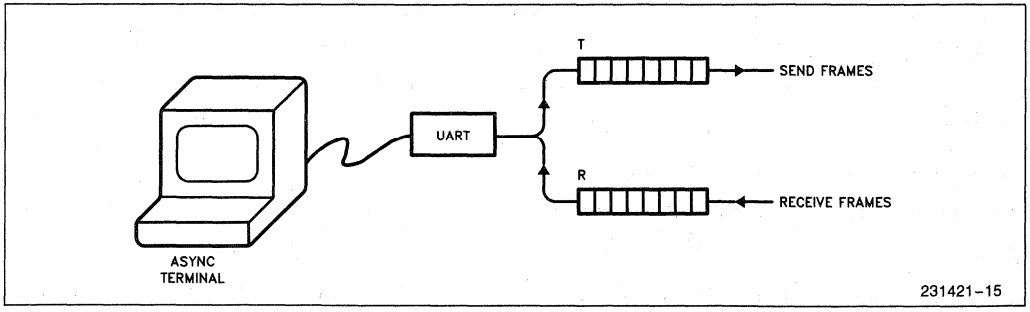


Figure 18

5.3.1 SENDING FRAMES

The Terminal Mode is entered when the Terminal_Mode( ) function is called from the Menu interface. The Terminal_Mode( ) function is one big loop, where each pass sends a frame. Receiving frames in the Terminal Mode is handled on an interrupt driven basis which will be discussed next.

The loop begins by getting a TBD from the 82586 handler. The first three bytes of the first buffer are loaded with the IEEE 802.2 header information. The loop then waits for the Transmit FIFO to become not EMPTY, at which point a byte is removed from the Transmit FIFO and placed in the TBD. After each byte is removed from the Transmit FIFO several conditions are tested to determine whether the frame needs to be transmitted, or whether a new buffer must be obtained. A frame needs to be transmitted if: a Carriage Return is received, the maximum frame length is reached, or the send_frame flag is set by the virtual terminal timer. A new buffer must be obtained if none of the above is true and the max buffer size is reached.

If a frame needs to be sent the last TBD's EOP bit is set and its buffer count is updated. The 82586 Handler's Send_Frame( ) function is called to transmit the frame, and continues to be called until the function returns TRUE.

The loop is repeated until a ^C followed by a Carriage Return is recieved.

5.3.2 RECEIVING FRAMES

Upon initialization the UAP module calls the Add_Dsap_Address(dsap, pfunc) function in the LLC module. This function adds the UAP's LSAP to the active list. The pfunc parameter is the address of the function to call when a frame has been received with the UAP's LSAP address. This function is Recv_Data_1( ). Recv_Data_1( ) looks at the control field of the frame received and determines the action required.

The commands and responses handled by Recv_Data_1( ) are the same as the Station Component's commands and responses given in Figure 15. One difference is that Recv_Data_1( ) will process a UI command while the Station Component will ignore a UI command addressed to it.

Recv_Data_1( ) will discard any UI frames received unless it is in the Terminal Mode. When in the Terminal Mode, Recv_Data_1( ) skips over the IEEE 802.2 header information and uses the length field to determine the number of bytes to place in the Receive FIFO. Before a byte is placed in the FIFO, the FIFO status is checked to make sure it is not full. Recv_Data_1( ) will move all of the data from the frame into the Receive FIFO before returning.

When a frame is received by the 82586 handler an interrupt is generated. While in the 82586 interrupt routine the receive frame is passed to the LLC layer and then to the UAP layer where the data is placed in the Receive FIFO by Recv_Octal_Data_1( ). Since Recv_Data_1( ) will not return until all of the data from the frame has been moved into the Receive FIFO, the 8274 transmit interrupt must be nested at a higher priority than the 82586 interrupt to prevent a software lock. For example if a frame is received which has more than 256 bytes of data, the Receive FIFO will fill up. The only way it can empty is if the 8274 interrupt can nest the 82586 interrupt service routine. If the 8274 could not interrupt the 82586 ISR then the software would be stuck in Recv_Data_1( ) waiting for the FIFO to empty.

5.4 Monitor Mode

The Monitor Mode dynamically updates 6 station related parameters on the terminal as shown below.

The Monitor_Mode( ) function consists of one loop. During each pass through the loop the counters are updated, and a frame is sent. Any size frame can be transmitted up to a size of the maximum number of transmit buffers available. Frame sizes less than the minimum frame length are automatically padded by the 82586 Handler.

The data in the frames transmitted in the Monitor Mode are loaded with all the printable ASCII characters. This way when one station is in the Monitor Mode transmitting to another station in the Terminal Mode, the Terminal Mode station will display a marching pattern of ASCII characters.

# of Good Frames Transmitted	# of Good Frames Received	CRC Errors	Alignment Errors	No Resource Errors	Receive Overrun Errors
32	0	00000	00000	00000	00000

5.5 High Speed Transmit Mode

The High Speed Transmit Mode demonstrates the throughput performance of the 82586 Handler. The Hs_Xmit_Mode( ) function operates in a tight loop which gets a TBD, sets the EOF bit, and calls Send_Frame( ). The flow chart for this loop is shown in Figure 19.

The loop is exited when a character is received from the terminal. Rather than polling the 8274 for a receive buffer full status, the 8274's receive interrupt is used. When the Hs_Xmit_Mode( ) function is entered, the hs_stat flag is set true. If the 8274 receive interrupt occurs, the hs_stat flag is set false. This way the loop only has to test the hs_stat flag rather than calling inb( ) function each pass through the loop to determine whether a character has been received.

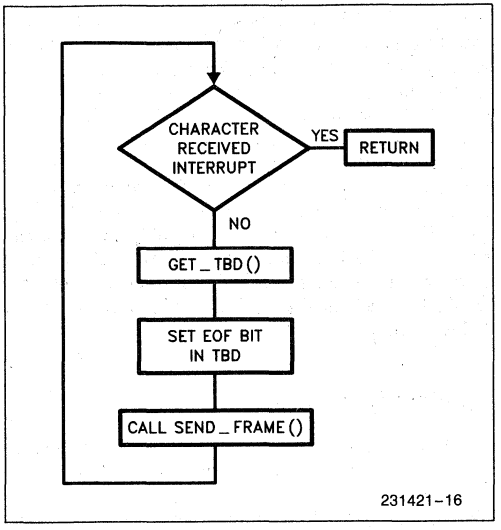


Figure 19. High Speed Transmit Mode Flow Chart

The performance measured on an 8 MHz 186/51 board is 593 frames per second. The bottle neck in the throughput is the software and not the 82586. The size of the buffer is not relevant to the transmit frame rate. Whether the buffer size is 128 bytes or 1500 bytes, linked or not, the frame rate is still the same. Therefore assuming a 1500 byte buffer at 593 frames per second, the effective data rate is 889,500 bytes per second.

This can easily be demonstrated by using two 186/51 boards running the Data Link software. The receiving stations counters should be cleared then placed in the Monitor mode. When placing it in the monitor mode, transmission should not be enabled. When the other station is placed in the High Speed Transmit Mode a timer should be started. One can use a stop watch to determine the time interval for transmission. The frame rate is determined by dividing the number of frames received in the Monitor station by the time interval of transmission.

APPENDIX A
COMPILING, LINKING, LOCATING, AND RUNNING THE SOFTWARE ON THE 186/51 BOARD

********* Instructions for using the 186/51 board *********

Use 27128A for no wait state operation. 27128s can be used but wait states will have to be added.

Copy HI.BYT and LO.BYT files into EPROMs
PROMs go into U34 - HI.BYT and U39 - LO.BYT on the 186/51 board

JUMPERS REQUIRED

Jumper the 186/51 board for 16K byte PROMs in U34 and U39 Table 2-5 in 186/51 HARDWARE REFERENCE MANUAL (Rev-001)

186/51(ES)	186/51 (S)/186/51
E151-E152 OUT	E199-E203 OUT
E152-E150 IN	E203-E191 IN
E94-E95 IN	E120-E119 IN
E100-E106 IN	E116-E112 IN
E107-E113 IN	E111-E107 IN
E133-E134 IN	E94-E93 IN

also change interrupt priority jumpers - switch 8274 and 82586 interrupt priorities

E36-E44 OUT	E43-E47 OUT
E39-E47 OUT	E46-ES0 OUT
E37-E45 OUT	E44-E48 OUT

WIRE WRAP

E36-E47 IN	E43-E50 IN
E39-E44 IN	E46-E47 IN
E79-E45 IN	E90-E48 IN

USE SDM MONITOR

The SDM Monitor should have the 82586's SCP burned into ROM. The ISCP is located at OFFFOH. Therefore for the SCP the value in the SDM ROM should be:

ADDRESS	DATA
FFFF6H	XXOOH
FFFF8H	XXXXH
FFFFAH	XXXXH
FFFFCH	FFFOH
FFFFEH	XXOOH

To run the program begin execution at 0D000:6H

```
I.E. G D000:6

GOOD LUCK!

**********    submit file for compiling one module:    **********

run

cc86.86  :F6:%0 LARGE ROM DEBUG DEFINE(DEBUG) include(:F6:)

exit

**********    submit file for linking and locating:    **********

run

link86     :F6:assy.obj, :F6:dld.obj, :F6:llc.obj, &

:F6:uap.obj, lclib.lib to :F6:dld.lnk segsize(stack(4000h)) notype

loc86 :F6:dld.lnk to :F6:dld.loc&

initcode (0D0000H) start(begin) order(classes(data, stack, code)) &
addresses(classes(data(3000H), stack(0CB00H), code(0D0020H)))

oh86 :F6:dld.loc to :F6:dld.rom

exit

**********    submit file for burning EPROMs using IPPS:    **********

ipps

i 86

f :F6:dld.rom (0d0000h)

3

2

1

0 to :F6:lo.byt

y

1 to :F6:hi.byt

y


t 27128

9

c :F6:lo.byt t p

n

C :F6:hi.byt t p

n

exit
```

```
/PCO/USR/CHUCK/CSRC/DLD.H

/*************************************************************************
*                                                                        *
*                        82586 Structures and Constants                  *
*                                                                        *
*************************************************************************/

/* general purpose constants */

#define INUSE    0
#define EMPTY    1
#define FULL     2
#define FREE     1
#define TRUE     1
#define FALSE    0
#define NULL     0xFFFF

/* Define Data Structures */

#define RBUF_SIZE        128 /* receive buffer size */
#define TBUF_SIZE        128 /* transmit buffer size */
#define ADD_LEN          6
#define MULTI_ADDR_CNT   16

typedef unsigned short int u_short;

/* results from Test_Link(): loaded into Self_Test char */

#define PASSED                    0
#define FAILED_DIAGNOSE           1
#define FAILED_LPBK_INTERNAL      2
#define FAILED_LPBK_EXTERNAL      3
#define FAILED_LPBK_TRANSCEIVER 4

/* Frame Commands */
#define     UI       0x03     /* Unnumbered Information Frame */
#define     XID      0xAF     /* Exchange Identification */
#define     TEST     0xE3     /* Remote Loopback Test */
#define     P_F_BIT 0x10      /* Poll/Final Bit Position */
#define     C_R_BIT 0x01      /* Command/Response bit in SSAP */


#define     DSAP_CNT    8     /* Number of allowable DSAPs; must be a multiple
                                 of 2**N, and DSAP addresses assigned must be
                                 divisible by 2**(8-N).
                                 (i.e. the N LSBs must be 0) */

#define     DSAP_SHIFT  5     /* DSAP_SHIFTS must equal 8-N */


#define     XID_LENGTH  6     /* Number of Info bytes for XID Response frame */


/* System Configuration Pointer SCP */

   struct SCP  {
               u_short sysbus; /* 82586 bus width, 0 - 16 bits
                                  1 - 8 bits */
```

231421-17

```
/PCO/USR/CHUCK/CSRC/DLD.H

                u_short junk[2];
                u_short iscpl;  /* lower 16 bits of iscp address */
                u_short iscph;  /* upper 8 bits of iscp address */
               };

/* Intermediate System Configuration Pointer ISCP */

   struct ISCP {
                u_short busy ;  /*set to 1 by cpu before its first CA,
                                 cleared by 82586 after reading */
                u_short offset  ;   /* offset of system control block */
                u_short base1 ; /* base of system control block */
                u_short base2 ;
               } ;

/* System Control Block SCB */

   struct SCB  {
                u_short stat;        /* Status word */
                u_short cmd;         /* Command word */
                u_short cbl_offset; /* Offset of first command block in CBL */
                u_short rfa_offset; /* Offset of first frame descriptor in RFA */
                u_short crc_errs;    /* CRC errors accumulated */
                u_short aln_errs;    /* Alignment errors */
                u_short rsc_errs;    /* Frames lost because of no Resources */
                u_short ovr_errs;    /* Overrun errors */
               };

/*  Command Block */

   struct  CB  {
                u_short stat;    /* Status of Command */
                u_short cmd;     /* Command */
                u_short link;    /* link field */
                u_short parm1;   /* Parameters */
                u_short parm2;
                u_short parm3;
                u_short parm4;
                u_short parm5;
                u_short parm6;
          };


/*  Multicast Address Command Block MA_CB */

   struct MA_CB{
                u_short stat;    /* Status of Command */
                u_short cmd;     /* Command */
                u_short link;    /* Link field */
                u_short mc_cnt; /* Number of MC addresses */
                char mc_addr[ADD_LEN*MULTI_ADDR_CNT]; /* MC address area */
               };


/* Transmit Buffer Descriptor TBD */

   struct TBD  {
```

231421-18

```
/PCO/USR/CHUCK/CSRC/DLD.H

                u_short act_cnt;     /* Number of bytes in buffer */
                u_short link;        /* offset to next TBD */
                u_short buff_l;      /* lower 16 bits of buffer address */
                u_short buff_h;      /* upper 8 bits of buffer address */
                struct TB *buff_ptr;     /* not used by the 586: used by the
                                         software to save address translation
                                         routine.  */
               };

/* Transmit Buffers */
    struct TB   {
                 char data [TBUF_SIZE];
                };

/* Frame Descriptor FD */

   struct  FD  {
                u_short stat;                /* Status Word of FD */
                u_short el_s;                /* EL and S bits */
                u_short link;                /* link to next FD */
                u_short rbd_offset;          /* Receive buffer descriptor offset */
                char dest_addr[ADD_LEN];/*Destination address */
                char src_addr[ADD_LEN]; /* Source address */
                u_short length;              /* Length field */
               };

/* Receive Buffer Descriptor RBD */

   struct RBD {
                u_short act_cnt;         /* Actual number of bytes received */
                u_short link;            /* Offset to next RBD */
                u_short buff_l;          /* Lower 16 bits of buffer address */
                u_short buff_h;          /* upper 8 bits of buffer address */
                u_short size;            /* size of buffer */
                struct RB *buff_ptr;     /* not used by the 586: used by the
                                         software to save address translation
                                         routine. */
               };


/* Receive Buffers */

   struct RB   {
                char data[RBUF_SIZE];
               };


struct  FRAME_STRUCT
        {
            unsigned char   dsap;    /* Destination Service Access Point */
            unsigned char   ssap;    /* Source Service Access Point */
            unsigned char   cmd;     /* ISO Data Link Command */
        };

    /* LSAP Address Table */
struct  LAT {
                char    stat;            /* INUSE or FREE */
```

231421-19

```
/PCO/USR/CHUCK/CSRC/DLD.H

                int     (*p_sap_func)();/* Pointer to LSAP function; associated
                                           with dsap address */
            };


struct MAT {                        /* Multicast Address Table */
        char    stat;               /* INUSE or FREE */
        char    addr[ADD_LEN];      /* actual mc address */
        };

/* general purpose flags */

struct FLAGS {
    unsigned diag_done : 1 ;        /* diagnose command complete */
    unsigned stat_on : 1 ;          /* network diagnostic statistics on/off */
    unsigned reset_sema: 1 ;        /* don't reset when this bit is set */
    unsigned reset_pend: 1 ;        /* reset when this bit is set */
    unsigned lpbk_test: 1 ;         /* loopback test flag */
    unsigned lpbk_mode: 1 ;         /* loopback mode on/off */
       } ;


/* General purpose bits */

#define ELBIT   0x8000
#define EOFBIT  0x8000
#define SBIT    0x4000
#define IBIT    0x2000
#define CBIT    0x8000
#define BBIT    0x4000
#define OKBIT   0x2000

/* SCB patterns */

#define CX          0x8000
#define FR          0x4000
#define CNA         0x2000
#define RNR         0x1000
#define RESET       0x0080
#define CU_START    0x0100
#define RU_START    0x0010
#define RU_ABORT    0x0040
#define CU_MASK     0x0700
#define RU_MASK     0x0070
#define RU_READY    0x0040

/* 82586 Commands */

#define NOP         0x0000
#define IA          0x0001
#define CONFIGURE   0x0002
#define MC_SETUP    0x0003
#define TRANSMIT    0x0004
#define TDR         0x0005
#define DUMP        0x0006
#define DIAGNOSE    0x0007
```

231421-20

```
/PCO/USR/CHUCK/CSRC/DLD.H

/* 82586 Command and Status Masks */

#define CMD_MASK          0x0007
#define NOERRBIT          0x2000
#define COLLMASK          0x000F
#define DEFERMASK         0x0080
#define NOCRSMASK         0x0400
#define UNDERRUNMASK      0x0100
#define SQEMASK           0x0040
#define MAXCOLMASK        0x0020
#define OUT_OF_RESOURCES 0x0200

/* Configure Parameters */

#define FIFO_LIM     0x0800       /* use FIFO lim of 8 */
#define BYTE_CNT     0x000B
#define SRDY         0x0040
#define SAV_BF       0x0080
#define ADDR_LEN     0x0600       /* address length of 6 bytes */
#define AC_LOC       0x0800
#define PREAM_LEN    0x2000       /* preamble length of 8 bytes */
#define INT_LPBCK    0x4000
#define EXT_LPBCK    0x8000
#define LIN_PRIO     0x0000       /* no priority */
#define ACR          0x0000
#define BOF_MET      0x0080
#define IFS          0x6000       /* IFS time 9.6 usec */
#define SLOT_TIME    0x0200       /* slot time 51.2 usec */
#define RETRY_NUM    0xF000       /* retry number 15 */
#define PRM          0x0001
#define BC_DIS       0x0002
#define MANCHESTER   0x0004
#define TONO_CRS     0x0008
#define NCRC_INS     0x0010
#define CRC_16       0x0020
#define BT_STUFF     0x0040
#define PAD          0x0080
#define CRSF         0x0000       /* no carrier sense filter */
#define CRS_SRC      0x0800
#define CDTF         0x0000       /* no collision detect filter */
#define CDT_SRC      0x8000
#define MIN_FRM_LEN 0x0040       /* 64 bytes */
#define MIN_DATA_LEN MIN_FRM_LEN - 18   /* assumes Ethernet/IEEE 802.3
                                           frames with 6 bytes of address */
#define MAX_FRAME_SIZE  1500 - 3
```

231421-21

```
/PCO/USR/CHUCK/CSRC/DLD.C

/*************************************************************************
*                                                                        *
*                          82586 Handler                                 *
*                                                                        *
*************************************************************************/


/* Define constants for storage area */

#define CB_CNT       8   /* Number of available Command Blocks */
#define FD_CNT       16  /* Number of available Frame Descriptors */
#define RBD_CNT      64  /* Number of available Receive Buffer descriptors */
#define TBD_CNT      16  /* Number of available Transmit Buffer descriptors */

/* loopback parameters passed to Configure() */

#define INTERNAL_LOOPBACK    0x4000
#define EXTERNAL_LOOPBACK    0x8000
#define NO_LOOPBACK          0x0000

#include "dld.h"             /* 586 Data Structures */


/* 186 Timer Addresses */

#define TIMER1_CTL   0xFF5E
#define TIMER1_CNT   0xFF58
#define TIMER2_CTL   0xFF66
#define TIMER2_CNT   0xFF60


/* external functions */

/* I/O */
int      inw();       /* input word : inw(address) */
void     outw();      /* output word: outw(address, value) */
void     init_intv(); /* initialize the interrupt vector table */
void     enable();    /* enable 80186 interrupts */
void     disable();   /* disable 80186 interrupts */

extern   char     *Build_Ptr();

u_short      SEGMT;       /* Data segment value */
char         *pNULL;      /* NULL pointer */

/* Macro 'type' of definitions */

#define CA  outw(0xC8,0)     /* the command to issue a Channel Attention */

#define ESI_LOOPBACK outw(0xCB,0) /* put the ESI in Loopback */
#define NO_ESI_LOOPBACK outw(0xCB,8) /* take the ESI out of Loopback */

#define EOI_80130          outb(0xE0,0x63)    /* End Of Interrupt */
#define TIMER1_EOI_80186 outw(0xFF22,0x04) /* EOI for Timer 1 on the 186 */
#define TIMER1_EOI_80130 outb(0xE0,0x64) /*EOI for 186's Timer1 on the 130 */
```

231421-22

```
/PCO/USR/CHUCK/CSRC/DLD.C


/******** memory  allocation ****************/

int Self_Test;       /* used for diagnostic purposes */
u_short temp;        /* temporary storage */

#define LPBK_FRAME_SIZE     4                /* loopback frame storage */
char    lpbk_frame[LPBK_FRAME_SIZE] = {
0x55, 0xAA, 0x55, 0xAA};

#define whoami_io_add   0x00F0  /* I/O address of Host Address Prom */
char    whoami[ADD_LEN];        /* Ram array where host address is stored */


/* transmission statistic variables */

unsigned long   good_xmit_cnt;
u_short         underrun_cnt;
u_short         no_crs_cnt;
unsigned long   defer_cnt;
u_short         sqe_err_cnt;
u_short         max_col_cnt;
unsigned long   recv_frame_cnt;
u_short         reset_cnt;

    /* Allocate storage for structures and buffers */


struct FLAGS flags;


/* 586 structures */

/* System Configuration Pointer: Rom Initialization */
/* struct SCP scp = {0x0000,0x0000,0x0000,0x1FF6,0x0000};  */

/* struct ISCP iscp;     Intermediate System Configuration Pointer */

struct SCB scb;      /* System Control Block */

struct CB cb[CB_CNT],    /* Command Blocks */
*cb_tos, *begin_cbl, *end_cbl;
                         /* pointer to the beginning of the free
                            command block list (cb_tos) and the
                            beginning and end of the 82586 cbl */

struct TBD tbd[TBD_CNT],    /* Transmit Buffer Descriptor */
*tbd_tos;                /* pointer to the free Transmit buffer
                            descriptors */

struct TB tbuf[TBD_CNT];     /* Transmit Buffers */


struct FD fd[FD_CNT],        /* Frame Descriptors */
*begin_fd, *end_fd;       /* pointers to the beginning and end of
                             the free FD list */

struct RBD rbd[RBD_CNT],     /* Receive Buffer Descriptors */
```

231421-23

```
/PCO/USR/CHUCK/CSRC/DLD.C


*begin_rbd, *end_rbd;          /* pointers to the beginning and the
                                  end of the rbd list */

struct RB rbuf[RBD_CNT];     /* Receive Buffers */

struct MAT mat[MULTI_ADDR_CNT];     /* Multicast Address Table */
struct MA_CB ma_cb;                 /* Multicast Address Command Block */


/* The following structures are used only in Reset_586() function */
struct      CB      res_cb;     /* Temporary CB for reinitializing the 586 */
struct      MA_CB   res_ma_cb;  /* Temporary MA_CB for reloading Multicast */

/*  Hardware Support Functions */

Enable_586_Int()
{
    int     c;

    c = inb(0xE2);           /* read the 80130 interrupt mask register */
    outb(0xE2, 0x00F7 & c); /* write to the 80130 interrupt mask register */
}

Disable_586_Int()
{
    int     c;

    c = inb(0xE2);
    outb(0xE2, 0x0008 | c);
}

Set_Timeout()
{
    outw(TIMER1_CNT, 0);    /* Write a 0 to Timer1 count register */
    outw(0xFF5E, 0xE009);   /* Set ENable bit in Timer1 Mode/Control register */
}

Reset_Timeout()
{
    outw(0xFF5E, 0x6009); /* Reset ENable bit in Timer1 Mode/Control register */
}

Init_Timer() /* 186's Timer 2 is a prescaler for Timer 1.  It clocks Timer 1
                every 32.7 msec.  The deadman timeout is set for 1.25 sec */
{
    outw(0xFF38, 0x000C);    /* Set Timer1 Interrupt Control register */
    outw(0xFF62, 0xFFFF);    /* set max count register for timer2 to OFFFFH */
    outw(0xFF5A,     38);    /* set max count register A for timer 1 */
    outw(0xFF66, 0xC001);    /* Set Timer2 Mode/Control register */
    outw(0xFF5E, 0x6009);    /* Set Timer1 Mode/Control register */
    outw(0xFF28, (inw(0xFF28) & 0xFFEF)); /* Enable 186 Timer1 interrupt */
    outb(0xE2,(inb(0xE2) & 0x00EF));  /* enable 80130 interrupt from 80186 */
}

/* end hardware support functions */

Clear_Cnt()
```

231421-24

```
/PCO/USR/CHUCK/CSRC/DLD.C

{
    scb.crc_errs = 0;        /* clear 586 error statistic counters */
    scb.aln_errs = 0;
    scb.rsc_errs = 0;
    scb.ovr_errs = 0;

    good_xmit_cnt = 0;       /* init data link statistics */
    underrun_cnt = 0;
    no_crs_cnt = 0;
    defer_cnt = 0;
    sqe_err_cnt = 0;
    max_col_cnt = 0;
    recv_frame_cnt = 0;
    reset_cnt = 0;
}

Init_586()
{
    struct ISCP *piscp;
    u_short i;
    struct  MAT *pmat;

    NO_ESI_LOOPBACK;  /* Done for 82501. Inactivates CRS if powered up
                         in loopback */
    ESI_LOOPBACK;

    init_intv();     /* Initialization DLDs interrupt vectors */

    Init_Timer();

    flags.reset_sema = 0;    /* Initialize Reset Flags */
    flags.reset_pend = 0;
    flags.stat_on = 1;

    Disable_586_Int();



    piscp =  0x0000FFF0 ; /* Initialize the ISCP pointer*/
    piscp->busy = 1;
    piscp->offset =  Offset(&scb);
    piscp->base1 = SEGMT << 4;
    piscp->base2 = (SEGMT >> 12) & 0x000F ;

    pNULL = Build_Ptr(NULL);     /* build a NULL pointer - 8086 type: 32 bits */
    Build_Rfa();     /* init Receive Frame Area */
    Build_Cb(); /* init Command Block list */
    ma_cb.cmd = 0;       /* multicast address semaphore init */

    Clear_Cnt();

    scb.stat = 0;

    CA;      /* wait for the 586 to complete initialization */


    for ( i = 0; i <= 0xFF00; i++)
```

231421-25

```
/PCO/USR/CHUCK/CSRC/DLD.C

        if (scb.stat == (CX | CNA))
            break;

    if (i >0xFF00)
    Fatal("DLD:init - Did not get an interrupt after Reset/CA\n");

    /* Ack the reset Interrupt */
    scb.cmd = (CX | CNA);
    CA;
    Wait_Scb();
    Enable_586_Int();

    scb.cbl_offset = Offset(&cb[0]);     /* link scb to cb and fd lists */
    scb.rfa_offset = Offset(&fd[0]);

     /* move the prom bytes into whoami array */

    for (i = 0; i < ADD_LEN; i++)
        whoami[(ADD_LEN - 1) - i] = inb(whoami_io_add + i*2);

        /* Initialization the Multicast Address Table */

    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        pmat->stat = FREE;

    Configure(INTERNAL_LOOPBACK);    /* Put 586 in internal loopback */

    SetAddress();    /* Set up the station address */

    /* run diagnostics */

    Test_Link();

    if (Self_Test != PASSED)
        return(Self_Test);

    Configure(NO_LOOPBACK); /* Configure the 82586 */

    return(Self_Test);
}

Build_Rfa()
{
    struct   FD      *pfd;
    struct   RBD     *prbd;
    struct   RB      *pbuf;
    unsigned long    badd;

    /* Build a linear linked frame descriptor list */

    for (pfd = &fd[0]; pfd <= &fd[FD_CNT - 1]; pfd++) {

        pfd->stat = pfd->el_s = 0;
        pfd->link = Offset(pfd+1);
        pfd->rbd_offset = NULL;
    }
```

231421-26

/PCO/USR/CHUCK/CSRC/DLD.C

```
    end_fd = --pfd;                 /* point to &fd[FD_CNT - 1] */
    pfd->link = NULL;               /* last fd link is NULL */
    pfd->el_s = ELBIT;              /* last fd has EL bit set */
    begin_fd = pfd = &fd[0];        /* point to first fd */
    pfd->rbd_offset = Offset(&rbd[0]);  /* link first fd to first rbd */


    /* Build a linear linked receive buffer descriptor list */

    for (prbd = &rbd[0], pbuf = &rbuf[0]; prbd <= &rbd[RBD_CNT - 1];
                                                    prbd++, pbuf++) {
        badd = SEGMT << 4;
        badd += Offset(pbuf);
        prbd->buff_l = badd;
        prbd->buff_h = badd >> 16;
        prbd->buff_ptr = pbuf;

        prbd->act_cnt =  0;
        prbd->link = Offset(prbd + 1);
        prbd->size = RBUF_SIZE;
    }

    end_rbd = --prbd;
    prbd->link = NULL;      /* last rbd points to NULL */
    prbd->size |= ELBIT;    /* last rbd has el bit set */

    begin_rbd = &rbd[0];

}

Build_Cb()  /* Build a stack of free command blocks */
{
    struct  CB  *pcb;
    struct  TBD *ptbd;
    struct  TB  *pbuf;
    unsigned    long    badd;

    for (pcb = &cb[0]; pcb <= &cb[CB_CNT - 1]; pcb++) {
        pcb->stat = 0;
        pcb->cmd = ELBIT;
        pcb->link = Offset(pcb + 1);
    }
    --pcb;
    begin_cbl = end_cbl = pNULL;
    pcb->link = NULL;
    cb_tos = &cb[0];

    /* Build a stack of transmit buffer descriptors */

    for (ptbd = &tbd[0], pbuf = &tbuf[0]; ptbd <= &tbd[TBD_CNT - 1];
                                                    ptbd++, pbuf++) {

        ptbd->act_cnt  = TBUF_SIZE;
        ptbd->link = Offset(ptbd + 1);

        badd = SEGMT << 4;
```

231421-27

/PCO/USR/CHUCK/CSRC/DLD.C

```
        badd += Offset(pbuf);
        ptbd->buff_l = badd;
        ptbd->buff_h = badd >> 16;
        ptbd->buff_ptr = pbuf;
    }

    --ptbd;
    ptbd->link = NULL;   /* last tbd link is NULL */
    tbd_tos = &tbd[0];   /* Set the Top Of the Stack */

}

    /* Get a Command Block from the free list */

struct  CB    *Get_Cb() /* return a pointer to a free command block */
{
    struct  CB *pcb;

    if (Offset(pcb = cb_tos) == NULL)
        return(pNULL);
    cb_tos = (struct CB *) Build_Ptr(pcb->link);
    pcb->link = NULL;
    return(pcb);
}

/* Put a Command Block back onto the free list */

Put_Cb(pcb)

    struct  CB *pcb;

{
    pcb->stat = 0;
    pcb->link = Offset(cb_tos);
    cb_tos = pcb;
}

struct TBD  *Get_Tbd()  /* return a pointer to a free transmit buffer
                           descriptor */
{
    struct   TBD    *ptbd;

    flags.reset_sema = 1;
    Disable_586_Int();
    if ((ptbd = tbd_tos) != pNULL) {
        tbd_tos = (struct TBD *) Build_Ptr(ptbd->link);
        ptbd->link = NULL;
    }
    Enable_586_Int();
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(ptbd);
}

Put_Tbd(ptbd)
```

231421-28

/PCO/USR/CHUCK/CSRC/DLD.C

```
struct      TBD     *ptbd;
{
    struct      TBD     *p ;

    /* find the end of the tbd list returned, ptbd is the beginning */

    for (p = ptbd; p->link != NULL; p = (struct TBD *) Build_Ptr(p->link)) ;

    p->act_cnt = TBUF_SIZE;      /* clear EOFBIT and update size on last tbd */
    p->link = Offset(tbd_tos);
    tbd_tos = ptbd;
}


SetAddress()
{
    struct  CB  *pcb;

#ifdef DEBUG

    if ((pcb = Get_Cb()) == pNULL)
    Fatal("dld.c - SetAddress - couldn't get a CB\n");

#else

    pcb = Get_Cb();

#endif  /* DEBUG */

    bcopy((char *)&pcb->parm1, &whoami[0], ADD_LEN); /* move the prom
                                                        address to IA cmd */
    pcb->cmd = IA | ELBIT;

    Issue_CU_Cmd(pcb);
}


Wait_Scb()  /* wait for the scb command word to be clear */
{
    u_short     i, stat;

    for (stat = FALSE; stat == FALSE; ) {

        for (i=0; i<=0xFF00; i++)
            if (scb.cmd == 0)
                break;

        if (i > 0xFF00) {
            Bug("DLD: Scb command not clear\n");
            CA;
        }
        else
            stat = TRUE;
    }
```

231421-29

```
/PCO/USR/CHUCK/CSRC/DLD.C

}


Issue_CU_Cmd(pcb) /* Queue up a command and issue a start CU command if no
                     other commands are queued */
    struct  CB  *pcb;
{
    Disable_586_Int();
    if (begin_cbl == pNULL) {   /* if the list is inactive start CU */
        begin_cbl = end_cbl = pcb;
        scb.cbl_offset = Offset(pcb);
        Wait_Scb();
        scb.cmd = CU_START;
        Set_Timeout();              /* set deadman timer for CU */
        CA;
    }
    else {
        end_cbl->link = Offset(pcb);
        end_cbl = pcb;
    }
    Enable_586_Int();
}

Isr7()
{
    outb(0xE0, 0x67);    /* EOI 80130 */
}

Isr6()
{
    Write("\nInterrupt 6\n");
    outb(0xE0, 0x66);    /* EOI 80130 */
}

Isr5()
{
    Write("\nInterrupt 5\n");
    outb(0xE0, 0x65);    /* EOI 80130 */
}

    /* Deadman Timer Interrupt Service Routine */

Isr_Timeout()   /* Interrupt 4 */
{
    Reset_Timeout();
    if (flags.reset_sema == 1)
        flags.reset_pend = 1;
    else
        Reset_586();

    TIMER1_EOI_80186;
    TIMER1_EOI_80130;
}

    /* Interrupt 0 is Uart in UAP Module */
    /* Interrupt 2 is Timer in UAP Module */
```

231421-30

```
/PCO/USR/CHUCK/CSRC/DLD.C


Isr1()
{
    Write("\nInterrupt 1\n");
    outb(0xE0, 0x61);    /* EOI 80130 */
}

    /* 586 Interrupt service routine: Interrupt 3 */

Isr_586()
{

    u_short     stat_scb;
    struct  CB      *pcb;

    enable();    /* nesting only the uart interrupt */

    Wait_Scb();
    scb.cmd = (stat_scb = scb.stat) & (CX | CNA | FR | RNR);
    CA;

    if (stat_scb & (FR | RNR))
        Recv_Int_Processing();

    if (stat_scb & CNA) {  /* end of cb processing */

        Reset_Timeout();     /* clear deadman timer  */
        pcb = Build_Ptr(scb.cbl_offset);

#ifdef DEBUG

        if (begin_cbl == pNULL){
            Bug("DLD: begin_cbl == NULL in interrupt routine\n");
            return;
        }

        if ((pcb->stat & 0xC000) != 0x8000)
            Fatal("DLD: C bit not set or B bit set in interrupt routine\n");

#endif /* DEBUG */

        switch (pcb->cmd & CMD_MASK) {

        case TRANSMIT:

            if (flags.stat_on == 1) {/* if Transmit Statistics are collected do */

                /* if sqe bit = 0 and there were no collisions -> sqe error
                   this condition will occur on the first transmission if
                   there were no collisions, or if the previous transmit
                   command reached the max collision count, and the current
                   transmission had no collisions */

                if ((pcb->stat & (SQEMASK | MAXCOLMASK | COLLMASK)) == 0)
                    ++sqe_err_cnt;

                if (pcb->stat & DEFERMASK)
                    ++defer_cnt;
```

231421-31

/PCO/USR/CHUCK/CSRC/DLD.C

```
                if (pcb->stat & NOERRBIT)
                    ++good_xmit_cnt;
                else {

                    if (pcb->stat & NOCRSMASK)
                        ++no_crs_cnt;
                    if (pcb->stat & UNDERRUNMASK)
                        ++underrun_cnt;
                    if (pcb->stat & MAXCOLMASK)
                        ++max_col_cnt;
                }
            }
            if (pcb->parm1 != NULL)
                Put_Tbd(Build_Ptr(pcb->parm1));
            break;

        case DIAGNOSE:

            flags.diag_done = 1;
            if ((pcb->stat & NOERRBIT) == 0)
                Self_Test = FAILED_DIAGNOSE;
            break;


        default:
                ;
        }


        /* check to see if another command is queued */

        if (pcb->link == NULL)
            begin_cbl = pNULL;

        else { /* restart the CU and execute the next command on the cbl */

            begin_cbl = Build_Ptr(pcb->link);
            scb.cbl_offset = pcb->link;
            Wait_Scb();
            scb.cmd = CU_START;
            CA;
            Wait_Scb();
            Set_Timeout();      /* START deadman timer */
        }
        if ((pcb->cmd & CMD_MASK) == MC_SETUP)
            pcb->cmd = 0; /* clear MC_SETUP cmd word, this will implement a
                             lock semephore so that it won't be reused until
                             it is completed */
        else
            Put_Cb(pcb); /* Don't return MC_SETUP cmd block.  It's not a
                            general purpose command block from free CB list */
    }
    disable();  /* disable cpu int so that the 586 isr will not nest */
    EOI_80130;
}
```

231421-32

```
/PCD/USR/CHUCK/CSRC/DLD.C


Recv_Int_Processing()
{
    struct      FD      *pfd;   /* points to the Frame Descriptor */
    struct      RBD     *q,     /* points to the last rbd for the frame */
                        *prbd;  /* points to the first rbd for the frame */

    for (pfd = begin_fd; pfd != pNULL; pfd = begin_fd)
        if (pfd->stat & CBIT) {
            begin_fd = (struct FD *) Build_Ptr(pfd->link);
            prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
            if (prbd != pNULL) { /* check to see if a buffer is attached */

#ifdef DEBUG
                if (prbd != begin_rbd)
                    Fatal("DLD: prbd != begin_rbd in Recv_Int_Processing\n");
#endif /* DEBUG */
                for (q = prbd; (q->act_cnt & EOFBIT) != EOFBIT;
                                        q = (struct RBD *) Build_Ptr(q->link));

                begin_rbd = (struct RBD *) Build_Ptr(q->link);
                q->link = NULL;
            }
            if (pfd->stat & OUT_OF_RESOURCES)
                Put_Free_RFA(pfd);
            else {
                /* if the DLD is in a loopback test, check the frame recv */
                if (flags.lpbk_mode == 1)
                    Loopback_Check(pfd);
                else

            /* if it's a multicast address check to see if it's
                    in the multicast address table, if not discard the frame */

              if ( ((pfd->dest_addr[0] & 01) == 01) && (!Check_Multicast(pfd)))
                    Put_Free_RFA(pfd);
              else
                {   Recv_Frame(pfd);
                    ++recv_frame_cnt;
                }
            }
        }
        else {
            Ru_Start(); /* If RU has gone into no resources, restart it */
            break;
        }
}


Loopback_Check(pfd)         /* Called by Recv_Int_Processing; checks address
                               and data of potential loopback frame */
    struct  FD  *pfd;
{
    struct RBD  *prbd;
    struct RB   *pbuf;
```

231421-33

/PCO/USR/CHUCK/CSRC/DLD.C

```
    if ( bcmp((char *) &pfd->src_addr[0], &whoami[0], ADD_LEN) != 0 ) {
        Put_Free_RFA(pfd);
        return;
    }
    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset); /* point to receive
                                                         buffer descriptor */
    pbuf = (struct RB *) prbd->buff_ptr; /* point to receive buffer */

    if ( bcmp((char *) pbuf, &lpbk_frame[0], LPBK_FRAME_SIZE) != 0) {
        Put_Free_RFA(pfd);
        return;
    }

    flags.lpbk_test = 1;  /* passed loopback test */
    Put_Free_RFA(pfd);
}

Check_Multicast(pfd)    /* returns true if multicast address is in MAT */
    struct      FD  *pfd;
{
    struct      MAT *pmat;

    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE &&
        (bcmp((char *) &pfd->dest_addr[0], &pmat->addr[0], ADD_LEN) == 0))
        break;

    if (pmat > &mat[MULTI_ADDR_CNT - 1])
        return(FALSE);
    return(TRUE);
}


    /* Test the Link function: executes Diagnose and Loopback tests */

Test_Link()
{
    Self_Test = PASSED;
    Diagnose();
    if (Self_Test == FAILED_DIAGNOSE)
        return;
    Ru_Start();     /* start up the RU for loopback tests */
    flags.lpbk_mode = 1; /* go into loopback mode */

    flags.lpbk_test = 0; /* set looback test to false */
    Send_Lpbk_Frame();  /* internal loopback test */
    if (flags.lpbk_test == 0) {
        Self_Test = FAILED_LPBK_INTERNAL;
        flags.lpbk_mode = 0;
        return;
    }

    flags.lpbk_test = 0;
    Configure(EXTERNAL_LOOPBACK);   /* external loopback test w/ ESI in lpbk */
    Send_Lpbk_Frame();
    if (flags.lpbk_test == 0) {
        Self_Test = FAILED_LPBK_EXTERNAL;
```

231421-34

/PCO/USR/CHUCK/CSRC/DLD.C

```
        flags.lpbk_mode = 0;
        return;
    }

    flags.lpbk_test = 0;     /* external loopback test through transceiver */
    NO_ESI_LOOPBACK;
    Send_Lpbk_Frame();
    if (flags.lpbk_test == 0)
        Self_Test = FAILED_LPBK_TRANSCEIVER;

    flags.lpbk_mode = 0; /* leave loopback mode */
}

Send_Lpbk_Frame()
{
    struct   TBD    *ptbd;
    int             i;

    for (i = 0; i < 8; i++) {   /* send lpbk frame 8 times, since it's
                                   best effort delivery */


#ifdef DEBUG
        if ((ptbd = Get_Tbd()) == pNULL)
            Fatal("dld - Send_Lpbk_Frame - couldn't get a TBD\n");
#else
        ptbd = Get_Tbd();

#endif /* DEBUG */

        ptbd->act_cnt = EOFBIT | LPBK_FRAME_SIZE;
        bcopy((char *) ptbd->buff_ptr, &lpbk_frame[0], LPBK_FRAME_SIZE);

        while(!Send_Frame(ptbd, &whoami[0]));
    }
}


Diagnose()
{
    struct    CB    *pcb;

#ifdef DEBUG
        if ((pcb = Get_Cb()) == pNULL)
        Fatal("dld - Diagnose - couldn't get a CB\n");
#else
        pcb = Get_Cb();

#endif /* DEBUG */

        flags.diag_done = 0;
        Self_Test = FALSE;
        pcb->cmd = DIAGNOSE | ELBIT;

        Issue_CU_Cmd(pcb);

        while (flags.diag_done == 0) ;  /* wait for Diag cmd to finish */
```

231421-35

/PCO/USR/CHUCK/CSRC/DLD.C

```
}

Configure(loopflag)

    u_short loopflag;
{
    struct  CB    *pcb;

#ifdef DEBUG
    if ((pcb = Get_Cb()) == pNULL)
        Fatal("dld - Configure - couldn't get a CB\n");
#else
    pcb = Get_Cb();

#endif /* DEBUG */


    /* Ethernet default parameters */

    pcb->parm1 = 0x080C;
    pcb->parm2 = 0x2600 | loopflag;
    pcb->parm3 = 0x6000;
    pcb->parm4 = 0xF200;
    pcb->parm5 = 0x0000;
    if (loopflag == NO_LOOPBACK)
        pcb->parm6 = 0x0040;
    else
        pcb->parm6 = 0x0006;     /* loopback frame is less bytes than
                                    the minimum frame length */
    pcb->cmd    = CONFIGURE | ELBIT;


    Issue_CU_Cmd(pcb);
}

/* Send a frame to the cable, pass a pointer to the destination address
   and a pointer to the first transmit buffer descriptor.  */

Send_Frame(ptbd, padd)  /* returns false if it can't get a Command block */
struct      TBD     *ptbd;
char                *padd;
{
    struct      CB  *pcb;

    u_short         length;

    flags.reset_sema = 1;

    if ((pcb = Get_Cb()) == pNULL) {
        flags.reset_sema = 0;
        if (flags.reset_pend == 1)
            Reset_586();
        return(FALSE);
    }

    pcb->parm1 = Offset(ptbd);
```

231421-36

```
/PCO/USR/CHUCK/CSRC/DLD.C

    /* move destination address to command block */
    bcopy((char *)&pcb->parm2, (char *)padd, ADD_LEN);


    /* calculate the length field by summing up all the buffers */

    for (length = 0; ptbd->link != NULL; ptbd = Build_Ptr(ptbd->link))
        length += ptbd->act_cnt;

    length += (ptbd->act_cnt & 0x3FFF);  /* add the last buffer */

    /* check to see if padding is required, do not do padding on loopback */

    /* this will not work if MIN_DATA_LEN > TBUF_SIZE */

    if ((length < MIN_DATA_LEN) &&  /* assumes a 4 byte CRC */
        (bcmp(&whoami[0], (char *)padd, ADD_LEN) != 0))

            ptbd->act_cnt = MIN_DATA_LEN | EOFBIT;

    pcb->parm5 = length;          /* length field */


    pcb->cmd = TRANSMIT | ELBIT;


    Issue_CU_Cmd(pcb);
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(TRUE);
}


Add_Multicast_Address(pma)  /* pma - pointer to multicast address */
char    *pma;               /* returning false means the Multicast address
                               table is full */
{
    struct      MAT     *pmat;

    flags.reset_sema = 1;

/* if the multicast address is a duplicate of one already in the MAT,
   then return */

    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE &&
            (bcmp( &pmat->addr[0], (char *) pma, ADD_LEN) == 0)) {
            return(TRUE);
        }


    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if (pmat->stat == FREE) {
            pmat->stat = INUSE;
            bcopy( &pmat->addr[0], (char *) pma, ADD_LEN);
            break;
```

231421-37

/PCO/USR/CHUCK/CSRC/DLD.C

```
        }

    if (pmat > &mat[MULTI_ADDR_CNT - 1]) {
        flags.reset_sema = 0;
        if (flags.reset_pend == 1)
            Reset_586();
        return(FALSE);
    }

    Set_Multicast_Address();
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(TRUE);
}


Delete_Multicast_Address(pma)  /* returning false means the multicast address
                                  was not found */
char    *pma;
{
    struct      MAT     *pmat;

    flags.reset_sema = 1;

    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE &&
            (bcmp( &pmat->addr[0], (char *) pma, ADD_LEN) == 0)) {
            pmat->stat = FREE;
            break;
        }

    if (pmat > &mat[MULTI_ADDR_CNT - 1]) {
        flags.reset_sema = 0;
        if (flags.reset_pend == 1)
            Reset_586();
        return(FALSE);
    }

    Set_Multicast_Address();
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(TRUE);
}

Set_Multicast_Address()
{
    struct      MAT     *pmat;
    struct      MA_CB   *pma_cb;
    u_short     i;

    i = 0;
    pma_cb = &ma_cb;
    while (pma_cb->cmd != 0) ; /* if the MA_CB is inuse, wait until it's free */
    pma_cb->link = NULL;
```

231421-38

```
/PCD/USR/CHUCK/CSRC/DLD.C

    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE) {
            bcopy( &pma_cb->mc_addr[i], &pmat->addr[0], ADD_LEN);
            i += ADD_LEN;
        }

    pma_cb->mc_cnt = i;
    pma_cb->cmd = MC_SETUP | ELBIT;

    Issue_CU_Cmd(pma_cb);

}


Put_Free_RFA(pfd)  /* Return Frame Descriptor and Receive Buffer
                      Descriptors to the Free Receive Frame Area */

    struct      FD      *pfd;
{
    struct      RBD     *prbd,  /* points to beginning of returned RBD list */
                        *q;     /* points to end of returned RBD list */
    char        ru_start_flag_fd,  /* indicates whether to restart RU */
                ru_start_flag_rbd;

    flags.reset_sema = 1;
    ru_start_flag_fd = ru_start_flag_rbd = FALSE;
    pfd->el_s = ELBIT;
    pfd->stat = 0;
    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);/* pick up the link to the rbd */
    pfd->link = pfd->rbd_offset = NULL;

    /*  Disable_586_Int(); this command is only necessary in a multitasking
    program.  However in this single task environment this routine is originally
    called from isr_586(), therefore interrupts are already disabled */


    if (begin_fd == pNULL)
        begin_fd = end_fd = pfd;
    else {
        end_fd->link = Offset(pfd);
        end_fd->el_s = 0;
        end_fd = pfd;
        ru_start_flag_fd = TRUE;
    }

    if (prbd != pNULL) {        /* if there is a rbd attached to the fd then
                                find the beginning and end of the rbd list */

        for (q = prbd; q->link != NULL; q = Build_Ptr(q->link))
            q->act_cnt = 0;

            /* now prbd points to the beginning of the rbd list and
               q points to the end of the list */

        q->size = RBUF_SIZE | ELBIT;
        q->act_cnt = 0;
```

231421-39

```
/PCO/USR/CHUCK/CSRC/DLD.C

        if (begin_rbd == pNULL) { /* if there is nothing on the list
                                     create a new list */

            begin_rbd = prbd;
            end_rbd = q;
            if (prbd != q)
                ru_start_flag_rbd = TRUE; /* if there is more than one rbd
                                             returned start the RU */
        }
        else {
                                  /* if the rbd list already exists add on
                                     the new returned rbds */
            end_rbd->link = Offset(prbd);
            end_rbd->size = RBUF_SIZE;
            end_rbd = q;
            ru_start_flag_rbd = TRUE;
        }
    }
    if (ru_start_flag_fd && ru_start_flag_rbd)
        Ru_Start();

    /*  Enable_586_Int(); if Disable_586_Int() is used above */

    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
}

Ru_Start()
{
    if ((scb.stat & RU_MASK) == RU_READY) /* if the RU is already 'ready'
                                             then return */
        return;

    if ((begin_fd->stat & CBIT) == CBIT)
        return;

    begin_fd->rbd_offset = Offset(begin_rbd); /* link the beginning of the rbd
                                                 list to the first fd */
    scb.rfa_offset = Offset(begin_fd);
    Wait_Scb();
    scb.cmd = RU_START;
    CA;
}

Software_Reset()
{
    scb.cmd = RESET;
    CA;
    Wait_Scb();
}

Issue_Reset_Cmds()
{
    Wait_Scb();
    scb.cmd = CU_START;
    CA;
```

231421-40

/PCO/USR/CHUCK/CSRC/DLD.C

```
    Wait_Scb();

    outw(0xFF5E, 0) ;          /* shut off timer 1 interrupt */
    outw(TIMER1_CNT, 0);
    outw(0xFF5E, 0xC009);      /* use timer 1 without interrupt as a deadman */

    while ((inw(0xFF5E) & 0x0020) == 0)  /* if Max Cnt bit is set before CNA
                                            is set, 586 Cmd deadlocked */
        if ((scb.stat & CNA) == CNA)
            break;

    if (scb.stat & CNA != CNA)
        Fatal("DLD:Issue_Reset_Cmds - Command deadlock during reset procedure\n");

    Reset_Timeout();

    scb.cmd = CNA;  /* Acknowledge CNA interrupt */
    CA;
    Wait_Scb();
}


/* Execute a reset, Configure, SetAddress, and MC_Setup, then restart the
    Receive Unit and the Command Unit */
Reset_586()
{
    struct      MAT     *pmat;
    u_short     i;

    ++reset_cnt;
    Disable_586_Int();
    ESI_LOOPBACK;
    Software_Reset();

    scb.stat = 0;

    CA;     /* wait for the 586 to complete initialization */


    for ( i = 0; i <= 0xFF00; i++)
        if (scb.stat == (CX | CNA))
            break;


    if (i >0xFF00)
        Fatal("DLD:init - Did not get an interrupt after Software Reset\n");

    /* Ack the reset Interrupt */
    Wait_Scb();
    scb.cmd = (CX | CNA);
    CA;
    Wait_Scb();

#ifdef  DEBUG
    if ( begin_cbl == pNULL)
        Fatal("DLD: begin_cbl = NULL in Reset_586");
#endif  /* DEBUG */
```

231421-41

/PCO/USR/CHUCK/CSRC/DLD.C

```
    /* Configure the 586 */
    /* Ethernet default parameters; Configure is not necessary when using
       default parameters  */

    res_cb.link = NULL;

    res_cb.parm1 = 0x080C;
    res_cb.parm2 = 0x2600;
    res_cb.parm3 = 0x6000;
    res_cb.parm4 = 0xF200;
    res_cb.parm5 = 0x0000;
    res_cb.parm6 = 0x0040;
    res_cb.cmd   = CONFIGURE | ELBIT;

    scb.cbl_offset = Offset(&res_cb.stat);

    Issue_Reset_Cmds();

    /* Set the Individual Address */

    bcopy((char *) &res_cb.parm1, &whoami[0], ADD_LEN); /* move the prom
                                                           address to IA cmd */
    res_cb.cmd = IA | ELBIT;

    Issue_Reset_Cmds();

    /* reload the multicast addresses */

    i = res_ma_cb.stat = 0;
    res_ma_cb.link = NULL;

    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE ) {
            bcopy( &res_ma_cb.mc_addr[i], &pmat->addr[0], ADD_LEN);
            i += ADD_LEN;
        }

    res_ma_cb.mc_cnt = i;
    res_ma_cb.cmd = MC_SETUP | ELBIT;
    scb.cbl_offset = Offset(&res_ma_cb.stat);


    Issue_Reset_Cmds();

    /* Restart the Command Unit and the Receive Unit */

    flags.reset_sema = 0;
    flags.reset_pend = 0;

    NO_ESI_LOOPBACK;

    Recv_Int_Processing();

    scb.cbl_offset = begin_cbl;
    Wait_Scb();
```

231421-42

```
/PCO/USR/CHUCK/CSRC/DLD.C

    scb.cmd = CU_START;
    Set_Timeout();          /* Set Deadman Timer */
    CA;
    Enable_586_Int();
}


/* bcopy -- byte copy routine */
bcopy(dst, src, nbytes)
char    *dst, *src;
int     nbytes;
{
    while (nbytes--) *dst++ = *src++;
}

/* bcmp -- byte compare */
bcmp(s1, s2, nbytes)
char    *s1, *s2;
int     nbytes;
{
    while (nbytes-- && *s1++ == *s2++);
    return(*--s1 - *--s2);
}
```

231421-43

/PCD/USR/CHUCK/CSRC/LLC.C

```
/*****************************************************************************
*                                                                            *
*                IEEE 802.2 Logical Link Control Layer                       *
*                        (Station Component)                                 *
*                                                                            *
*****************************************************************************/

#include "dld.h"

extern  char    *pNULL;

extern  struct  TBD *Get_Tbd();
extern  char        *Build_Ptr();

readonly char   xid_frame[XID_LENGTH] = { 0, 0, XID, 0x81, 0x01, 0};
/* DSAP, SSAP, XID, xid class 1 response */

struct LAT lat[DSAP_CNT];

Init_Llc()
{
    struct  LAT     *plat;

    for (plat = &lat[0]; plat <= &lat[DSAP_CNT - 1]; plat++)
        plat->stat = FREE;
    return(Init_586());
}

/* Function for adding a new DSAP */

Add_Dsap_Address(dsap, pfunc) /* DSAP must be divisible by 2**(8-N), where
                                 2**N = DSAP_CNT.  (i.e.  N LSBs must be 0).
                                 The function will return FALSE if does not
                                 meet the above requirements, or the Lsap
                                 Address Table is full, or the address has
                                 already been used.  NULL DSAP address is
                                 reserved for the Station Component */
int dsap, (*pfunc) ();
{
    struct  LAT     *plat;

    if ((dsap << (8-DSAP_SHIFT) & 0x00FF) != 0 || dsap == 0)
        return (FALSE);

    /* Check for duplicate dsaps.  */
    if ( (plat = &lat[dsap >> DSAP_SHIFT])->stat == FREE) {
            plat->stat = INUSE;
            plat->p_sap_func = pfunc;
            return (TRUE);
    }
    else
        return(FALSE);
}

/* Function for deleting DSAPs */

Delete_Dsap_Address(dsap) /* If the specified connection exists, it is severed.
                             If the connection does not exist, the command is ignored.  */
```

231421-44

```
/PCO/USR/CHUCK/CSRC/LLC.C

int dsap;
{
    lat[dsap >> DSAP_SHIFT].stat = FREE;
}

Recv_Frame(pfd)
    struct  FD              *pfd;
{
    struct  RBD             *prbd;
    struct  FRAME_STRUCT    *pfs;
    struct  LAT             *plat;

    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
    pfs  = (struct FRAME_STRUCT *) prbd->buff_ptr;

    if (pfd->rbd_offset != NULL) { /* There has to be a rbd attached
                                      to the fd, or else the frame is
                                      too short. */
        if (pfs->dsap == 0) {   /* if the frame is addressed to the Station
                                   Component, then a response may be required */

            if ( !(pfs->ssap & C_R_BIT) ) {/* if the frame received is a response,
                                              instead of a command, then reject it.
                                              Because this software does not implement
                                              DUPLICATE_ADDRESS_CHECK. -> no response
                                              frames should be recv'd */
                Station_Component_Response(pfd);
            }
        }
        /* not addressed to Station Component, */
        /* check to see if the dsap addressed is active */
        else if ((pfs->dsap << (8-DSAP_SHIFT) & 0x00FF) == 0 &&
                 (plat = &lat[(pfs->dsap) >> DSAP_SHIFT])->stat == INUSE ) {
            (*plat->p_sap_func)(pfd);   /* call the function associated
                                           with the dsap received */
            return;
        }
    }
    Put_Free_RFA(pfd); /* return the pfd if not given to the user saps */
}


Station_Component_Response(pfd)

    struct  FD      *pfd;
{
    struct FRAME_STRUCT     *prfs, *ptfs;
    struct TBD              *ptbd, *begin_ptbd, *q;
    struct RBD              *prbd;

    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
    prfs  = (struct FRAME_STRUCT *) prbd->buff_ptr;

    switch (prfs->cmd & ~P_F_BIT)
        {
        case    XID:
```

231421-45

/PCO/USR/CHUCK/CSRC/LLC.C

```
            while ((ptbd = Get_Tbd()) == pNULL);
            ptbd->act_cnt = EOFBIT | XID_LENGTH;
            bcopy ((char *) ptbd->buff_ptr, &xid_frame[0], XID_LENGTH);
            ptfs = (struct FRAME_STRUCT *) ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT; /* return the frame
                                                  to the sender */
            ptfs->ssap = 0;
            while(!Send_Frame(ptbd, Build_Ptr(pfd->src_addr)));
            break;

        case    TEST:

            for (prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset),
                        q = begin_ptbd = pNULL; prbd != pNULL;
                                prbd = Build_Ptr(prbd->link)) {

                while ((ptbd = Get_Tbd()) == pNULL);
                if (q != pNULL)
                    q->link = Offset(ptbd);
                else
                    begin_ptbd = ptbd;
                ptbd->act_cnt = prbd->act_cnt;
                bcopy((char *) ptbd->buff_ptr, (char *) prbd->buff_ptr,
                                                    ptbd->act_cnt & 0x3FFF);
                q = ptbd;
            }

            ptfs = (struct FRAME_STRUCT *) begin_ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT;  /* return the frame to
                                                   the sender */
            ptfs->ssap = 0;
            while(!Send_Frame(begin_ptbd, Build_Ptr(pfd->src_addr)));
            break;

        }
}
```

231421-46

```
/PCO/USR/CHUCK/CSRC/UAP.C


/*************************************************************************
*                                                                        *
*                        User Application Program                        *
*                Async to IEEE 802.2/802.3 Protocol Converter            *
*                                                                        *
*************************************************************************/

#include "dld.h"

/* ASCII Characters */
#define ESC      0x1B
#define LF       0x0A
#define CR       0x0D
#define BS       0x08
#define BEL      0x07
#define SP       0x20
#define DEL      0x7F
#define CTL_C        0x03

/* Hardware */
#define CH_B_CTL     0x00DE
#define CH_A_CTL     0x00DC
#define CH_B_DAT     0x00DA
#define CH_A_DAT     0x00D8
#define UART_STAT_MSK    0x70

/* Interrupt cases for 8274 */
#define UART_TX_B        0
#define UART_RECV_B      0x08
#define UART_RECV_ERR_B 0x0C
#define  EXT_STAT_INT_B 0x04
#define  EXT_STAT_INT_A 0x14

char     fifo_t[256];
char     fifo_r[256];
char     wra[5], wrb[5];
unsigned     char     in_fifo_t, out_fifo_t, in_fifo_r, out_fifo_r, actual;
u_short      t_buf_stat, r_buf_stat;

char     cbuf[80];    /* Command line buffer */
char     line[81];    /* Monitor Mode display line */

unsigned     char     dsap, ssap, send_flag, local_echo;
char     Dest_Addr[ADD_LEN];
char     Multi_Addr[ADD_LEN];

int tmstat; /* terminal mode status: for leaving terminal mode */
int dhex, monitor_flag, hs_stat;         /* flags */


extern   struct TBD       *Get_Tbd();
extern   char             *Build_Ptr();

extern   struct FLAGS     flags;

extern   char     xid_frame[];
extern   char     whoami[];
```

231421-47

/PCO/USR/CHUCK/CSRC/UAP.C

```
extern  struct  MAT     mat[];
extern  struct  LAT     lat[];
extern  char    *pNULL;

extern  unsigned long   good_xmit_cnt;
extern  u_short         underrun_cnt;
extern  u_short         no_crs_cnt;
extern  unsigned long   defer_cnt;
extern  u_short         sqe_err_cnt;
extern  u_short         max_col_cnt;
extern  unsigned long   recv_frame_cnt;
extern  u_short         reset_cnt;

extern  struct  SCB     scb;

/* Macro 'type' of definitions */

#define RTS_ONB outb(CH_B_CTL,0x05);outb(CH_B_CTL,wrb[5]=wrb[5]|0x02)
#define RTS_OFFB outb(CH_B_CTL,0x05);outb(CH_B_CTL,wrb[5]=wrb[5]&0xFD)
#define RTS_ONA outb(CH_A_CTL,0x05);outb(CH_A_CTL,wra[5]=wra[5]|0x02)
#define RTS_OFFA outb(CH_A_CTL,0x05);outb(CH_A_CTL,wra[5]=wra[5]&0xFD)
#define UART_TX_DI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]&0xFD)
#define UART_TX_EI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]|0x02)
#define UART_RX_DI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]&0xE7)
#define UART_RX_EI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]|0x10)
#define RESET_TX_INT outb(CH_B_CTL,0x28)
#define EOI_8274 outb(CH_A_CTL,0x38) /* 8274 int is IR3 on 80130 */
#define EOI_80130_8274 outb(0xE0,0x60)
#define EOI_80130_TIMER outb(0xE0,0x62)


Enable_Uart_Int()
{
    int     c;

    c = inb(0xE2);  /* read the 80130 interrupt mask register */
    outb(0xE2, 0x00FE & c); /* write to the 80130 interrupt mask register */
}

Disable_Uart_Int()
{
    int     c;

    c = inb(0xE2);
    outb(0xE2, 0x0001 | c);
}

Enable_Timer_Int()
{
    int     c;

    outb(0xEA, 125);
    outb(0xEA, 0x00); /* Timer 1 interrupts every .125 sec */
    send_flag = FALSE;
    c = inb(0xE2);  /* read the 80130 interrupt mask register */
    outb(0xE2, 0x00FB & c); /* write to the 80130 interrupt mask register */
}
```

231421-48

```
/PCO/USR/CHUCK/CSRC/UAP.C


Disable_Timer_Int()
{
    int     c;

    c = inb(0xE2);
    outb(0xE2, 0x0004 | c);
}


Co(c)
    char    c;
{
    while ( (inb(CH_B_CTL) & 4) == 0 );
    outb(CH_B_DAT, c);
}

Ci()
{
    while ( (inb(CH_B_CTL) & 1) == 0 );
    return(inb(CH_B_DAT) & 0x7F);
}

Read(pmsg, cnt, pact)
    char    *pmsg;
    unsigned char   cnt, *pact;
{
    unsigned char   i;
    char    c,  buf[200];

    for (i = c = 0; (c != CR) && (c != LF) && (i < 198); ) {
        c = Ci() & 0x7F;
        if (c == BS || c == DEL) {
            if (i > 0) {
                --i;
                Co(BS); Co(SP); Co(BS);
            }
        }
        else
            if (c >= SP) {
                Co(c);
                buf[i++] = c;
            }
        else
            if ((c == CR) || (c == LF)) {
                buf[i++] = CR;
                buf[i++] = LF;
            }
        else Co(BEL);
    }
    Co(CR); Co(LF);
    if (i > cnt)
        *pact = cnt;
    else
        *pact = i;
    for (i = 0; i < *pact ; i++)
        *pmsg++ = buf[i];
```

231421-49

/PCO/USR/CHUCK/CBRC/UAP.C

```
}

Read_Char()
{
    unsigned char   i;

    Read(&cbuf[0], 80, &actual);
    i = Skip(&cbuf[0]);
    return(cbuf[i]);
}

Write(pmsg)
    char    *pmsg;
{
    while (*pmsg != '\0') {
        if (*pmsg == '\n')
            Co(CR);
        Co(*pmsg++);
    }
}

Fatal(pmsg) /* write a message to the screen then stop */
    char    *pmsg;
{
    Write("Fatal: ");
    Write(pmsg);
    for(;;);
}

Bug(pmsg)   /* write a message to the screen then continue */
    char    *pmsg;
{
    Write("Bug: ");
    Write(pmsg);
}

Ascii_To_Char(c) /* convert ASCII-Hex to Char */
    char    c;
{
    if (('0' <= c) && (c <= '9'))
        return(c - '0');
    if (('A' <= c) && (c <= 'F'))
        return(c - 0x37);
    if (('a' <= c) && (c <= 'f'))
        return(c - 0x57);
    return(0xFF);
}

Lower_Case(c)
char    c;
{
    if (('a' <= c) && (c <= 'z'))
        return(c);
    if (('A' <= c) && (c <= 'Z'))
        return(c + 0x20);
    return(0);
}
```

231421-50

```
/PCO/USR/CHUCK/CSRC/UAP.C


Char_To_Ascii(c, ch) /* convert char to ASCII-Hex */
    unsigned char   c, ch[];
{
    unsigned char   i;

    i = (c & 0xF0) >> 4;
    if (i < 10)
        ch[0] = i + 0x30;
    else
        ch[0] = i + 0x37;
    i = (c & 0x0F) ;
    if (i < 10)
        ch[1] = i + 0x30;
    else
        ch[1] = i + 0x37;
    ch[2] = '\0';
}

Skip(pmsg)  /* skip blanks */
    char    *pmsg;
{
    int     i;

    for (i = 0; *pmsg == ' '; i++, pmsg++);
    return(i);
}

Read_Int()  /* Read a 16 bit Integer */
{
    u_short     wd, wh, wd1, wh1, j;
    char        i, done, hex, dover, hover;

    for (done = FALSE; done == FALSE; ) {
        Read(&cbuf[0], 80, &actual);
        i = Skip(&cbuf[0]);

        for (hex = dover = hover = FALSE, wd = wh = wd1 = wh1 = 0;
                        (j = Ascii_To_Char(cbuf[i])) <= 15; i++) {
            if (j > 9)
                hex = TRUE;
            wd = wd*10 + j;
            wh = wh*16 + j;
            if (wd < wd1)
                dover = TRUE;
            if (wh < wh1)
                hover = TRUE;
            wd1 = wd; wh1 = wh;
        }
        if (cbuf[i] == 'H' || cbuf[i] == 'h' || cbuf[i] == CR ||
                                    cbuf[i] == LF || cbuf[i] == ' ') {
            if (cbuf[i] == 'H' || cbuf[i] == 'h')
                hex = TRUE;
            if (hex == TRUE && hover == FALSE)
                done = TRUE;
            if (hex == FALSE && dover == FALSE)
                done = TRUE;
```

231421-51

/PCO/USR/CHUCK/CSRC/UAP.C

```
            if (!done) {
                Write("\n This number is too big.\n It has to be less than 65536.\n");
                Write("\n Enter number --> ");
            }
        }
        else
            Write(" Illegal Character\n Enter a number -->");
    }
    if (hex)
        return(wh);
    return(wd);
}

Int_To_Ascii(value, base, ld, ch, width) /* convert an integer to an ASCII string */
    unsigned long   value;
    u_short      base, width;
    char    ch[], ld;
{
    u_short i, j;

    for (i = 0; i < width; i++) {
        j = value % base;
        if (j < 10)   ch[i] = j + 0x30;
        else ch[i] = j + 0x37;
        value = value / base;
    }
    for (i = width - 1; ch[i] == '0' && i > 0; i--)
        ch[i] = ld;
    ch[width] = '\0';
}

Write_Long_Int(dw, i)
    unsigned long   dw;
    u_short     i;
{
    u_short     j;
    char     ch[11];

    if (dhex)
        Int_To_Ascii(dw, 16, ' ', &ch[0], 8);
    else
        Int_To_Ascii(dw, 10, ' ', &ch[0], 10);
    for (j = 0; ch[j] != '\0'; i--, j++)
        line[i] = ch[j];
}

Write_Short_Int(w, i)
    u_short w, i;
{
    u_short j;
    char    ch[6];
    unsigned long   dw;

    dw = w;
    if (dhex)
        Int_To_Ascii(dw, 16, '0', &ch[0], 4);
    else
```

231421-52

/PCD/USR/CHUCK/CSRC/UAP.C

```
        Int_To_Ascii(dw, 10, '0', &ch[0], 5);
    for (j = 0; ch[j] != '\0'; i--, j++)
        line[i] = ch[j];
}


Yes()
{
    char    b;

    for ( ; ; ) {
        b = Read_Char();
        if ((b == 'Y') || (b == 'y'))
            return(TRUE);
        if ((b == 'N') || (b == 'n'))
            return(FALSE);
        Write(" Enter a Y or N --> ");
    }
}

Read_Addr(pmsg, add, cnt) /* pmsg - pointer to the output message */
                          /* add - pointer to the address */
                          /* cnt - number of bytes in the address */
    char    *pmsg, add[], cnt;
{
    char    i, j;

    for ( ; ; ) {
        Write(pmsg);
        Read(&cbuf[0], 80, &actual);
        for (j = skip(&cbuf[0]), i = 0; i < 2*cnt ; i++, j++) {
            if (('0' <= cbuf[j]) && (cbuf[j] <= '9'))
                cbuf[i] = cbuf[j] - '0';
            else
                if (('A' <= cbuf[j]) && (cbuf[j] <= 'F'))
                    cbuf[i] = cbuf[j] - 0x37;
            else
                if (('a' <= cbuf[j]) && (cbuf[j] <= 'f'))
                    cbuf[i] = cbuf[j] - 0x57;
            else {
                Write(" Illegal Character\n");
                break;
            }
        }
        if (i >= 2*cnt - 1)
            break;
    }
    for (i = 0; i <= cnt - 1; i++)
        add[(cnt - 1) - i] = cbuf[2*i] << 4 | cbuf[2*i + 1];
}

Write_Addr(padd, cnt)
    char    padd[], cnt;
{
    unsigned char   i, c[3];

    for ( ; cnt >0 ; cnt--) {
```

231421-53

/PCD/USR/CHUCK/CSRC/UAP.C

```
        i = padd[cnt-1];
        Char_To_Ascii(i, &c[0]);
        Write(&c[0]);
    }
    c[0] = '\n';
    c[1] = '\0';
    Write(&c[0]);
}

Recv_Data_1(pfd)    /* Receives the frame from the 802.2 module */

    struct  FD              *pfd;
{
    struct FRAME_STRUCT     *prfs, *ptfs;
    struct TBD              *ptbd, *begin_ptbd, *q;
    struct RBD              *prbd;
    char                    *prbuf;
    int                     cnt;


    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
    prfs  = (struct FRAME_STRUCT *) Build_Ptr(prbd->buff_ptr);

    switch (prfs->cmd & ~P_F_BIT)   {
        case    UI:
            if (monitor_flag)
                break;  /* Don't put data in fifo unless in terminal mode */
            prbuf = (char *) prfs;
            prbuf += 3; /* skip over the header info and point to the data */
            cnt = 3;
            pfd->length -= 3;
            for (; prbd != pNULL; cnt = 0, prbuf = (char *) prbd->buff_ptr){
                for ( ; cnt < (prbd->act_cnt & 0x03FFF) && pfd->length > 0;
                                        cnt++, prbuf++, pfd->length--) {
                    while(r_buf_stat == FULL);
                    Fifo_R_In(*prbuf);
                }
                prbd = Build_Ptr(prbd->link);
#ifdef DEBUG
                if (pfd->length == 0 && prbd != pNULL)
                    Fatal("Uap: Recv_Data_1(pfd) ");
#endif /* DEBUG */
            }
            break;

        case    XID:

            while ((ptbd = Get_Tbd()) == pNULL);
            ptbd->act_cnt = EOFBIT | XID_LENGTH;
            bcopy ((char *) ptbd->buff_ptr, &xid_frame[0], XID_LENGTH);
            ptfs = (struct FRAME_STRUCT *) ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT; /* return the frame
                                                  to the sender */
            ptfs->ssap = ssap;
            while(!Send_Frame(ptbd, Build_Ptr(pfd->src_addr)));
```

231421-54

/PCO/USR/CHUCK/CSRC/UAP.C

```
            break;

        case    TEST:

            for (prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset),
                        q = begin_ptbd = pNULL; prbd != pNULL;
                                prbd = Build_Ptr(prbd->link)) {
                while ((ptbd = Get_Tbd()) == pNULL);
                if (q != pNULL)
                    q->link = Offset(ptbd);
                else
                    begin_ptbd = ptbd;
                ptbd->act_cnt = prbd->act_cnt;
                bcopy((char *) ptbd->buff_ptr, (char *) prbd->buff_ptr,
                                                ptbd->act_cnt & 0x3FFF);
                q = ptbd;
            }

            ptfs = (struct FRAME_STRUCT *) begin_ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT;  /* return the frame to
                                                   the sender */
            ptfs->ssap = ssap;
            while(!Send_Frame(begin_ptbd, Build_Ptr(pfd->src_addr)));
            break;

    }
    Put_Free_RFA(pfd);  /* return the frame */
}


Fifo_T_Out()    /* called by main program */
{
    char    c;

    c = fifo_t[out_fifo_t++];

    Disable_Uart_Int();
    if (out_fifo_t == in_fifo_t)    /* if the fifo is empty */
        t_buf_stat = EMPTY; /* stop filling Transmit Buffer Descriptors */
    else        /* if the fifo was full and is now draining */
        if (t_buf_stat == FULL && out_fifo_t - 80 == in_fifo_t) { /* turn on
                                                                    the spigot */
            RTS_ONB;
            t_buf_stat = INUSE;
        }
    Enable_Uart_Int();
    return(c);
}

Fifo_T_In(c)    /* called by Uart receive interrupt */
    char    c;
{

    fifo_t[in_fifo_t++] = c;
    if (t_buf_stat == EMPTY)
```

231421-55

```
/PCO/USR/CHUCK/CSRC/UAP.C

        t_buf_stat = INUSE; /* start filling Transmit Buffer Descriptor */
    else        /* if there are only 20 locations left, turn off the spigot */
        if (t_buf_stat == INUSE && in_fifo_t + 20 == out_fifo_t) {
            RTS_OFFB;
            t_buf_stat = FULL;
        }
}

Fifo_R_Out()    /* called by transmit interrupt */
{
    char    c;

    c = fifo_r[out_fifo_r++];

    if (out_fifo_r == in_fifo_r)    /* if the fifo is empty */
        r_buf_stat = EMPTY;
    else        /* if the fifo was full and is now draining */
        if (r_buf_stat == FULL && out_fifo_r - 81 == in_fifo_r)
            r_buf_stat = INUSE;
    return(c);
}

Fifo_R_In(c)    /* called by Recv_Data_1() */
    char    c;
{
    fifo_r[in_fifo_r++] = c;
    Disable_Uart_Int();
    if (r_buf_stat == EMPTY) {
        UART_TX_EI_B;
        Co(0);              /* prime the interrupt */
        r_buf_stat = INUSE;
    }
    else        /* if the buffer is full, indicate it */
        if (r_buf_stat == INUSE && in_fifo_r == out_fifo_r)
            r_buf_stat = FULL;
    Enable_Uart_Int();
}

Isr_Uart()
{
    int     stat;
    char    c;

    outb(CH_B_CTL, 2);  /* point to RR2 in 8274 */

    switch(inb(CH_B_CTL) & 0x1C){ /* read 8274 interrupt vector and service it */

    case UART_TX_B:

        if (r_buf_stat == EMPTY) {
            UART_TX_DI_B;           /* if fifo is empty disable transmitter */
            RESET_TX_INT;
        }
        else
            outb (CH_B_DAT, Fifo_R_Out());
        break;
```

231421-56

```
/PCO/USR/CHUCK/CSRC/UAP.C

    case UART_RECV_ERR_B:

        outb(CH_B_CTL, 1); /* point to RR1 in 8274 */
        stat = inb(CH_B_CTL);
        outb(CH_B_CTL, 0x30);
        if (stat & 0x0010)
            Write("\nParity Error Detected\n");
        if (stat & 0x0020)
            Write("\nOverrun Error Detected\n");
        if (stat & 0x0040)
            Write("\nFraming Error Detected\n");
        break;

    case UART_RECV_B:

        c = inb(CH_B_DAT);

        if (hs_stat == TRUE) {
            hs_stat = FALSE;  /* Flag to terminate High Speed Transmit mode */
            break;
        }

        if (local_echo)
            Co(c);              /* echo the char back to the terminal; could cause
                                   a transmit overrun if Tx interrupt is enabled */
        if (c == CTL_C)
            tmstat = FALSE;
        else
            Fifo_T_In(c);
        break;

    case EXT_STAT_INT_B:

        outb(CH_B_CTL, 0x10);   /* reset external status interrupts */
        break;

    case EXT_STAT_INT_A:

        outb(CH_A_CTL, 0x10);
        break;


    default:
        ;
    }
    EOI_80130_8274;
    EOI_8274;
}

Isr2()
{
    send_flag = TRUE;
    outb(0xEA, 125);
    outb(0xEA, 0x00); /* Timer 1 interrupts every .125 sec */
    outb(0xE0, 0x62);   /* EOI 80130 */
}
```

231421-57

/PCO/USR/CHUCK/CSRC/UAP.C

```
Load_Lsap()
{
    int     Recv_Data_1();

    for(;;) {
        Read_Addr("\n\nEnter this Station's LSAP in Hex --> ", &ssap, 1);
        if (!Add_Dsap_Address(ssap, Recv_Data_1)) {
            Write("\n\nError: LSAP Address must be one of the following:\n");
            Write("\n        20H, 40H, 60H, 80H, A0H, C0H, E0H \n");
        }
        else break;
    }
}

Load_Multicast()
{

    for ( ; ; ) {
        Read_Addr("\nEnter the Multicast Address in Hex -->",
                                                &Multi_Addr[0], ADD_LEN);
        if ((Multi_Addr[0] & 0x01) == 0)
            Write("\nSorry, the LSB of the Multicast Address must be 1\n");
        else { if (!Add_Multicast_Address(&Multi_Addr[0])) {
                    Write("\n\nSorry, Multicast Address Table is full!\n");
                    break;
                }
                else {
                    Write("\n\nWould you like to add another Multicast Address?");
                    Write(" (Y or N) --> ");
                    if (!Yes())
                        break;
                }
        }
    }
}

Remove_Multicast()
{

    for ( ; ; ) {
        Read_Addr("\nEnter the Multicast Address that you want to delete in Hex -->",
                                                &Multi_Addr[0], ADD_LEN);
        if ((Multi_Addr[0] & 0x01) == 0)
            Write("\nSorry, the LSB of the Multicast Address must be 1\n");
        else { if (!Delete_Multicast_Address(&Multi_Addr[0])) {
                    Write("\n\nSorry, that Multicast Address doesn't exist!\n");
                    break;
                }
                else {
                    Write("\n\nWould you like to delete another Multicast Address?");
                    Write(" (Y or N) --> ");
                    if (!Yes())
                        break;
                }
        }
    }
}
```

231421-58

```
/PCO/USR/CHUCK/CSRC/UAP.C


Print_Addresses()
{
    struct  MAT *pmat;
    int         stat;

    Write("\n This Stations Host Address is: ");
    Write_Addr(&whoami[0], ADD_LEN);
    Write("\n The Address of the Destination Node is: ");
    Write_Addr(&Dest_Addr[0], ADD_LEN);
    Write("\n This Stations LSAP Address is: ");
    Write_Addr(&ssap, 1);
    Write("\n The Address of the Destination LSAP is: ");
    Write_Addr(&dsap, 1);
    stat = FALSE;
    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if (pmat->stat == INUSE) {
            stat = TRUE;
            break;
        }
    if (stat)   {
        Write("\n The following Multicast Addresses are enabled: ");
        for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
            if (pmat->stat == INUSE) {
                Write_Addr(&pmat->addr[0], ADD_LEN);
                Write("                                                    ");
            }
    }
    else
        Write("\n There are no Multicast Addresses enabled.\n");
}

Init_DataLink()
{
    int     stat;

    if ((stat = Init_Llc()) == PASSED)
        Write("\n\nPassed Diagnostic Self Tests\n\n\n");
    else
        if(stat == FAILED_DIAGNOSE)
            Write("\n\nFailed: Self Test Diagnose Command\n");
    else
        if(stat == FAILED_LPBK_INTERNAL)
            Write("\n\nFailed: Internal Loopback Self Test\n");
    else
        if(stat == FAILED_LPBK_EXTERNAL)
            Write("\n\nFailed: External Loopback Self Test\n");
    else
        if(stat == FAILED_LPBK_TRANSCEIVER)
            Write("\n\nFailed: External Loopback Through Transceiver Self Test\n");
}

Init_Uap()
{

    outb(0xE0, 0x31);   /*initalize 80130 pic - ICW1 */
    outb(0xE2, 0x20);   /* ICW2 */
```

231421-59

```
/PCD/USR/CHUCK/CSRC/UAP.C

    outb(0xE2, 0x10);    /* ICW3 */
    outb(0xE2, 0x0D);    /* ICW4 */
    outb(0xE2, 0x10);    /* ICW6 */
    outb(0xE2, 0xFF);    /* mask all interrupts */

    outw(0xFF20, 0x0020);    /* set 80186 vector base */

    /* Initialize the 80130 timers for Terminal Mode */

    outb(0xEE, 0x34);
    outb(0xE8, 0x88);
    outb(0xE8, 0x0B); /* SYSTICK set for 1 msec */
    outb(0xEE, 0x70);
    outb(0xEA, 125);
    outb(0xEA, 0x00); /* Timer 1 interrupts every .125 sec */

    /* Initialize the 8274 */
    outb(CH_B_CTL, 0x10); outb(CH_B_CTL, 0x28); outb(CH_B_CTL, 0x30);
    outb(CH_A_CTL, 0x38);
    outb(CH_B_CTL, 2); outb(CH_B_CTL, wrb[2] = 0x14);
    outb(CH_B_CTL, 1); outb(CH_B_CTL, wrb[1] = 0x15);
    outb(CH_B_CTL, 5); outb(CH_B_CTL, wrb[5] = 0xEA);

    Write("\n\n\n\n\n\n\n\n\n\n\n\n\n");
    Write("          ******************************************************\n");
    Write("          * 82586 IEEE 802.2/802.3 Compatible Data Link Driver *\n");
    Write("          ******************************************************\n");
    Write("\n\n\n\n\n\n\n");

    Init_DataLink();

    dhex = FALSE;
    monitor_flag = TRUE;

    Read_Addr("\n\nEnter the Address of the Destination Node in Hex --> ",
                                                &Dest_Addr[0], ADD_LEN);

    Load_Lsap();

    Read_Addr("\n\nEnter the Destination Node's LSAP in Hex --> ", &dsap, 1);

    Write("\n\nDo you want to Load any Multicast Addresses? (Y or N) -->");

    if (Yes())
        Load_Multicast();

    Print_Addresses();
}

Terminal_Mode()
{
    int     frame_cnt, buf_cnt;
    struct  TBD     *ptbd, *q, *begin_ptbd;
    char            *pbuf, c;

    Write("\n Would you like the local echo on? (Y or N)-->");

    if(Yes())
```

231421-60

/PCO/USR/CHUCK/CSRC/UAP.C

```
        local_echo = TRUE;
    else
        local_echo = FALSE;

    Write("\n This program will now enter the terminal mode.\n\n");
    Write("\n Press ^C then CR to return back to the menu\n\n");

    /* Initialize Fifo variables */

    out_fifo_t = in_fifo_t = out_fifo_r = in_fifo_r = 0;
    t_buf_stat = EMPTY;  r_buf_stat = EMPTY;

    EOI_80130_8274;
    Enable_Uart_Int();
    Enable_Timer_Int();
    monitor_flag = FALSE;
    tmstat = TRUE;
    while (tmstat)  {

        for (frame_cnt = 0; frame_cnt < MAX_FRAME_SIZE;  q = ptbd) {

            while ((ptbd = Get_Tbd()) == pNULL); /* get a xmit buffer from the
                                                    data link */
            pbuf = (char *) ptbd->buff_ptr;  /* point to the buffer */
            buf_cnt = 0;

            if (frame_cnt == 0) { /* if this is the first buffer, add on IEEE 802.2
                                     header information */
                begin_ptbd = ptbd;
                *pbuf++ = dsap;
                *pbuf++ = ssap;
                *pbuf++ = UI;
                buf_cnt = 3;
            }
            else  q->link = Offset(ptbd); /* if this isn't the first buffer
                                     link the previous buffer with the new one */
            /* fill up a data link xmit buffer from async transmit fifo */
            for ( ; buf_cnt < TBUF_SIZE && frame_cnt < MAX_FRAME_SIZE;
                                        buf_cnt++, pbuf++, frame_cnt++) {
                if (frame_cnt != 0 && send_flag)
                    break;

                while (t_buf_stat == EMPTY);     /* wait until fifo has data */
                if ((c = *pbuf = Fifo_T_Out()) == CR) {
                    ++buf_cnt ; ++pbuf; ++frame_cnt;
                    break;
                }
            }
            if (c == CR || buf_cnt < TBUF_SIZE || send_flag) { /* last buffer in list */
                ptbd->act_cnt = buf_cnt | EOFBIT;
                send_flag = FALSE;
                break;
            }
        }
        while(!Send_Frame(begin_ptbd, &Dest_Addr[0])); /* keep trying until
                                                          successful */
    }
```

231421-61

```
/PCO/USR/CHUCK/CSRC/UAP.C

    Disable_Uart_Int();
    Disable_Timer_Int();
    monitor_flag = TRUE;
}

struct  TBD     *Build_Frame(cnt)
    u_short     cnt;
{
    u_short         buf_cnt, frame_cnt, i;
    struct  TBD     *ptbd, *q, *begin_ptbd;
    char            *pbuf;

    i = 0x20; frame_cnt = 0;

    for ( ; ; q = ptbd) {

        while ((ptbd = Get_Tbd()) == pNULL); /* get a xmit buffer from the
                                                data link */

        pbuf = (char *) ptbd->buff_ptr;  /* point to the buffer */
        buf_cnt = 0;

        if (frame_cnt == 0) { /* if this is the first buffer, add on IEEE 802.2
                                 header information */
            begin_ptbd = ptbd;
            *pbuf++ = dsap;
            *pbuf++ = ssap;
            *pbuf++ = UI;
            buf_cnt = 3;
        }
        else  q->link = Offset(ptbd); /* if this isn't the first buffer
                                 link the previous buffer with the new one */
        /* fill up a data link xmit buffer with ASCII characters */
        for (; buf_cnt < TBUF_SIZE && cnt > 0;
                                 i++, buf_cnt++, pbuf++, cnt--, frame_cnt++) {
            *pbuf = i;
            if (i > 0x7E)
                i = 0x1F;
        }
        if (cnt == 0) { /* last buffer in list */
            ptbd->act_cnt = buf_cnt | EOFBIT;
            break;
        }
    }
    return(begin_ptbd);
}

Monitor_Mode()
{
    u_short     xmit, cnt, i;
    struct  TBD     *Build_Frame(), *ptbd;

    Write(" Do you want this station to transmit? (Y or N) --> ");
    if (Yes())
```

231421-62

```
/PCO/USR/CHUCK/CSRC/UAP.C

        for (xmit = FALSE; xmit == FALSE; ) {
        Write("\n Enter the number of data bytes in the frame --> ");
        cnt = Read_Int();
        if (cnt > 2045)
            Write ("\n Sorry, the number has to be less than 2046!\n");
        else
            xmit = TRUE;
        }

    else xmit = FALSE;

    Write("\n Hit any key to exit Monitor Mode.\n\n");

Write("  # of Good     # of Good     CRC      Alignment     No           Receive\n");
Write("    Frames        Frames     Errors     Errors      Resource     Overrun\n");
Write(" Transmitted     Received                           Errors       Errors\n");

   /* "012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
        xxxxxxxxxx    xxxxxxxxxx    xxxx       xxxx          xxxx          xxxx
          xxxxxxxx      xxxxxxxx
                11            25      33         44            57            71 */

    for (i = 0; i < 79; i++)
        line[i] = 0x20;
    line[79] = CR;
    line[80] = '\0';

    while ((inb(CH_B_CTL) & 1) == 0) {
        for (i = 0; i < 72; i++)
            line[i] = 0x20;
        Write_Long_Int(good_xmit_cnt, 11);
        Write_Long_Int(recv_frame_cnt, 25);
        Write_Short_Int(scb.crc_errs, 33);
        Write_Short_Int(scb.aln_errs, 44);
        Write_Short_Int(scb.rsc_errs, 57);
        Write_Short_Int(scb.ovr_errs, 71);
        Write(&line[0]);
        if (xmit) {
            ptbd = Build_Frame(cnt);
            while(!Send_Frame(ptbd, &Dest_Addr[0]));
        }
    }
    i = Ci();
}

Hs_Xmit_Mode()
{
    struct  TBD  *ptbd;

    Write("\n Hit any key to exit High Speed Transmit Mode.\n\n");

    hs_stat = TRUE;
    EOI_80130_8274;
    Enable_Uart_Int();

    /* Execute this loop until a recv char interrupt happends at Uart */
```

231421-63

/PCO/USR/CHUCK/CSRC/UAP.C

```
    while (hs_stat) {
        while ((ptbd = Get_Tbd()) == pNULL); /* get a xmit buffer from
                                                the data link */
        ptbd->act_cnt |= EOFBIT;          /* set the End Of Frame bit */
        while(!Send_Frame(ptbd, &Dest_Addr[0])); /* Send Frame */
    }

    Disable_Uart_Int();
}

Print_Cnt()
{
    char    ch[11], base, dwidth, width, i;
    unsigned    long    temp;

    if (dhex) {
        dwidth = 8;
        width = 4;
        base = 16;
    }
    else {
        base = 10;
        dwidth = 10;
        width = 5;
    }

    Write("\n\n Good frames transmitted: ");
    for (i = 1; i <= 11 - dwidth; i++)
        Co(SP);
    Int_To_Ascii(good_xmit_cnt, base, ' ', &ch[0], dwidth);
    for (i = dwidth - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Good frames received: ");
    for (i = 1; i <= 15 - dwidth; i++)
        Co(SP);
    Int_To_Ascii(recv_frame_cnt, base, ' ', &ch[0], dwidth);
    for (i = dwidth - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n CRC errors received: ");
    for (i = 1; i <= 15 - width; i++)
        Co(SP);
    temp = scb.crc_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Alignment errors received: ");
    for (i = 1; i <= 10 - width; i++)
        Co(SP);
    temp = scb.aln_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n Out of Resource frames: ");
    for (i = 1; i <= 12 - width; i++)
        Co(SP);
    temp = scb.rsc_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
```

231421-64

/PCD/USR/CHUCK/CBRC/UAP.C

```
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Receiver overrun frames: ");
    for (i = 1; i <= 12 - width; i++)
        Co(SP);
    temp = scb.ovr_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n 82586 Reset: ");
    for (i = 1; i <= 23 - width; i++)
        Co(SP);
    temp = reset_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Transmit underrun frames: ");
    for (i = 1; i <= 11 - width; i++)
        Co(SP);
    temp = underrun_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n Lost CRS: ");
    for (i = 1; i <= 26 - width; i++)
        Co(SP);
    temp = no_crs_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  SQE errors: ");
    for (i = 1; i <= 25 - width; i++)
        Co(SP);
    temp = sqe_err_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n Maximum retry: ");
    for (i = 1; i <= 21 - width; i++)
        Co(SP);
    temp = max_col_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Frames that deferred: ");
    for (i = 1; i <= 15 - dwidth; i++)
        Co(SP);
    Int_To_Ascii(defer_cnt, base, ' ', &ch[0], dwidth);
    for (i = dwidth - 1; i >= 0; i--)
        Co(ch[i]);
}


Print_Help()
{
    Write ("\n\n Commands are:\n\n");
    Write (" T - Terminal Mode                    M - Monitor Mode\n");
```

231421-65

```
/PCO/USR/CHUCK/CSRC/UAP.C

    Write (" X - High Speed Transmit Mode          V - Change Transmit Statistics\n");
    Write (" P - Print All Counters                C - Clear All Counters\n");
    Write (" A - Add a Multicast Address           Z - Delete a Multicast Address\n");
    Write (" S - Change the SSAP Address           D - Change the DSAP Address\n");
    Write (" N - Change Destination Node Address   L - Print All Addresses\n");
    Write (" R - Re-Initialize the Data Link       B - Change the number Base\n");
}

Main()
{
    int     c;

    Init_Uap();
    Print_Help();

    for (;;) {
        Write ("\n\n Enter a command, type H for Help --> ");
        c = Read_Char();
        switch (Lower_Case(c)) {

        case 'h':
            Print_Help();
            break;
        case 'm':
            Monitor_Mode();
            break;
        case 't':
            Terminal_Mode();
            break;
        case 'x':
            Hs_Xmit_Mode();
            break;
        case 'v':
            Write("\n Transmit Statistics are now ");
            if (flags.stat_on == 1)
                Write("on. \n Would you like to change it ? (Y or N) --> ");
            else
                Write("off. \n Would you like to change it ? (Y or N) --> ");
            if (Yes()) {
                if (flags.stat_on == 1)
                    flags.stat_on = 0;
                else flags.stat_on = 1;
            }
            break;
        case 'p':
            Print_Cnt();
            break;
        case 'c':
            Clear_Cnt();
            break;
        case 'a':
            Load_Multicast();
            break;
        case 'z':
            Remove_Multicast();
            break;
        case 's':
```

231421-66

/PCO/USR/CHUCK/CSRC/UAP.C

```
            Delete_Dsap_Address(ssap);
            Load_Lsap();
            break;
        case 'd':
            Read_Addr("\n\nEnter the Destination Node's LSAP in Hex --> ", &dsap, 1);
            break;
        case 'n':
            Read_Addr("\n\nEnter the Address of the Destination Node in Hex --> ",
                                                    &Dest_Addr[0], ADD_LEN);
            break;
        case 'l':
            Print_Addresses();
            break;
        case 'r':
            Software_Reset();
            Init_DataLink();
            Add_Dsap_Address(ssap, Recv_Data_1);
            break;
        case 'b':
            Write("\n The current base is ");
            if (dhex == TRUE)
                Write("Hex.\n Would you like to change it ? (Y or N) --> ");
            else
                Write("Decimal.\n Would you like to change it ? (Y or N) --> ");
            if (Yes()) {
                if (dhex == TRUE)
                    dhex = FALSE;
                else dhex = TRUE;
            }
            break;

        default:
            Write ("\n Unknown command\n");
            break;
        }
    }
}
```

231421-67

```
/PCD/USR/CHUCK/CSRC/ASSY.ASM

name    c_assy_support

stack   segment stack   'stack'
stktop  label   word
stack   ends

DLD_DATA                segment public  'DATA'
extrn   SEGMT_:word                     ; data segment address
DLD_DATA                ends

UAP_DATA        segment public   'DATA'
UAP_DATA        ends

DLD_CODE        segment public  'CODE'
        extrn    Isr_Timeout_:far, Isr_586_:far, Isr7_:far
        extrn    Isr6_:far, Isr5_:far, Isr1_:far
DLD_CODE        ends

UAP_CODE        segment public  'CODE'
        extrn   Isr_Uart_:far, Isr2_:far, Main_:far
UAP_CODE        ends


DG_CODE         segment public  'CODE'

        public   inw_, outw_, init_intv_, enable_, disable_, Build_Ptr_
        public   Offset_, begin, inb_, outb_

arg1    equ     [BP + 6]
arg2    equ     [BP + 8]

        assume  CS:DG_CODE
        assume  DS:DLD_DATA

;+
;       initialization program for the 82586 data link driver
;-

begin:

        sti
        mov     ax, DLD_DATA ;get base of dgroup and
        mov     SEGMT_, ax      ;pass the segment value to the c program
        mov     ds, ax
        call Main_              ;go to the c program
        hlt


inb_    proc    far
        push    BP
        mov     BP, SP
        push    DX
        mov     DX, arg1
        in      AL, DX
        pop     DX
        mov     SP, BP
```

231421-68

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

        pop     BP
        ret
inb_    endp

outb_   proc    far
        push    BP
        mov     BP, SP
        push    DX
        push    AX
        mov     DX, arg1
        mov     AX, arg2
        out     DX, AL
        pop     AX
        pop     DX
        mov     SP, BP
        pop     BP
        ret
outb_   endp

inw_    proc    far
        push    BP
        mov     BP, SP
        push    DX
        mov     DX, arg1
        in      AX, DX
        pop     DX
        mov     SP, BP
        pop     BP
        ret
inw_    endp

outw_   proc    far
        push    BP
        mov     BP, SP
        push    DX
        push    AX
        mov     DX, arg1
        mov     AX, arg2
        out     DX, AX
        pop     AX
        pop     DX
        mov     SP, BP
        pop     BP
        ret
outw_   endp

Build_Ptr_      proc    far
        push    BP
        mov     BP, SP
        mov     DX, DLD_DATA
        mov     AX, arg1
        mov     SP, BP
        pop     BP
        ret
Build_Ptr_      endp

Offset_ proc    far
```

231421-69

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

        push    BP
        mov     BP, SP
        mov     AX, arg1
        mov     SP, BP
        pop     BP
        ret
Offset_ endp

serve_int_isr   proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr_586_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int_isr   endp

serve_int_8274  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, UAP_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr_Uart_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
```

231421-70

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

        pop     CX
        pop     BX
        pop     AX
        iret
serve_int_8274  endp

serve_int_timeout       proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr_Timeout_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int_timeout       endp

serve_int7_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr7_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
```

231421-71

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

        iret
serve_int7_isr  endp

serve_int6_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr6_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int6_isr  endp

serve_int5_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr5_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int5_isr  endp
```

231421-72

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

serve_int2_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, UAP_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr2_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int2_isr  endp

serve_int1_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr1_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int1_isr  endp


enable_ proc    far
        sti
```

231421-73

```
/PCD/USR/CHUCK/CSRC/ASSY.ASM


        ret
enable_ endp

disable_        proc    far
        cli
        ret
disable_        endp

init_intv_      proc    far
        push    DS
        push    AX

        xor     AX, AX
        mov     DS, AX

        ; Interrupt types for the 186/51 COMMputer

        mov     DS:word ptr  80h, offset serve_int_8274         ; int 0
        mov     DS:word ptr  82h, DG_CODE
        mov     DS:word ptr  84h, offset serve_int1_isr         ; int 1
        mov     DS:word ptr  86h, DG_CODE
        mov     DS:word ptr  88h, offset serve_int2_isr         ; int 2
        mov     DS:word ptr  8Ah, DG_CODE
        mov     DS:word ptr  8Ch, offset serve_int_isr          ; int 3
        mov     DS:word ptr  8Eh, DG_CODE
        mov     DS:word ptr  90h, offset serve_int_timeout      ; int 4
        mov     DS:word ptr  92h, DG_CODE
        mov     DS:word ptr  94h, offset serve_int5_isr         ; int 5
        mov     DS:word ptr  96h, DG_CODE
        mov     DS:word ptr  98h, offset serve_int6_isr         ; int 6
        mov     DS:word ptr  9Ah, DG_CODE
        mov     DS:word ptr  9Ch, offset serve_int7_isr         ; int 7
        mov     DS:word ptr  9Eh, DG_CODE


        pop     AX
        pop     DS
        ret

init_intv_      endp

DG_CODE ends
                end             begin, ds:dld_data, ss:stack:stktop
```

231421-74

APPLICATION NOTE

AP-236

November 1986

Implementing StarLAN with the Intel 82588

ADI GOLBERT
DATA COMMUNICATIONS OPERATION

SHARAD GANDHI
FIELD APPLICATIONS-EUROPE

Order Number: 231422-003

1.0 INTRODUCTION

Personal computers have become the most prolific workstation in the office, serving a wide range of needs such as word processing, spreadsheets, and data bases. The need to interconnect PCs in a local environment has clearly emerged, for purposes such as the sharing of file, print, and communication servers; downline loading of files and application programs; electronic mail; etc. Proliferation of the PC makes it the workstation of choice for accessing the corporate mainframe/s; this function can be performed much more efficiently and economically when clusters of PCs are already interconnected through Local Area Networks (LANs). According to market surveys, the installed base of PCs in business environments reached about 10 million units year-end '85, with only a small fraction connected via LANs. The installed base is expected to double by 1990. There is clearly a great need for locally interconnecting these machines; furthermore, end users expect interconnectability across vendors. Thus, there is an urgent need for industry standards to promote cost effective PC LANs.

A large number of proprietary PC LANs have become available for the office environment over the past several years. Many of these suffer from high installed cost, technical deficiencies, non-conformance to industry standards, and general lack of industry backing. StarLAN, in Intel's opinion, is one of the few networks which will emerge as a standard. It utilizes a proven network access method, it is implemented with proven VLSI components; it is cost effective, easily installable and reconfigurable; it is technically competent; and it enjoys the backing of a large cross section of the industry which is collaborating to develop a standard (IEEE 802.3, type 1BASE5).

1.1 StarLAN

StarLAN is a 1 Mb/s network based on the CSMA/CD access method (Carrier Sense, Multiple Access with Collision Detection). It works over standard, unshielded, twisted pair telephone wiring. Typically, the wiring connects each desk to a wiring closet in a star topology (from which the IEEE Task Force working on the standard derived the name StarLAN in 1984). In fact, telephone and StarLAN wiring can coexist in the same twisted pair bundle connecting a desk to the wiring closet. Abundant quantities of unused phone wiring exist in most office environments, particularly in the U.S. The StarLAN concept of wiring and networking concepts was originated by AT&T Information Systems.

1.2 The 82588

The 82588 is a single-chip LAN controller designed for CSMA/CD networks. It integrates in one chip all functions needed for such networks. Besides inplementing the standard CSMA/CD functions like framing, deferring, backing off and retrying on collisions, transmitting and receiving frames, it performs data encoding and decoding in Manshester or NRZI format, carrier sensing and collision detection, all up to a speed of 2 Mb/s (independent of the chosen encoding scheme). These functions make it an optimum controller for a StarLAN node. The 82588 has a very conventional microcomputer bus interface, easing the job of interfacing it to any processor.

1.3 Organization of the Application Note

This application note has two objectives. One is to describe StarLAN in practical terms to prospective implementers. The other is to illustrate designing with 82588, particularly as related to StarLAN which is expected to emerge as its largest application area.

Section 2 of this Application Note describes the StarLAN network, its basic components, collision detection, signal propagation and network parameters. Sections 3 and 4 describe the 82588 LAN controller and its role in the StarLAN network. Section 5 goes into the details of designing a StarLAN node for the IBM PC. Section 6 describes the design of the HUB. Both these designs have been implemented and operated in an actual StarLAN environment. Section 7 documents the software used to drive the 82588. It gives the actual procedures used to do operations like, configure, transmit and receive frames. It also shows how to use the DMA controller and interrupt controller in the IBM PC and goes into the details of doing I/O on the PC using DOS calls. Appendix A shows oscilloscope traces of the signals at various points in the network. Appendix B describes the multiple point extension (MPE) being considered by IEEE. Appendixes C and D talk about advanced usages of the 82588; working with only one DMA channel, and measuring network delays with the 82588.

1.4 References

For additional information on the 82588, see the Intel Microcommunications Handbook. StarLAN specification are currently available in draft standard form through the IEEE 802.3 Working Group.

2.0 StarLAN

StarLAN is a low cost 1 Mb/s networking solution aimed at office automation applications. It uses a star

topology with the nodes connected in a point-to-point fashion to a central HUB. HUBs can be connected in a hierarchical fashion. Up to 5 levels are supported. The maximum distance between a node and the adjacent HUB or between two adjacent HUBs is 800 ft. (about 250 meters) for 24 gauge wire and 600 ft. (about 200 meters) for 26 gauge wire. Maximum node-to-node distance with one HUB is 0.5 km, hence IEEE 802.3 designation of type 1BASE5. 1 stands for 1 Mb/s and BASE for baseband. (StarLAN doesn't preclude the use of more than 800 ft wiring provided 6.5 dB maximum attenuation is met, and cable propagation delay is no more than 4 bit times).

One of the most attractive features of StarLAN is that it uses telephone grade twisted pair wire for the transmission medium. In fact, existing installed telephone wiring can also be used for StarLAN. Telephone wiring is very economical to buy and install. Although use of telephone wiring is an obvious advantage, for small clusters of nodes, it is possible to work around the use of building wiring.

Factors contributing to low cost are:

1) Use of telephone grade, unshielded, 24 or 26 gauge twisted pair wire transmission media.
2) Installed base of redundant telephone wiring in most buildings.
3) Buildings are designed for star topology wiring. They have conduits leading to a central location.
4) Availability of low cost VLSI LAN controllers like the 82588 for low cost applications and the 82586 for high performance applications.
5) Off-the-shelf, Low cost RS-422, RS-485 drivers/receivers compatible with the StarLAN analog interface requirements.

2.1 StarLAN Topology

StarLAN, as the name suggests, uses a star topology. The nodes are at the extremities of a star and the central point is called a HUB. There can be more than one HUB in a network. The HUBs are connected in a hierarchical fashion resembling an inverted tree, as shown in Figure 1, where nodes are shown as PCs. The HUB at the base (at level 3) of the tree is called the Header Hub (HHUB) and others are called Intermediate HUBs (IHUB). It will become apparent, later in this section, that topologically, this entire network of nodes and HUBs is equivalent to one where all the nodes are connected to a single HUB. Also StarLAN doesn't limit the number of nodes or HUBS at any given level.

2.1.1 TELEPHONE NETWORK

StarLAN is structured to run parallel to the telephone network in a building. The telephone network has, in fact, exactly the same star topology as StarLAN. Let us now examine how the telephone system is typically laid out in a building in the USA. Figure 2 shows how a typical building is wired for telephones. 24 gauge unshielded twisted pair wires emanate from a Wiring Closet. The wires are in bundles of 25 or 50 pairs. The bundle is called D inside wiring (DIW). The wires in these cables end up at modular telephone jacks in the wall. The telephone set is either connected directly to

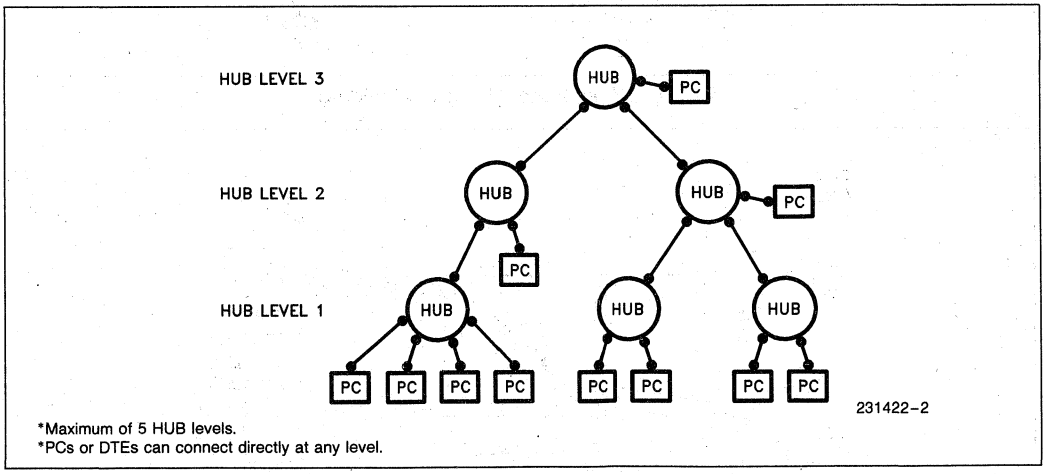


Figure 1. StarLAN Topology

the jack or through an extension cable. Each telephone generally needs one twisted pair for voice and another for auxilliary power. Thus, each modular jack has 2 twisted pairs (4 wires) connected to it. A 25 pair DIW cable can thus be used for up to 12 telephone connections. In most buildings, not all pairs in the bundle are used. Typically, a cable is used for only 4 to 8 telephone connections. This practice is followed by telephone companies because it is cheaper to install extra wires initially, rather than retrofitting to expand the existing number of connections. As a result, a lot of extra, unused wiring exists in a building. The stretch of cable between the wiring closet and the telephone jack is typically less than 800 ft. (250 meters). In the wiring closet the incoming wires from the telephones are routed to another wiring closet, a PABX or to the central office through an interconnect matrix. Thus, the wiring closet is a concentration point in the telephone network. There is also a redundancy of wires between the wiring closets.

2.1.2 StarLAN AND THE TELEPHONE NETWORK

StarLAN does not have to run on building wiring, but the fact that it can significantly adds to its attractiveness. Figure 3 shows how StarLAN piggybacks on telephone wiring. Each node needs two twisted pair wires to connect to the HUB. The unused wires in the 25 pair DIW cables provide an electrical path to the wiring closet, where the HUB is located. Note that the telephone and StarLAN are electrically isolated. They only use the wires in the same bundle cable to connect to the wiring closet. Within the wiring closet, StarLAN wires connect to a HUB and telephone wires are routed to a different path. Similar cable sharing can occur in connecting HUBs to one another. See Figure 4 for a typical office wired for StarLAN through telephone wiring.

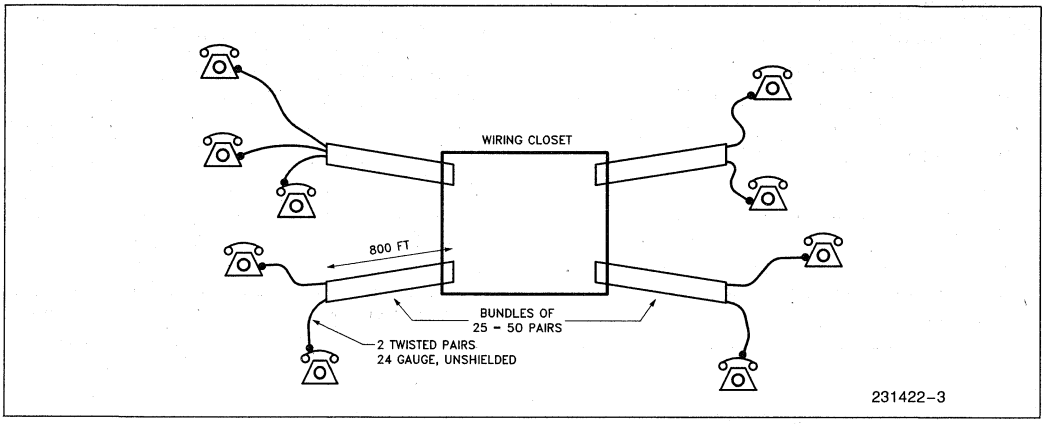


Figure 2. Telephone Wiring in a Building

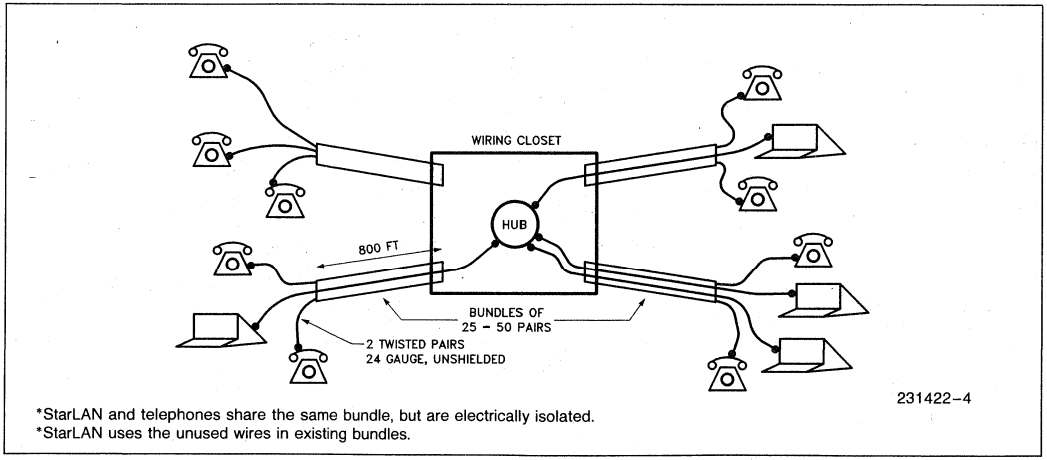


Figure 3. Coexistence of Telephone and StarLAN

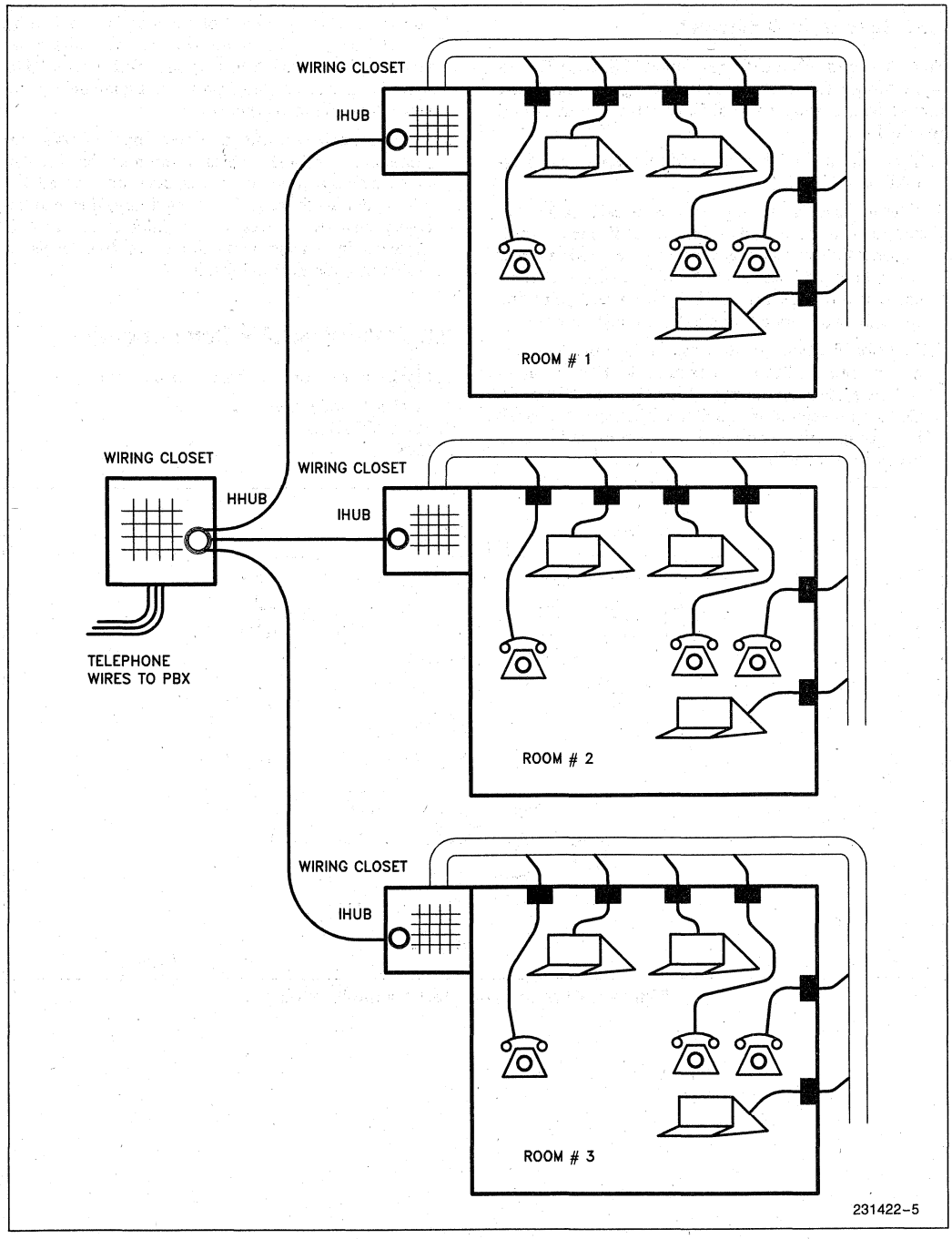


Figure 4. A Typical Office Using Telephone Wiring for StarLAN

2.1.3 StarLAN AND Ethernet

StarLAN and Ethernet are similar CSMA/CD networks. Since Ethernet has existed longer and is better understood, a comparison of Ethernet with StarLAN is worthwhile.

1. The data rate of Ethernet is 10Mb/s and that of StarLAN is 1 Mb/s.
2. Ethernet uses a bus topology with each node connected to a coaxial cable bus via a 50 meter transceiver cable containing four shielded twisted pair wires. StarLAN uses a star topology, with each node connected to a central HUB by a point to point link through two pairs of unshielded twisted pair wires.
3. Collision detection in Ethernet is done by the transceiver connected to the coaxial cable. Electrically, it is done by sensing the energy level on the coax cable. Collision detection in StarLAN is done in the HUB by sensing activity on more than one input line connected to the HUB.
4. In Ethernet, the presence of collision is signalled by the transceiver to the node by a special collision detect signal. In StarLAN, it is signalled by the HUB using a special collision presence signal on the receive data line to the node.
5. Ethernet cable segments are interconnected using repeaters in a non-hierarchical fashion so that the distance between any two nodes does not exceed 2.8 kilometers. In StarLAN, the maximum distance between any two nodes is 2.5 kilometers. This is achieved by wiring a maximum of five levels of HUBs in a hierarchical fashion.

2.2 Basic StarLAN Components

A StarLAN network has three basic components:

1. StarLAN node interface
2. StarLAN HUB
3. Cable

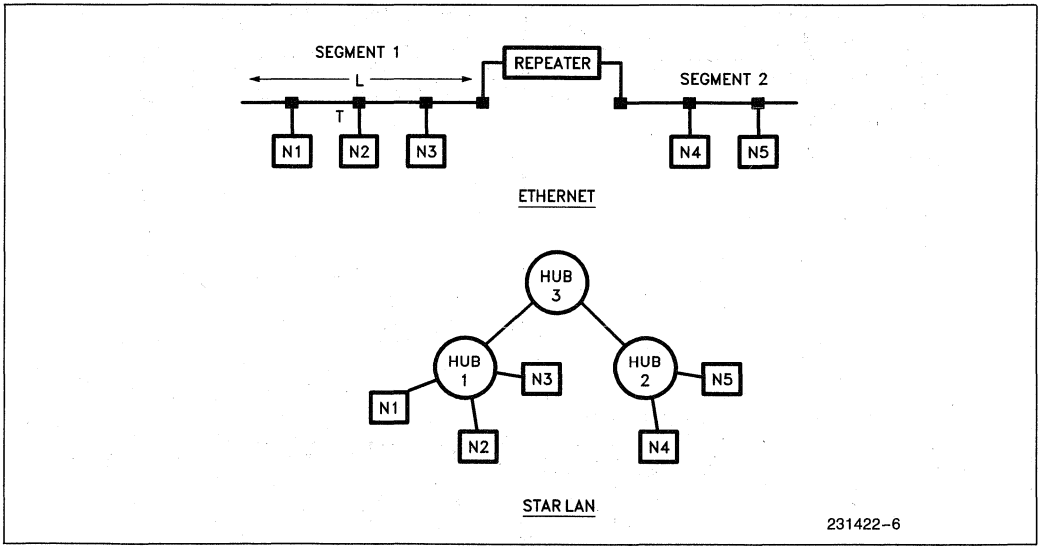


Figure 5. Ethernet and StarLAN Similarities

2.2.1 A StarLAN NODE INTERFACE

Figure 6 shows a typical StarLAN node interface. It interfaces to a processor on the system side. The processor runs the networking software. The heart of the node interface is the LAN controller which does the job of receiving and transmitting the frames in adherence to the IEEE 802.3 standard protocol. It maintains all the timings—like the slot time, interframe spacing etc.—required by the network. It performs the functions of framing, deferring, backing-off, collision detection which are necessary in a CSMA/CD network. It also does Manchester encoding of data to be transmitted and clock separation—or decoding—of the Manchester encoded data that is received. These signals before going to the unshielded twist pair wire, may undergo pulse shaping (optional) pulse shaping basically slows down the fall/rise times of the signal. The purpose of that is to diminish the effects of cross-talk and radiation on adjacent pairs sharing the same bundle (digital voice, T1 trunks, etc). The shaped signal is sent on to the twisted pair wire through a pulse transformer for DC isolation. The signals on the wire are thus differential, DC isolated from the node and almost sinusoidal (due to shaping and the capacitance of the wire).

NOTE:
Work done by the IEEE 802.3 committee has shown that no slew rate control on the drivers is required. Shaping by the transformer and the cable is sufficient to avoid excessive EMI radiation and crosstalk.

The squelch circuit prevents idle line noise from affecting the receiver circuits in the LAN controller. The squelch circuit has a 600 mv threshold for that purpose. Also as part of the squelch circuitry an envelope detector is implemented. Its purpose is to generate an envelope of the transitions of the RXD line. Its output serve as a carrier sense signal. The differential signal from the HUB is received using a zero-crossing RS-422 receiver. Output of the receiver, qualified by the squelch circuit, is fed to the RxD pin of the LAN controller. The RxD signal provides three kinds of information:

1) Normal received data, when receiving the frame.
2) Collision information in the form of the collision presence signal from the HUB.
3) Carrier sense information, indicating the beginning and the end of frame. This is useful during transmit and receive operations.

2.2.2 StarLAN HUB

HUB is the point of concentration in StarLAN. All the nodes transmit to the HUB and receive from the HUB. Figure 7 shows an abstract representation of the HUB. It has an upstream and a downstream signal processing unit. The upstream unit has N signal inputs and 1 signal output. And the downstream unit has 1 input and N output signals. The inputs to the upstream unit come from the nodes or from the intermediate HUBs (IHUBs) and its output goes to a higher level HUB. The downstream unit is connected the other way around; input from an upper level HUB and the outputs to nodes or lower level IHUBs. Physically each input and output consist of one twisted pair wire carrying a differential signal. The downstream unit essentially just re-times the signal received at the input, and sends it to all its outputs. The functions performed by the upstream unit are:

1. Collision detection
2. Collision Presence signal generation
3. Signal Retiming
4. Jabber Function
5. Start of Idle protection timer

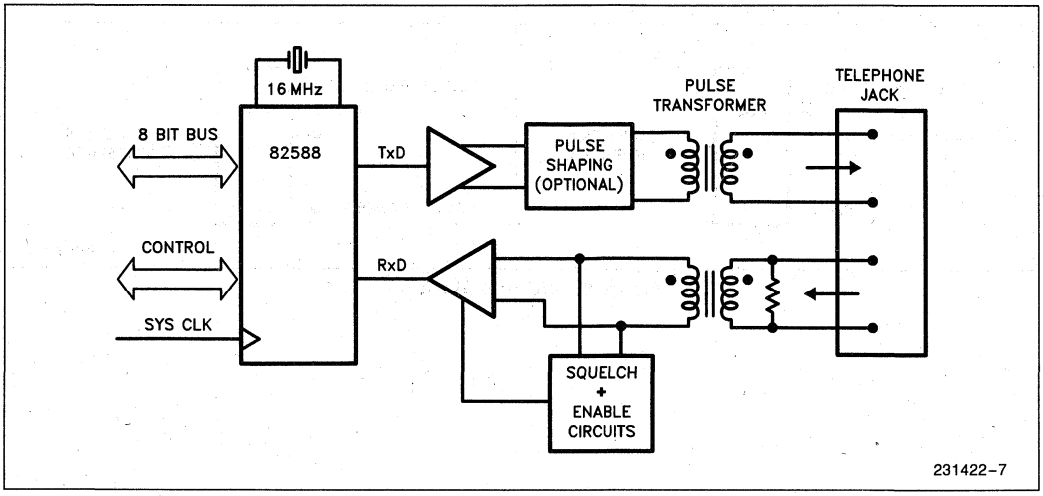


Figure 6. 82588 Based StarLAN Node

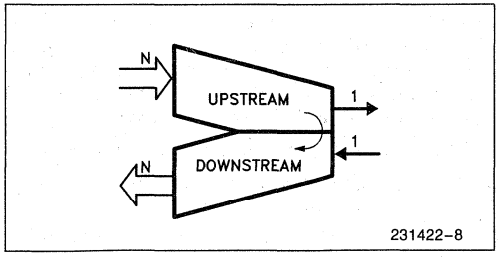


Figure 7. A StarLAN HUB

The collision detection in the HUB is done by sensing the activity on the inputs. If there is activity (or transitions) on more than one input, it is assumed that more than one node is transmitting. This is a collision. If a collision is detected, a special signal called the Collision Presence Signal is generated. This signal is generated and sent out as long as activity is sensed on any of the input lines. This signal is interpreted by every node as an occurrence of collision. If there is activity only on one input, that signal is re-timed—or cleaned up of any accumulated jitter—and sent out. Figure 8 shows the input to output relations of the HUB as a black box.

If a node transmits for too long the HUB exercises a Jabber function to disable the node from interfering with traffic from other nodes. There are two timers in the HUB associated with this function and their operation is described in section 6.

The last function implemented by the HUB is the start of Idle protection timer. During the end of reception, the HUB will see a long undershoot at its input port. This undershoot is a consequence of the transformer discharging accumulated charge during the 2 microseconds of high of the idle pattern. The HUB should implement a protection mechanism to avoid the undesirable effects of that undershoot.

Figure 9 shows a block diagram of the HUB. A switch position determines whether the HUB is an IHUB or a HHUB (Header HUB). If the HUB is an IHUB, the switch decouples the upstream and the downstream units. HHUB is the highest level HUB; it has no place to send its output signal, so it returns its output signal (through the switch) to the outputs of the downstream unit. There is one and only one HHUB in a StarLAN network and it is always at the base of the tree. The returned signal eventually reaches every node in the network through the intermediate nodes (if any). StarLAN specifications do not put any restrictions on the number of IHUBS at any level or on number of inputs to any HUB. The number of inputs per HUB are typically 6 to 12 and is dictated by the typical size of clusters in a given networking environment.

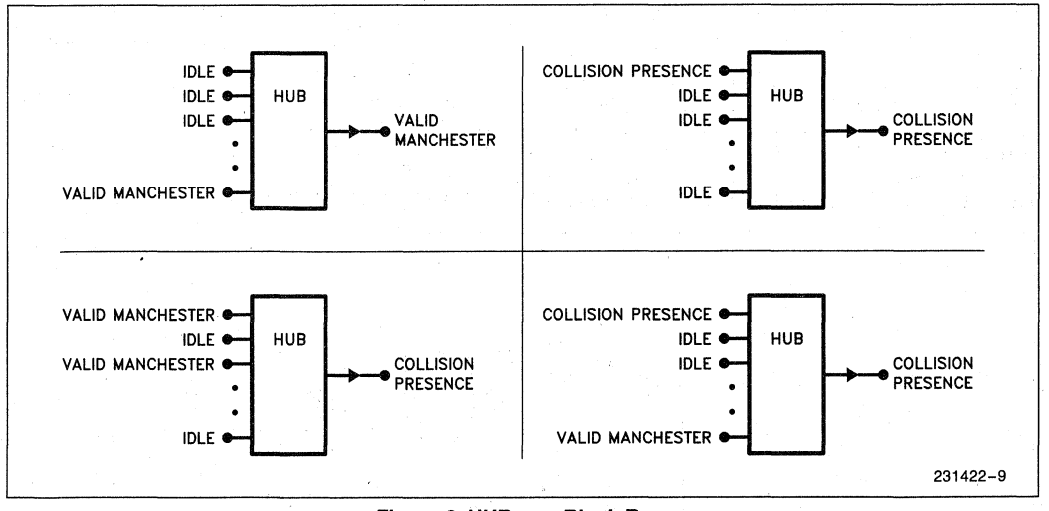


Figure 8. HUB as a Black Box

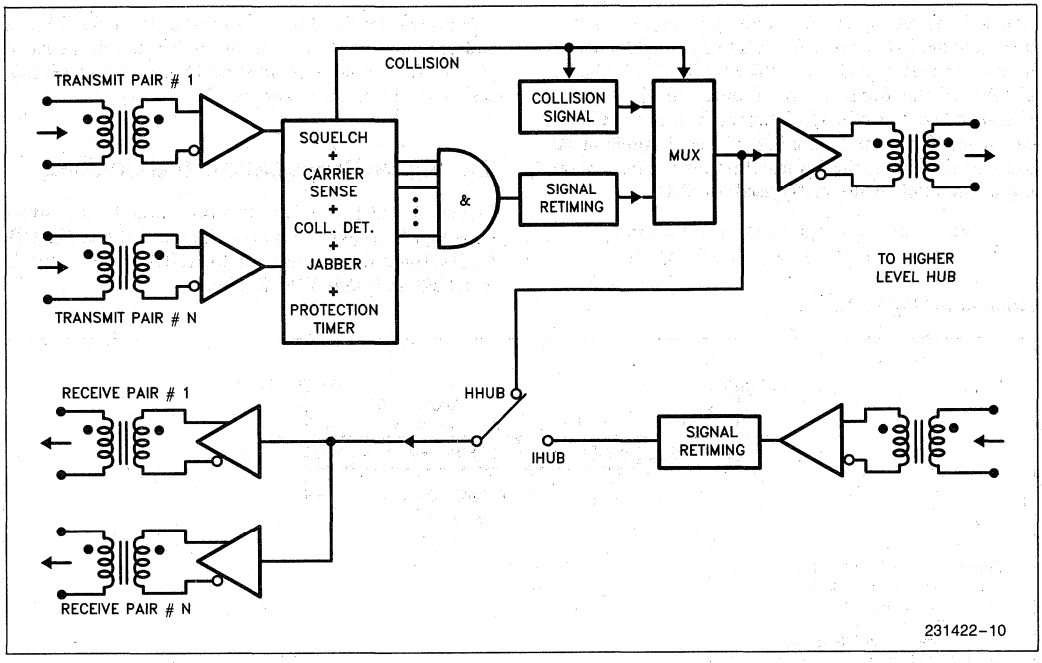


Figure 9. StarLAN HUB Block Diagram

2.2.3 StarLAN CABLE

Unshielded telephone grade twisted pair wires are used to connect a node to a HUB or to connect two HUBs. This is one of the cheapest types of wire and an important factor in bringing down the cost of StarLAN.

Although the 24 gauge wire is used for long stretches, the actual connection between the node and the telephone jack in the wall is done using extension cable, just like connecting a telephone to a jack. For very short StarLAN configurations, where all the nodes and the HUB are in the same room, the extension cable with plugs at both ends may itself be sufficient for all the wiring. (Extension cables must be of the twisted pair kind, no flat cables are allowed).

The telephone twisted pair wire of 24 gauge has the following characteristics:

Attenuation	: 42.55 db/mile @ 1 MHz
DC Resistance	: 823.69 Ω/mile
Inductance	: 0.84 mH/mile
Capacitance	: 0.1 μF/mile
Impedance	: 92.6Ω, −4 degrees @ 1 MHz

Experiments have shown that the sharing of the telephone cable with other voice and data services does not cause any mutual harm due to cross-talk and radiation, provided every service meets the FCC limits.

Although it is outside the scope of the IEEE 802.3 1BASE5 standard, there is considerable interest in using fiber optics and coaxial cable for node to HUB or HUB to HUB links especially in noisy and factory environments. Both these types of cables are particularly suited for point-to-point connections. Even mixing of different types of cables is possible (this kind of environments are not precluded).

NOTE:
StarLAN IEEE 802.3 1BASE5 draft calls for a maximum attenuation of 6.5 dB between the transmitter and the corresponding receiver at all frequencies between 500 KHz to 1 MHz. Also the maximum allowed cable propagation delay is 4 microseconds.

2.3 Framing

Figure 10 shows the format of a 802.3 frame. The beginning of the frame is marked by the carrier going active and the end marked by carrier going inactive. The preamble has a 56 bit sequence of 101010 . . . . ending in a 0. This is followed by 8 bits of start of frame delimiter (sfd) − 10101011. These bits are transmitted with the MSB (leftmost bit) transmitted first. Source and destination fields are 6 bytes long. The first byte is the least significant byte. These fields are transmitted with LSB first. The length field is 2 bytes long and gives the length of data in the Information field. The entire information field is a minimum of 46 bytes and a maximum of 1500 bytes. If the data content of the Informa-

tion field is less than 46, padding bytes are used to make the field 46 bytes long. The Length field indicates how much real data is in the Information field. The last 32 bits of the frame is the Frame Check Sequence (FCS) and contains the CRC for the frame. The CRC is calculated from the beginning of the destination address to the end of the Information field. The generating polynomial (Autodin II) used for CRC is:

$$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X + 1$$

No need for Figure N.

The frames can be directed to a specific node (LSB of address must be 0), to a group of nodes (multicast or group—LSB of address must be 1) or all nodes (broadcast—all address bits must be 1).

2.4 Signal Propagation and Collision

Figure 11 will be used to illustrate three typical situations in a StarLAN with two IHUBs and one HHUB. Nodes A and B are connected to HUB1, nodes C and D to HUB2 and node E to HUB3.

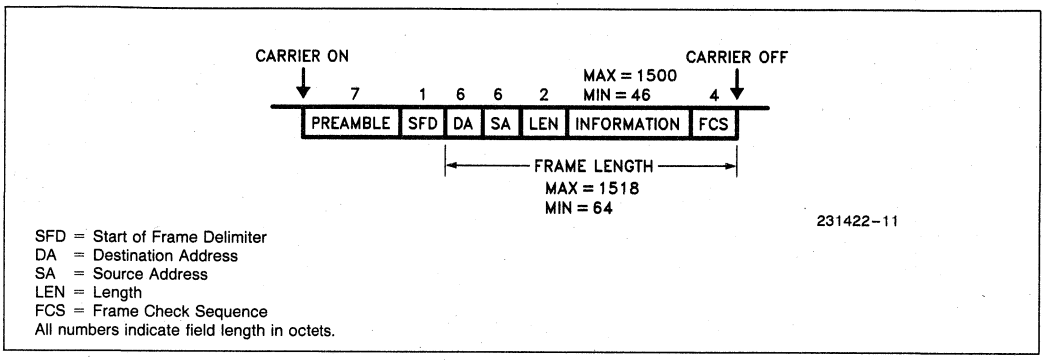
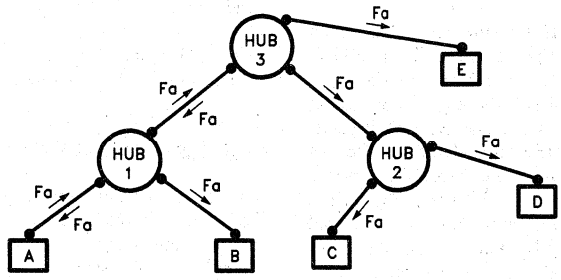
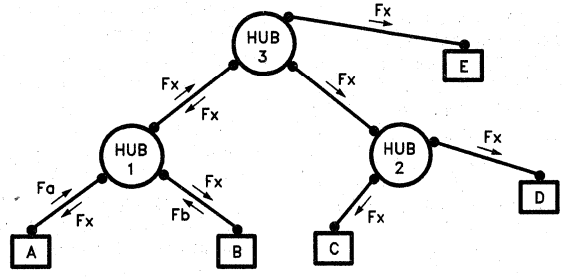


Figure 10. Framing



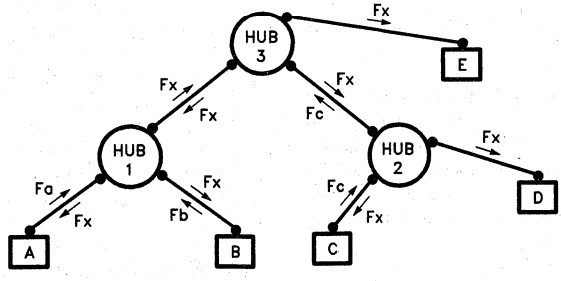
231422-12

Situation #1. A Transmitting



231422-13

Situation #2. A & B Transmitting



231422-14

Situation #3. A, B & C Transmitting

HUB1, HUB2 are IHUBs
HUB3 is the HHUB

Fa, Fb, Fc—Frames from nodes A, B & C
Fx—Collision Presence Signal

Figure 11. Signal Propagation and Collisions

2.4.1 Situation #1

Whenever node A transmits a frame Fa, it will reach HUB1. If node B is silent, there is no collision. HUB1 will send Fa to HUB3 after re-timing the signal. If nodes C, D and E are also silent, there is no collision at HUB2 or HUB3. Since HUB3 is the HHUB, it sends the frame Fa to HUB1, HUB2 and to node E after re-timing. HUB1 and HUB2 send the frame Fa to nodes A, B and C, D. Thus, Fa reaches all the nodes on the network including the originator node A. If the signal received by node A is a valid Manchester signal and not the Collision Presence Signal (CPS) for the entire duration of the slot time, then the node A assumes that it was a successful transmission.

2.4.2 Situation #2

If both nodes A and B were to transmit, HUB1 will detect it as a collision and will send signal Fx (the Collision Presence Signal) to the HUB3—Note that HUB1 does not send Fx to nodes A and B yet. HUB 3 receives a signal from HUB1 but nothing from node E or HUB2, thus it does not detect the situation as a collision and simply re-times the signal Fx and sends it to node E, HUB2 and HUB1. Fx ultimately reach all the nodes. Nodes A and B detect this signal as CPS and call it a collision.

2.4.3 Situation #3

In addition to nodes A and B, if node C were also to transmit, the situation at HUB1 will be the same as in situation #2. HUB2 will propagate Fc from C towards HUB3. HUB3 now sees two of its inputs active and hence generates its own Fx signal and sends it towards each node.

These situations should also illustrate the point made earlier in the chapter that, the StarLAN network, with nodes connected to multiple HUBs is, logically, equivalent to all the nodes connected to a single HUB (Yet there are some differences between stations connected at different HUB levels, those are due to different delays to the header hub HHUB).

2.5 StarLAN System and Network Parameters

Preamble length (incl. sfd) ..... 64 bits
Address length ..... 6 bytes
FCS length CRC (Autodin II) ..... 32 bits
Maximum frame length ..... 1518 bytes
Minimum frame length ..... 64 bytes
Slot time ..... 512 bit times
Interframe spacing ..... 96 bit times
Minimum jam timing ..... 32 bit times
Maximum number of collisions 16
Backoff limit 10
Backoff method ..... Truncated binary exponential
Encoding ..... Manchester

Clock tolerance ..... ±0.01% (100 ppm)
Maximum jitter per segment ..... ±62.5 ns

3.0 LAN CONTROLLER FOR StarLAN

One of the attractive features of StarLAN is the availability of the 82588, a VLSI LAN controller, designed to meet the needs of a StarLAN node. The main requirements of a StarLAN node controller are:

1. IEEE 802.3 compatible CSMA/CD controller.
2. Configurable to StarLAN network and system parameters.
3. Generation of all necessary clocks and timings.
4. Manchester data encoding and decoding.
5. Detection of the Collision Presence Signal.
6. Carrier Sensing.
7. Squelch or bad signal filtering.
8. Fast and easy interface to the processor.

82588 performs all these functions in silicon, providing a minimal hardware interface between the system processor and the StarLAN physical link. It also reduces the software needed to run the node, since a lot of functions, like deferring, back off, counting the number of collisions etc., are done in silicon.

3.1 IEEE 802.3 Compatibility

The CSMA/CD control unit on the 82588 performs the functions of deferring, maintaining the Interframe Space (IFS) timing, reacting to collision by generating a jam pattern, calculating the back-off time based on the number of collisions and a random number, decoding the address of the incoming frame, discarding a frame that is too short, etc. All these are performed by the 82588 in accordance to the IEEE 802.3 standards. For inter-operability of different nodes on the StarLAN network it is very important to have the controllers strictly adhere to the same standards.

3.2 Configurability of the 82588

Almost all the networking parameters are programmable over a wide range. This means that the StarLAN parameters form a subset of the total potential of the 82588. This is a major advantage for networks whose standards are being defined and are in a flux. It is also an advantage when carrying over the experience gained with the component in one network to other applications, with differing parameters (leveraging the design).

The 82588 is initialized or configured to its working environment by the CONFIGURE command. After the execution of this command, the 82588 knows its system and network parameters. A configure block in

memory is loaded into the 82588 by DMA. This block contains all the parameters to be programmed as shown in Figure 12. Following is a partial list of the parameters with the programmable range and the StarLAN value:

Parameter	Range	StarLAN Value
Preamble length	2, 4, 8, 16 bytes	8
Address length	0 to 6 bytes	6
CRC type	16, 32 bit	32
Minimum frame length	6 to 255 bytes	64
Interframe spacing	12 to 255 bit times	96
Slot time	1 to 2047 bit times	512
Number of retries	0 to 15	15
Data encoding	NRZI, Man., Diff. Man.	Manch.
Collision detection	Code viol., Bit comp.	Code Viol.

Beside these, there are many other options available, which may or may not apply to StarLAN:

Data sampling rate of 8 or 16
Operating in Promiscuous mode
Reception of Broadcast frames
Internal loopback operation
External loopback operation
Transmit without CRC
HDLC Framing
:
:

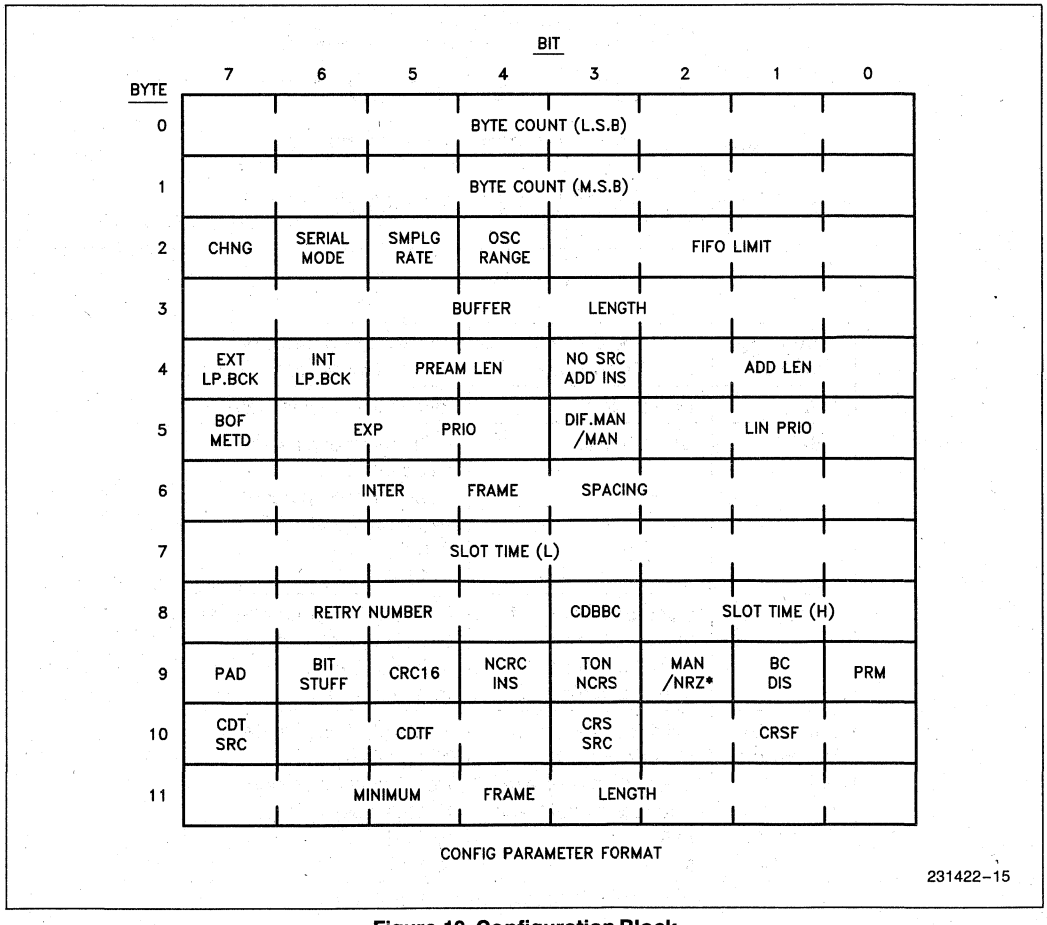


Figure 12. Configuration Block

3.3 Clocks and Timers

The 82588 requires two clocks; one for the operation of the system interface and another for the serial side. Both clocks are totally asynchronous to each other. This permits transmitting and receiving frames at data rates that are virtually independent of the speed at which the system interface operates.

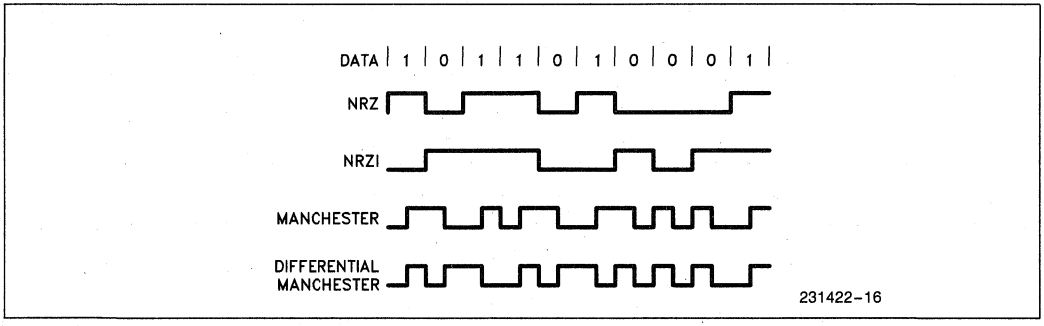
The serial clock can be generated on chip using just an external crystal of a value 8 or 16 times the desired bit rate. An external clock may also be used.

The 82588 has a set of timers to maintain various timings necessary to run the CSMA/CD control unit. These are timings for the Slot time, Interframe spacing time, Back off time, Number of collisions, Minimum frame length, etc. These timers are started and stopped automatically by the 82588.

3.4 Manchester Data Encoding and Decoding

In StarLAN the data transmitted by the node must be encoded in Manchester format. The node should also be able to decode Manchester encoded data when receiving a frame—a process also known as clock recovery. The 82588 does the encoding and decoding of data bits on chip for data rates up to 2 Mb/s.

Besides Manchester, the 82588 can also do encoding and decoding in NRZI and Differential Manchester formats. Figure 13 shows samples of encoding in



Encoding Method	Mid Bit Cell Transitions	Bit Cell Boundary Transitions
NRZ	Do not exist.	Identical to original data.
NRZI	Do not exist.	Exist only if original data bit equals 0. Dependent on present encoded signal level: to 0 if 1 to 1 if 0
Manchester	Exist for every bit of the original data: from 0 to 1 for 1 from 1 to 0 for 0	Exist for consequent equal bits of original data: from 1 to 0 for 1 1 from 0 to 1 for 0 0
Differential Manchester	Exist for every bit of the original data. Dependent on present Encoded signal level: to 0 if 1 to 1 if 0	Exist only if original data bit equals 0. Dependent on present Encoded signal level: to 0 if 1 to 1 if 0

Figure 13. 82588 Data Encoding Rules

these three formats. The main advantage of NRZI over the other two is that NRZI requires half the channel bandwidth, for any given data rate. On the other hand, since the NRZI signal does not have as many transitions as the other two, clock recovery from it is more difficult. The main advantage of Differential Manchester over straight Manchester is that for a signal that is differentially driven (as in RS 422), crossing of the two wires carrying the data does not change the data received at the receiver. In other words, NRZI and Differential Manchester encoding methods are polarity insensitive (Even though NRZI, Differential Manchester are polarity insensitive, the 82588 expects a high level in the RXD line to detect carrier inactive at the end of frames).

3.5 Detection of the Collision Presence Signal

In a StarLAN network, HUB informs the nodes that a collision has occurred by sending the Collision Presence Signal (CPS) to the nodes. The CPS signal is a special signal which contains violations in Manchester encoding. Figure 14 shows the CPS signal. It has a 5 ms period, looking very much like a valid Manchester signal except for missing transitions (or violations) at periodic intervals. When the 82588 decodes this signal, it fails to see mid-cell transitions repeatedly at intervals of 2.5 bit times and hence calls it a code violation. The edges of CPS are marked for illustration as a, b, c, d, . . . l. Let us see how the 82588 interprets the signal if it starts calling the edge 'a' as the mid-cell transition for '1'. Then edge at 'b' is '0'. Now the 82588 expects to see an edge at '*' but since there is none, it is a Manchester code violation. The edge that eventually does occur at 'd' is then used to re-synchronize and, since it is a falling edge, it is taken as a mid-cell transition for '0'. The edge at 'e' is for a '1' and then again there is no edge at '*'. This goes on, with the 82588 flagging code violation and re-synchronizing again every 2.5 bit times. When a transmitting node sees this CPS signal being returned by the HUB (instead of a valid Manchester signal it transmitted), it assumes that a collision occurred. The 82588 has two built-in mechanisms to detect collisions. These mechanisms are very general and can be used for a very broad class of applications to detect collisions in a CSMA/CD network. Using these mechanisms, the 82588 can detect collisions (two or more nodes transmitting simultaneously) by just receiving the collided signal during transmission, even if there was no HUB generating the CPS signal.

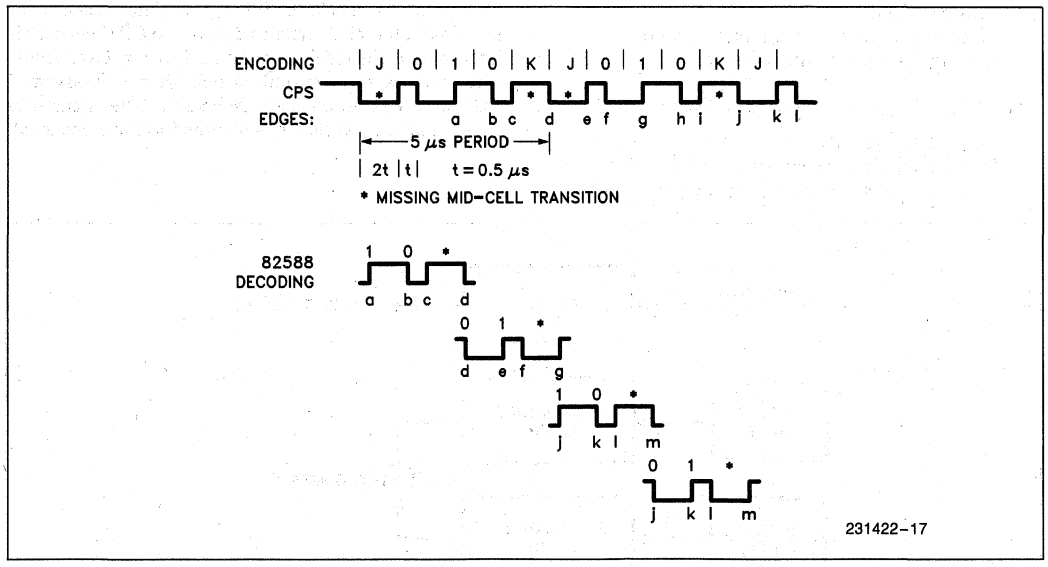


Figure 14. 82588 Decoding the Collision Presence Signal

3.5.1 COLLISION DETECTION BY CODE VIOLATION

If during transmission, the 82588 sees a violation in the encoding (Manchester, NRZI or Differential Manchester) used, then it calls it a collision by aborting the transmission and transmitting a 32 bit jam pattern. The algorithm used to detect collisions, and to do the data decoding, is based on finding the number of sampling clocks between an edge to the next one. Suppose an edge occurred at time 0, the sampling instant of the next edge determines whether it was a collision (C), a long pulse (L)—with a nominal width of 1 bit time—,or a short pulse (S)—nominal width of half a bit time. The following two charts show the decoding and collision detection algorithm for sampling rates of 8 and 16 when using Manchester encoding. The numbers at the bottom of the line indicate sampling instances after the occurrence of the last edge (at 0). The alphabets on the top show what would be inferred by the 82588 if the next edge were to be there.

Sampling rate = 8 (clock is 8x bit rate)

```
C  C  S  S  S  L  L  L  L  L  C  C
|  |  |  |  |  |  |  |  |  |  |  |  |
0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Collision also if:
RxD stays low for 13 samples or more
A mid cell transition is missing

Sampling rate = 16 (clock is 16x bit rate)

```
CCCCCSSSSSCLLLLLLLLCCCC
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
0  2  4  6  8  10 12 14 16 18 20 22 24 26
```

Collision also if:
RxD stays low for 25 samples or more
A mid cell transition is missing

A single instance of code violation can qualify as collision. The 82588 has a parameter called collision detect filter (CDT Filter) that can be configured from 0 to 7. This parameter determines for how many bit times the violation must remain active to be flagged as a collision. For StarLAN CDT Filter must be configured to 0—that is disabled.

3.5.2 COLLISION DETECTION BY SIGNATURE (OR BIT) COMPARISON

This method of collision detection compares a signature of the transmitted data with that of the data received on the RxD pin while transmitting. Figure 15 shows a block diagram of the logic. As the frame is transmitted it flows through the CRC generation logic. A timer, called the Tx slot timer, is started at the same time that the CRC generation starts. When the count in the timer reaches the slot time value, the current value of the CRC generator is latched in as the transmit signature. As the frame is returned back (through the HUB) it flows through the CRC checker. Another timer—Rx slot timer—is started at the same time as the CRC checker starts checking. When this timer reaches the slot time value, the current value of the CRC checker is latched in as the receive signature. If the received signature matches the transmitted one, then it is assumed that there was no collision. Whereas, if the signatures do not match, a collision is assumed to have occurred.

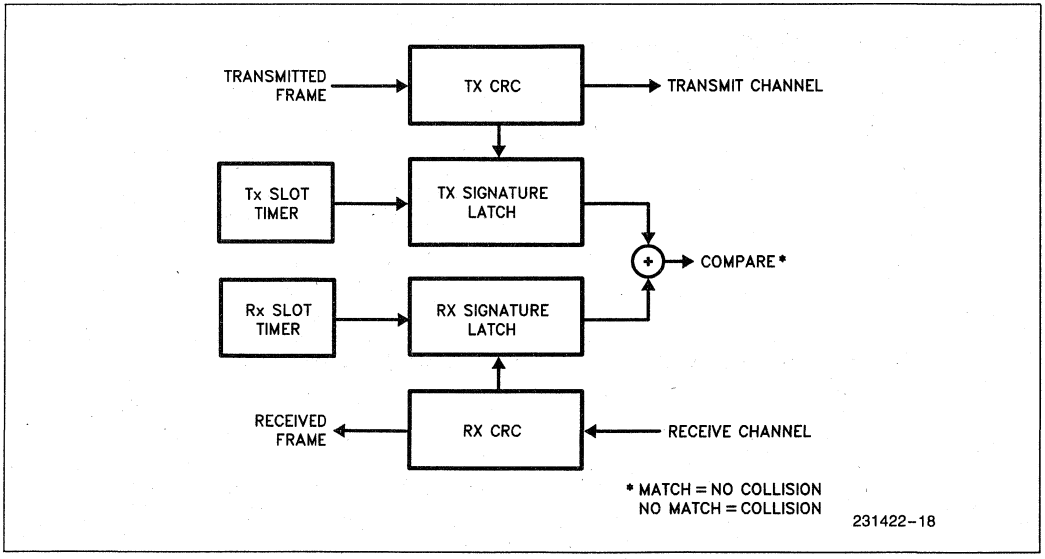


Figure 15. Collision Detection by Signature Comparison

Note that, even if the collision were to occur in the first few bits of the frame, a slot time must elapse before it is detected. In the code violation method, collision is detected within a few bit times. However, since the signature method compares the signatures, which are characteristic of the frame being transmitted, it is more robust. The code violation method can be fooled by returning a signal to the 82588 which is not the same as the transmitted signal but is a valid Manchester signal—like a 1 MHz signal. Both methods can be used simultaneously giving a combination of speed and robustness.

NOTE:
In order to reliably detect a collision using the collision by bit comparison mode, the transmitter must still be transmitting up to the point where the receiver has seen enough bits to complete its signature. Otherwise, the transmitter may be done before the RX signature is completed resulting in an undetected collision. A sufficient condition to avoid this situation is to transmit frames with a minimum length of 1.5 * slot-time (see Figure 16).

3.5.3 ADDITIONAL COLLISION DETECTION MECHANISM

In addition to the collision detection mechanisms described in the preceding sections, the 82588 also flags collision when after starting a transmission any of the following conditions become valid:

a) Half a slot time elapses and the carrier sense of 82588 is not active.

b) Half a slot time + 16 bit times elapse and the opening flag (sfd) is not detected.

c) Carrier sense goes inactive after an opening flag is received with transmitter still active.

These mechanisms add a further robustness to the collision detection mechanism of the 82588. It is also possible to OR an externally generated collision detect signal to the internally generated condition by bit comparison (see Figure 17).

3.6 Carrier Sensing

A StarLAN network is considered to be busy if there are transitions on the cable. Carrier is supposed to be active if there are transitions. Every node controller needs to know when the carrier is active and when not. This is done by the carrier sensing circuitry. On the 82588 this circuit is on chip. It looks at the RxD (receive data) pin and if there are transitions, it turns on an internal carrier sense signal. It turns off the carrier sense signal if RxD remains in idle (high) state for 13/8 bit times. This carrier sense information is used to mark the start of the interframe space time and the back off time. The 82588 also defers transmission when the carrier sense is active.

When operating in the NRZI encoded mode, carrier sense is turned off if RxD pin is in the idle state for 8 bit times or more (see Figure 18).

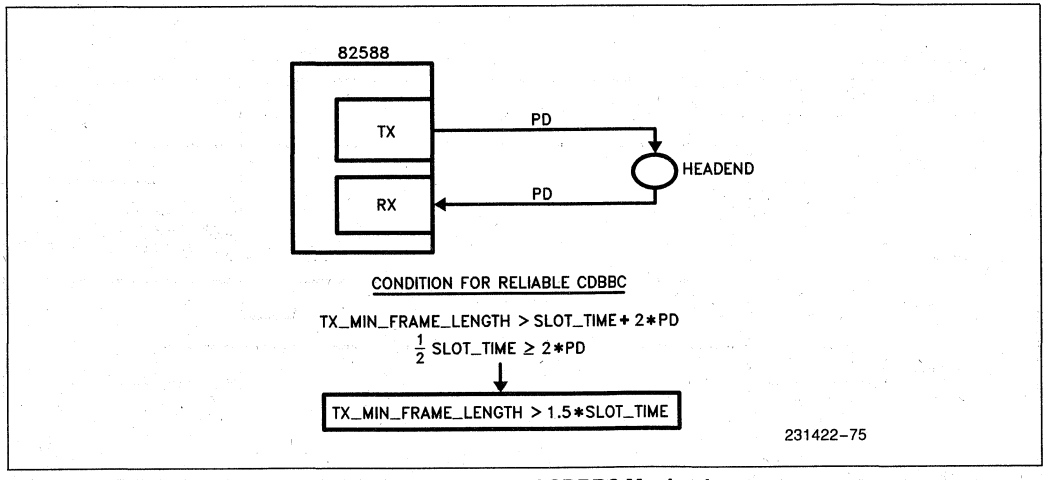


Figure 16. Limitation of CDBBC Mechanism

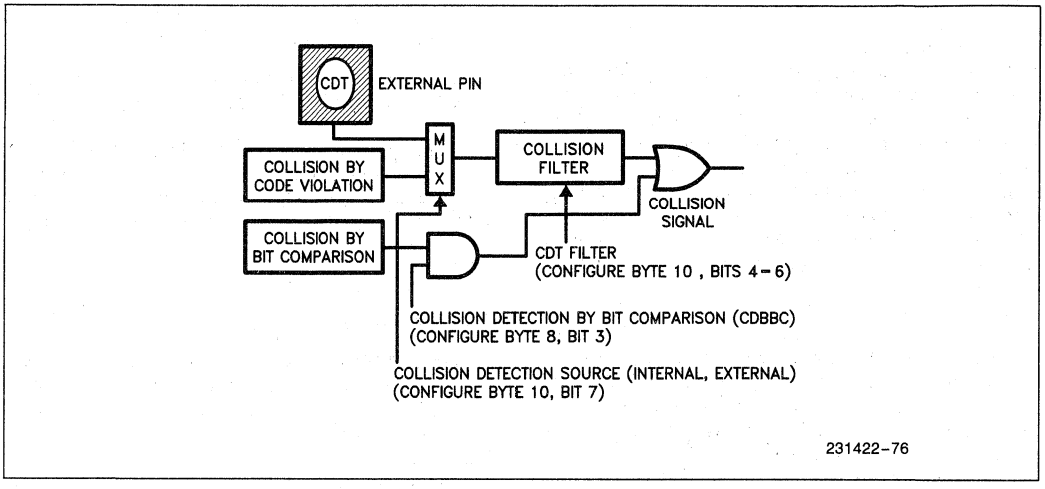


Figure 17. Mode 0, Collision Detection

3.7 Squelching the Input

Squelch circuit is used to filter idle noise on the receiver input. Basically two types of squelch may be used: Voltage and time. Voltage squelch is done to filter out signals whose strength is below a defined voltage threshold (0.6 volts for StarLAN). It prevents idle line noise from disturbing the receive circuits on the controller. The voltage squelch circuit is placed right after the receiving pulse transformer. It enables the input to the RxD pin of the 82588 only when the signal strength is above the threshold.

If the signal received has the proper level but not the proper timing, it should not bother the receiver. This is accomplished by the time squelch circuit on the 82588. Time squelching is essential to weed out spikes, glitches and bad signal especially at the beginning of a frame. The 82588 does not turn on its carrier sense (or receive enable) signal until it receives three consecutive edges, each separated by time periods greater than the fast time clock high time but less than 13/8 bit-times as shown in Figure 18.

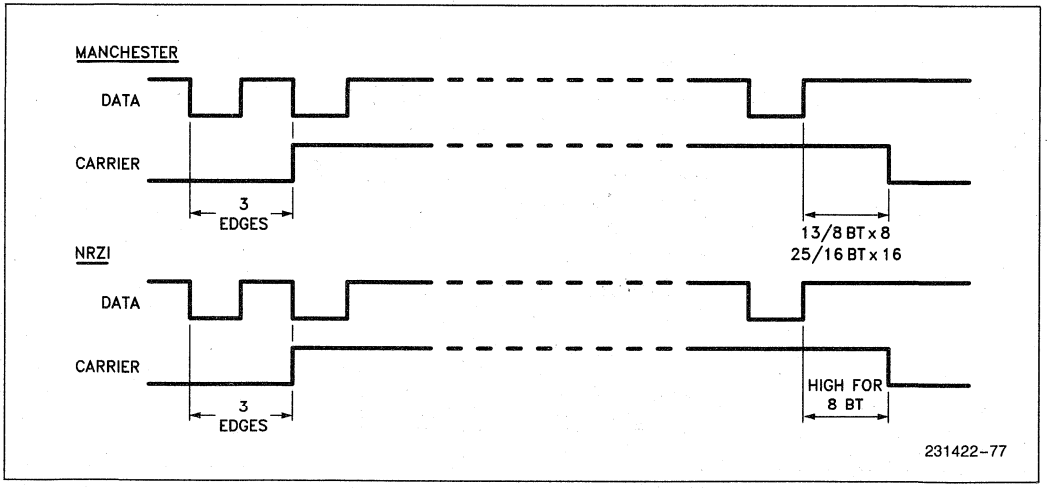


Figure 18. Carrier Sensing

The carrier sense activation can be programmed for a further delay by up to 7 bit times by a configuration parameter called carrier sense filter.

3.8 System Bus Interface

The 82588 has a conventional bus interface making it very easy to interface to any processor bus. Figure 19 shows that it has an 8 bit data bus, read, write, chip select, interrupt and reset pins going to the processor bus. It also needs an external DMA controller for data transfer. A system clock of up to 8 MHz is needed. The read and write access times of the 82588 are very short—95 ns—as shown by Figure 20. This further facilitates interfacing the controller to almost any processor.

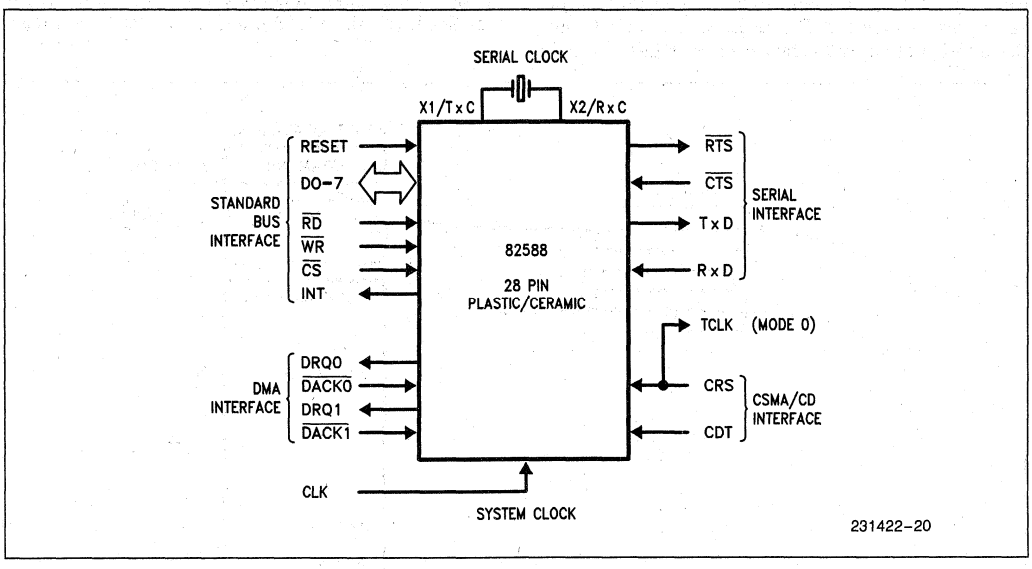


Figure 19. Chip Interface

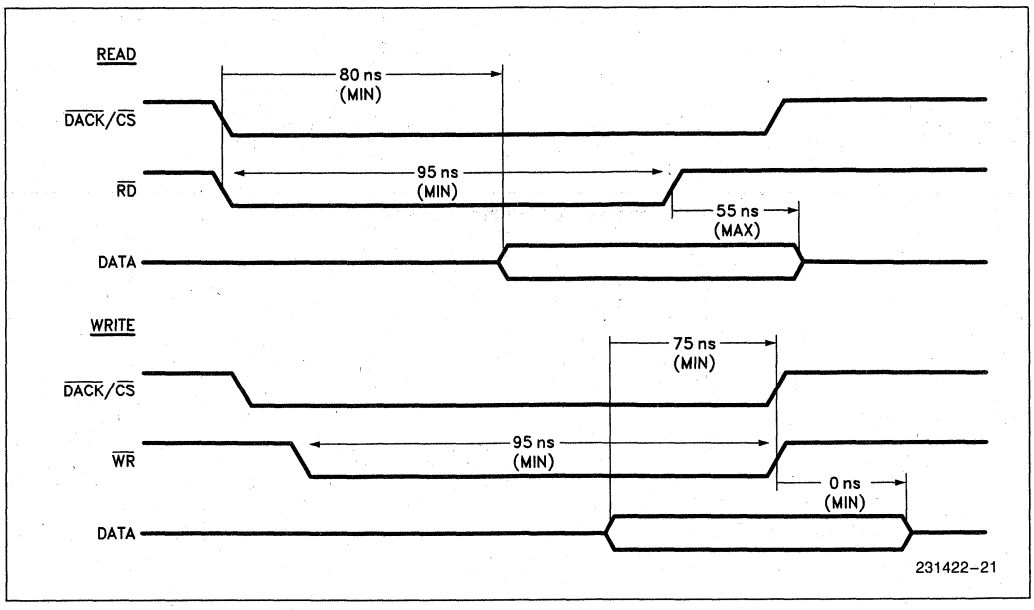


Figure 20. Access Times

The 82588 has over 50 bytes of registers, and most are accessed only indirectly. Figure 21 shows the register access mechanism of the 82588. It has one I/O port and 2 DMA channel ports. These are the windows into the 82588 for the CPU and the DMA controller. An external CPU can write into the Command register and read from the Status registers using I/O instructions and asserting chip select and write or read lines. Although there is just one I/O port and 4 status registers, they can be read out in a round robin fashion through the same port as shown in Figure 22. Other registers like the Configuration, Individual Address registers can be accessed only through DMA. All the internal registers can be dumped into memory by DMA using the Dump command. The execution of some of the commands is described in section 4. See the 82588 Reference Manual for details on these commands.

3.9 Debug and Diagnostic Aids

Besides the standard functions that can be used directly for StarLAN, the 82588 offers many debug and diag-

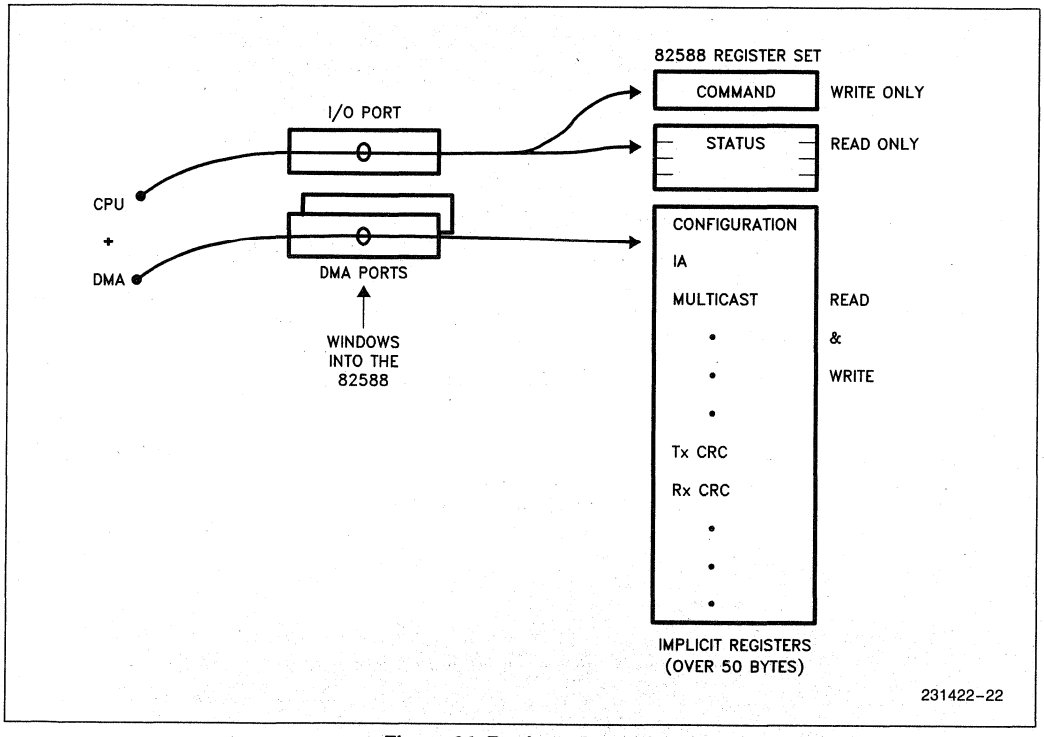
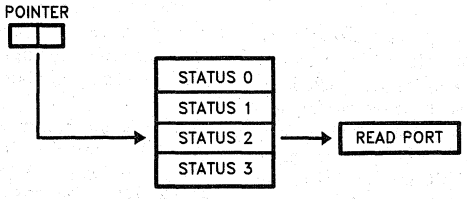


Figure 21. Register Access

4 Status registers are accessed through one read port



231422-23

The pointer can be changed using a command or can be automatically incremented.

```
READ_STATUS_588: PROCEDURE;                    /* COMMAND 15 */
     OUTPUT (CS_588) = 15;                   /* RELEASE POINTER, INITIAL = 00 */
     STATUS_588(0)=INPUT (CS_588);           /* REFRESH STATUS REGISTER IMAGE */
     STATUS_588(1)=INPUT (CS_588);           /* IN MEMORY.
     STATUS_588(2)=INPUT (CS_588);
     STATUS_588(3)=INPUT (CS_588);
     RETURN
END READ_STATUS_588;
```

READING 4 STATUS REGISTERS

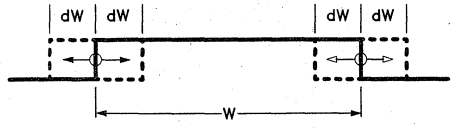
Figure 22. Reading the Status Register

nostics functions. The DIAGNOSE command of the 82588 does a self-test of most of the counters and timers in the 82588 serial unit. Using the DUMP command, all the internal registers of the 82588 can be dumped into the memory. The TDR command does Time Domain Reflectometery on the network. The 82588 has two loopback modes of operation. In the internal loopback mode, the TXD line is internally connected to the RXD one. No data appears outside the chip, and the 82588 is isolated from the link. This mode enables checking of the receive and transmit machines without link interference. In the external loopback mode, the 82588 becomes a full duplex device, being able to receive its own transmitted frames. In this mode data goes through the link and all CSMA/CD mechanisms are involved.

3.10 Jitter Performance

When the 82588 receives a frame from the HUB, the signal has jitter. Jitter is the shifting of the edges of the signal from their nominal position due to the transmission over a length of cable. Many factors like, intersymbol interference (pulses of different widths have different delays through the transmission media), rise and fall times of drivers and receivers, cross talk etc., contribute to the jitter. StarLAN specifies a maximum jitter of ±62.5 ns whenever the signal goes from a NODE/HUB or HUB/HUB. Figure 23 shows that the jitter tolerance of the 82588 is exactly the required ±62.5 ns at 1 Mbs for both 8X, 16X Manchester encoded data.

Jitter = ± variation of an edge from its nominal position.
Jitter can occur on every edge.



231422-78

	x8	x16
Manchester	±1/16	±1/16
NRZI (Code Violations Enabled)	±1/16 BT	±3/32 BT
NRZI (Code Violations Disabled)	±3/16 BT	±3/16 BT

Figure 23. 82588 Jitter Performance

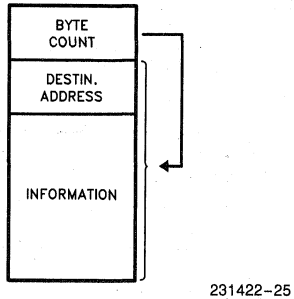
4.0 THE 82588

This chapter describes the basic 82588 operations. Please refer to the 82588 reference manual in Intel Microcommunications Handbook for a detailed description. Basic operations like transmitting a frame, receiving a frame, configuring the 82588 and dumping the register contents are discussed here to give a feel for how the 82588 works.

4.1 Transmit and Retransmit Operations

To transmit a frame, the CPU prepares a block in the memory called the transmit data block. As shown in Figure 24, this block starts with a byte count field, indicating how long the rest of the block is. The destination address field contains the node address of the destination. The rest of the block contains the information or the data field of the frame. The CPU also programs the DMA controller with the start address of the transmit data block. The DMA byte count must be equal to or greater than the block length. The 82588 is then issued a TRANSMIT command—an OUT instruction to the command port of the 82588. The 82588 starts generating DMA requests to read in the transmit data block by DMA. It also determines whether and how long it must defer on the link and after that, it starts transmitting the preamble. The 82588 constructs the frame on the fly. It takes the destination address from the memory, source address from its own individual address memory (previously programmed), data field from the memory and the CRC, is generated on chip, at the end of the frame.

1. Prepare Transmit Data—Block in Memory
2. Program DMA Controller
3. Issue Transmit Command on the Desired Channel



Transmit Data Block

4. Interrupt is received on completion of command or if the command was aborted or there was a collision. The status bytes 1 and 2 indicate the result of the operation.

7	6	5	4	3	2	1	0	
TX DEF	HRT BEAT	MAX COLL		NUM. OF COLLISIONS				STATUS 1
COLL		TX OK			LOST CRS	LOST CTS	UNDER RUN	STATUS 2

231422-26

Transmit & Retransmit Results Format

Figure 24. Transmit Operation

At the conclusion of transmission the 82588 generates an interrupt to the CPU. The CPU can then read the status registers to find out if the transmission was successful. If a collision occurs during transmission, the 82588 aborts transmission and generates the jam sequence, as required by IEEE 802.3, and informs the CPU through interrupt and the status registers. It also starts the back-off algorithm.

To re-attempt transmission, the CPU must reinitialize the DMA controller 7to the start of the transmit data block and issue a RETRANSMIT command to the 82588. When the 82588 receives the retransmit command and the back-off timer has expired, it transmits again. Interrupt and the status register contents again indicate the success or failure of the (re)transmit attempt.

The main difference between transmit and retransmit commands is that retransmit does not clear the internal count for the number of collisions occurred, whereas transmit does. Moreoever, retransmit takes effect only when the back-off timer has expired.

4.2 Configuring the 82588

To initialize the 82588 and program its network and system parameters, a configure operation is performed. It is very similar to the transmit operation. Instead of a transmit data block as in transmit command, a configure data block—shown in Figure 12—is prepared by the CPU in the memory. The first two bytes of the block specify the length of the rest of the block, which specify the network and system parameters for the 82588. The DMA controller is then programmed by the CPU to the beginning of this block and a CONFIGURE command is issued to the 82588. The 82588 reads in the parameters by DMA and loads the parameters in the on-chip registers.

Similarly, for programming the INDIVIDUAL ADDRESS and MULTICAST ADDRESSes, the DMA controller is used to load the 82588 registers.

4.3 Frame Reception

Before enabling the 82588 for reception the CPU must make a buffer available for the frame to be received. The CPU must program the DMA controller with the starting address of the buffer and then issue the RX_ ENABLE command to the 82588. When a frame arrives at the RxD pin of the 82588, it starts being received. Only if the address in the destination address matches either the Individual address, Multicast address or if it is a broadcast address, is the frame deposited into memory by the 82588 using DMA. The format of storage in the memory is shown in Figure 25. At the end, a two byte field is attached which shows the status of the received frame. If CRC, alignment or overrun errors are encountered, they are reported. An inter-

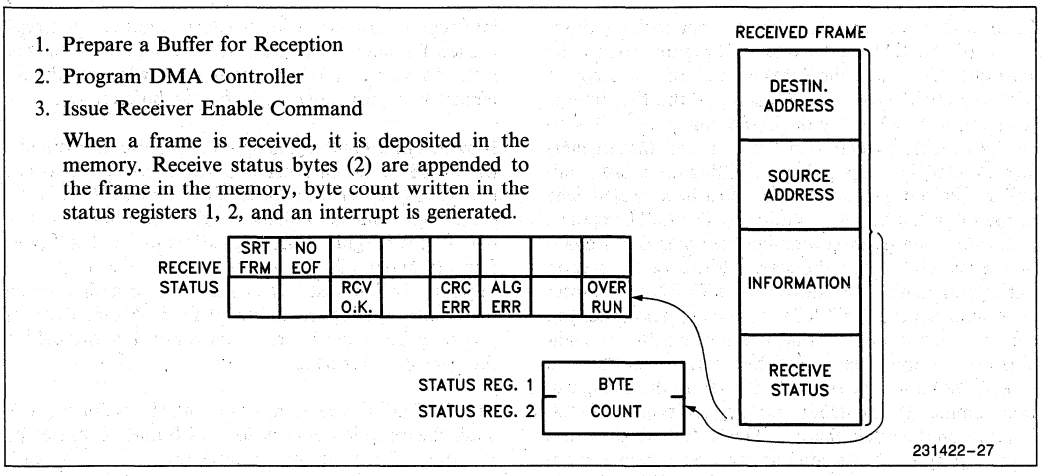


Figure 25. Receive Operation (Single Buffer)

rupt from 82588 occurs when all the bytes have been transferred to the memory. This informs the CPU that a new frame has been received.

If the received frame has errors, the CPU must recover (or re-use) the buffer. Note that the entire frame is deposited into one buffer. The 82588 when NOT configured for the external loopback mode, will detect collisions (code violations) during receptions. If a collision is detected, the reception is aborted and status updated. CPU is then informed by an interrupt (if the collided frame fragment is shorter than the address length, no reception will be started), and no interrupt will happen.

4.3.1 Multiple Buffer Frame Reception

It is also possible to receive a frame into a number of fixed size buffers. This is particularly economical if the received frames vary widely in size. If the single buffer scheme were used as described above, the buffer required would have to be bigger than the longest expected frame and would be very wasteful for very short (typically acknowledge or control) frames. The multiple buffer reception is illustrated in Figure 26. It uses two DMA channels for reception.

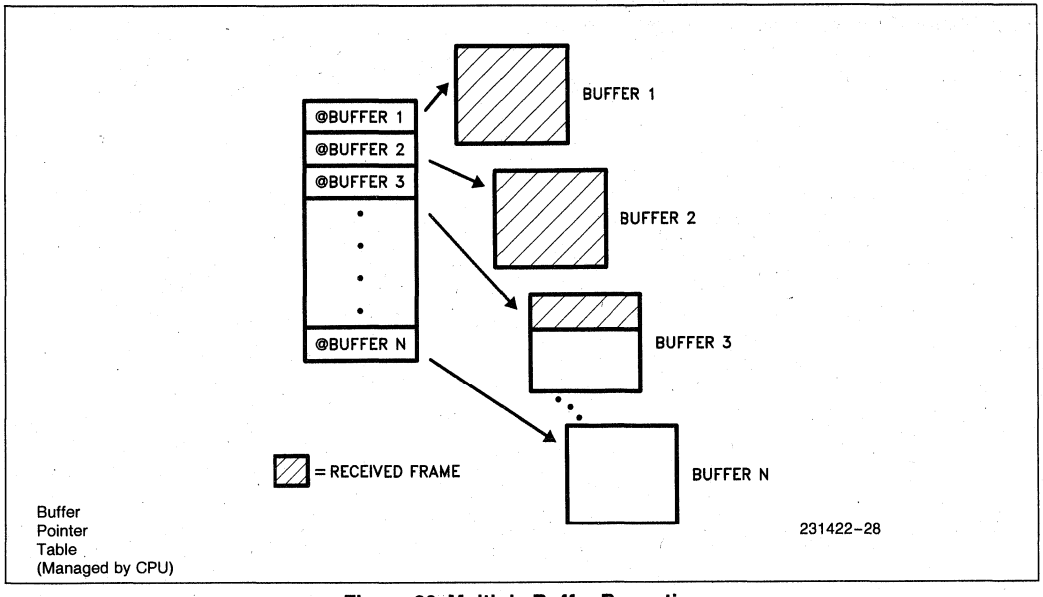


Figure 26. Multiple Buffer Reception

As in single buffer reception, the one channel, say channel 0, of the DMA controller is programmed to the start of buffer 1, and the 82588 is enabled for reception with the chaining bit set. As soon as the first byte is read out of the 82588 by the DMA controller and written into the first location of buffer 1, the 82588 generates an interrupt, saying that it is filling up its last available buffer and one more buffer must be allocated. The filling up of the buffer 1 continues. The CPU responds to the interrupt by programming the other DMA channel—channel 1—with the start address of the second buffer and issuing an ASSIGN ALTERNATE buffer command with an INTACK (interrupt acknowledge). This informs the 82588 that one more buffer is available on the other channel. When buffer 1 is filled up (the 82588 knows the size of buffers from the configuration command), the 82588 starts generating the DMA requests on the other channel. This automatically starts filling up buffer 2. As soon as the first byte is written into buffer 2, the 82588 interrupts the CPU again asking for one more buffer. The CPU programs the channel 0 of the DMA controller with the start address of buffer 3, issues an ASSIGN ALTERNATE buffer command with INTACK. This keeps the buffer 3 ready for the 82588. This switching of channels continues until the entire frame is received generating an end of frame interrupt. The CPU maintains the list of pointers to the buffers used.

Since a new buffer is allocated at the time of filling up of the last buffer, the 82588 automatically switches to the new buffer to receive the next frame as soon as the last frame is completely received. It can start receiving the new frame almost immediately, even before the end of frame interrupt is serviced and acknowledged by the CPU. If a new frame comes in, and the previous frame interrupt is not yet acknowledged, another interrupt needed for new buffer allocation is buffered (and not lost). As soon as the first one is acknowledged, the interrupt line goes active again for the buffered one.

If by the time a buffer fills up no new buffer is available, the 82588 keeps on receiving. An overrun will occur and will be reported in the received frame status. However, ample time is available for the allocation of a new buffer. It is roughly equal to the time to fill up a buffer. For 128 byte buffers it is $128 \times 8 = 1024$ ms or approximately 1 millisec. You get 1 ms to assign a new buffer after getting the interrupt for it. Hence the process of multiple buffer reception is not time critical for the system performance.

This method of reception is particularly useful to guarantee the reception of back-to-back frames separated by IFS time. This is because a new buffer is always available for the new frame after the current frame is received.

Although both the DMA channels get used up in receiving, only one channel is kept ready for reception and the other one can be used for other commands until the reception starts. If an execution command like transmit or dump command is being executed on a channel which must be allocated for reception, the command gets automatically aborted when the ASSIGN ALTERNATE BUFFER command is issued to the channel used for the execution command. The interrupt for command abortion occurs after the end of frame interrupt.

4.4 Memory Dump of Registers

All the 82588 internal registers can be dumped in the memory by the DUMP command. A DMA channel is used to transfer the register contents to the memory. It is very similar to reception of a frame; instead of data from the serial link, the data from the registers gets written into the memory. This provides a software debugging and diagnostic tool.

4.5 Other Operations

Other 82588 operations like DIAGNOSE, TDR, ABORT, etc. do not require any parameter or data transfer. They are executed by writing a command to the 82588 command register and knowing the results (if any) through the status registers.

5.0 StarLAN NODE FOR IBM PC

This chapter deals with the hardware—the StarLAN board—to interface the IBM PC to a StarLAN Network. This is a slave board which takes up one slot on the I/O channel of the IBM PC. Figure 27 shows an abstract block diagram of the board. It requires the IBM PC resources of the CPU, memory, DMA and interrupt controller on the system board to run it. Such a board has two interfaces. The IBM PC I/O Channel on the system or the parallel side and the telephone grade twisted pair wire on the serial side. Figures 28, 29 show the circuit diagram of the board.

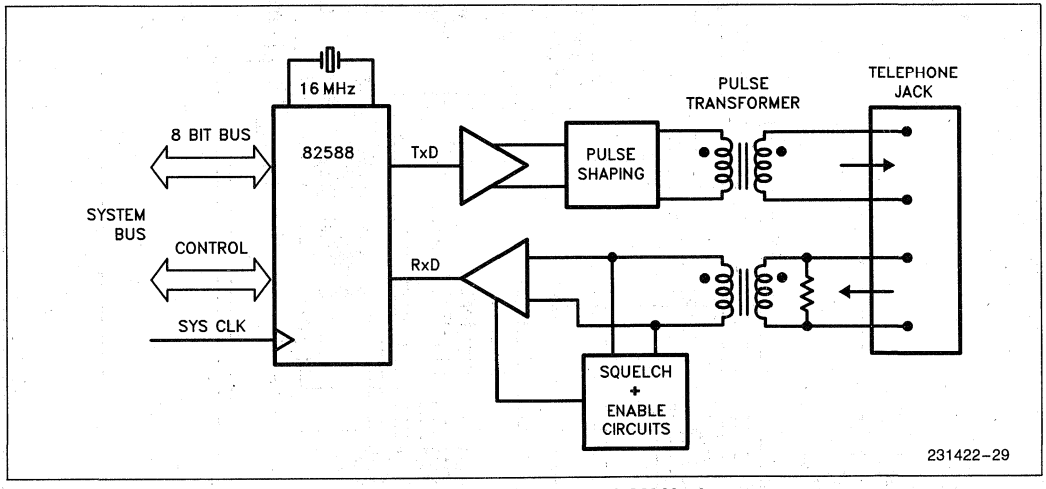


Figure 27. 82588 Based StarLAN Node

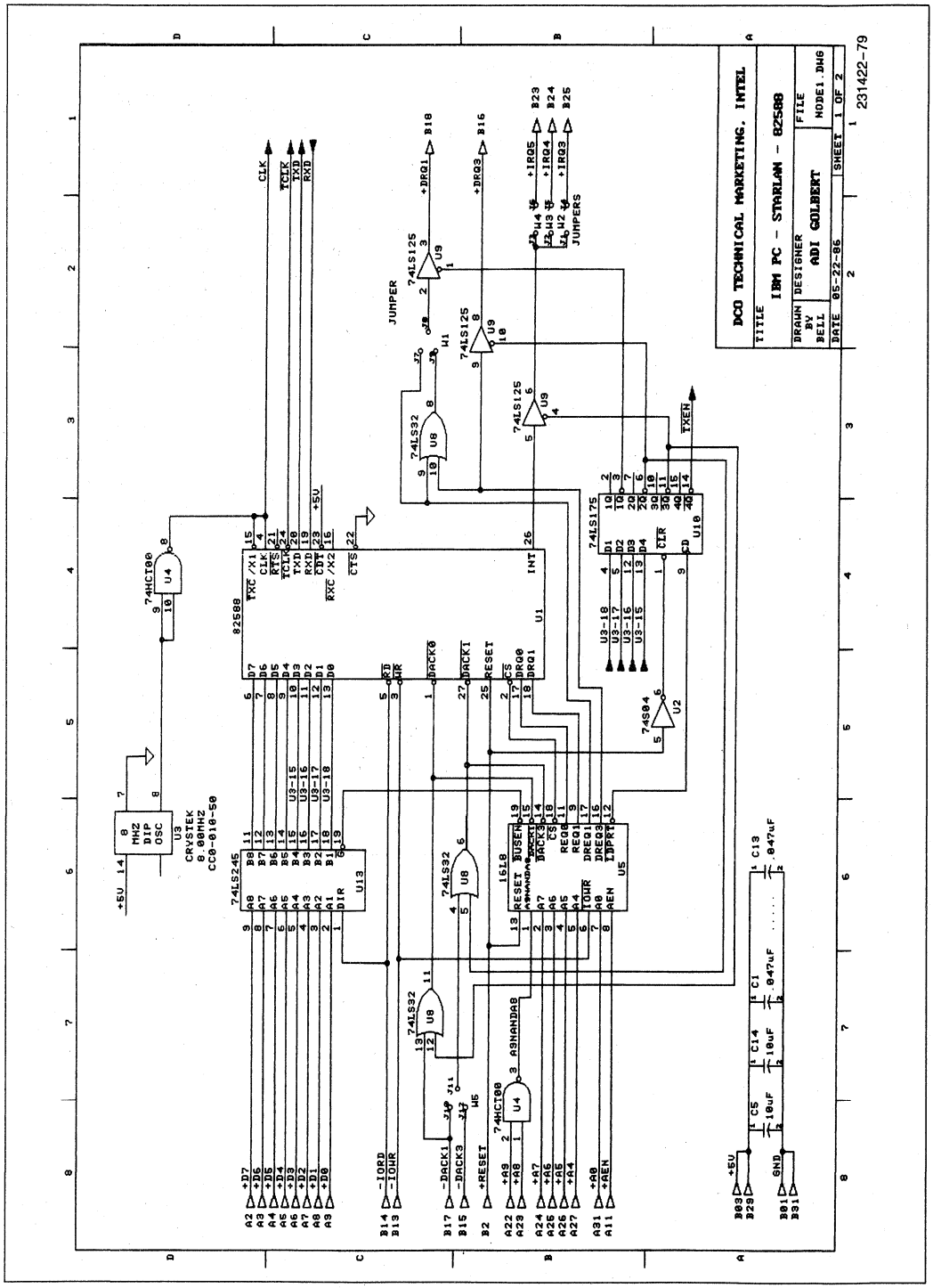

DCD TECHNICAL MARKETING, INTEL
IBM PC - STARLAN - 82588
DESIGNER
ADI GOLDBERT
NODE1.DWG
SHEET 1 OF 2
231422-79
82588
74LS245
74LS32
74LS125
74LS175
74HCT00
74HCT86
74S04
CRYSTEK
8.00MHZ
CC8-010-50
JUMPER
JUMPERS
TXEN
CLK
TCLK
TXD
RXD
CTS
INT
+5V
GND
+DRQ1
+DRQ3
+IRQ5
+IRQ4
+IRQ3
-IORD
-IOWR
-DACK1
-DACK3
+RESET
ASHANDA8
+AEN

Figure 28

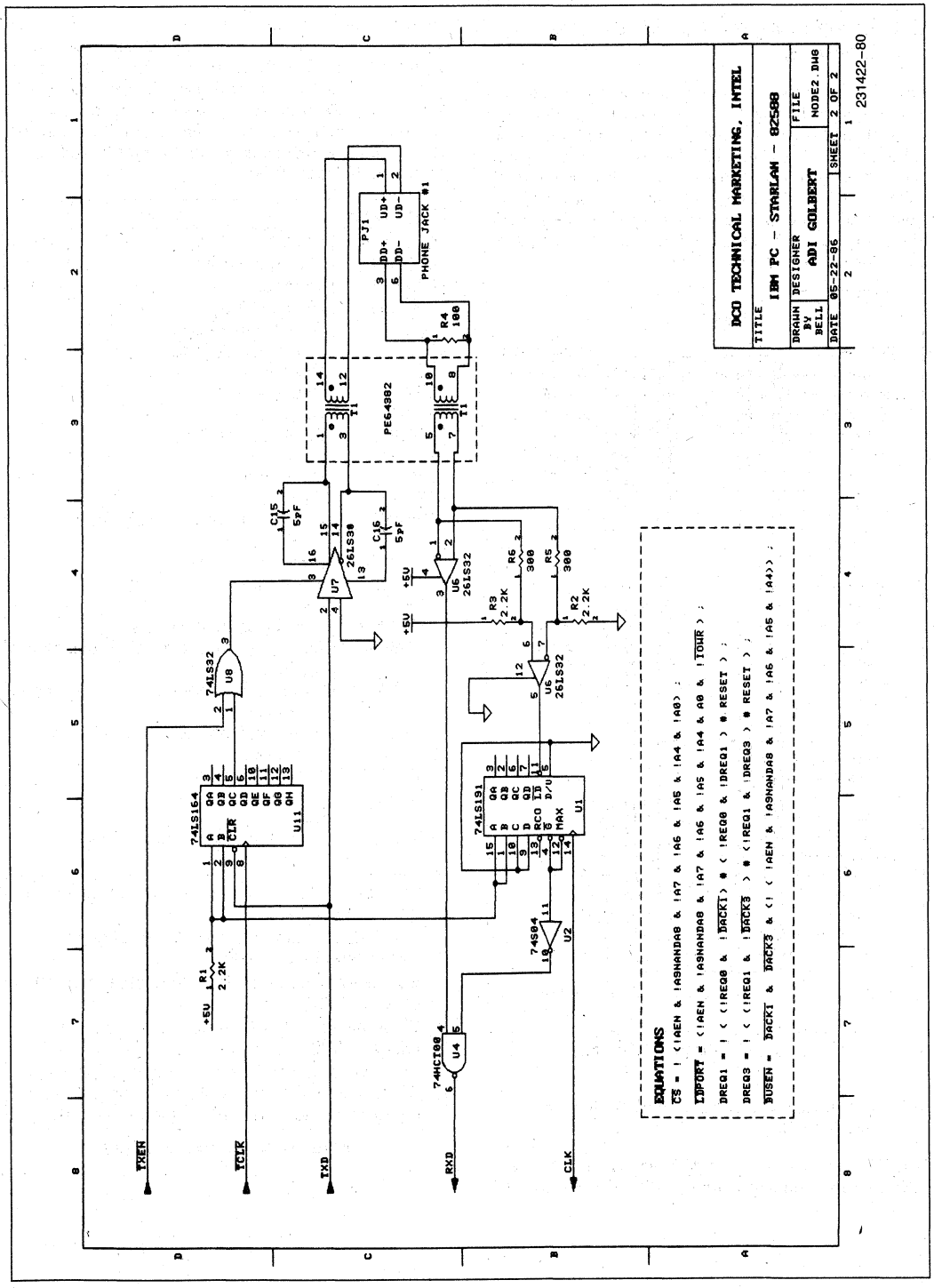

DCO TECHNICAL MARKETING, INTEL
TITLE
IBM PC - STARLAN - 82588
DRAWN BY BELL
DESIGNER ADI GOLBERT
FILE NODE2.DWG
DATE 05-22-86
SHEET 2 OF 2
231422-80
EQUATIONS
CS = ! (!AEN & !A9HAND8 & !A7 & !A6 & !A5 & !A4 & !A0) ;
IOPORT = (!AEN & !A9HAND8 & !A7 & !A6 & !A5 & !A4 & A0 & !IOWR) ;
DREQ1 = ! ((!REQ0 & !DACK1) # (!REQ0 & !DREQ1) # RESET) ;
DREQ3 = ! ((!REQ1 & !DACK3) # (!REQ1 & !DREQ3) # RESET) ;
BUSEN = DACK1 & DACK3 & (! (!AEN & !A9HAND8 & !A7 & !A6 & !A5 & !A4)) ;
PJ1
PHONE JACK #1
PE64382
T1
U7
26LS30
U6
26LS32
U5
26LS32
U8
74LS32
U11
74LS164
U1
74LS191
U2
74S04
U4
74HCT00_4
C15
5PF
C16
5PF
R1
2.2K
R2
2.2K
R3
2.2K
R4
100
R5
300
R5
300
+5V
TXEN
TCLK
TXD
RXD
CLK

Figure 29

5.1 Interfacing to the IBM PC I/O Channel

IBM PC has 8 slots on the system board to allow expansion of the basic system. All of them are electrically identical and the I/O channel is the bus that links them all to the 8088 system bus. The I/O channel contains an 8 bit bidirectional data bus, 20 address lines, 6 levels of interrupt, 3 channels of DMA control lines and other control lines to do I/O and memory read/write operations. Figure 30 shows the signals and the pin assignment for the I/O Channel.

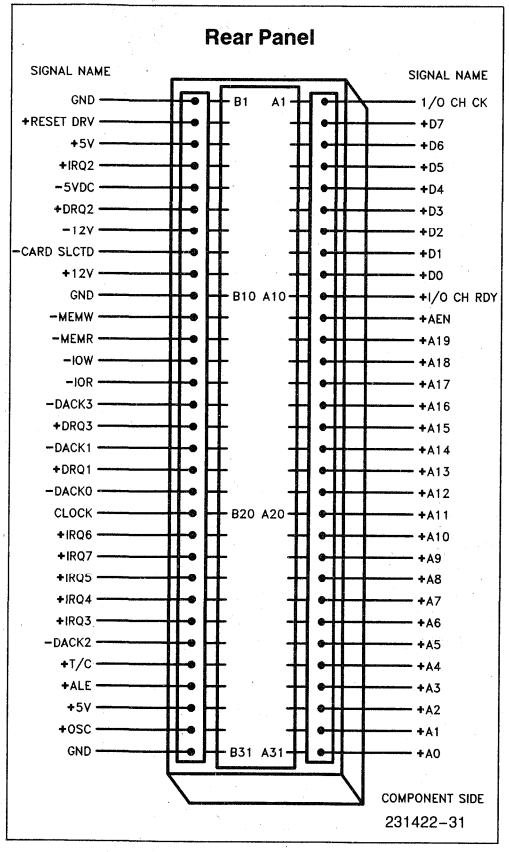


Figure 30. I/O Channel Diagram

5.1.1 REGISTER ACCESS AND DATA BUS INTERFACE

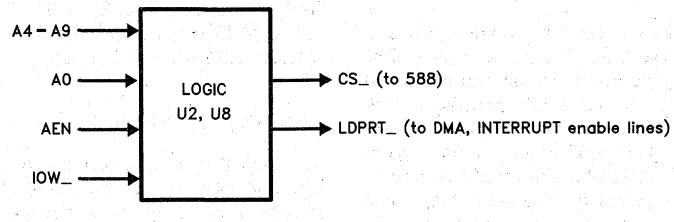
The CPU accesses the StarLAN adapter card through 2 I/O address windows. Address 300H is used to access to 82588 for commands and status, address 301H accesses an on board control port that enables the various interrupt and DMA lines. Even though only two addresses are needed, the card uses all the 16 addresses spaces from 300H to 30FH. This was done to keep simplicity and minimum component count. Registers address decoding is done using a PAL (16L8) and an external NAND gate (U8).

Hex Range	Usage
000-00F	DMA Chip 8237A-5
020-021	Interrupt 8259A
040-043	Timer 8253-5
060-063	PPI 8255A-5
080-083	DMA Page Registers
0AX*	NMI Mask Register
0CX	Reserved
0EX	Reserved
200-20F	Game Control
210-217	Expansion Unit
220-24F	Reserved
278-27F	Reserved
2F0-2F7	Reserved
2F8-2FF	Asynchronous Communications (Secondary)
300-31F	Prototype Card
320-32F	Fixed Disk
378-37F	Printer
380-38C**	SDLC Communications
380-389**	Binary Synchronous Communications (Secondary)
3A0-3A9	Binary Synchronous Communications (Primary)
3B0-3BF	IBM Monochrome Display/Printer
3C0-3CF	Reserved
3D0-3DF	Color/Graphics
3E0-3E7	Reserved
3F0-3F7	Diskette
3F8-3FF	Asynchronous Communications (Primary)

* At power-on time, the Non Mask Interrupt into the 8088 is masked off.
This mask bit can be set and reset through system software as follows:
Set mask: Write hex 80 to I/O Address hex A0 (enable NMI)
Clear mask: Write hex 00 to I/O Address hex A0 (disable NMI)

** SDLC Communications and Secondary Binary Synchronous Communications cannot be used together because their hex addresses overlap.

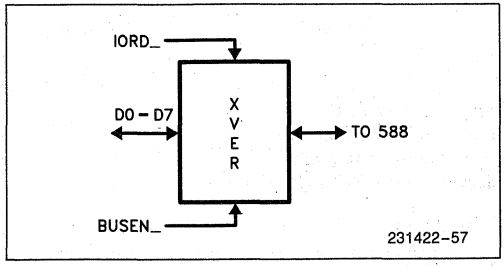
Figure 31. I/O Address Map



Register Access

```
Format of Following Equations Will Be According To
The Following Specifications:
    !  INVERT
    _  SIGNAL ACTIVE LOW
    &  LOGIC AND
    #  LOGIC OR
A9NANDA8 = ! (A9 & A8)
CS_ = ! ( !AEN & !A9NANDA8 & !A7 & !A6 & !A5 & !A4 & !A0 )
LDPORT_ = ! ( !AEN & !A9NANDA8 & !A7 & !A6 & !A5 & !A4 & A0 & !IOWR_ )
BUSEN_ = DACK1_ & DACK2_ & (! ( !AEN & !A9NANDA8 & !A7 & !A6 & !A5 & !A4));
```

The signal CS__ decodes address 300H, it is only active when AEN is inactive meaning CPU and not DMA cycles. LDPORT__ has exactly the same logic for address 301H, but it is only active during I/O write cycles. The I/O port sitting on address 301H is write only. The data BUS lines D0 to D7 are buffered from the 82588 to the PC bus using an 74LS245 transceiver chip.



Data Bus Interface

The Bus transceiver is enabled if: A DMA access is taking place, or I/O ports 300H to 30FH are being accessed.

5.1.2 Control Port

As mentioned the StarLAN adapter port has a 4-bit write only control port. The purpose of this port is to selectively enable the DMA and INTERRUPT request lines. Also it can completely disable the transmitter.

Control Port Definition

ENDRQ1	ENDRQ3	ENINTER	TXEN

ENDRQ1, ENDRQ2 : "1" Enable DMA requests.
ENINTER : "1" Enable INTERRUPT request.
TXEN : "1" Enable the transmitter.

On power up all bits default to "0".

5.1.3 CLOCK GENERATION

The 82588 requires two clocks for operation. The system clock and the serial clock. The serial clock can be generated on chip by putting a crystal across X1 and X2 pins. Alternatively, an externally generated clock can be fed in at pin X1 (with X2 left open). In both cases, the frequency must be either 8 or 16 times (sampling factor) the desired bit rate. For StarLAN, 8 or 16 MHz are the correct values to generate 1 Mb/s data rate. A configuration parameter is used to tell the 82588 what the sampling factor is. An externally supplied clock must have MOS levels (0.6V–3.9V). Specifications for the crystal and the circuit are shown in Figure 32.

The system clock has to be supplied externally. It can be up to 8 MHz. This clock runs the parallel side of the 82588. Its frequency does not have any impact on the read and write access times but on the rate at which data can be transferred to and from the 82588 (Maximum DMA data rate is one byte every two system clocks). This clock doesn't require MOS levels.

The I/O channel of the IBM PC supplies a 4.77 MHz signal of 33% duty cycle. This signal could be used as a system clock. It was decided, however, to generate a separate clock on the StarLAN board to be independent of the I/O channel clock so that this board can also be used in other IBM PCs and also in some other compatibles. The 8 MHz system clock is generated using a DIP OSCILLATOR which have the required 50 ppm tolerance to meet StarLAN. This clock is converted to MOS levels by 74HCT00 and fed into both the system and serial clock inputs.

5.1.4 DMA INTERFACE

The 82588 requires either one or two DMA channels for full operation. In this application, one channel is dedicated for reception and the other is used for transmissions and the other commands. Use of only one DMA channel is possible but may require more complex software, also some RX frames may be lost during switches of the DMA channel from the receiver to the transmitter (Those frames will be recovered by higher layers of the protocol). Also using only one DMA channel will limit the 82588 loopback functionality. So the recommendation is to operate with two DMA channels if available. Appendix C describes a method of operating with only one DMA channel without loosing RX frames.

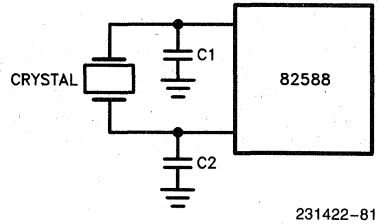
The IBM PC system board has one 8237A DMA controller. Channel 0 is used for doing the refresh of DRAMs. Channels 1, 2 and 3 are available for add-on boards on the I/O Channel. The floppy disk controller board uses the DMA channel 2 leaving exactly two channels (1 and 3) for the 82588. The situation is worse if the IBM PC/XT is used, since it uses channel 3 for the Winchester hard disk leaving just the channel 1 for

Series Resonance

—Frequency Will Drift by About 400 PPM from Nominal
—No Capacitors Needed
—Doesn't Meet StarLAN Requirements

Meeting StarLAN 100 PPM Requirements

—Use Parallel Resonance Crystal
—Recommended For Precise Frequencies
—82588 X-TAL Oscillator Stability ±35 PPM (0–70°C)



Crystal: Load Capacitance = 20 pF
Shunt Capacitance = 7 pF Maximum
Series Resistance = 30Ω Maximum
Frequency Tolerance = 50 PPM (0–70°C)

C1, C2 → 27 pF or 39 pF, 5%

Figure 32. Crystal Specifications

the 82588. On the other hand, the IBM PC/AT has 5 free DMA channels. We will assume that 8237A DMA channels 1 and 3 are available for the 82588 as in the case of the IBM PC.

Since the channel 0 of 8237A is used to do refresh of DRAMs all the channels should be operated in single byte transfer mode. In this mode, after every transfer for any channel the bus is granted to the current highest priority channel. In this way, no channel can hog the bus bandwidth and, more important, the refresh of DRAMs is assured every 15 microseconds since the refresh channel (number 0) has the highest priority. This mode of operation is very slow since the HOLD is dropped by the 8237A and then asserted again after every transfer. Demand mode of operation is a lot more suitable to 82588 but it cannot be used because of the refresh requirements.

Whenever the 82588 interfaces to the 8237A in the single transfer mode, there is a potential 8237A lock-up problem. The 82588 may deactivate its DMA request line (DREQ) before receiving an acknowledge from the DMA controller. This situation may happen during command abortions, or aborted receptions. The 8237A under those circumstances may lock-up. In order to solve this potential problem, an external logic must be used to insure that DREQ to the DMA controller is never deactivated before the acknowledge is received. Figure 33 shows the logic to implement this function. This logic is implemented in the 16L8 PAL.

The 82588 DREQ lines are connected to the IBM/PC bus through tri-state buffers which are enabled by writing to I/O port 301H. This function enables the use of either one or two DMA channels and also the sharing of DMA channels with other adapter boards.

5.1.5 INTERRUPT CONTROLLER

The 82588 interrupts the CPU after the execution of a command or on reception of a frame. It uses the 8259A interrupt controller on the system board to interrupt the CPU. There are 6 interrupt request lines, IRQ2 to IRQ7, on the I/O channel. Figure 34 shows the assignment of the lines. In fact, none of the lines are completely free for use. To add any new peripheral which uses a system board interrupt, this interrupt needs to have the capability to share the specific line, by driving the line with a tri-state driver. The 82588 StarLAN adapter board can optionally drive interrupt lines IRQ3, IRQ4 or IRQ5 (An 74LS125 driver is used).

Number	Usage
NMI	Parity
0	Timer
1	Keyboard
2	Reserved
3	Asynchronous Communications (Secondary) SDLC Communications BSC (Secondary)
4	Asynchronous Communications (Primary) SDLC Communications BSC (Primary)
5	Fixed Disk
6	Diskette
7	Printer

Figure 34. IBM PC Hardware Interrupt Listing

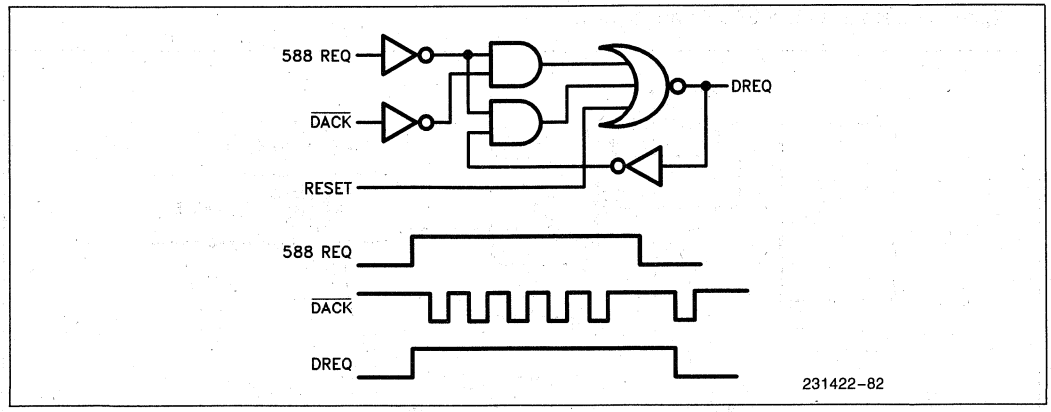


Figure 33. DMA Request Logic

5.2 Serial Link Interface

A typical StarLAN adapter board is connected to the twisted pair wiring using an extension cable (typically up to 8 meters—25 ft.). See Figure 35. One end of the cable plugs into the telephone modular jack on the StarLAN board and the other end into a modular jack in the wall. The twisted pair wiring starts at the modular jack in the wall and goes to the wiring closet. In the wiring closet, another telephone extension cable is used to connect to a StarLAN HUB. The transmitted signal from the 82588 reach the on-board telephone jack through a RS-422 driver with pulse shaping and a pulse transformer. The received signals from the telephone jack to the 82588 come through a pulse transformer, squelch circuit and a receive enable circuit.

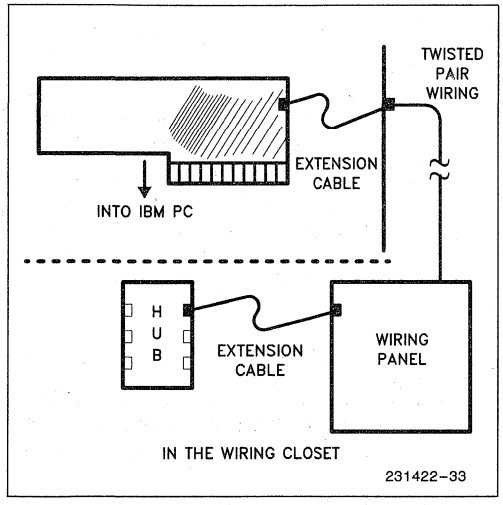


Figure 35. Path from StarLAN Board to HUB

5.2.1 TRANSMIT PATH

The single ended transmit signal on the TxD pin is converted to a differential signal and the rise and fall times are increased to 150 to 200 ns before feeding it to the pulse transformer (this pulse shaping is not a requirement, but proves to give good results). Am26LS30 is a RS-422 driver which converts the TxD signal to a differential signal. It also has slew rate control pins to increase to rise and fall times. A large rise and fall time reduces the possibility of crosstalk, interference and radiation. By the other hand a slower edge rate increases the jitter. In the StarLAN adapter card, the first approach was used. The 26LS30 converts a square pulse to a trapezoidal one—see Figure 36. The filtering effect of the cable further adds to reduce the higher frequency components from the waveform so that on the cable the signal is almost sinusoidal. The pulse transformer is for DC isolation. The pulse transformers from Pulse Engineering—type PE 64382—was used in this design. This is a dual transformer package which introduces an additional rise and fall time of about 70–100 ns on the signal, helping the former discussed waveshaping.

5.2.2 IDLE PATTERN GENERATION

StarLAN requires transmitters to generate an IDLE pattern after the last transmitted data bit. The IDLE pattern is defined to be a constant high level for 2–3 microseconds. The purpose of this pattern is to insure that receivers will decode properly the last transmitted data bits before signal decay. Currently the 82588 needs one external component to generate the IDLE. The operation principle is to have an external shift register (74LS164) that will kind of act as an envelope detector of the TXD line. Whenever the TXD line goes low

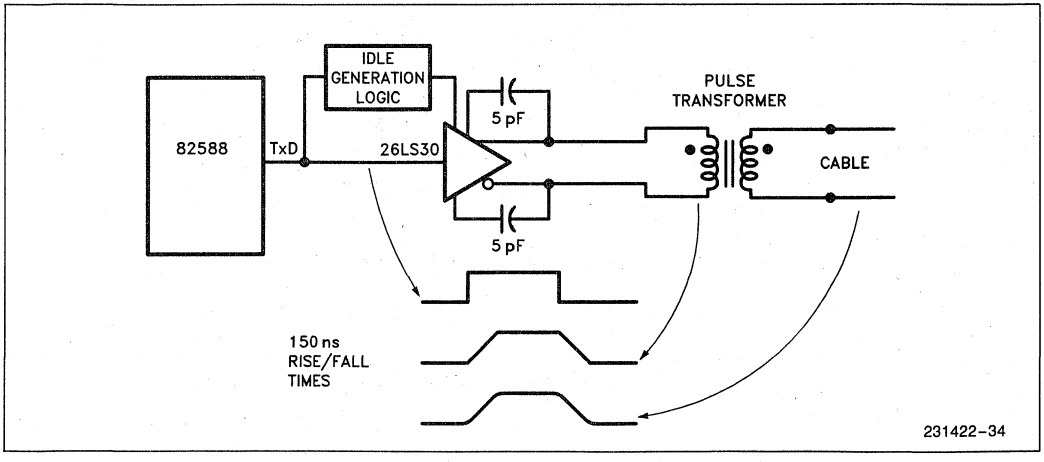


Figure 36. Wave Shaping

(first preamble bit), the output of the shift register (third cell) will immediately go low, enabling the RS-422 driver, the shift register being clocked by TCLK—will time the duration of the TXD high times. If the high time is more than 2 microseconds, meaning that the 82588 has gone idle, the transmitter will be disabled (See Figure 37). Another piece of this logic is the ORing of the output of the shift register with TXEN—signal which comes from the board control port. This signal completely disables the transmitter. The other purpose of this enable signal, is to make sure that after power-up, before the 82588 is configured, the RS-422 drivers won't be enabled (TCLK__ is not active before the configure command). See Figures 28, 29 for the complete circuit.

5.3 RECEIVE PATH

The signal coming from the HUB over the twisted pair wire is received on the StarLAN board through a 100Ω line termination resistor and a pulse transformer. The pulse transformer is of the same type as for the transmit side and its function is dc isolation. The received signal which is differential and almost sinusoidal is fed to the Am26LS32 RS-422 receiver. As seen from Figure 38 the pulse transformer feeds two RS-422 receivers. The one on the bottom is for squelch filtering and the one above is the real receiver which does real zero crossing detection on the signal and regenerates a square digital waveform from the sinusoidal signal that is received. Proper zero crossing detection is very essential; if the edges of the regenerated signal are not at zero crossings, the resulting signal may not be a proper Manchester encoded signal (self introduced jitter) even if the original signal is valid Manchester. The resistors in the lower receiver keep its differential inputs at a voltage difference of 600 mV. These bias resistors ensure that the output remains high as long as the input signal is more than −600 mV. It is very important that the RxD pin remains HIGH (not LOW or floating) whenever the receive line is idle. A violation of this may cause the 82588 to lock-up on transmitting. Remember, that based on the signal on the RxD pin, the 82588 extracts information on the data being received, Carrier Sense and Collision Detect. This squelch of 600 mV keeps the idle line noise from getting to the 82588. Figure 39 shows that when the differential input of the receiver crosses zero, a transition occurs at the output. It also shows that if the signal strength is higher than −600 mV, the output does not change. (This kind of squelching is called negative squelching, and it is done due to the fact that the preamble pattern starts with a going low transition). Note that the differential voltage at the upper receiver input is zero when the line is idle. The output of the squelch goes to a pulse stretcher which generates an envelope of the received frame. The envelope is a receive enable signal and is used to AND the signal from the real zero crossing receiver before feeding it to the RxD pin of the 82588.

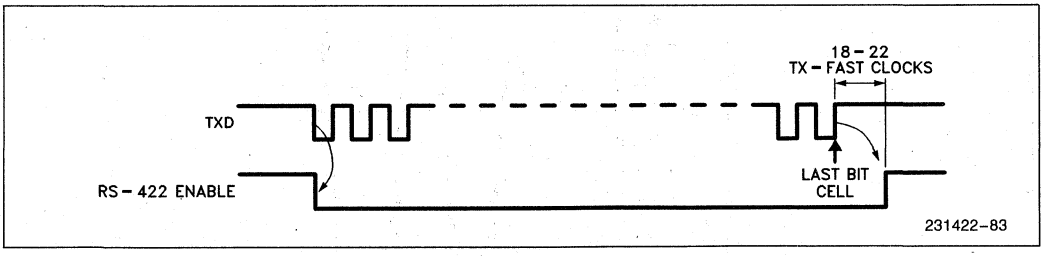


Figure 37. Idle Generation

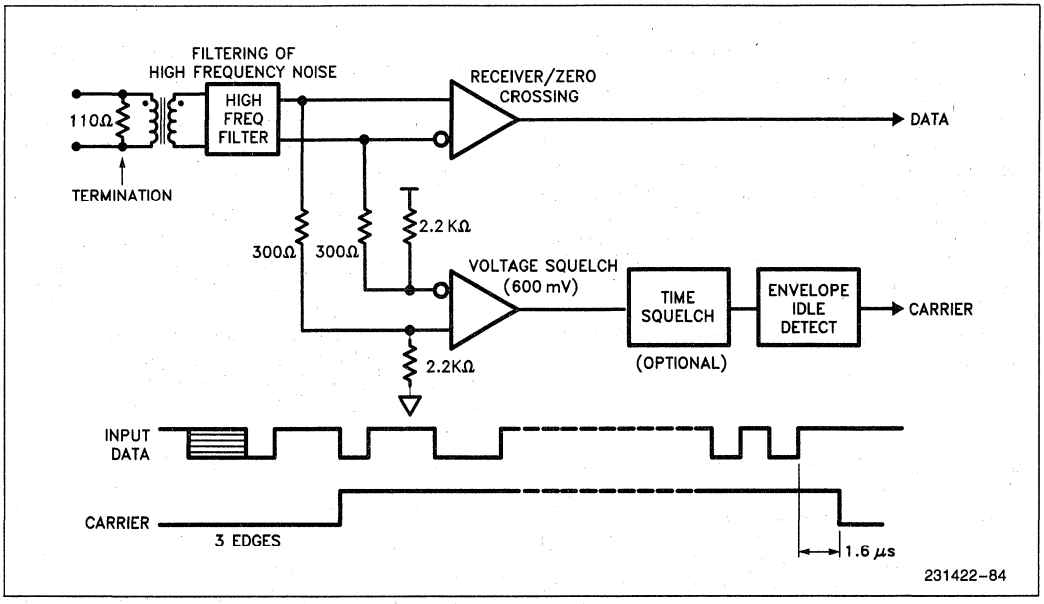


Figure 38. Input Ports

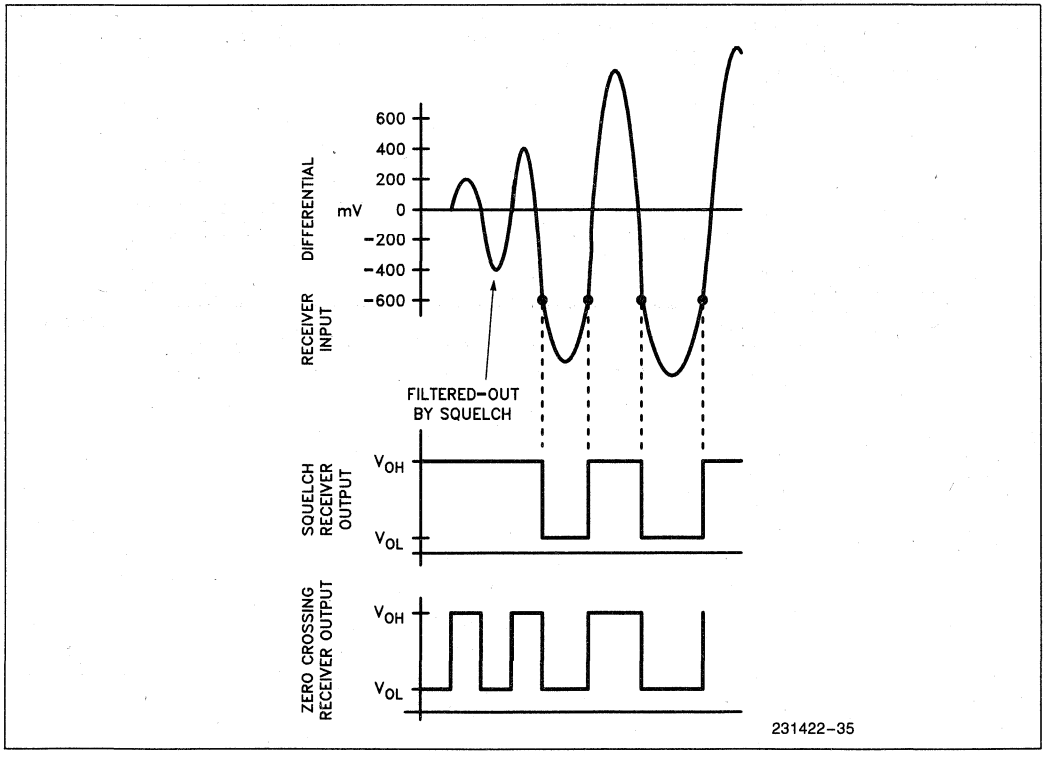


Figure 39. Squelch Circuit Output

5.4 80188 Interface to 82588

Although the 82588 interfaces easily to almost any processor, no processor offers as much of the needed functionality as the 80186 or its 8 bit cousin, the 80188. The 80188 is 8088 object code compatible processor with DMA, timers, interrupt controller, chip select logic, wait state generator, ready logic and clock generator functions on chip. Figure 40 shows how the 82588, in a StarLAN environment interfaces to the 80188. It uses the clock, chip select logic, DMA channels, interrupt controller directly from the 80188. The interface components between the CPU and the 82588 are totally eliminated.

5.5 iSBX Interface to StarLAN

Figure 41 shows how to interface the 82588 in a StarLAN environment to the iSBX bus. It uses 2 DMA channels—tapping the second DMA channel from a neighboring iSBX connector. Such a board can be used to make a StarLAN to an Ethernet or a SNA or DECNET gateway when it is placed on an appropriate SBC board. It may also be used to give a StarLAN access to any SBC board (with an iSBX connector) independent of the type of processor on the board.

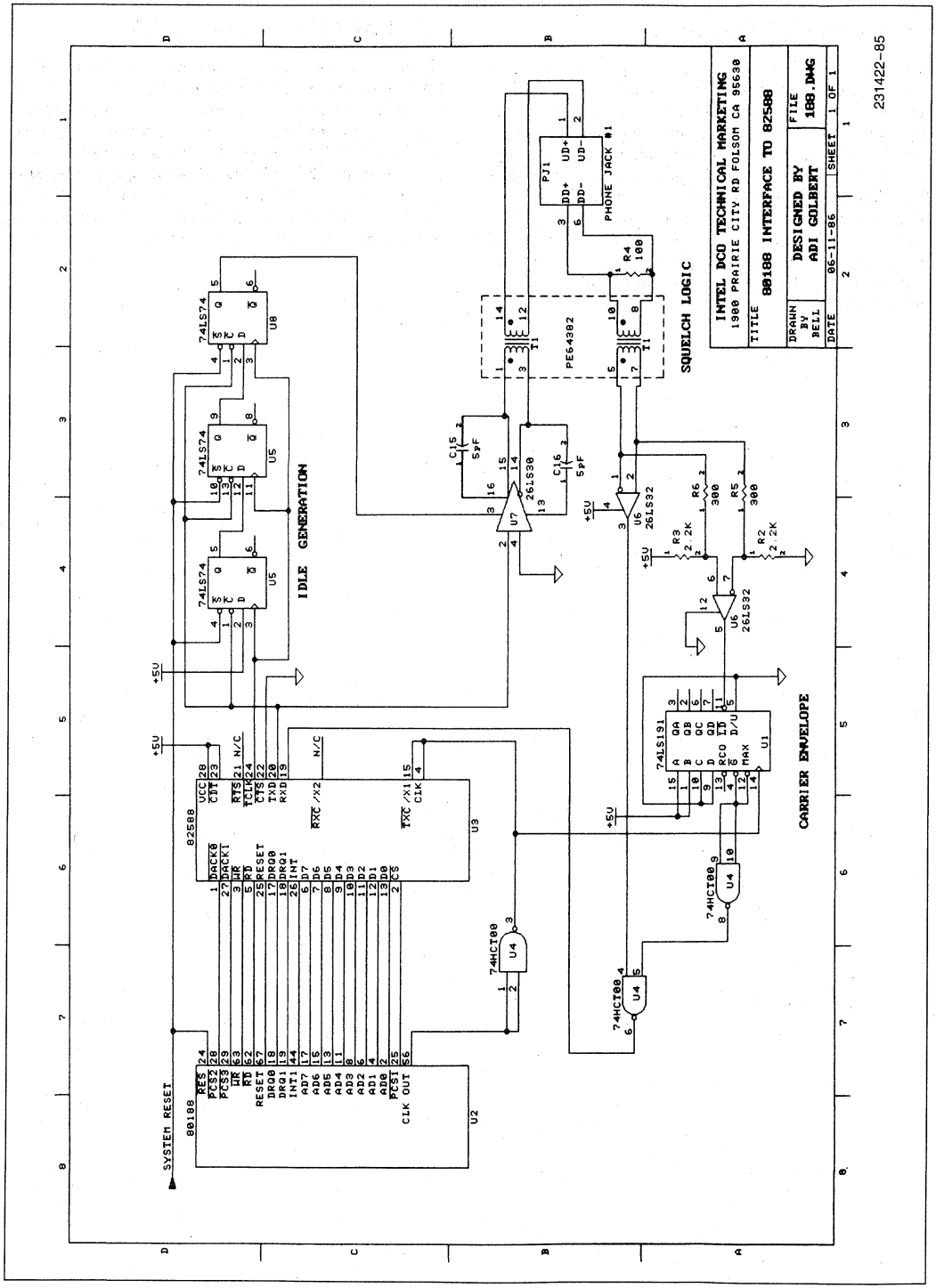


Figure 40. 80188 Interface to 82588

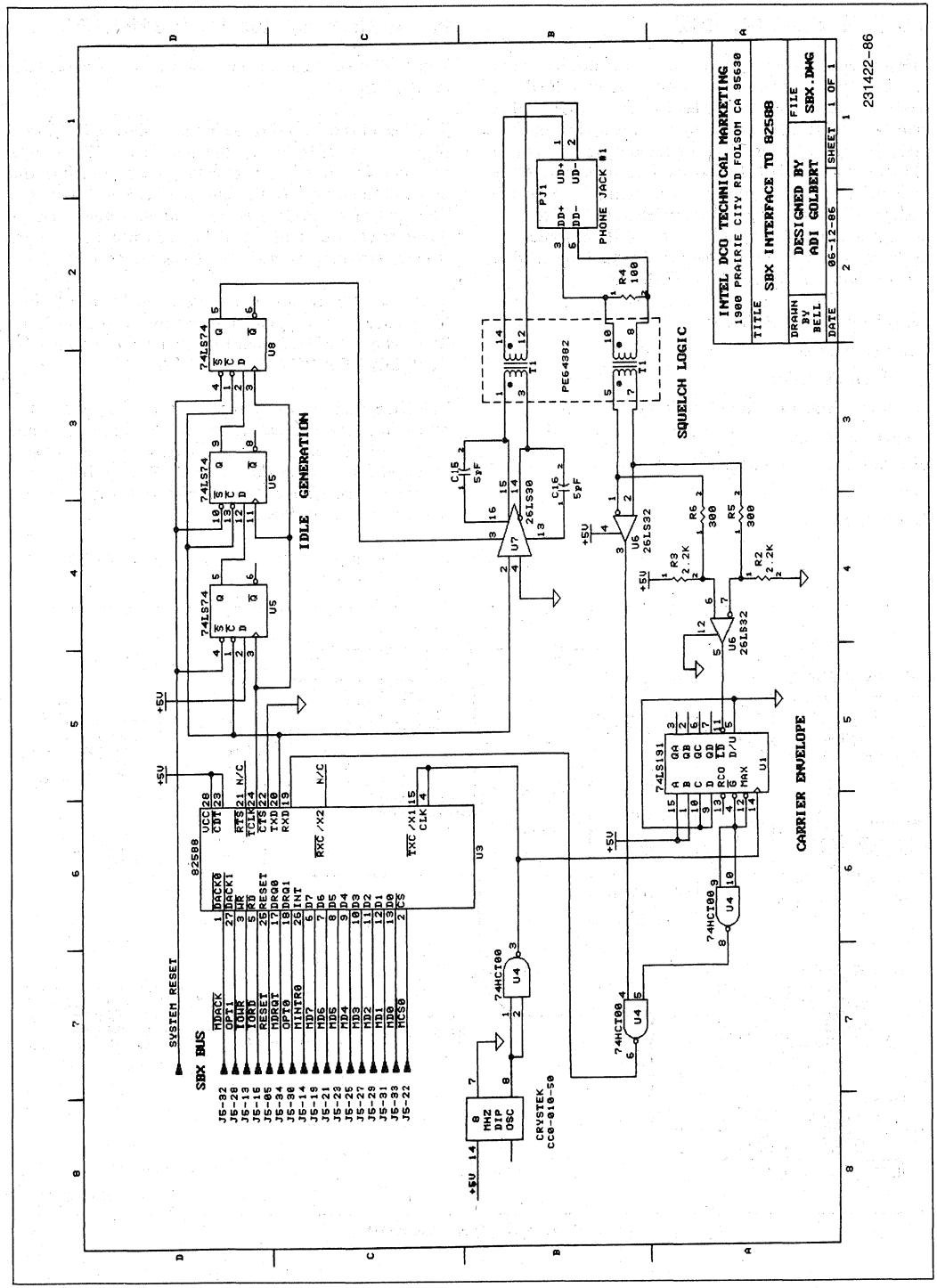


Figure 41. iSBX Interface to 82588

6.0 THE StarLAN HUB

The function of a StarLAN HUB is described in section 2.0. Figure 42 shows a block diagram of a HUB. It receives signals from the nodes (or lower level HUBs) detects if there is a collision, generates the collision presence signal, re-times the signal and sends it out to the higher level HUB. It also receives signals from the higher level HUB, re-times it and sends it to all the nodes and lower level HUBs connected to it. If there is no higher level HUB, a switch on the HUB routes the upstream received signal down to all the lower nodes. The functions performed by a HUB are:

*Receiving signals, squelch
*Carrier Sensing
*Collision Detection
*Collision Presence Signal Generation
*Signal Retiming
*Driving signals on to the cable
*Jabber Function
*Receive protection Timer

6.1 A StarLAN Hub for the IBM/PC

Figure 43 shows the implemention of a 5/6 port HUB for the IBM/PC.

The idea of the following design is to show a HUB that plugs into the IBM/PC backplane. This HUB not only gets its power from the backplane, but also enables the host PC to be one NODE into the StarLAN network. This embedded node scheme enables further savings due to the fact that all the analog interface for this port is saved (receiver, transmitter, transformer, etc).

This kind of board would suit very much a small cluster topology (very typical in departments and small offices) where the HUB board would be plugged into the FILE SERVER PC (PC/XT, PC/AT).

The HUB design doesn't implement the Jabber and the protection timers as called by the 1BASE5 draft standard. Those functions are optional and were not closed during the writing of this AP-NOTE. This HUB does implement the RETIMING circuit which is an essential requirement of StarLAN.

Figures 44 to 49 show a complete set of schematics for the HUB design.

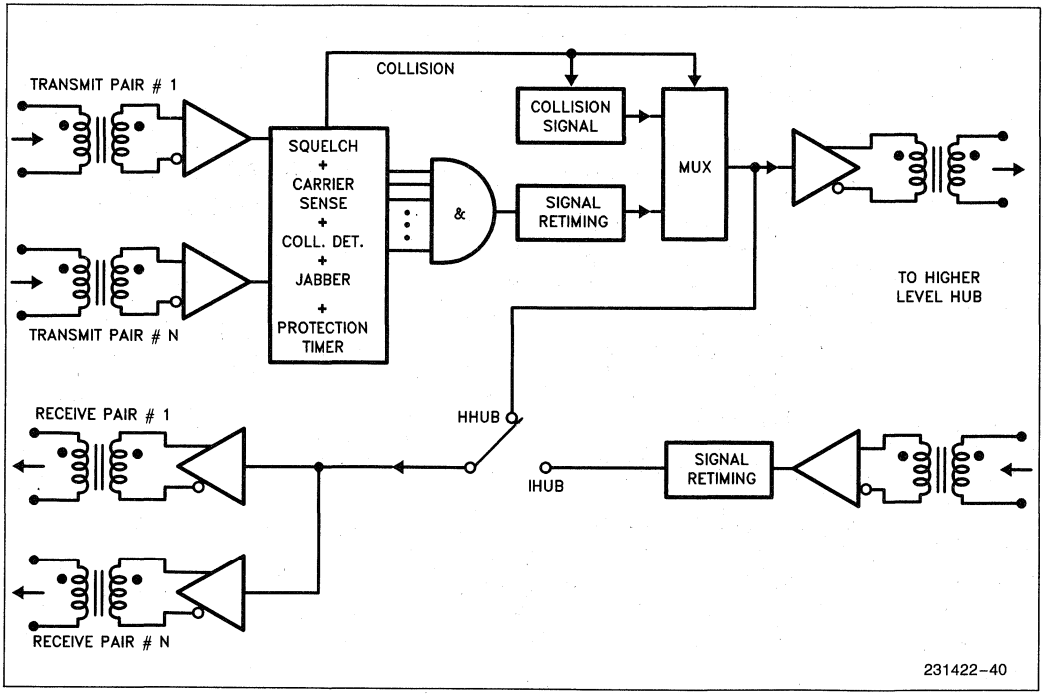
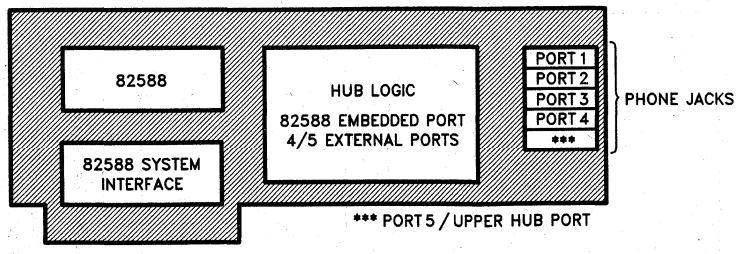


Figure 42. StarLAN HUB



- Low Cost HUB, Uses IBM/PC Power Supply
- 82588, Embedded Port Savings
  - Transformers
  - 422 Drivers
- Functional StarLAN Cluster, For Low Cost/Small Topologies

Figure 43. IBM/PC Resident HUB

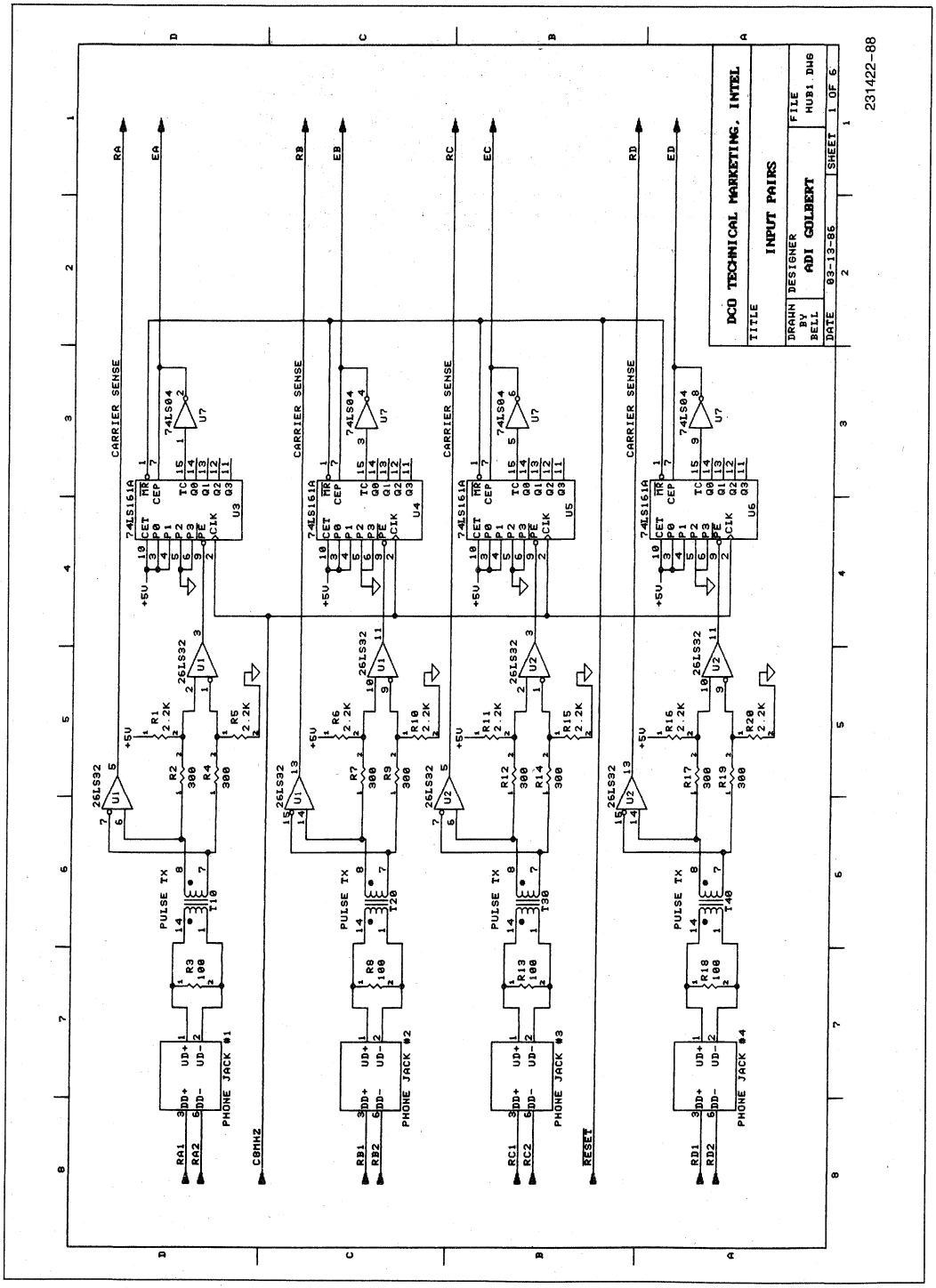

DCD TECHNICAL MARKETING, INTEL
INPUT PAIRS
DESIGNER
ADI GOLBERT
DRAWN BY BELL
DATE 03-13-86
SHEET 1 OF 6
FILE HUB1.DWG
231422-88
CARRIER SENSE
PULSE TX
PHONE JACK #1
PHONE JACK #2
PHONE JACK #3
PHONE JACK #4
COMM2
RESET
74LS161A
74LS04
26LS32
+5V

Figure 44

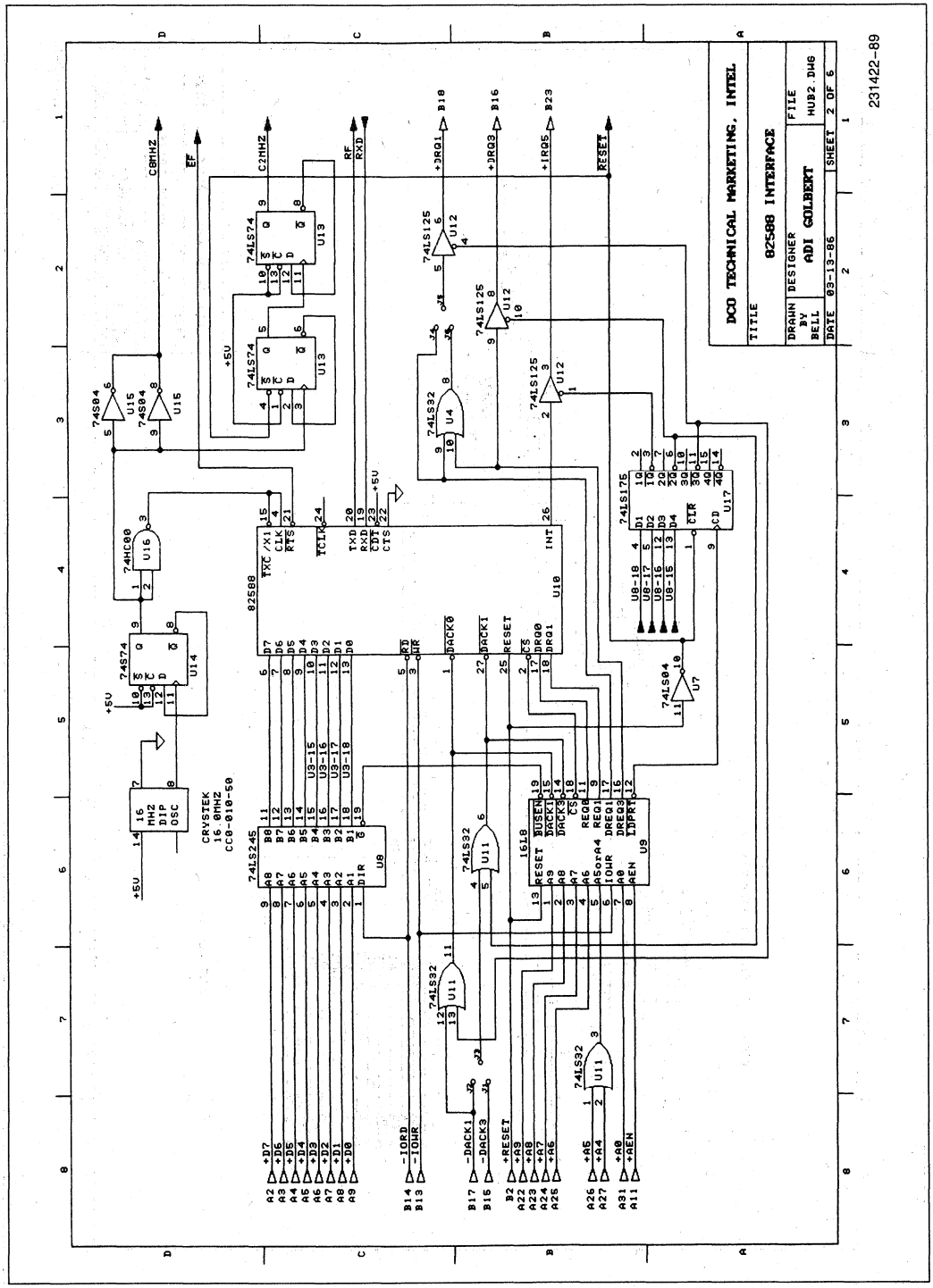

DCO TECHNICAL MARKETING, INTEL
TITLE
82588 INTERFACE
DRAWN BY BELL
DESIGNER ADI GOLBERT
FILE HUB2.DWG
DATE 03-13-86
SHEET 2 OF 6
231422-89
82588
74LS245
74LS32
74LS74
74LS125
74LS175
74LS04
74S04
74HC00
CRYSTEK 16.0MHZ CC8-010-50
16 MHZ DIP OSC
C8MHZ
C2MHZ
RF
RXD
EF
RESET
+5V
+JRQ1
+JRQ3
+IRQ5
-IORD
-IOWR
-DACK1
-DACK3
+RESET

Figure 45

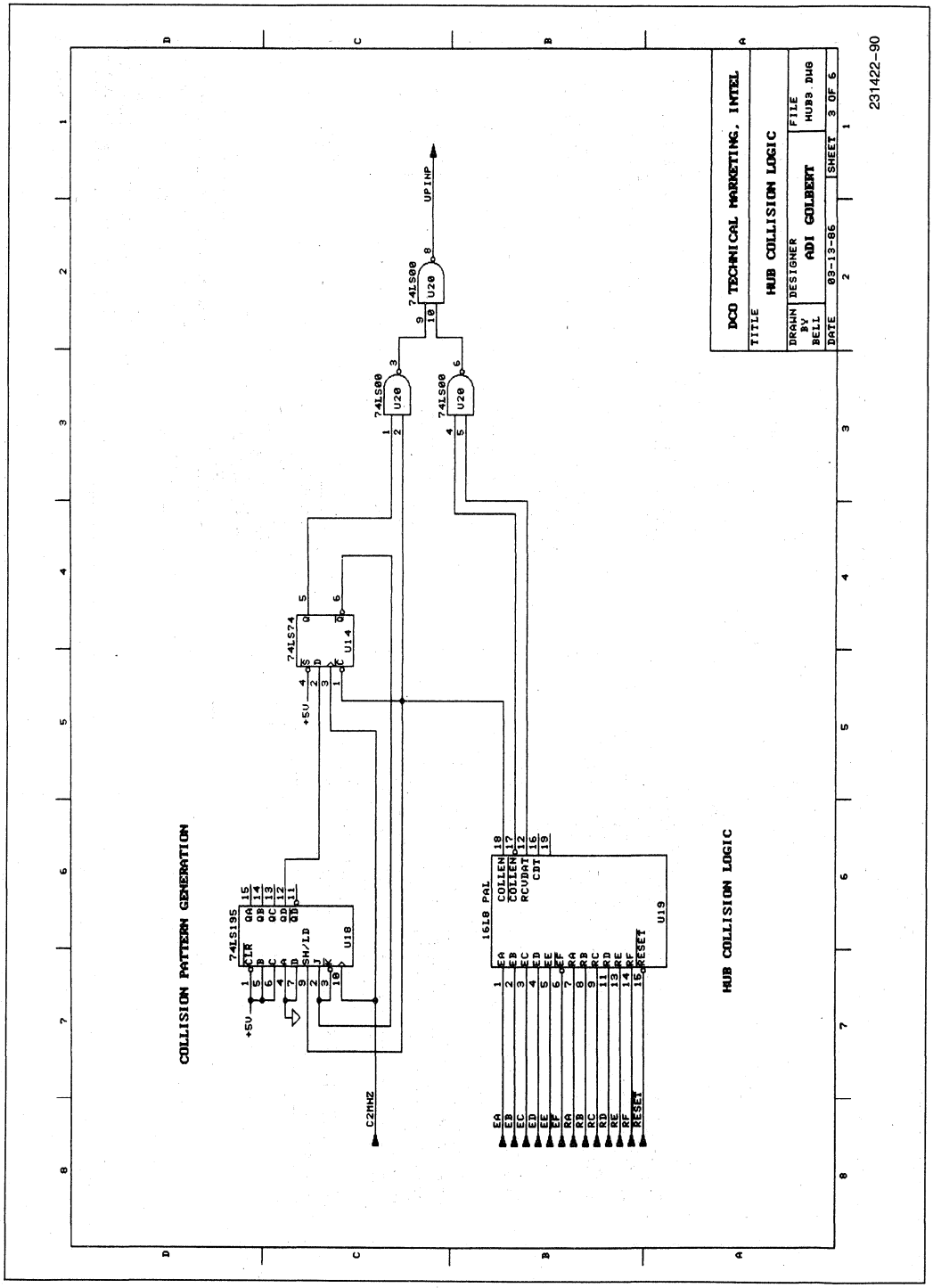

DCO TECHNICAL MARKETING, INTEL
TITLE
HUB COLLISION LOGIC
DRAWN BY BELL
DESIGNER ADI GOLBERT
FILE HUB3.DWG
DATE 03-13-86
SHEET 3 OF 6
231422-90
COLLISION PATTERN GENERATION
HUB COLLISION LOGIC
74LS195
74LS74
74LS00
16L8 PAL
+5V
C2MHZ
UPINP
RESET

Figure 46

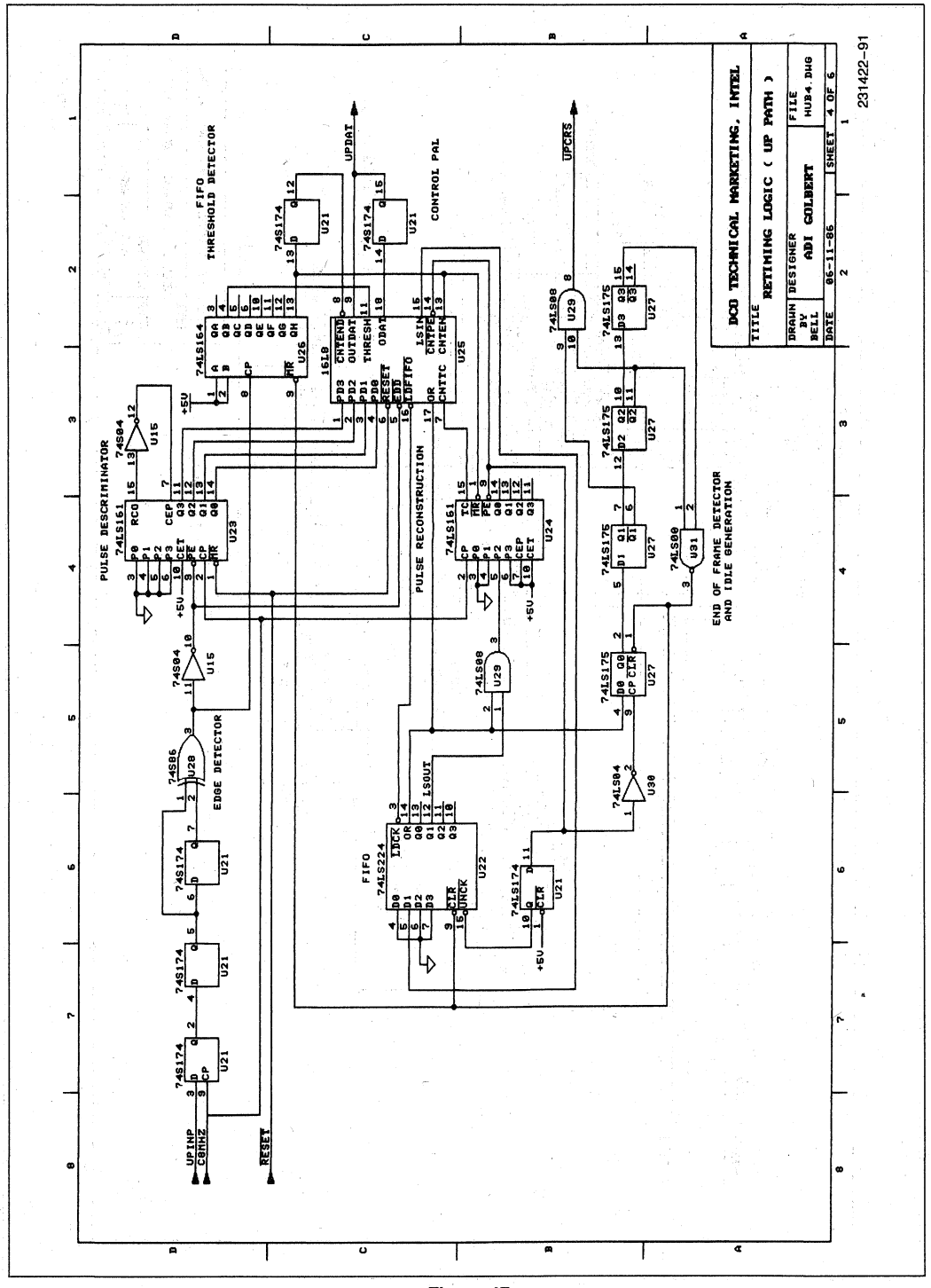

DCU TECHNICAL MARKETING, INTEL
TITLE
RETIMING LOGIC (UP PATH)
DRAWN BY
BELL
DESIGNER
ADI GOLBERT
FILE
HUB4.DWG
DATE
06-11-86
SHEET 4 OF 6
231422-91
PULSE DESCRIMINATOR
FIFO THRESHOLD DETECTOR
CONTROL PAL
PULSE RECONSTRUCTION
EDGE DETECTOR
END OF FRAME DETECTOR AND IDLE GENERATION
UPDAT
UPCRS
UPINP
C8MHZ
RESET
+5V

Figure 47

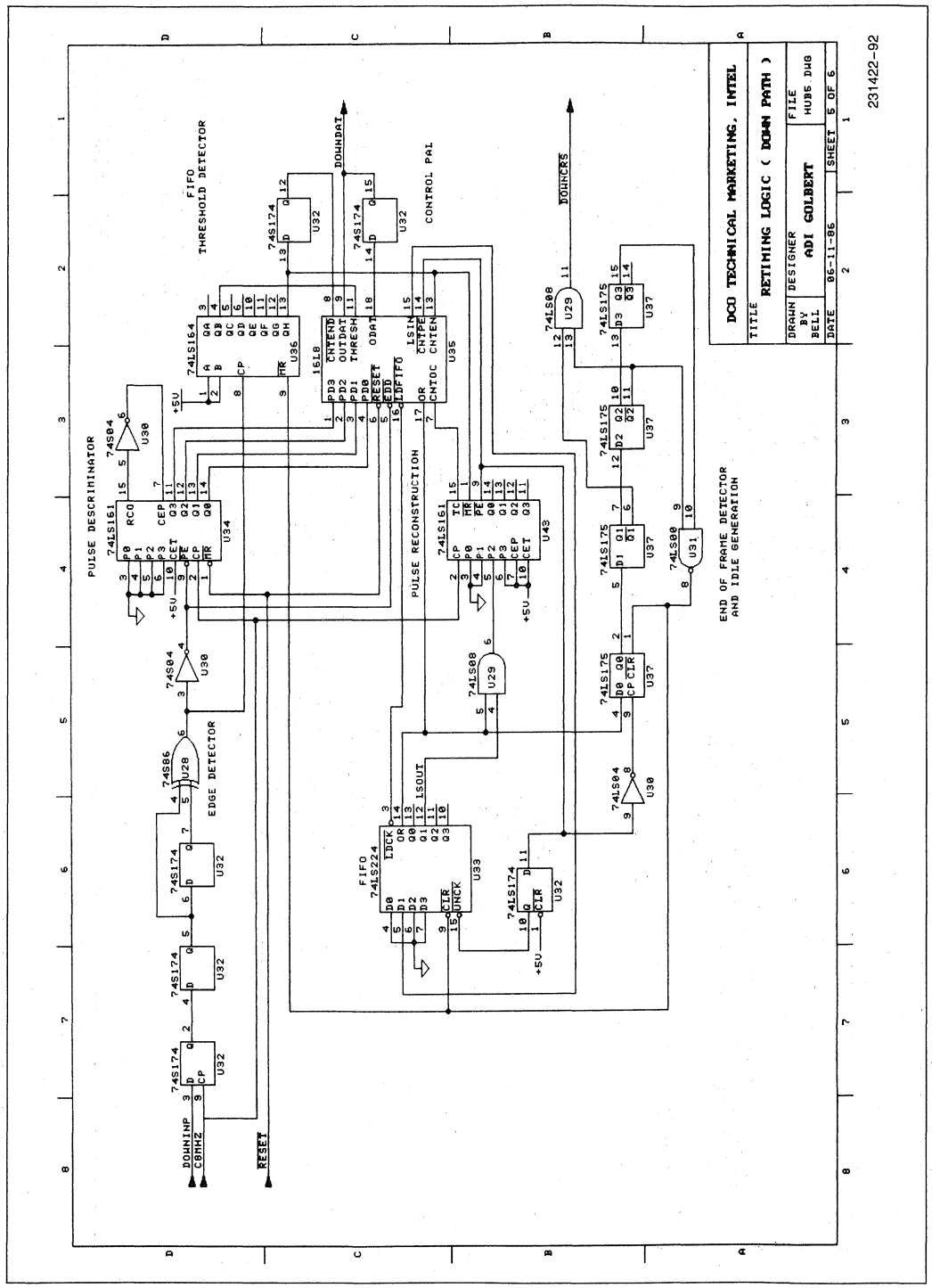

DCD TECHNICAL MARKETING, INTEL
TITLE RETIMING LOGIC (DOWN PATH)
DRAWN BY BELL
DESIGNER ADI GOLBERT
FILE HUBS DWG
DATE 06-11-86
SHEET 5 OF 6
231422-92
PULSE DESCRIMINATOR
FIFO THRESHOLD DETECTOR
CONTROL PAL
PULSE RECONSTRUCTION
EDGE DETECTOR
FIFO
END OF FRAME DETECTOR AND IDLE GENERATION
DOWNDAT
DOWNCKS
DOWNINP
CBRHI2
RESET
+5V

Figure 48

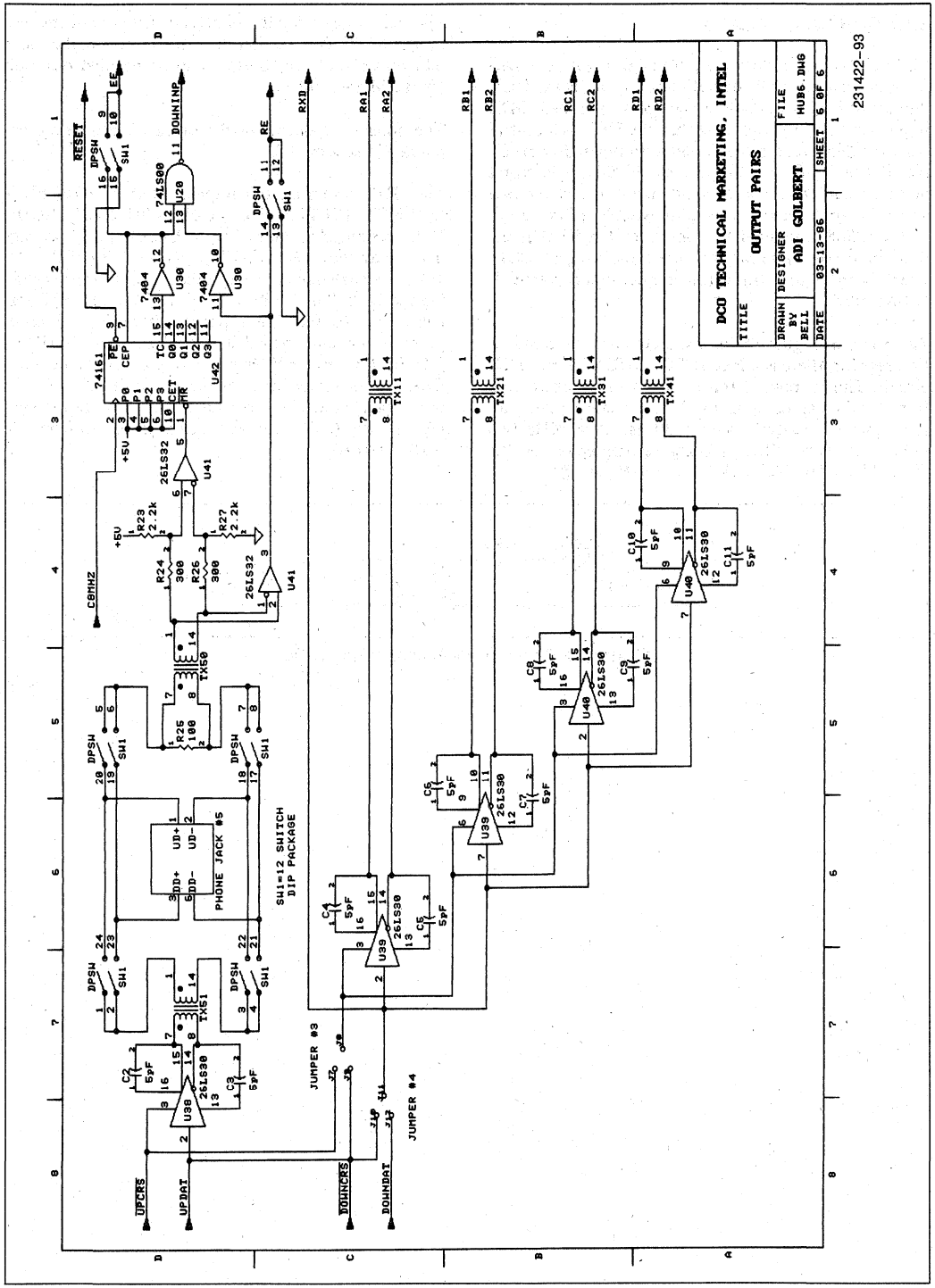

DCD TECHNICAL MARKETING, INTEL
TITLE
OUTPUT PAIRS
DRAWN BY
BEIL
DESIGNER
ADI GOLBERT
FILE
HUB6.DWG
DATE
03-13-86
SHEET 6 OF 6
231422-93
RESET
DPSW
SW1
EE
RE
RXD
DOWNINP
74LS00
U20
7404
U30
74161
U42
261S32
U41
CBMHZ
+5V
R23 2.2k
R27 2.2k
R24 300
R26 300
TX50
R25 100
TX51
TX11
TX21
TX31
TX41
RA1
RA2
RB1
RB2
RC1
RC2
RD1
RD2
261S30
U38
U39
U40
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
C11
5pF
PHONE JACK #5
SW1=12 SWITCH DIP PACKAGE
JUMPER #3
JUMPER #4
UPCKS
UPDAT
DOWNCKS
DOWNDAT

Figure 49

6.1.1 HUB INPUT PORTS

Figure 38 shows a block diagram of an input port. Differently than the implementation in Figure 29 the HUB input port is potentially more complex than the NODE input port. The reason being that the HUB is a central resource and much more sensitive to noise. For example, if the NODE input port would falsely interpret noise on an IDLE line as valid signal, the worst case situation would be that this noise would be filtered out by the 82588 time squelch circuitry, on the HUB by the other hand, this false carrier sense could trigger a COLLISION and a good frame (on another input) potentially discarded.

As shown in Figure 38 immediately after the termination resistor, there is a HIGH FREQUENCY FILTER circuit. The purpose of this circuit is to eliminate high frequency noise components keeping noise jitter into the allocated budget (about ±30 ns). A 4 MHz two pole butterworth filter is being recommended by the IEEE 802.3 1BASE5 task force (see Figure 50).

The time squelch for the NODE board is implemented by the 82588 (see section 3.7) this circuit makes sure that pulses that are shorter than a specified duration will be filtered out.

The other components of the block diagram were explained in section 3.0.

The HUB design doesn't implement the HIGH FREQUENCY FILTER and TIME SQUELCH. In the HUB design as an output of each input port, two signals are available: Rn, En, (RA, RB . . . , EA, EB . . .). The Rn signals are the receive data after the zero crossing receivers. The En lines are CARRIER SENSE signals. The HUB design supports either 5 or 6 input ports, dependent upon if it is configured as IHUB or HHUB. Port RE, EE (Figure 49) is bidirectional, configurable for either input or output. Port RF, EF__ is the embedded 82588 port, and doesn't require the analog circuitry (EF is inverted, being generated from the RTS__ signal).

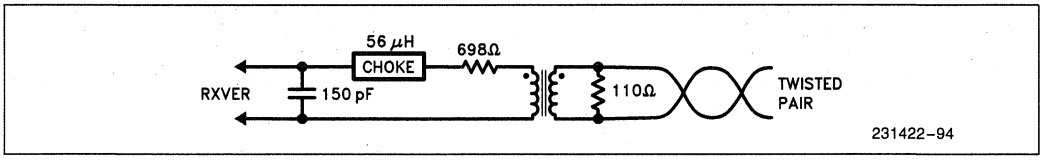


Figure 50. Receiver High Frequency Filter

6.1.2 COLLISION DETECTION

Rn and En signals from each channel are fed to a 16L8 PAL, where the collision detection function is performed.

Collision Detection in the StarLAN HUB is performed by detecting the presence of activity on more than one input channels. This means if the signal En is active for more than one channel, a collision is said to occur. This translates to the PAL equations:

COLLISION DETECTION:

```
CDT = ! (EA & !EB & !EC & !ED & !EE & EF__ #                (only EA active)
         ! EA & EB & !EC & !ED & !EE & EF__ #               (only EB active)
         ! EA & !EB & EC & !ED & !EE & EF__ #               (only EC active)
         ! EA & !EB & !EC & ED & !EE & EF__ #               (only ED active)
         ! EA & !EB & !EC & !ED & EE & EF__ #               (only EE active)
         ! EA & !EB & !EC & !ED & !EE & !EF__ #             (only EF active)
         ! EA & !EB & !EC & !ED & !EE & EF__ );         (none of the inputs active)
```

COLLISION DETECTION SR-FF:

```
COLLEN__ = ! (CDT # COLLEN );                               (set with collision)

COLLEN__ = ! ( RESET__ # COLLEN__ #
               ( !CDT & !EA & !EB & !EC & !ED & !EE & EF__);

                                        ( reset when all inputs inactive )
```

RECEIVE DATA OUTPUT:

```
RCVDAT = (   ( RA # !EA ) & ( RB # !EB ) & ( RC # !EC ) &

             ( RD # !ED ) & (RE # !EE ) & (RF # EF__ ) );

                                        ( output is high if no active input)
```

The COLLEN signal once triggered will stay active until all inputs go quiet. This signal is used externally to either enable passing RCVDAT or the collision presence signal (CPS) to the retiming logic. An external multiplexer using 3 nand gates is used for this function. Note that in this specific implementation the CPS/RCVDAT multiplexer is before the retiming logic, which is different from Figure 42 diagram. StarLAN provides enough BIT-BUDGET delay to allow the CPS signal to be generated through the retiming FIFO. In this HUB implementation it was decided to use this option to make sure that the CPS startup is synchronized with the previously transmitted bit as required by the 1BASE5 draft.

6.1.3 THE LOCAL 82588

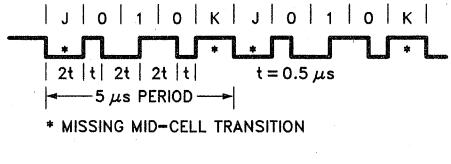
As described before, the purpose of the local 82588 is to enable the Host IBM/PC to also be a node into the StarLAN network. The interface of this 82588 is exactly similar to the one explained in section 5. The RTS__ signal serves as the carrier EF__ signal, and TXD as RF signal. This local node interfaces to the HUB without any analog interface which is a significant saving.

6.1.4 THE COLLISION PRESENCE SIGNAL

The Collision Presence Signal (CPS) is generated by the HUB whenever the HUB detects a collision. It then propagates the CPS to the higher level HUB. The CPS signal pattern is shown in Figure 51. Whenever a StarLAN node receives this signal, it should be able to detect within a very few bit times that a collision occurred. Since the nodes detect the occurrence of a collision by detecting violations in Manchester encoding, the CPS must obviously be a signal which violates Manchester encoding. Section 3.5 shows that the CPS has missing mid-cell transitions occurring every two and a half bit cells. These are detected as Manchester code violations. Thus, the StarLAN node is presented with collision detection indications every two and a half ms. This results in fast and reliable detection of collisions. CPS has a period of 5 ms.

One may wonder why such a strange looking signal was selected for CPS. The rationale is that this CPS looks very much like a valid Manchester signal—edges are 0.5 or 1.0 microsec. apart—resulting in identical radiation, cross-talk and jitter characteristics as a true Manchester. This also makes the re-timing logic for the signals simpler—it need not distinguish between valid Manchester and CPS. Moreover, this signal is easy to generate.

A few important requirements for CPS signal are: a) it should be generated starting synchronized with the last transmitted bit cell. CPS is allowed to start either low or high, but no bit cell of more than 1 microsecond is allowed (Avoid false idles, very long "low" bits). b) once it starts, it should continue until all the input lines to the HUB die out. Typically, when the collision occurs, the multiplexor in the HUB switches from RCV signal to the CPS. This switch is completely asynchronous to the currently being transmitted data, and by such may violate the requirement of not having bit cells longer than 1 μs. In order to avoid those long pulses, the output of the CPS/RCVDAT multiplexer is passed through the retiming circuitry which will correct those long pulses to their nominal value. The reason for restriction b) is to ensure that the CPS is seen by all nodes on the network since it is generated until every node has finished generating the Jam pattern.



- Collision Presence Signal (CPS) is generated by the HUB when it detects more than one input line active.
- CPS violates Manchester encoding rules—due to missing mid-cell transitions—hence is detected as a collision by the DTE (82588).

Choice of Collision Presence Signal

- It is a Manchester look-alike signal—edges are 0.5 or 1.0 μs apart.
  - — Identical radiation, crosstalk and jitter characteristics
  - — Eases retiming of the signal in the HUB
- It is easy to generate—1.5 TTL pack, or in a PAL

Figure 51. Collision Presence Signal

CPS is generated using a 4-bit shift register and a flip-flop as shown in Figure 52. It works off a 2 MHz clock. A closer look at the CPS waveform shows that it is inverse symmetric within the 5 μs period. The circuit is a 5-bit shift register with a complementary feedback from the last to the first bit. The bits remain in defined states (01100) till collision occurs. On collision the bits start rotating around generating the pattern of 0011011001, 0011011001, 00110 ... with each state lasting for 0.5 μs.

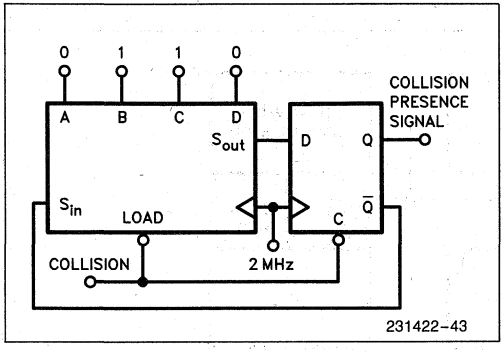


Figure 52. Collision Presence Signal Generation

6.1.5 SIGNAL RETIMING

Whenever the signal goes over a cable it suffers jitter. This means that the edges are no longer separated by the same 0.5 or 1.0 μs as at the point of origin. There are various causes of jitter. Drivers, receivers introduce some shifting of edges because of differing rise and fall times and thresholds. A random sequence of bits also produces a jitter which is called intersymbol interference, which is a consequence of different propagation delays for different frequency harmonics in the cable. Meaning short pulses have a longer delay than long ones. A maximum of 62.5 ns of jitter can accumulate in a StarLAN network from a node to a HUB or from a HUB to another HUB. The following values show what are the jitter components:

Transmitter skew	±10 ns
Cable Intersymbol interference	±9 ns
Cable Reflections	±8 ns
Reflections due to receiver termination mismatch	±5 ns
HUB fan-in, fan-out	±5 ns
Noise	±25.5
Total	±62.5 ns

It is important for the signal to be cleaned up of this jitter before it is sent on the next stretch of cable because if too much jitter accumulates, the signal is no longer meaningful. A valid Manchester signal would, as a result of jitter, may no longer be decodable. The process of either re-aligning the edges or reconstructing the signal or even re-generating the signal so that it once again "looks new" is called re-timing. StarLAN requires for the signal to be re-timed after it has travelled on a segment of cable. In a typical HUB two re-timing circuits are necessary; one for the signals going upstream towards the higher level HUB and the other for signals going downstream towards the nodes.

6.1.6 RETIMING CIRCUIT, THEORY OF OPERATION

This section will discuss the principles of designing a re-timing circuit. Figure 53 shows the block diagram of a re-timing circuit. The data coming in is synchronized using an 8 MHz sampling clock. Edges in the waveform are detected doing an XOR of two consecutive samples. A counter counts the number of 8 MHz clocks between two edges. This gives an indication of long (6 to 10 clocks) or short (3 to 5 clocks) pulses in the received waveform. Pulses shorter than 3 clocks are filtered out. Every time an edge occurs, the length—(S)hort or (L)ong—of the pulse is fed into the FIFO. Retiming of the waveform is done by actually generating a new waveform based on the information being pumped into the FIFO. The signal regeneration unit reads the FIFO and generates the output waveform out of 8 MHz clock pulses based on what it reads, either short or longs. In summary every time a bit is read from the fifo, it indicates that a transition needs to occur, and when to fetch the next bit. When idle the output of the retiming logic starts with a "high" level.

FIFO	Output
empty	1111
S	0000
S	1111
L	00000000
L	11111111

It can be seen that the output always has edges separated by 4 or 8 clock pulses—0.5 or 1.0 μs.

The FIFO is primarily needed to account for a difference of clock frequencies at the source and regeneration end. Due to this difference, data can come in faster or slower than the regeneration circuit expects. A 16 deep FIFO can handle frequency deviations of up to 200 ppm for frame lengths up to 1600 bytes. The FIFO also overcomes short term variations in edge separation. It is essential that the FIFO fills in up to about half before the process of regeneration is started. Thus, if the regeneration is done at a clock slightly faster than the source clock, there is always data in the FIFO to work from. That is why the FIFO threshold detect logic is necessary, which counts 8 edges and then enables the signal regeneration logic.

Example:

Input Waveform ...1111000111100000001111111111000111100...

Input into the FIFO <S> <S> <L> <L><S> <S>

Regenerated Output:

Output: ...1111100001111000000001111111110000111...

FIFO: <S> <S> <L> <L> <S> <S>

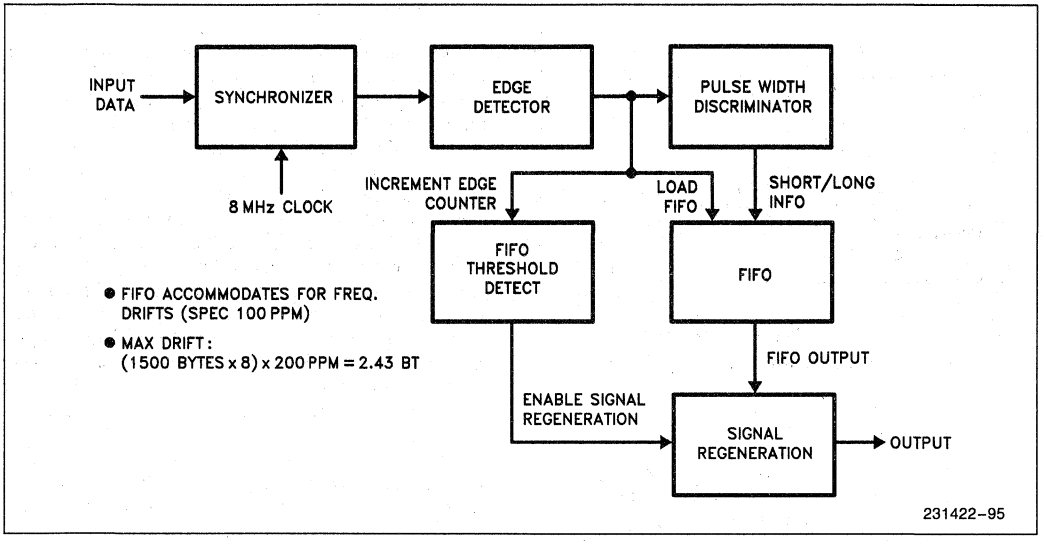


Figure 53. Retiming Block Diagram

6.1.7 RETIMING CIRCUIT IMPLEMENTATION

The retiming circuit implementation can be seen in Figures 47, 48. Both figures implement exactly the same function, one for the upstream, and the other for the downstream. The retiming circuit was implemented using about 8 SSI, MSI TTL components, one fifo chip and one PAL. The purpose of implementing this function with discrete components was to show the implementation details. The discussion of the implementation will refer to Figure 47 for unit numbers.

The signal UPIMP which is an output of the HUB multiplexing logic, is asynchronous to the local clock. This signal is synchronized by two flip-flops and fed into an edge generation logic (basically an XOR gate that compares the present sample with the previous one). On every input transition a 125 ns pulse will be generated at the output of the edge detector (U28). This pulse will reset the 74LS161 counter that is responsible for measuring pulse widths (in X8 clock increments). The output of the pulse discriminator will reflect the previous pulse width every time a new edge is detected. The following events will take place on every detected edge:

1. U26 which is the threshold detector will shift one "1" in. The outputs of U26 will be used by the control PAL to start the reconstruction process.
2. The output of U23 which specifies the last pulse width will be input into the control PAL for determining if it was a long or short pulse. The result of this evaluation will be the LSIN signal which will be loaded into the fifo (U22).

U22 is the retiming FIFO, it is 16x4 fifo, but only one bit is necessary to store the SHORT/LONG information.

CONTROL LOGIC PAL functions (U25):

Signals definition:

INPUTS:

PD0..PD3: Outputs of the pulse descriminator, indicate the width of the last measured pulse.

EDD__: Output of the edge detector, pulse of 125 ns width, indicates the occurrence of an edge in the input data.

THRESH: Output of the threshold logic, indicates at least one bit was already received.

CNTEN: Output of the Threshold logic, indicates 7 bits have been loaded into the FIFO, and that signal reconstruction can begin.

CNTEND: The same signal as before delayed by one clock.

OUTDAT: Output of the retiming logic, is feedback into the PAL to implement a clocked T-FF.

RESET__: Resets the retiming logic.

CNTTC: Terminal count of the reconstruction counter, indicating that reconstruction of a new bit will get started.

OR: Output of the FIFO indicating, that the FIFO is empty and that IDLE generation can get started.

OUTPUTS:

LDFIFO__: Loads SHORT/LONG indications into the FIFO.

LSIN: Indicates SHORT/LONG

CNTPE__: Loads FIFO SHORT/LONG output into the reconstruction counter.

ODAT: Together with the external U21 flip-flop and OUTDAT implement a clocked T-FF.

Loading the FIFO will be done every time there is an edge, we have passed the one bit filter threshold level, and the pulse width is longer than two 8X clocks. This one bit threshold level serves as a time domain filter discarding the first received preamble bit.

```
LDFIFO_ = ! ( PD1 # PD2 # PD3 ) & !EDD_ & THRESH ) ;
```

Whenever there is an edge, we are above the first received bit threshold and the pulse width is longer than "1" the fifo is loaded.

```
LSIN    = ! (PD3 # (PD2 & PD0) # (PD2 & PD1)) ;
```

Every pulse longer than 6 is considered to be a long pulse.

```
CNTPE_  = ! ( (CNTEN & !CNTEND) # CNTTC ) ;
```

The reconstruction counter is loaded in two conditions:

Whenever CNTEN comes active, meaning the FIFO threshold of seven was exceeded.
Whenever the terminal count of U24 is active meaning a new pulse is going to be reconstructed.

```
ODAT    = !RESET_ # ( !CNTPE_ & !OUTDAT)    (A)
                  # ( CNTPE_ &  OUTDAT)    (B)
                  # ( !CNTPE_ & !OR)        (C)
```

Minterm (A) and (B) implement a T-FF, whenever CNTPE is "low" ODAT will toggle. The external U21 is part of this flip-flop. Minterm (C) insures the output of the flip-flop will go inactive "high" when the FIFO is empty. RESET causes the output to go "high" on initialization.

U24 as mentioned is the reconstruction counter. This counter is loaded by the control logic with either 8 or 12, it counts up and is reloaded on terminal count. Essentially generating at the output nominal length longs and shorts.

U22 is the retiming FIFO, and its function as mentioned is to accommodate frequency skews between the incoming and outgoing signal.

U27 is the IDLE generation logic. The purpose of this logic is to detect when the FIFO is empty, meaning that no more data needs to be transmitted. On detection of this event this component will generate 2 ms of IDLE time. On the end of IDLE the whole retiming logic will be reset.

6.1.8 DRIVER CIRCUITS

The signal coming out of the RETIMING LOGIC is fed into 26LS30s and pulse transformers to drive the twisted pair lines (See section 5.0 for details).

6.1.9 HEADER/INTERMEDIATE HUB SWITCH

As seen on Figure 43 this hub can be configured as either an intermediate hub, or a Header one. One of the phone jacks, more specifically JACK #5 is either an input port or an output one. In order to implement this function, an 8 position DIP SWITCH (SW1) is used. The phone jacks are marked with UD, DD notation, meaning upstream data, and downstream data respectively. As specified in the StarLAN 1BASE5 draft NODES transmit data on UD pair, and HUBS on the DD pair. Switch SW1 has the function to invert UD, DD in PHONE JACK #5 to enable it to be either input or output port.

6.1.10 JABBER FUNCTION

This design does not implement the jabber unit but it is described here for completeness. IEEE 802.3 does not mandate this feature, but it is "Strongly Recommended". The jabber function in the HUB protects the network from abnormally long transmissions by any node.

Two timers T1, T2 are used by the JABBER function. They may be implemented either as local timers (one for each HUB port) or as global timers shared by all ports. After detecting an input active, timers T1, T2 will be started, and T1 will time out after 25 to 50 ms. T2 will time-out after 51 to 100 ms. During T2 time, after T1 expired, the HUB will send the CP-PATTERN informing any jamming stations to quit their transmissions. If on T2 time-out there are still jamming ports, their input is going to be disabled. A disabled port, will be reenabled whenever its input becomes again active and the downward side is idle.

The following is an explanation of the requirement that the downward side be idle to reenable an input port. Consider the case of Figure 54. The figure shows a two port HUB. Port A has two wires A_u, A_d for the up and down paths. Port B has B_u, B_d respectively. Port C is the output port, that broadcasts to the other HUBs higher in the hierarchy. Consider the case as shown, where B_u and B_d are shorted together. Suppose the case that port A_u is active. Its signal will propagate up in the hierarchy through C_u and come down from C_d to A_d, and B_d. Due to the short between B_d and B_u the signal will start a loop, that will first cause a collision and jam the network forever. This kind of fault is taken care of by the jabber circuitry. T1 and T2 will expire, causing the jabber logic to disable B_u input. Upon this disabling B_u is going to go Idle and be a candidate for future enabling. Suppose now that A_u is once again active. If the reenable condition would not require C_d to be IDLE, B_u would be reenabled causing the same loop to happen once again. Note that in this case C_d will be active before B_u causing this port to continue to be disabled and avoiding the jamming situation (Figure 55) gives a formal specification of the jabber function).

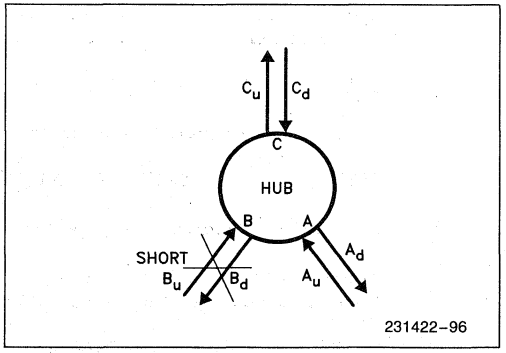


Figure 54. Jabber Function

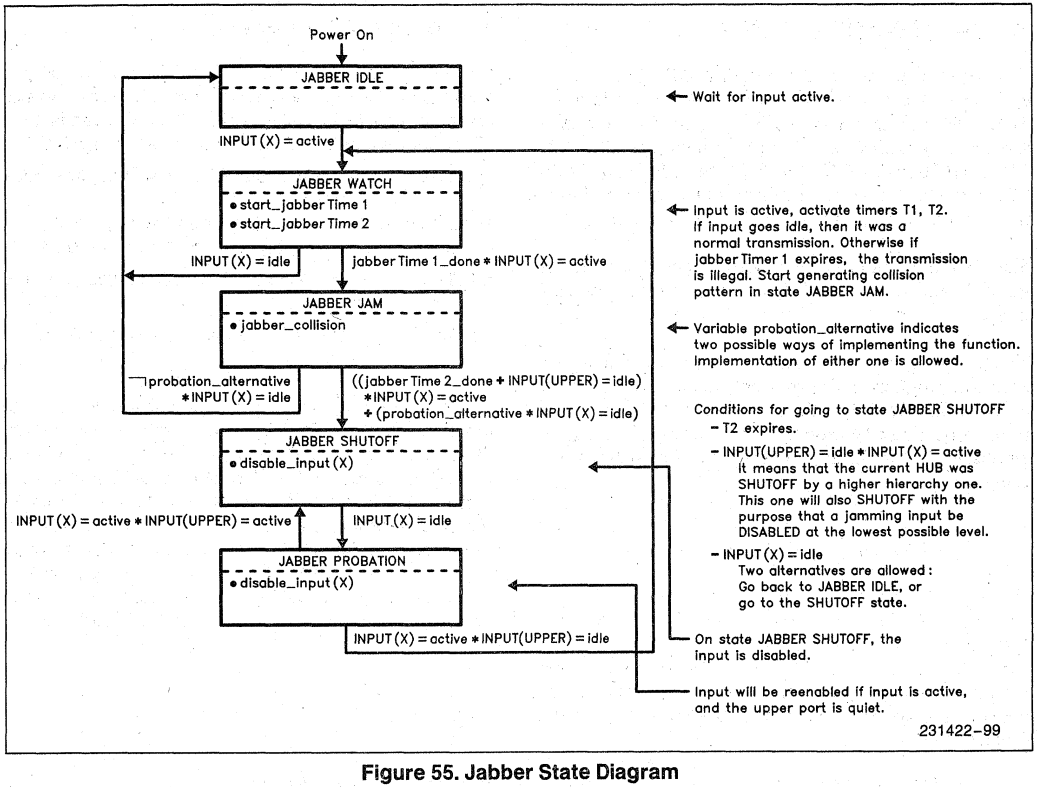


Figure 55. Jabber State Diagram

6.1.11 HUB RECEIVER PROTECTION TIMER

On the end of a transmission, during the transition from IDLE to high impedance state, the transmitter will exhibit an undershoot and/or ringing, as a consequence of transformer discharge. This undershoot/ringing will be transmitted to the receiver which needs to protect itself from false carriers due to this effect. One way of implementing this protection mechanism is to implement a blind timer, which upon IDLE detection will "blind" the receiver for a few microseconds.

Causes of the transmitter undershoot/ringing:

1. Difference in the magnitudes of the differential output voltage between the high and the low output stages.
2. Waveform assymmetry due to transmitter jitter.
3. Transmitter and receiver inductance (transformer L).
4. Two to three microseconds of IDLE pattern.

All the described elements will contribute to energy storage into the transformer inductor, which will discharge during the transition of the driver to high impedance.

The blinding timer is currently defined to be from 20 to 30 microseconds for the HUBs, being from 0 to 30 microseconds for the nodes (optional). The 82588 has built-in this function. It won't receive any frames for an inter-frame-spacing (IFS) from the idle detection.

6.1.12 HUB RELIABILITY

Since the StarLAN HUBs form focal points in the network, it is important for them to be very reliable, since they are single points of failure which can affect a number of nodes or can even bring down the whole network. StarLAN 1BASE5 draft requires HUBs to have a mean time between failures (MTBF) of at least 5 years of continuous operation.

7.0 SOFTWARE DRIVER

The software needed to drive the 82588 in a StarLAN environment is not different from that needed in a generic CSMA/CD environment. This section goes into specific procedures used for operations like TRANSMIT, RECEIVE, CONFIGURE, DUMP, ADDRESS SET-UP, etc. A special treatment will be given to interfacing with the IBM PC—DMA, interrupt and I/O.

Since all the routines were written and tried out in PLM-86 and ASM-86, all illustrations are in these languages.

The following software examples are pieces of an 82588 exerciser program. This program's main purpose was to exercise the 82588 functionality and provide the functions of traffic generation and monitoring. By such the emphasis was on speed and accuracy of statistics gathering.

7.1 Interfacing to IBM PC

The StarLAN board interfaces to the CPU, DMA controller and the interrupt controller on the IBM PC system board. The software to operate the 82588 runs on the system board CPU. The illustrated routines in this section show exactly how the software interface works between the system resources on the IBM PC and the StarLAN board.

7.1.1 DOING I/O ON IBM PC

The safest way to use the PC monitor as an output device and the keyboard as the input device is to use them through DOS system calls. The following is a set of routines which are handy to do most of the I/O:

key$stat	—to find out if a new key has been pressed
keyin$noecho	—to read a key from the keyboard
char$out	—to display a character on the screen
msg$out	—to display a character string on the screen
line$in	—to read in a character string from the keyboard

The exact semantics and the protocol for doing these functions through DOS system calls is shown in the listing in Figure 56. Refer to the DOS Manual for a more detailed description. To make a DOS system call, register AH of 8088 is loaded with the call Function Number and then, a software interrupt (or trap) 21 hex is executed. Other 8088 registers are used to transfer any parameters between DOS and the calling program. The code is written in Assembly language for register access. Let us see an example of the 'msg$out' routine:

```
lds dx,STRING_POINTER    ; load pointer to string in reg. ds:dx
mov ah,09h               ; 9 = function number for string o/p
int 21h                  ; DOS System Call
```

These procedures are called from another module, written in a higher level language like PLM-86. The parameters are transferred to the ASM-86 routines on the stack.

Examples of using the I/O routines:

```
KEY_STATUS = key$stat;                    /* INQUIRE KEYBOARD STATUS      */
NEW_KEY = keyin$noecho;                   /* INPUT NEW KEY                */
call line$in(@LINE_BUFFER);               /* STRING INPUT                 */
call char$out(CHAR_OUT);                  /* TO OUTPUT CHAR_OUT ON SCREEN*/
call msg$out(@('THIS IS A MESSAGE.$'));   /* OUTPUT STRING                */
                                          /* NOTE $ TERMINATOR            */
```

```
/*-----------------------------------------------------------------------*/
/*     Declarations for external IBM PC I/O routines                     */
/*-----------------------------------------------------------------------*/

key$stat: procedure byte external;          /* key status routine */
end key$stat;

key$in$noecho: procedure byte external;       /* console input routine */
end key$in$noecho;

char$out: procedure(char) external;    /* console output routine */
declare char byte;
end char$out;

msg$out: procedure(msg$ptr) external;         /* console string output routine
declare msg$ptr pointer;
end msg$out;

line$in: procedure(line$ptr) external;        /* console string input routine
declare line$ptr pointer;
end line$in;


Assembly Language implementation of the routines

$TITLE(IBM/PC  DOS CALLS PROCEDURES)

            NAME    DOSPROCS
;
DGROUP              GROUP   DATA
CGROUP              GROUP   CODE
;
DATA                SEGMENT WORD PUBLIC 'DATA'
DATA                ENDS
;
DOS         EQU     21H
;
CODE                SEGMENT WORD PUBLIC 'CODE'
                    ASSUME  CS:CGROUP,DS:DGROUP
```

231422-58

```
;
; CHAR$OUT: PROCEDURE(CHAR) EXTERNAL;
; DECLARE CHAR BYTE;
; END CHAR$OUT;
; Outputs character to the screen.
; DOS system call 2
;
CHAR                EQU     [BP+4]        STACK
;
CHAROUT             PROC    NEAR          +------+
              PUBLIC CHAROUT              ! CHAR ! x
              PUSH   BP                   +------+
              MOV    BP,SP                !IP lo ! x-1
              MOV    DL,CHAR              +------+
              MOV    AH,2                 !IP hi ! x-2
              INT    DOS                  +------+
              POP    BP                   !BP lo ! x-3
              RET    2                    +------+
CHAROUT             ENDP                  !BP hi ! x-4   <--SP
;                                         +------+
; KEYIN$NOECHO: PROCEDURE BYTE EXTERNAL;
; END KEYIN$NOECHO;
; Reads character without echoing to display
;
KEYINNOECHO PROC    NEAR
              PUBLIC KEYINNOECHO
              MOV    AH,8                 (DOS call 8)
              INT    DOS
              RET
KEYINNOECHO ENDP
```

Figure 7-56. I/O Routines for IBM/PC (continued)

231422-59

Figure 56. I/O Routines for IBM/PC

```
;
;
; MSG$OUT: PROCEDURE(MSG$PTR) EXTERNAL;
; DECLARE MSG$PTR POINTER;
; END MSG$OUT;
; /* NOTE: MESSAGE IS TERMINATED WITH A DOLLAR SIGN */
; MSG$PTR is double word pointer SEG:OFFSET

MSG_L               EQU     [BP+4]
MSG_H               EQU     [BP+6]
;
MSGOUT              PROC    NEAR
            PUBLIC  MSGOUT
;
            PUSH    BP
            MOV     BP,SP
            MOV     DX,MSG_L
            PUSH    DS
            MOV     AX,MSG_H
            MOV     DS,AX
            MOV     AH,9                    (DOS call 9)
            INT     DOS
            POP     DS
            POP     BP
            RET     4
MSGOUT              ENDP

;
; LINE$IN: PROCEDURE(LINE$PTR) EXTERNAL;
; DECLARE LINE$PTR POINTER;
; END LINE$IN
;

LINE_L              EQU     [BP+4]
LINE_H              EQU     [BP+6]
;
LINEIN              PROC    NEAR
            PUBLIC  LINEIN
            PUSH    BP
            MOV     BP,SP
            PUSH    DS
            MOV     AX,LINE_H
            MOV     DS,AX
            MOV     DX,LINE_L
            MOV     AH,10                   (DOS call 10)
            INT     DOS
            POP     DS
            POP     BP
            RET     4
LINEIN              ENDP                                    231422-60


  ;
  ; KEY$STAT: PROCEDURE BYTE EXTERNAL;
  ; END KEY$STAT;
  ; Indicates whether any keyboard key was pressed.

  KEYSTAT           PROC    NEAR
              PUBLIC  KEYSTAT
              MOV     AH,11               (DOS call 11)
              INT     DOS
              RET
  KEYSTAT           ENDP
  ;
  ;
  CODE              ENDS
              END
                                                          231422-61
```

Figure 56. I/O Routines for IBM/PC (Continued)

7.2 Initialization and Declarations

Figure 57 shows some declarations describing what addresses the devices have and also some literals to help understand the other routines in this section.

Figure 58 shows the initialization routines for the IBM PC and for the 82588. It also shows some of the typical values taken by the memory buffers for Configure, IA_Set, Multicast and transmit buffers.

Following are some literal declarations that are used in the procedure examples

```
Following are some literal declarations that are used in the
procedure examples
  declare

    cs_588          literally '0300h' , /* 82588 COMMAND/STATUS       */
    brd_port            literally '0301h' , /* DMA/INTERUPT ENABLE PORT   */
    pic_mask            literally '021h'  , /* 8259A MASK REGISTER        */
    pic_ocw2            literally '020h'  , /* 8259A COMMAND WORD 2       */
    dma_mask            literally '0ah'   , /* 8237A MASK REGISTER        */
    dma_mode            literally '0bh'   , /* 8237A MODE REGISTER        */
    dma_flff            literally '0ch'   , /* 8237A 1ST/2ND BYTE FLOP    */
    dma_addr_1          literally '02h'   , /* 8237A CHANNEL 1 ADDR. REG. */
    dma_bc_1            literally '03h'   , /* 8237A CHANNEL 1 BYTE COUNT */
    dma_addrh_1         literally '083h'  , /* CHANNEL 1 PAGE REGISTER    */
    dma_addr_3          literally '06h'   , /* 8237A CHANNEL 3 ADDR. REG. */
    dma_bc_3            literally '07h'   , /* 8237A CHANNEL 3 BYTE COUNT */
    dma_addrh_3         literally '082h'  , /* CHANNEL 3 PAGE REGISTER    */
    dma_on_1            literally '01h'   , /* START CHANNEL 1            */
    dma_on_3            literally '03h'   , /* START CHANNEL 3            */
    dma_off_1           literally '05h'   , /* STOP CHANNEL  1            */
    dma_off_3           literally '07h'   , /* STOP CHANNEL  3            */
    enable_588          literally '0dfh'  , /* UNMASK INTERRUPT LEVEL 5   */
    seoi_pico           literally '065h'  , /* SPECIFIC EOI LEVEL 5       */
    tx_dir              literally  '1'    , /* MEMORY TO 82588            */
    rx_dir              literally  '0'    , /* 82588 TO MEMORY            */
    dma_rx_mode_1       literally '045h'  , /* RX ON CHANNEL # 1          */
    dma_rx_mode_3       literally '047h'  , /* RX ON CHANNEL # 3          */
    dma_tx_mode_1       literally '049h'  , /* TX ON CHANNEL # 1          */
    dma_tx_mode_3       literally '04bh'  , /* TX ON CHANNEL # 3          */
                                                                  231422-62
```

Figure 57. Literal Declarations

Initialization Routines

```
Initialization routines

/* SYSTEM INITIALIZE */

sys_init: procedure;

   call set$interrupt (13,intr_588);                  /* BASE 8, LEVEL 5    */
   output(pic_mask) = input(pic_mask) and enable_588; /* ENABLE 588 INTERR. */
   output(pic_ocw2) =    seoi_pico;                   /* ACKS PENDING INTERR*/

   wr_ptr,rd_ptr,fifocnt=0;                           /* RESET STATUS FIFO  */

   /************************************************/
   /* CONVERT SEG:OFFSET FORMAT TO 20 BIT ADDRESSES   */
   /* FOR ALL THE BUFFERS                             */
   /************************************************/

   iaset_dma_addr = convert_20bit_addr(@ia_set_buff_588(0));
   cnf_dma_addr   = convert_20bit_addr(@config_588(0));
   dmp_dma_addr   = convert_20bit_addr(@dump_buff_588(0));
   mc_dma_addr    = convert_20bit_addr(@multicast_buff_588(0));
   tx_dma_addr    = convert_20bit_addr(@tx_buffer_588(0));
   do i=0 to 7 ;
     rx_dma_addr(i)=convert_20bit_addr(@rx_buffer(i).buff(0));
   end;

   output(brd_port)=0ffh;  /* ENABLE DMA AND INTERRUPT DRIVERS */

   end sys_init;

82588 initialization

init_588: procedure;

  config_588(00) = 10;          /* TO CONFIGURE ALL 10 PARAMETERS          */
  config_588(01) = 00;
  config_588(02  = 00001000b;   /* MODE 0, 8  MHZ CLOCK, 1 MB/S            */
  config_588(03) = buff_len/4;  /* RECEIVE BUFFER LENGTH                   */
  config_588(04) = 00100110b;   /* NO LOOPBACK, ADDR LEN = 6, PREAMBLE = 8 */
  config_588(05) = 00000000b;   /* DIFFERENTIAL MANCHESTER = OFF           */
  config_588(06) = 96;          /* IFS = 96 TCLK                           */
  config_588(07) = 0;           /* SLOT TIME = 512 TCLK                    */
  config_588(08) = 11110010b;   /* MAX. NO. RETRIES = 15                   */
  config_588(09) = 00000100b;   /* MANCHESTER ENCODING                     */
  config_588(10) = 10001000b;   /* INTERNAL CRS AND CDT, CRSF = 0          */
  config_588(11) = 64;          /* MIN FRAME LENGTH = 64 BYTES = 512 BITS  */
                                                                  231422-63
```

Figure 58. Initialization Routines

```
ia_set_buff_588(0) = 6;
ia_set_buff_588(1) = 0;
ia_set_buff_588(2) = 000h;
ia_set_buff_588(3) = 041h;
ia_set_buff_588(4) = 000h;
ia_set_buff_588(5) = 000h;
ia_set_uff_588(6)  = 000h;
ia_set_buff_588(7) = 000h;

multicast_buff_588(00) = 12;
multicast_buff_588(01) = 00h;
multicast_buff_588(02) = 11h;
multicast_buff_588(03) = 12h;
multicast_buff_588(04) = 13h;
multicast_buff_588(05) = 14h;
multicast_buff_588(06) = 15h;
multicast_buff_588(07) = 16h;
multicast_buff_588(08) = 21h;
multicast_buff_588(09) = 22h;
multicast_buff_588(10) = 23h;
multicast_buff_588(11) =24h;
multicast_buff_588(12) = 25h;
multicast_buff_588(13) = 26h;

tx_buffer_588(00) = tx_frame_len mod 256;
tx_buffer_588(01) = tx_frame_len / 256;
tx_buffer_588(02) = 011h;   /* INITIAL DESTINATION ADDRESS = MC(1) */
tx_buffer_588(03) = 012h;
tx_buffer_588(04) = 013h;
tx_buffer_588(05) = 014h;
tx_buffer_588(06) = 015h;
tx_buffer_588(07) = 016h;

end init_588;
```

231422-64

Figure 58. Initialization Routines (Continued)

7.3 General Commands

Operations like Transmit, Receive, Configure, etc. are done by a simple sequence of loading the DMA controller with the necessary parameters and then writing the command to the 82588.

Example: Configure Command

To configure the operating environment of the 82588. This command must be the first one to be executed after a RESET.

```
call
DMA_LOAD(1,1,12,@CONFIG_588_ADDR);
output (CS_588) = 12h;
```

The first statement is the prologue to the configure command to the 82588 which calls a routine to load and initialize the DMA controller for the desired operation. This routine is described in section 7.4. The parameters for DMA_LOAD are:

```
first parameter  = 82588 channel
                   number ( = 1)
second parameter = direction ( = 1,
                   memory >> 82588)
third parameter  = length of DMA
                   transfer ( = 12)
fourth parameter = pointer to a 20 bit
                   address of the
                   memory buffer
                   (=@CONFIG_588_ADDR)
```

The second statement writes 12h to the command register of the 82588 to execute a Configure command on channel 1.

When the command execution is complete (successfully or not), 82588 interrupts the 8088 CPU through the 8259A, on the system board. This executes the interrupt service routine, described in section 7.5, which takes the epilogue action for the command.

Most operations are very similar in structure to Configure. The 82588 Reference Manual describes them in detail. Figure 59 shows a listing of the most commonly used operations like:

```
CONFIGURE     INDIVIDUAL-ADDRESS (IA)
              SET-UP
TRANSMIT      MULTICAST-ADDRESS (MC)
              SET-UP
DIAGNOSE      RECEIVE (RCV)-ENABLE
DUMP          RECEIVE (RCV)-DISABLE
TDR           RECEIVE (RCV)-STOP
RETRANSMIT    READ-STATUS
```

```
ia_set: procedure public;                    /* COMMAND - 01 */

    call dma_load(cmd_channel,tx_dir,8,@iaset_dma_addr);

    /* SET DMA CHANNEL 0 OR 1 TO TRANSFER FROM MEMORY
       TO THE 82588. iaset_dma_addr VARIABLE STORES THE
       20 BIT POINTER TO THE INDIVIDUAL ADDRESS BUFFER  */

    if cmd_channel then output (cs_588) = 11h;
    else output(cs_588) = 01h;

    /* EVERY COMMAND CAN BE EXECUTED IN EITHER DMA CHANNEL 0 OR 1.
       THE VARIABLE cmd_channel INDICATES THE REQUIRED CHANNEL  */

end ia_set;

/*-------------------------------------------------------------------------*/

config: procedure public;                    /* COMMAND - 02 */

    call dma_load(cmd_channel,tx_dir,12,@conf_dma_addr);
    if cmd_channel then output (cs_588) = 12h;
    else output(cs_588) = 02h;

end config;

/*-------------------------------------------------------------------------*/

multicast: procedure public;                 /* COMMAND - 03 */

    call dma_load(cmd_channel,tx_dir,14,@mc_dma_addr);
    if cmd_channel then output (cs_588) = 13h;
    else output(cs_588) = 03h;

end multicast;

/*-------------------------------------------------------------------------*/

transmit: procedure(buffer_len) public;      /* COMMAND - 04 */

     declare buffer_len word;

    tx_buffer_588(00) = low(buffer_len);
    tx_buffer_588(01) = high(buffer_len);

    call dma_load(cmd_channel,tx_dir,1536,@tx_dma_addr);

    if cmd_channel then output (cs_588) = 14h;
    else output(cs_588) = 04h;

end transmit;
```

231422-65

Figure 59. General Commands

```
tdr: procedure public;                          /* COMMAND - 05 */

    if cmd_channel then output (cs_588) = 15h;
    else output(cs_588) = 05h;

end tdr;

/*-------------------------------------------------------------------------*/

dump_588: procedure public;                     /* COMMAND - 06 */

    call dma_load(cmd_channel,rx_dir,64,@dmp_dma_addr);
    if cmd_channel then output (cs_588) = 16h;
    else output(cs_588) = 06h;

end dump_588;

/*-------------------------------------------------------------------------*/

diagnose: procedure public;                     /* COMMAND - 07 */

    if cmd_channel then output (cs_588) = 17h;
    else output(cs_588) = 07h;

end diagnose;

/*-------------------------------------------------------------------------*/

rcv_enable: procedure(channel,buffer_no,len) public;    /* COMMAND - 08 */

    declare channel byte;
    declare len  word;
    declare buffer_no byte;
    call dma_load(channel,rx_dir,len,@rx_dma_addr(buffer_no));
    if rx_channel then output (cs_588) = 18h;
    else output(cs_588) = 08h;

end rcv_enable;

/*-------------------------------------------------------------------------*/

rcv_disable: procedure public;                  /* COMMAND - 10 */

    enable_rcv=0;
    output(cs_588)=0ah;

end rcv_disable;
```

231422-66

```
/*-------------------------------------------------------------------------*/

rcv_stop: procedure public;                     /* COMMAND - 11 */

    enable_rcv=0;
    output(cs_588)= 0bh;

end rcv_stop;

/*-------------------------------------------------------------------------*/

retransmit: procedure public;                   /* COMMAND - 12 */

    call dma_load(cmd_channel,tx_dir,1536,@tx_dma_addr);
    if cmd_channel then output (cs_588) = 1ch;
    else output(cs_588) = 0ch;

end retransmit;

/*-------------------------------------------------------------------------*/

abort: procedure public;                        /* COMMAND - 13 */

    output(cs_588)= 1dh;
    call new_status(1);

end abort;

/*-------------------------------------------------------------------------*/

reset_588: procedure public;                    /* COMMAND - 14 */

    enable_rcv=0;
    output(cs_588) = 1eh;
    call config;

end reset_588;
```

231422-67

Figure 59. General Commands (Continued)

7.4 DMA Routines

DMA__LOAD procedure is used to program the 8237A DMA controller for all the operations requiring DMA service. It also starts or enables the programmed DMA channel after programming it. Figure 60 shows the listing of this procedure. It accepts 4 parameters from the calling routine to decide the programming configuration for the 8237A. The parameters for DMA__LOAD are: Channel, direction, buff__len, and buff__addr.

```
Converting a pointer SEG:OFFSET to a 20 bit address
convert_20bit_addr:procedure(ptr) dword public;

   declare   ptr       pointer,
             ptr_addr  pointer,
             ptr_20bit dword   ,
             (wrd  based ptr_addr)(2) word;

   ptr_addr=@ptr;
   ptr_20bit=shl((ptr_20bit:=wrd(1)),4)+wrd(0);
   return(ptr_20bit);

end convert_20bit_addr;


IBM/PC DMA loading procedure

dma_load: procedure(channel,direction,buff_len,buff_addr) reentrant public;

    declare channel byte;              /* CHANNEL #, 0 or 1                  */
    declare direction byte;            /* 0=RX, 588 -> MEM; 1=TX, MEM -> 588 */
    declare buff_len word;             /* BYTE COUNT                         */
    declare buff_addr pointer;         /* BUFFER ADDR IN 20 BITS FORM        */
    declare (wrd based buff_addr)(2) word;

    channel=channel and 1;             /* GET LEAST SIGNIFICANT BIT          */

    if channel=0 then                  /* EXECUTE COMMAND ON CHANNEL 1       */
      do;
      output(dma_flff)  = 0;           /* CLEAR FIRST/LAST FLIP-FLOP         */
      if direction=0
          then output(dma_mode)=dma_rx_mode_1;  /* DIRECTION BIT, TELLS    */
      else output(dma_mode)=dma_tx_mode_1;      /* TRANSMIT OR RECEIVE     */
      output(dma_addr_1)  = low (wrd(0));       /* LOAD LSB ADDRESS BYTE   */
      output(dma_addr_1)  = high(wrd(0));       /* LOAD MSB ADDRESS BYTE   */
      output(dma_addrh_1) = low (wrd(1));       /* LOAD PAGE REGISTER      */
      output(dma_bc_1)    = low (buff_len);     /* LOAD LSB BYTE COUNT     */
      output(dma_bc_1)    = high(buff_len);     /* LOAD MSB BYTE COUNT     */
      output(dma_mask) = dma_on_1;              /* START CHANNEL 1         */
      end;
    else do;                          /* SAME AS BEFORE FOR CHANNEL 3        */
      output(dma_flff)  = 0;
      if direction=0
        then output(dma_mode)=dma_rx_mode_3;
      else output(dma_mode)=dma_tx_mode_3;
      output(dma_addr_3)  = low (wrd(0));
      output(dma_addr_3)  = high(wrd(0));
      output(dma_addrh_3) = low (wrd(1));
      output(dma_bc_3)    = low (buff_len);
      output(dma_bc_3)    = high(buff_len);
      output(dma_mask) = dma_on_3;
      end;

end dma_load;
```

231422-68

Figure 60. DMA Routine

One peculiarity about this procedure is that in order to speed up the DMA step-up, this procedure doesn't get a pointer to the buffer, but a pointer to a 20 bit address in the 8237 format. The 8088/8086 architecture define pointers as 32 bits seg:offset entities, where seg and offset are 16 bit operands. By the other hand the IBM/PC uses an 8237A and a page register, requiring a memory address to be a 20 bit entity. The process of converting a seg:offset pointer to a 20 bit address is time consuming and could negatively affect the performance of the 82588 driver software. The decision was to make the pointer/address conversions during initialization, considering that the buffers are static in memory (essentially removing this calculation from the real time response loops).

Figure 61 is a listing of the DMA_LOAD procedure for the 80188 or 80188 on-chip DMA controller. It has the same caller interface as the 8237A based one.

```
dma_load: procedure(channel,direction,trans_len,buff_addr) reentrant;

/* To load and start the  80186 DMA controller for the desired operation */

  declare dma_rx_mode   literally '1010001001000000b'; /* rx channel */
        /* src=IO, dest=M(inc), sync=src, TC, noint, priority, byte */

  declare dma_tx_mode   literally '0000110100000000b'; /* tx channel */
        /* src=M(inc), dest=IO, sync=dest, TC, noint, noprior, byte */


  declare channel byte;       /* channel #                                */
  declare direction byte;     /* 0 = rx, 588 -> mem; 1 = tx, mem -> 588 */
  declare trans_len word;     /* byte count                               */
  declare buff_addr pointer; /* buffer pointer in 20 bit addr. form      */

  declare (wrd based buff_addr)(2) word;

  do case channel and 00000001b;
    do case direction and 00000001b;
      do;               /* channel 0, 588 to memory  */
      output(dma_0_dpl)  = wrd(0);
      output(dma_0_dph)  = wrd(1);
      output(dma_0_spl)  = ch_a_588;
      output(dma_0_sph)  = 0;
      output(dma_0_tc)   = trans_len;
      output(dma_0_cw)   = dma_rx_mode or 0006h; /* Start DMA chl 0 */
      end;

      do;              /* channel 0, memory to 588 */
      output(dma_0_dpl)  = ch_a_588;
      output(dma_0_dph)  = 0;
      output(dma_0_spl)  = wrd(0);
      output(dma_0_sph)  = wrd(1);
      output(dma_0_tc)   = trans_len;
      output(dma_0_cw)   = dma_tx_mode or 0006h; /* Start DMA chl 0 */
      end;
    end;
```

231422-69

Figure 61. 80186 DMA Routines

```
    do case direction and 00000001b;
        do;                 /* channel 1, 588 to memory  */
        output(dma_1_dpl)   = wrd(0);
        output(dma_1_dph)   = wrd(1);
        output(dma_1_spl)   = ch_b_588;
        output(dma_1_sph)   = 0;
        output(dma_1_tc)    = trans_len;
        output(dma_1_cw)    = dma_rx_mode or 0006h; /* Start DMA chl 1 */
        end;

        do;                 /* channel 1, memory to 588 */
        output(dma_1_dpl)   = ch_b_588;
        output(dma_1_dph)   = 0;
        output(dma_1_spl)   = wrd(0);
        output(dma_1_sph)   = wrd(1);
        output(dma_1_tc)    = trans_len;
        output(dma_1_cw)    = dma_tx_mode or 0006h; /* Start DMA chl 1 */
        end;
      end;

    end;

end dma_load;
```

231422-70

Figure 61. 80186 DMA Routines (Continued)

7.5 Interrupt Routine

The interrupt service routine, 'intr_588', shown in Figure 62, is invoked whenever the 82588 interrupts. The main difficulty in designing this interrupt routine was to speed its performance. Fast status processing was a basic requirement to be able to handle back to back frames.

The interrupt handler will read 82588 status, and put them into a 64 byte long EVENT_FIFO. Those statuses are going to be used in the main loop for updating screen counters. All the statistics are updated as fast as possible in the interrupt handler to fulfill the back-to-back frame processing requirement.

The interrupt handler is not reentrant, interrupts are disabled at the beginning and reenabled on exit.

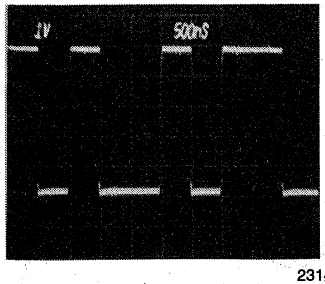
```
Interrupt service routine

intr_588:procedure interrupt 13;

    declare stat                    byte,
            event                   byte,
            i                       byte,
            (st0,st1,st2,st3)       byte,
            rx_st0                  byte,
            rx_st1                  byte;

   /* FOLLOWING LITERALS HAVE THE PURPOSE OF ENABLE ACTING
      ON EITHER CHANNEL 1 OR 3 SELECTIVELY                    */

   declare

     stop_cmd_dma  literally 'if cmd_channel
                                then output(dma_mask)=dma_off_3;
                              else output(dma_mask)=dma_off_1',
     stop_rx_dma   literally 'if rx_channel
                                then output(dma_mask)=dma_off_3;
                              else output(dma_mask)=dma_off_1',

     issue_rtx_cmd literally 'if cmd_channel
                                then output(cs_588)=1Ch;
                              else output(cs_588)=0ch',
     issue_tx_cmd  literally 'if cmd_channel
                                then output(cs_588)=14h;
                              else output(cs_588)=04h';

   disable;                                    /* DISABLE INTERRUPTS   */
                                               /* NO INTERR. NESTING   */
   output(cs_588)  =0fh;                       /* RLS 588 PTR, START 0 */

   event_fifo(wr_ptr).st0,st0=input(cs_588);   /* READ 82588 STATUS    */
   event_fifo(wr_ptr).st1,st1=input(cs_588);   /* REGISTERS, PASSING   */
   event_fifo(wr_ptr).st2,st2=input(cs_588);   /* THEM TO THE MAIN     */
   event_fifo(wr_ptr).st3,st3=input(cs_588);   /* PROGRAM ON THE FIFO  */

   wr_ptr=(wr_ptr+1) and 0fh;                  /* INCREMENT FIFO       */
   fifocnt=(fifocnt+1) and 0fh;                /* COUNTERS             */

   event=st0 and 0fh;                          /* GET EVENT FIELD      */

   output(cs_588)=80h;                         /* ACKNOWLEDGE 82588    */
                                               /* INTERRUPT            */   231422-71


   do case event;

   ev_00: ;                            /* NOP COMMAND          */
   ev_01: stop_cmd_dma;                /* IA_SETUP, STOP DMA   */
   ev_02: stop_cmd_dma;                /* CONFIGURE, STOP DMA  */
   ev_03: stop_cmd_dma;                /* MULTICAST, STOP DMA  */
   ev_04: do;                          /* TRANSMIT DONE        */
          stop_cmd_dma;

          /* CHECK IF THERE WAS A COLLISION AND IS NOT THE
             MAX COLLISION                                     */

          stat=(st2 and 10000000b) or (st1 and 00100000b);
          if (stat=80h)
             then do;                  /* RETRANSMIT           */
                  call dma_load(cmd_channel,tx_dir,1536,@tx_dma_addr);
                  issue_rtx_cmd;
                  /* UPDATE STATISTICS                        */
                  total_tx_count=total_tx_count+1;
                  coll_cnt(17) = coll_cnt(17) + 1; /*TOTAL COLL*/
                  bad_tx_count = bad_tx_count + 1;
                  end;
          else do;
               if in_loop    /* EXECUTING TRANSMISSIONS IN LOOP  */
                then do;     /* RE ISSUE TRANSMIT COMMAND        */
                     call dma_load(cmd_channel,tx_dir,1536,@tx_dma_addr
                     issue_tx_cmd;
                     total_tx_count=total_tx_count+1;
                     end;
               if (st2 and 00100000b) = 0          /* BAD TRANSMIT*/
                 then do;
                      bad_tx_count = bad_tx_count + 1;
                      /* INCREMENT UNDERRUN COUNTER            */
                      tmp=scr(tmp:=st2,1);
                      tx_under=tx_under plus 0;
                      /* INCREMENT LOST CTS COUNTER            */
                      tmp=scr(tmp,1);
                      lost_cts=lost_cts plus 0;
                      /* INCREMENT LOST CRS COUNTER            */
                      tmp=scr(tmp,1);
                      lost_crs=lost_crs plus 0;
                      if (stat=0A0h)  /* INC COLLISIONS COUNTER  */
                         then coll_cnt(17) = coll_cnt(17) + 1;
                      end;
               end;

               /* INCREMENT DEFER  COUNTER                    */
               tmp=scl((tmp:=st1),1);
               tx_defer=tx_defer plus 0;
          end;                                                 231422-72
```

Figure 62. Interrupt Routine

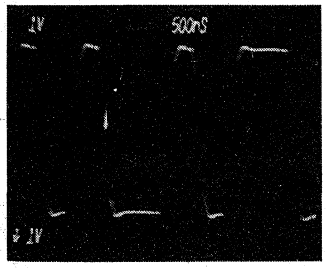
```
ev_05: stop_cmd_dma;                    /* TDR COMMAND, STOP DMA   */
ev_06: stop_cmd_dma;                    /* DUMP COMMAND, STOP DMA  */
ev_07: stop_cmd_dma;                    /* DIAGNOSE CMD, STOP DMA  */
ev_08:                                  /* RECEIVED FRAME          */
     do;
     stop_rx_dma;
     i=(current_buff+1) and 00000111b; /* INC BUFFER NO. MOD 8*/
     if enable_rcv<>0                   /* IF RECEIVER IS ON       */
         then do;                       /* PREPARE NEXT BUFFER */
              call dma_load(rx_channel,rx_dir,1532,@rx_dma_addr(i));
              if rx_channel then output(cs_588)= 18h;
              else output(cs_588)=08h;
              rx_buffer(i).chain_cnt=0;
              end;
     else call rcv_disable;             /* DISABLE RECEIVER    */

     /* FIND ADDRESS OF END OF CURRENTLY RECEIVED BUFFER      */
     /* BY CALCULATING IT WITH THE 82588 BYTE COUNT REGS.     */
     rx_buff_off=(shl(double(st2),8) or double(st1));
     /* READ STATUS BYTES FROM MEMORY                         */
     rx_st0=rx_buffer(current_buff).buff(rx_buff_off-2);
     rx_st1=rx_buffer(current_buff).buff(rx_buff_off-1);
     /* UPDATE ACTUAL BUFFER SIZE                             */
     rx_buffer(current_buff).actual_size=rx_buff_off;
     rx_buffer(current_buff).st0=rx_st0;
     rx_buffer(current_buff).st1=rx_st1;
     current_buff=i;
     /* UPDATE TOTAL RECEIVED BUFFERS                         */
     total_rcv_count=total_rcv_count+1;
     /* UPDATE STATISTICS                                     */
     if (rx_st1 and 00100000b)=0
         then do;
              bad_rcv_count=bad_rcv_count+1;
              /* INCREMENT  NO END OF FRAME COUNTER           */
              tmp=scr(tmp:=rx_st0,7);
              no_eof=no_eof plus 0;
              /* INCREMENT  SHORT FRAME COUNTER               */
              tmp=scr(tmp,1);
              srt_frm=srt_frm plus 0;
              /* INCREMENT  RX OVERRUN COUNTER                */
              tmp=scr(tmp:=rx_st1,1);
              rx_over=rx_over plus 0;
              /* INCREMENT  ALIGNMENT ERROR COUNTER           */
              tmp=scr(tmp,2);
              alg_err=alg_err plus 0;
              /* INCREMENT  CRC ERROR COUNTER                 */
              tmp=scr(tmp,1);
              crc_err=crc_err plus 0;
              end;
     end;                                                          231422-73


      /*  EV_09 REQUESTS ASSIGNMENT OF A NEW BUFFER       */
ev_09: call allocate_new_buffer(not(rol(st3,1)) and 00000001b);
ev_10: stop_rx_dma;     /* RECEIVE DISABLE                 */
ev_11: stop_rx_dma;     /* STOP RECEIVE                    */
ev_12: do;              /* RE-TRANSMIT DONE                */
       stat=(st2 and 10000000b) or (st1 and 00100000b);
       if (stat=80h)
       then do;         /* RETRANSMIT                      */
            call dma_load(1,tx_dir,1536,@tx_dma_addr);
            issue_rtx_cmd;
            coll_cnt(17) = coll_cnt(17) + 1;
            total_tx_count=total_tx_count+1;
            bad_tx_count=bad_tx_count +1;
            end;
       else do;
            if in_loop
             then do;   /* LOOP RETRANSMISSIONS            */
                  call dma_load(cmd_channel,tx_dir,1536,@tx_dma_a
                  issue_tx_cmd;
                  total_tx_count=total_tx_count+1;
                  end;
            if (stat=0A0h) /* MAX COLLISION                */
               then do;
                    coll_cnt(16) = coll_cnt(16)+1;
                    coll_cnt(17) = coll_cnt(17)+1;
                    bad_tx_count=bad_tx_count +1;
                    end;
            /* UPDATE SPECIFIC COLLISION COUNTER           */
            else  coll_cnt(st1 and 0fh)
                        = coll_cnt(st1 and 0fh) + 1;
            end;
       end;
ev_13: stop_cmd_dma;       /* EXECUTION ABORTED            */
ev_14: ;
ev_15: stop_cmd_dma;       /* DIAGNOSE FAILED              */
       end;

/*  ACKNOWLEDGE 8259A  INTERRUPT                           */
output(pic_ocw2)= seoi_pico;   /* SPECIFIC EOI FOR 8259    */

end intr_588;                                                      231422-74
```

Figure 62. Interrupt Routine (Continued)

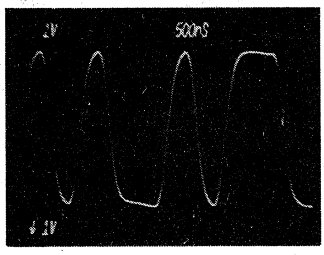
APPENDIX A
STARLAN SIGNALS



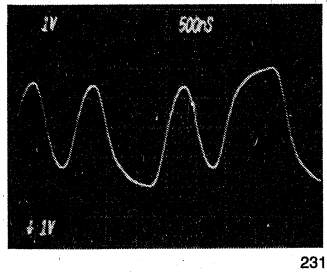
231422-51



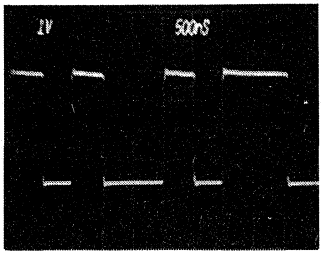
231422-52



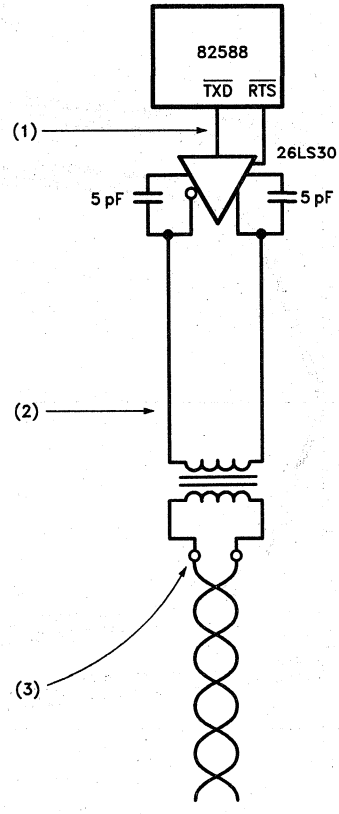
231422-53



231422-54



231422-55



231422-47

Figure 63. StarLAN Signals

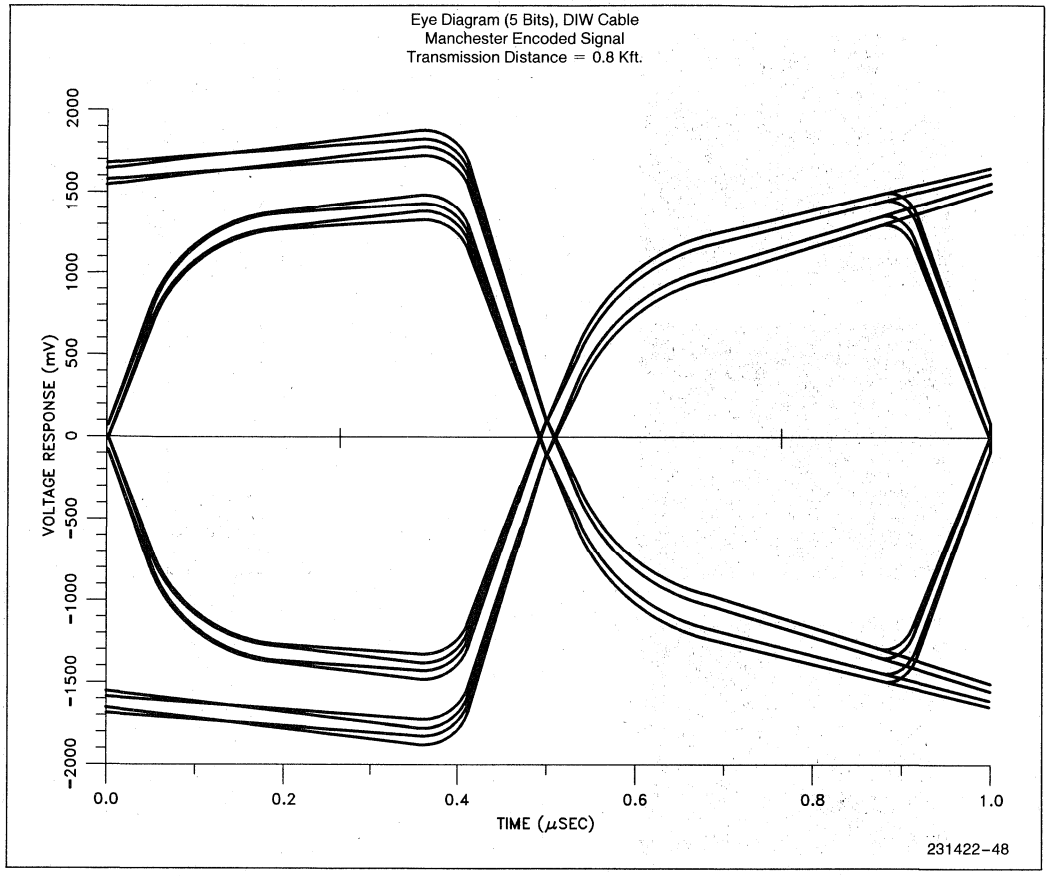


Figure 64. Received Signal Eye Diagram

APPENDIX B
802.3 1BASE5 MULTI-POINT EXTENSION (MPE)

As previously stated, one of the most important advantages of StarLAN is being able to work on already installed phone wires. This advantage is considerably diminished in Europe where numerous constraints exist to the using of those wires:

1. Wire belongs to local PTTs.
2. Not enough spare wires.

This same issue is raised when talking about small businesses where in a lot of cases no wiring closets and/or spare wires are available.

In summary, in a lot of cases rewiring will be necessary, in which case the STAR topology may not be the most economical one.

Recently the StarLAN 802.3 1BASE5 task force has been considering the extension of the StarLAN base topology. This extension called MULTI POINT EXTENSION (MPE) is going to be developed to address the previously described marketing requirements.

Currently no agreement has been reached by the StarLAN task force on the MPE exact topology and implementation. Multiple approaches have been presented, but no consensus met. It was decided though that the MPE is going to be an addendum to the STAR topology, and that its final specification will happen after the approval of the current 1BASE5 STAR topology (July 1986).

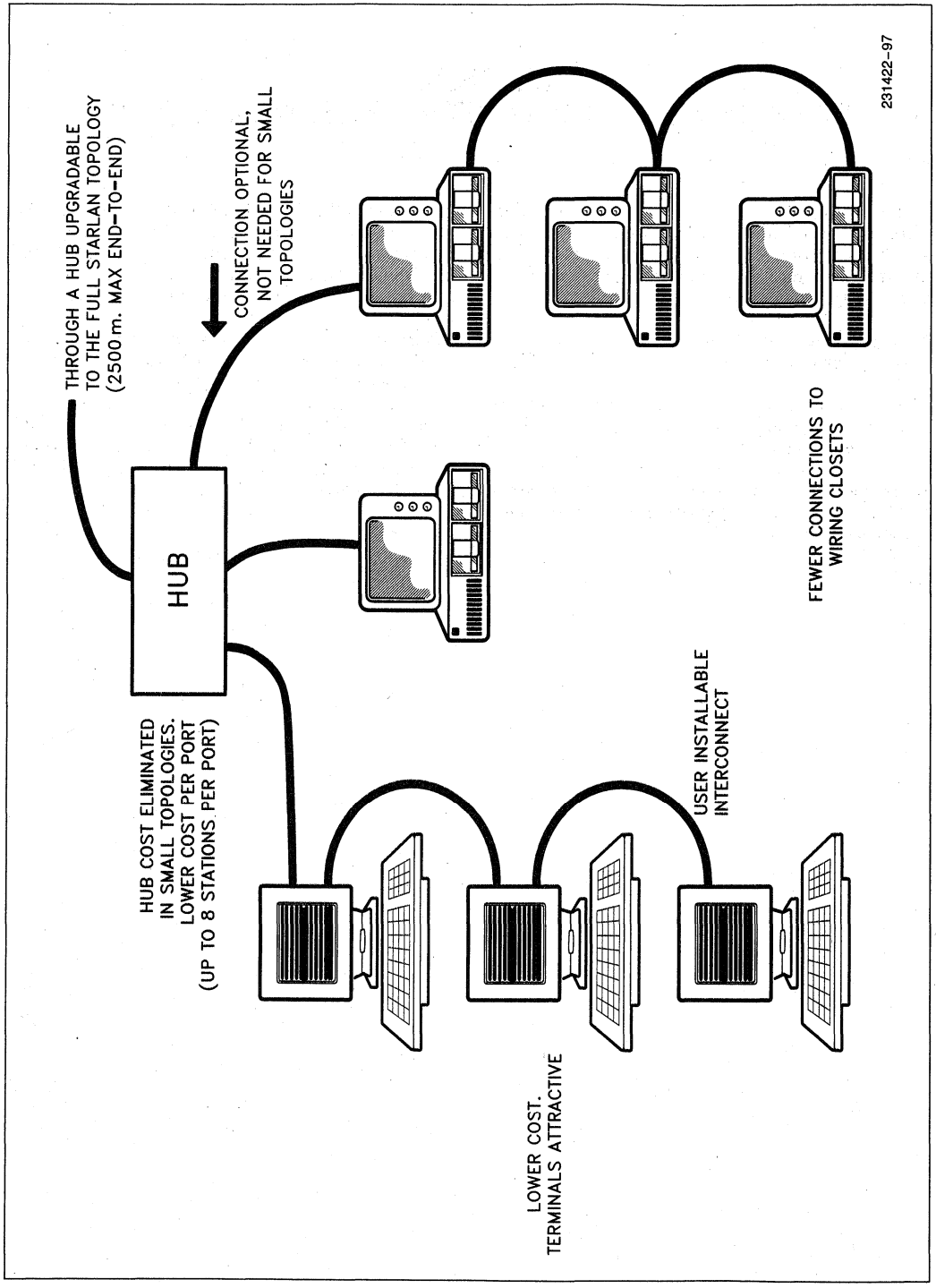


Figure 65. Multipoint Extension

APPENDIX C
SINGLE DMA CHANNEL INTERFACE

In a typical system, the 82588 needs 2 DMA channels to operate in a manner that no received frames are lost as discussed in section 5.1.3. If an existing system has only one DMA channel available, it is still possible to operate the 82588 in a way that no frames are lost. This method is recommended only in situations where a second DMA channel is impossible to get.

Figure 66 shows how the 82588 DMA logic is interfaced to one channel of a DMA controller. Two DRQ lines are ORed and go to the DMA controller DRQ line and the DACK line from the DMA controller is connected to DACK0 and DACK1 of the 82588. The 82588 is configured for multiple buffer reception (chaining), although the entire frame is received in a single buffer. Let us assume that channel CH-0 is used as the first channel for reception. After the ENAble RECeive command, CH-0 is dedicated to reception. As long as no frame is received, the other channel, CH-1, can be used for executing any commands like transmit, multicast address, dump, etc., by programming the DMA channel for the execution command. The status register should be checked for any ongoing reception, to avoid issuing an execution command when reception is active.

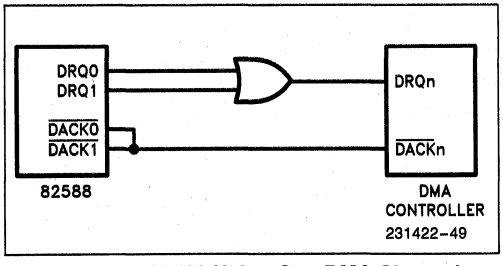


Figure 66. 82588 Using One DMA Channel

If a frame is received, an interrupt for additional buffer occurs immediately after an address match is established, as shown in Figure 67. After this, the received bytes start filling up the on-chip FIFO. The 82588 activates the DRQ line after 15—FIFO LIMIT + 3 bytes are ready for transfer in the FIFO (about 80 microseconds after the interrupt). The CPU should react to the interrupt within 80 μs and disable the DMA controller. It should also issue an ASSIGN ALTERNATE BUFFER command with INTACK to abort any execution command that may be active. The FIFO fills up in about 160 μs after interrupt. To prevent an underrun, the CPU must reprogram the DMA controller for frame reception and re-enable the DMA controller within 160 μs after the interrupt (time to receive about 21 bytes). No buffer switching actually takes place, although the 82588 generates request for alternate buffer every time it has no additional buffer. The CPU must respond to these interrupts with an ASSIGN ALTERNATE BUFFER command with INTACK. To keep the CPU overhead to a minimum, the buffer size must be configured to the maximum value of 1 kbyte.

If a frame transmission starts deferring due to the reception occurring just prior to an issued transmit command, the transmission can start once the link is free after reception. A maximum of 19 bytes are transmitted (stored in the FIFO and internal registers) followed by a jam pattern and then an execution aborted interrupt occurs. The aborted frame can be transmitted again.

If the transmit command is issued and the 82588 starts transmitting just prior to receiving a frame then transmit wins over receive—but this will obviously lead to a collision.

Note that the interrupt for additional buffer is used to abort an ongoing execution command and to program the DMA channel for reception just when a frame is received. This scheme imposes real time interrupt handling requirements on the CPU and is recommended only when a second DMA channel is not available.

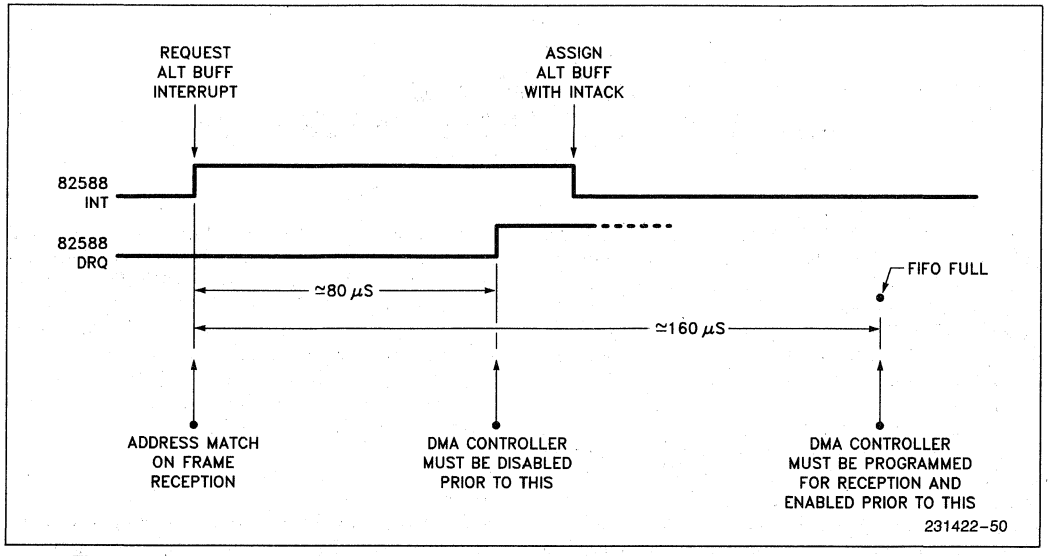


Figure 67. Timing at the Beginning of Frame Reception for Single DMA Channel Operation

APPENDIX D
MEASURING NETWORK DELAYS WITH THE 82588

Knowing networks round-trip delays in local area networks is an important capability. The round-trip delay very much defines the slot time parameter which by itself has a direct relationship to network efficiency and throughput. Very often the slot-time parameter is not flexible, due to standards requirements. Whenever it is flexible, optimization of this number may lead to significant improvement in network performance.

Another possible usage of the network delay knowledge is in balancing the inter-frame -spacing (IFS) on broadband networks. On those networks, stations nearer to the HEAD-END hear themselves faster than farther ones. Effectively having a shorter IFS than stations far from the HEAD-END. This difference causes an inbalance in network access time for different stations at different distances from the HEAD-END. Knowing the STATION/HEAD-END delay allows the user to reprogram the 82588 IFS accordingly, and by that balance the effective IFS for all the stations.

The 82588 has an internal mechanism that allows the user to measure this delay in BIT-TIME units. The method is based on the fact that the 82588 when configured for internal collision detection, requires that the carrier sense be active within half a slot-time after transmission has started. If this requirement is not fulfilled the 82588 notifies that a collision has occurred. Thus it is possible to configure the 82588 to different slot time values, then transmit a long frame (of at least half a slot-time). If the transmission succeeds, the network round-trip delay is less than half the programmed slot-time. If a collision is reported, the delay is longer. The value of the round-trip delay can be found by repeating this experiment process while scanning the slot-time configuration parameter value and searching the threshold. A binary search algorithm is used for that purpose. First the slot-time is configured for the maximum (2048 bits) and according if there was a collision or not, the number changed for the next try. (See Figure 68)

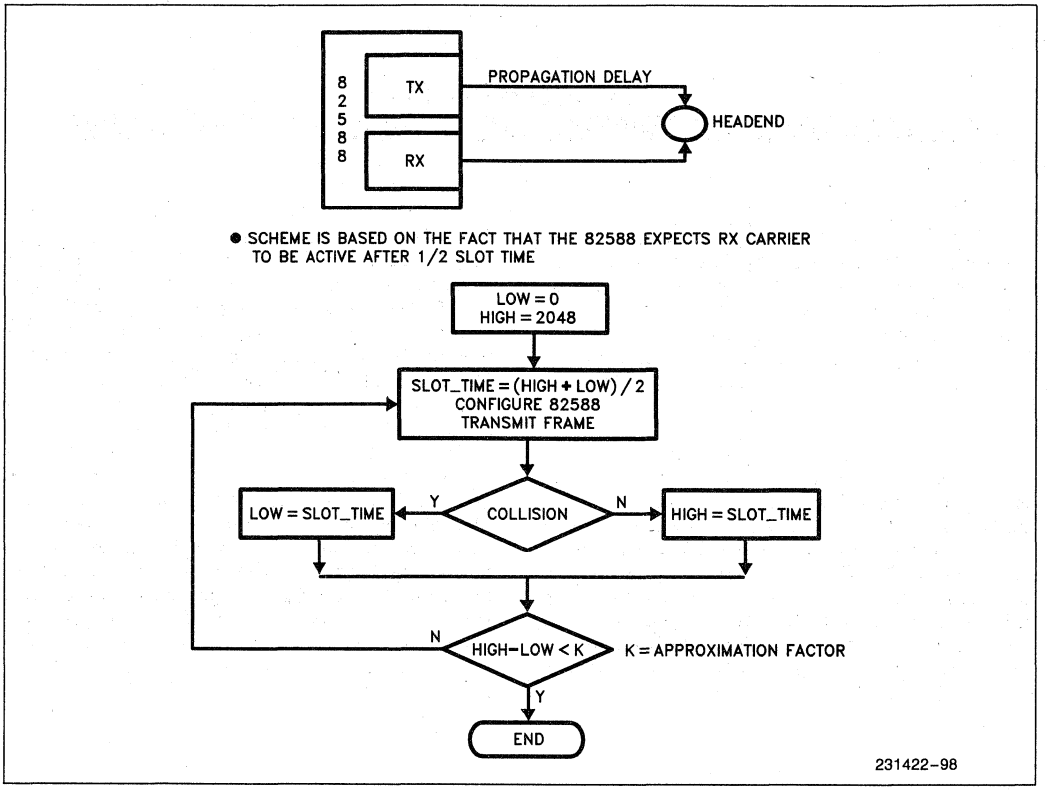


Figure 68. Network Delay Measurement using the 82588

intel®

APPLICATION NOTE

AP-320

November 1988

Using the Intel 82592 to Integrate a Low-Cost Ethernet Solution into a PC Motherboard

MICHAEL ANZILOTTI
TECHNICAL MARKETING ENGINEER

Order Number: 290189-001

1.0 INTRODUCTION

During the past several years office networking has become an increasingly efficient method of resource sharing for companies looking to increase productivity while reducing cost. Networking allows multiuser access to a data base of files or programs via a network file server; it allows sharing of expensive peripherals; e.g., laser printers; and it offers a greater degree of data security by centralizing the hard disk and backup facilities. This type of network allows a user to concentrate his resources; e.g., a high-capacity, high-performance hard disk, at the network file server, allowing the other nodes, or PC workstations, on the network to function with limited or no mass data storage capability.

As Local Area Networks (LANs) have become more common in the office and in industry, some clear market development trends have emerged. Possibly the most significant development in the LAN marketplace is the concern for cost reduction. This need is driven by intense competition between network vendors for market share. Today's LAN marketplace requires low-cost, simple network solutions that do not sacrifice performance. Another significant development in the LAN marketplace is the acceptance of Ethernet, or a derivative (e.g., Cheapernet or Twisted Pair Ethernet), as the industry standard for high-performance LANs. Because of Ethernet's popularity, there is a great need for cost reduction in this market.

Personal computers (PCs) have also seen significant changes over the past several years. PCs have become firmly entrenched in the office. Their popularity, coupled with a highly competitive market, has compelled PC vendors to both reduce costs for their LAN solutions and to attempt to distinguish their product from the competition's. The means of this cost reduction range from eliminating expensive hardware, such as disk drives and their associated hardware, to using highly integrated VLSI devices that implement the functions of a PC in a combination chip set containing several devices. Differentiation has been achieved by integrating peripheral functions, normally contained on an external adapter card, into the main processor board, or motherboard, of the PC. Video Graphics Array (VGA) and LAN connections are examples of this strategy.

The Intel 82592 LAN controller is uniquely suited for integration into a PC AT style motherboard. It meets the demands of today's market by providing the PC vendor (1) a means of reducing cost while maintaining high performance, and (2) a path for differentiation. An 82592 integrated into a PC motherboard provides a very low cost and very simple implementation because it uses the host system's existing resources to complete the LAN solution; e.g., system memory and DMA. This leaves the 82592, the serial interface, and some control logic as the only components required to complete a motherboard LAN solution.

1.1 Objective

This Application Note presents the general concept of integrating a Local Area Networking into a PC motherboard, and how the 82592 suits this purpose. The design of the 82592 Embedded LAN Module, which plugs into an Intel SYP301 motherboard (or any standard PC AT style motherboard), is explained in detail—providing a demonstration of an integrated Ethernet LAN solution.

1.2 Acknowledgements

For their contributions to this Application Note, and for their work in developing the architecture of the 82592 Embedded LAN Module, I would like to acknowledge, and thank, Uri Elzur, Dan Gavish, and Haim Sadger, of the Intel Israel System Validation group; and Joe Dragony, of Intel's (Folsom) Data Communications Focus Group.

2.0 THE EVOLUTION OF LAN SOLUTION ARCHITECTURES

LAN solutions have undergone an evolution in architecture—from expensive and complex to more cost-efficient and streamlined. A definite trend in office networking can be seen, as these solutions permit the host system to perform functions that were previously included in the LAN solution.

The first LAN solutions were usually intelligent buffered adapter cards, with a CPU, large memory requirements (up to 512 kB), firmware, a LAN controller, and a serial interface. As networking became more prevalent in the office environment—linking PCs and workstations via Ethernet—this complex architecture evolved into simpler and more streamlined nonintelligent, buffered adapters. In this architecture the CPU is no longer part of the LAN solution; its processing power is supplied by the host system. This architecture does not need memory to support a local CPU. Memory is only needed to supply a buffer space to store data before moving it to system memory or onto the serial link. The memory requirement for nonintelligent, buffered architectures is typically 8 kBytes to 32 kBytes. The firmware to boot the CPU is also no longer needed. The evolution to a nonintelligent, buffered architecture has resulted in significant cost savings and reduced complexity.

Significant increases in speed and processing power have been made to PCs during the past several years. This trend to higher performance host systems has allowed further streamlining of the LAN solution's architecture, resulting in even greater cost reduction and simplification. This is accomplished by using host system resources whenever possible. A nonintelligent, nonbuffered architecture is the result. In this architecture, the host system's memory and DMA are used by the LAN controller. The complexity associated with buffered LAN solutions (e.g., supplying a dual-port arbitration scheme for local memory access by both the CPU and the LAN controller) is reduced; this complexity is removed from the LAN solution and returned to the host system, which is designed for these complex tasks. The result of this architectural optimization is a very simple, low component count, cost-efficient solution for a LAN connection. The 82592 Embedded LAN Module is the realization of this optimization. The trend to optimization of LAN architectures is shown in Figure 1.

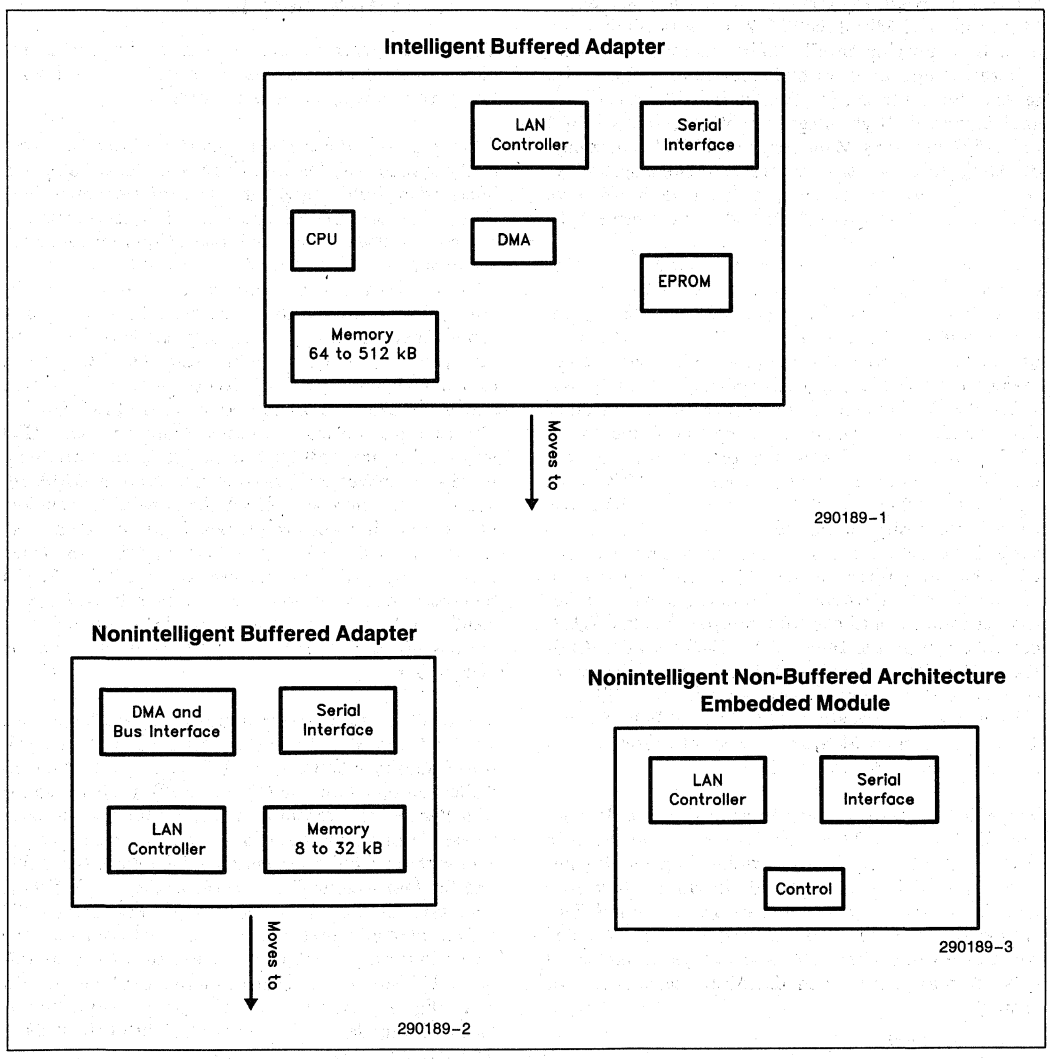


Figure 1. Architectural Optimization of LAN Solutions

3.0 THE 82592 LAN CONTROLLER

3.1 General Features

The 82592 is a second generation, CMOS, advanced CSMA/CD LAN controller with a 16-bit data path. Along with its 8-bit version, the 82590, it is the follow-on design to the 82588 LAN controller. The 82592 is upwards software compatible from the 82588. The 82592 has two modes of serial operation, High Speed Mode and High Integration Mode. In High Speed Mode (up to 20 Mb/s) the 82592 couples with the Intel 82C501 to provide an all CMOS kit for IEEE 802.3 Ethernet applications. In this mode the 82592 can also serve as the controller for Twisted Pair Ethernet (TPE) applications. In High Integration Mode (up to 4 Mb/s) the 82592 performs Manchester and NRZI encoding/decoding, collision detection, transmit clocking, and receive clock recovery on chip; in this mode it can serve as a controller for StarLAN and other midrange LANs.

The 82592 provides several features that allow an efficient system interface to a wide variety of Intel microprocessors (e.g., iAPX 188, 186, 286, and 386) and industry standard buses (e.g., the IBM PC I/O channel or the PS/2™ Micro Channel™). To issue a command to the 82592 (e.g., TRANSMIT or CONFIGURE) the CPU only needs to set up a block in memory that contains the parameters to be transferred to the 82592, program the DMA controller to point to that location and issue the proper opcode to the 82592. The 82592 and DMA controller perform the functions needed to complete the command, with the 82592 interrupting the CPU when the command is complete. The 82592 has a high-performance, 16-bit bus interface, operating at up to 16 MHz. It also implements a specialized hardware handshake with industry standard DMA controllers (e.g., the Intel 8237, 82380, and 82370) or the Intel 82560. This allows for back-to-back frame reception, and automatic retransmission on collision—without CPU intervention. The 82592 FIFOs (Rx and Tx) can have their 64 bytes divided into combinations of 32/32, 16/48, 48/16, or 16/16.

The 82592 features a Deterministic Collision Resolution (DCR) mode. When a collision is detected while in this mode, all nodes in a deterministic network enter into a time-division-multiplexed algorithm where each node has its own unique slot in which to transmit. This ensures that the collision is resolved within a calculated worst-case time. The 82592 also features a number of network management and diagnostic capabilities; for example,

- Monitor mode
- A 24-bit timer
- Three 16-bit event counters
- Internal and external loopback
- Internal register dump
- A TDR mechanism
- Internal diagnostics

For further information on the 82592, please refer to the Intel *Microcommunications Handbook.*

3.2 Unique Features for Embedded LAN Applications

The 82592 has several unique features that enable implementing a high-performance embedded LAN solution with minimal cost and complexity.

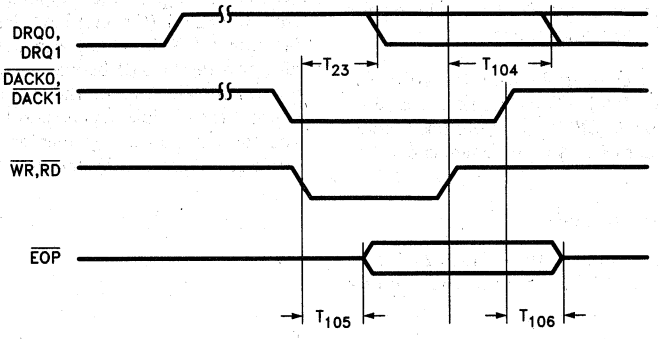
Peripherals on a motherboard must compete for access to the system bus. Because there is no local buffer for intermediate buffering of data, data transfers take place in real-time over the system bus to the system memory. A LAN controller must have a large internal data storage area to be able to wait for access to the system bus while serial data is being received or transmitted. Without sufficient internal data storage, a LAN controller cannot take advantage of the cost efficiency and simplicity of a non-buffered architecture. The 82592 has a total of 64 bytes of FIFOs. This expanded FIFO section allows the 82592 to tolerate long system bus latencies. For example, during a Receive (with the Rx FIFO length configured to 48 bytes) the 82592 can tolerate up to 38.4 μs of bus latency—the time from a DMA request to reception of a DMA Acknowledge from the DMA controller—before the possibility of a data overrun occurring in a 10 Mb/s Ethernet application. Once access to the system bus has been obtained, the 82592's high-performance, 16-bit bus interface provides efficient data transfer over the system bus, thus reducing the bus utilization load for a LAN connection on the host system.

The 82592 features a specialized hardware handshake with industry standard DMA controllers. This hardware handshake between the 82592 and the DMA controller (on signal lines DRQ and $\overline{EOP}$) relays the status of a Receive or Transmit and allows for back-to-back frame reception and automatic retransmission on collision without CPU intervention. This allows the 82592 and the DMA controller to perform these time-critical operations in real-time without depending on the CPU via an interrupt service routine, and without the time delays inherent in such routines. For the 82592 Embedded LAN Module, this hardware handshake is enabled by configuring the 82592 to the Tightly Coupled Interface (TCI) mode. Figure 2 shows details of the 82592's TCI signals.

Transmit/Receive Status Encoding on DRQ and $\overline{EOP}$

DRQ	$\overline{EOP}$	Status Information
0	Hi-Z	Idle
1	Hi-Z	DMA Transfer
0	0	Transmission or Reception Terminated OK
1	0	Transmission or Reception Aborted

Tightly Coupled Interface Timings



290189-4

Symbol	Parameter	Min	Max	Units	Notes
t_{23}	$\overline{WR}$ or $\overline{RD}$ Low to DRQ0 or DRQ1 Inactive		45	ns	C_L = 50 pF
t_{104}	$\overline{WR}$ or $\overline{RD}$ High to DRQ0 or DRQ1 Inactive	2.5	65	ns	C_L = 50 pF
t_{105}	$\overline{WR}$ or $\overline{RD}$ Low to $\overline{EOP}$ Active		45	ns	Open Drain I/O Pin
t_{106}	$\overline{EOP}$ Float after $\overline{DACK0}$ or $\overline{DACK1}$ Inactive		40	ns	Open Drain I/O Pin

Figure 2. TCI Encoding and Timings

These three features (FIFO depth, high-performance bus interface, and TCI) allow the 82592 to operate successfully in a high-performance motherboard LAN application. The application of these features will be discussed further in Section 4.

4.0 SYP301 INTERFACE

This section will discuss the details of the interface of the 82592 Embedded LAN Module to the Intel SYP301. The basic architecture will be presented, demonstrating that the 82592 Embedded LAN Module is a low-cost, low component count Ethernet solution for networking office PCs or workstations.

The Intel SYP301 is compatible with the IBM PC AT™. It features an Intel 80386™ microprocessor, running at 16 MHz, as its CPU. Its system bus is compatible with the standard PC AT I/O-channel bus.

4.1 Basic Architecture

Figure 3 shows the basic architecture of the 82592 Embedded LAN Module, and Figure 4 shows the module's schematics. The module consists of an 82592, two 20L10 PALs, and two 8-bit LS573 address latches that combine to provide a 16-bit address latch. The module contains no DMA unit or local memory.

The 82592 Embedded LAN Module is a simple, low-cost, low component count solution because it uses the available system resources (DMA and memory) to provide for those functions normally added to a LAN solution. Removing DMA and local memory from a LAN solution reduces cost and complexity. Two host DMA channels, one for receive and one for transmit, are needed to support the module. The DMA interface from the 82592 (through PAL B) is the standard combination of DRQ, $\overline{DACK}$ and $\overline{EOP}$. These three signals also provide the TCI between the 82592 and the DMA controller. The size of the memory buffer needed to support the module depends on the specific application and the amount of free memory available; the buffer size can be specified by the programmer.

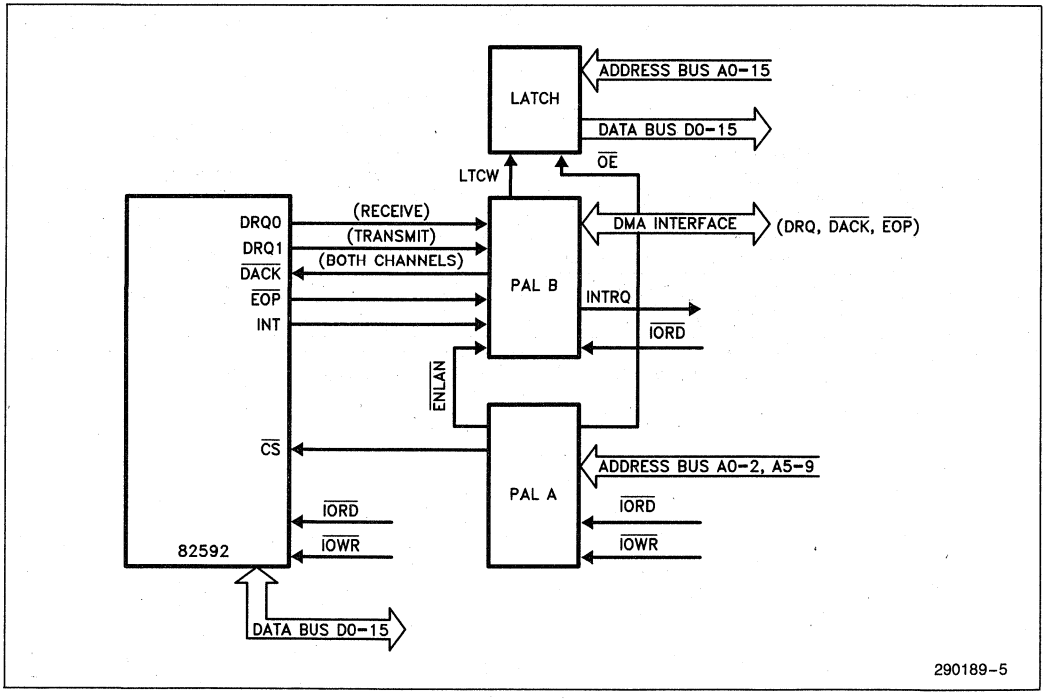


Figure 3. 82592 Embedded LAN Module Basic Architecture

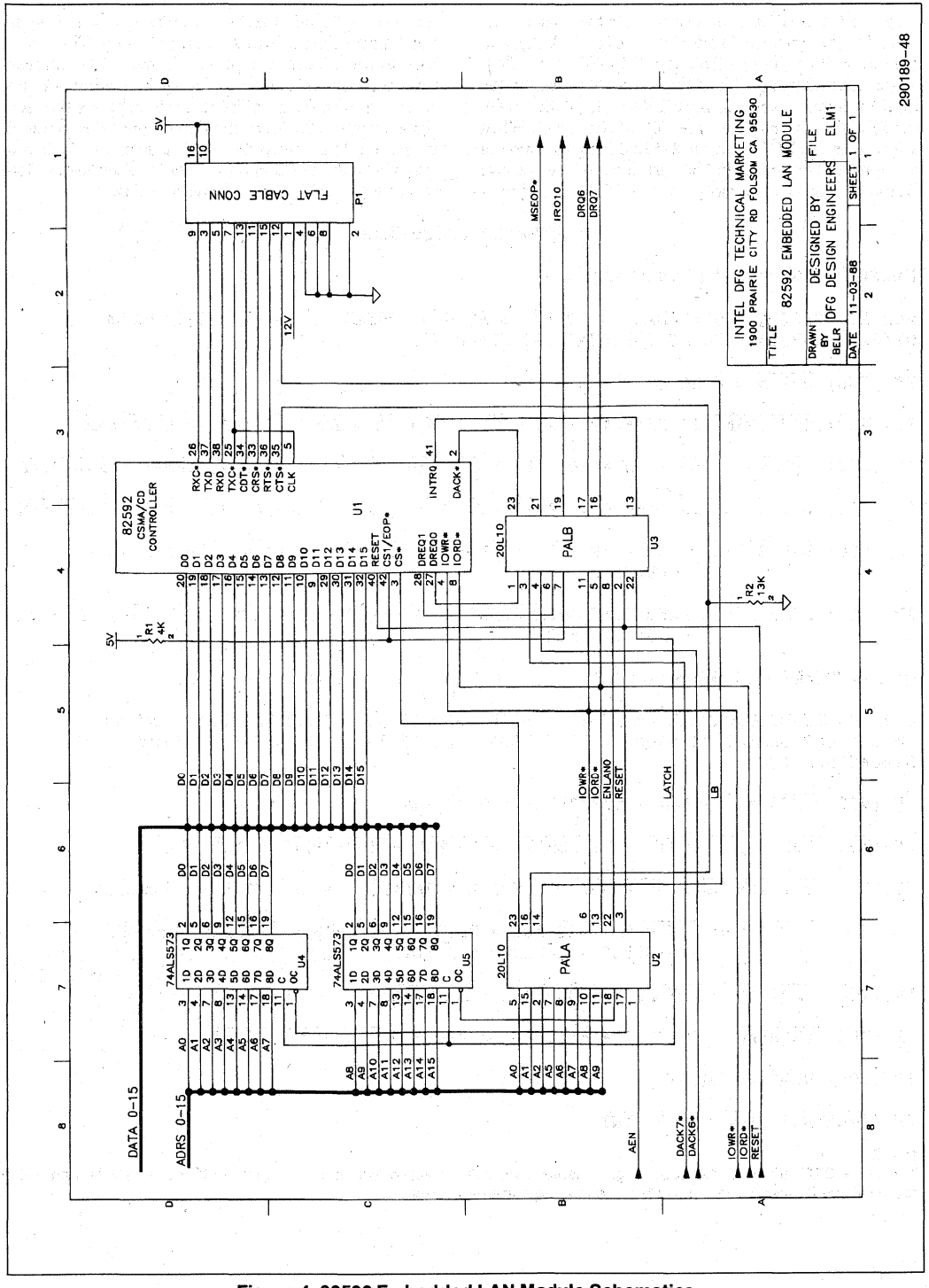


Figure 4. 82592 Embedded LAN Module Schematics

The two PALs (PAL A and B) provide two major junctions for the module: (1) address decode (PAL A), and (2) interpreting the TCI from the 82592 (PAL B). PAL A decodes addresses for $\overline{CS}$ to the 82592, $\overline{OE}$ for the address latches, and an Enable/Disable of the LAN module. PAL B interprets the TCI of the 82592. When PAL B detects $\overline{EOP}$ from the 82592 during reception of a frame ($\overline{EOP}$ indicates the last byte of the receive frame) it loads the memory address of the last byte of the receive frame (the byte count) into the Address Latch at the time it is written into memory. This allows back-to-back frame reception without CPU intervention, and will be covered in detail in Section 4.2. For Auto-Retransmit on collision, PAL B passes the $\overline{EOP}$ signal from the 82592 to the DMA controller, reinitializing the DMA controller for retransmission. This process will be discussed in more detail in Section 4.3. Both sets of PAL equations are listed in Table 1.

Table 1. PAL Equations

PAL20L10 MMI—PAL A (Version 1.1)

```
AEN A2 RESET NC A0 IOWBAR A5 A6 A7 A8 A9 GND IORBAR 501LB A1 59CTS OE2BAR
OE1BAR LANRSTBAR NC NC ENLANBAR 592CS0BAR VCC

IF (VCC) 501LB = 592CTS

IF (VCC) 592CS0BAR = AEN • A9 • A8 • A7 • A6 • A5 • A2 • A1 • A0 • ENLANBAR

IF (VCC) OE2BAR = AEN • A9 • A8 • A7 • A6 • A5 • A2 • A1 • A0 • IORBAR • ENLANBAR

IF (VCC) OE1BAR = AEN • A9 • A8 • A7 • A6 • A5 • A2 • A1 • A0 • IORBAR • ENLANBAR

IF (VCC) LANRSTBAR = AEN • A9 • A8 • A7 • A6 • A5 • A2 • A1 • A0 • IOWBAR •
                     ENLANBAR

IF (VCC) ENLANBAR = LANRSTBAR • ENLANBAR + AEN • A9 • A8 • A7 • A6 • A5 • A2 • A1
                    • A0 • IOWBAR
```

PAL20L10 MMI—PAL B (Version 1.1)

```
592DRQ0 RESET DACK7BAR DACK6BAR 10RBAR 592DRQ1 592E0PBAR ENLANBAR AEN NC
IOWBAR GND 592INT NC DRQ6BAR DRQ7 DRQ6 DISDACK IRQ10 NC MSE0PBAR LTCW
592DACKBAR VCC

IF (VCC) LTCW = IORBAR + 592E0PBAR + DACK7BAR

IF (ENLANBAR • 592E0PBAR • DACK6BAR) MSE0PBAR = 592E0PBAR • DACK6BAR

IF (VCC) 592DACKBAR = DACK6BAR • DISDACK • ENLANBAR + DACK7BAR • ENLANBAR

IF (VCC) DISDACK = IOWRBAR • DISDACK • RESET + 592DRQ0 • DISDACK • RESET
                   + 592DRQ0 • IOWRBAR • RESET

IF (VCC) DRQ7 = 592DRQ1 + 592E0PBAR • DACK7BAR

IF (VCC) DRQ6BAR = 592DRQ0 • RESET + DACK6BAR • DRQ6BAR • RESET

IF (VCC) DRQ6 = DRQ6BAR

IF (ENLANBAR) IRQ10 = 592INT
```

NOTE:
The suffix BAR added to the above signal names indicates an active low signal. A signal in these equations with a line drawn above it indicates this signal is to be in a low state for the equation.

4.2 Back-to-Back Frame Reception

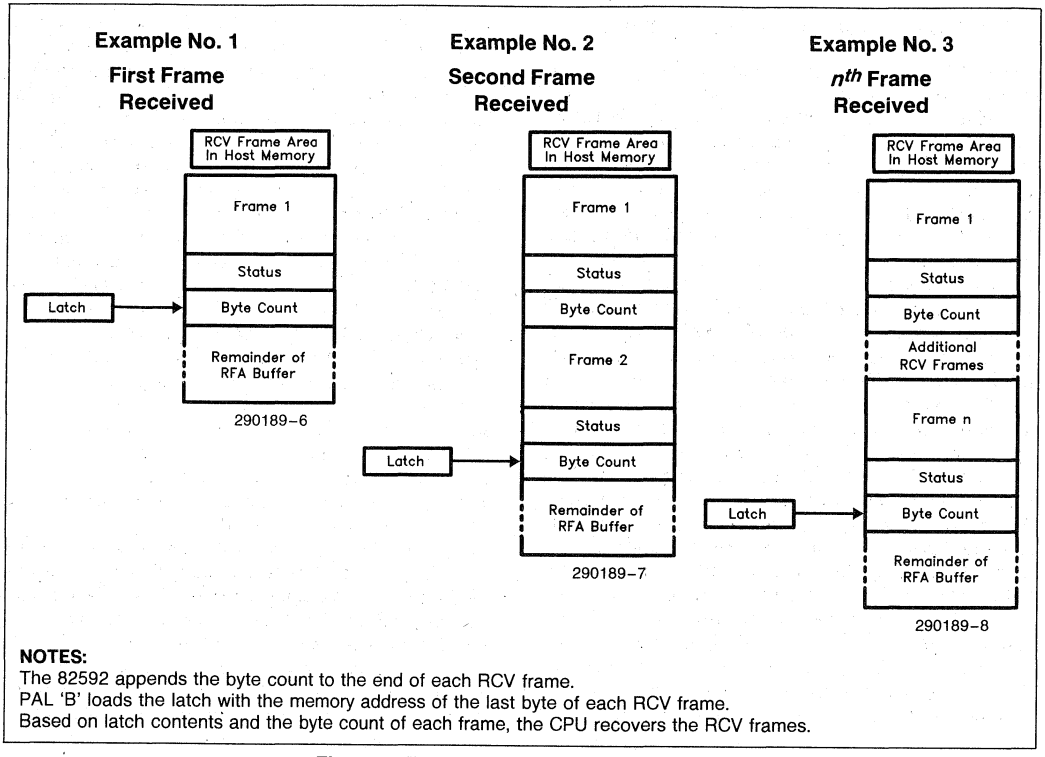
The architecture of the 82592 Embedded LAN Module allows it to receive back-to-back frames without CPU intervention. It uses a contiguous Receive Frame Area (RFA) buffer in host system memory where receive frames can be continuously stored. This sequential storage of receive frames can continue until the buffer space is exhausted. The size of the RFA buffer can be specified by the programmer. Its size will be programmed as the byte count of the Rx DMA channel. The Base Address Register contents of that channel serve as the start address of the RFA buffer. The receive frames will be stored sequentially in the RFA buffer based on the contents of the Current Address Register of the Rx DMA channel. The module's architecture, and the 82592 receive frame memory structure, allows the CPU to recover the addresses of each Receive frame in memory for processing. The CPU can also reinitialize the RFA buffer (by reinitializing the Rx DMA channel) as the RFA buffer fills up and its contents are processed. Alternatively, configuring the Rx DMA channel to Auto-Initialize mode will allow the Rx buffer to automatically wrap around, back to the beginning of the buffer, when its end is reached. This creates a virtual "endless" circular buffer. When using this approach, care must be taken to avoid writing over unprocessed Rx frames—either by the addition of a hardware Stop Register, or by guaranteeing that the Rx frames can be processed faster than the buffer can wrap around.

Back-to-back frame reception without CPU intervention—and eventual recovery of the frames for processing by the CPU—is based on PAL B's decoding of the TCI signals of the 82592 (PAL B loads the address latch with the address of the last byte of the received frame) and the structure of the received frame transferred from the 82592 to memory. Figure 5 shows the format of an 82592 receive frame in TCI mode. After the information fields are written to memory, the Status and byte count of the received frame are appended to the frame in memory. These four bytes (two bytes of Status and two bytes of byte count) are the last four bytes of the receive frame written to memory. The high byte of the byte count is the last byte transferred from the 82592 to memory. As this last byte is transferred to memory, the 82592 asserts the $\overline{EOP}$ signal. When PAL B detects the assertion of $\overline{EOP}$ by the 82592, it loads the address of the last byte of the receive frame into the Address Latch as this byte is written into memory. This action ensures that there will always be a pointer (the contents of the Address Latch) to the byte count of the last frame stored in the RFA buffer in system memory. Based on the value of the byte count, the beginning address of the receive frame in memory can be calculated; i.e., Byte Count Address Pointer − Byte Count = Beginning of Frame. The byte count of a previous receive frame would reside one address location before the first byte of the current receive frame. That frame, and any additional receive frames that may have preceded it, can have their start addresses recovered by the same calculation used to recover the last frame received. This process allows frames to be continually stored in the RFA buffer without CPU intervention, and to be recovered by the CPU for processing. Figure 6 illustrates the process of back-to-back frame reception.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DESTINATION ADDRESS SECOND BYTE								DESTINATION ADDRESS FIRST BYTE							
DESTINATION ADDRESS LAST BYTE															
SOURCE ADDRESS SECOND BYTE								SOURCE ADDRESS FIRST BYTE							
SOURCE ADDRESS LAST BYTE															
INFORMATION (LENGTH FIELD, HIGH)								INFORMATION (LENGTH FIELD, LOW)							
INFORMATION LAST BYTE															
CRC BYTE 1*								CRC BYTE 0*							
CRC BYTE 3*								CRC BYTE 2*							
X	X	X	X	X	X	X	X	SHORT FRAME	NO EOF	TOO LONG	1	NO SFD	NO ADD MATCH	I-A MATCH	Rx CLD
X	X	X	X	X	X	X	X	0	0	Rx OK	LEN ERR	CRC ERROR	ALG ERROR	0	OVER RUN
X	X	X	X	X	X	X	X	BYTE COUNT LOW							
X	X	X	X	X	X	X	X	BYTE COUNT HIGH							

*The CRC bytes are transferred to memory only when the device is so configured

Figure 5. Receive Format for the 82592 in 16-Bit Mode (Tightly Coupled Interface Enabled)



NOTES:
The 82592 appends the byte count to the end of each RCV frame.
PAL 'B' loads the latch with the memory address of the last byte of each RCV frame.
Based on latch contents and the byte count of each frame, the CPU recovers the RCV frames.

Figure 6. Back-to-Back Frame Reception

4.3 Automatic Retransmission on Collision

Automatic retransmission on collision detection is accomplished by the TCI between the 82592 and the host 8237 DMA controller and requires no CPU intervention. The transmit channel of the 8237 should be configured for Auto-Initialize mode. The transmit block (data to be transmitted) starts at the location pointed to by the Base Address Register of the Tx DMA channel. During a Transmit command, the 82592 DMA requests begin at the start of the transmit block and work sequentially through the block (by incrementing the contents of the 8237's Current Address Register) until the transmission is complete. Should a collision occur, the 82592 asserts the $\overline{EOP}$ signal and DRQ* to the 8237 (these signals pass through PAL B) causing the 8237 to auto-initialize back to the beginning of the transmit block (the Current Address Register is loaded with the value in the Base Address Register). Internally, the 82592 generates a Retransmit command and begins making DMA requests to the 8237, which is now pointing to the beginning of the transmit block. The 82592 also enters into a back-off algorithm (counting to a random number to resolve the collision). When the back-off algorithm is complete, and the 82592 regains access to the serial link, retransmission is attempted. The 82592 will repeat this process until the retransmission is completed successfully or until the maximum allowable number of collisions per Transmit command is reached—at that point all retransmit attempts stop. No CPU involvement is required to carry out a retransmission. The process of automatic retransmission is shown in Figure 7.

NOTE:
*For Auto-Initialization of the 8237, the signal DRQ must be asserted to the 8237 along with assertion of $\overline{EOP}$. With the 82380 and 82370 DMA controllers, Auto-Initialization can be triggered by asserting the $\overline{EOP}$ signal alone.

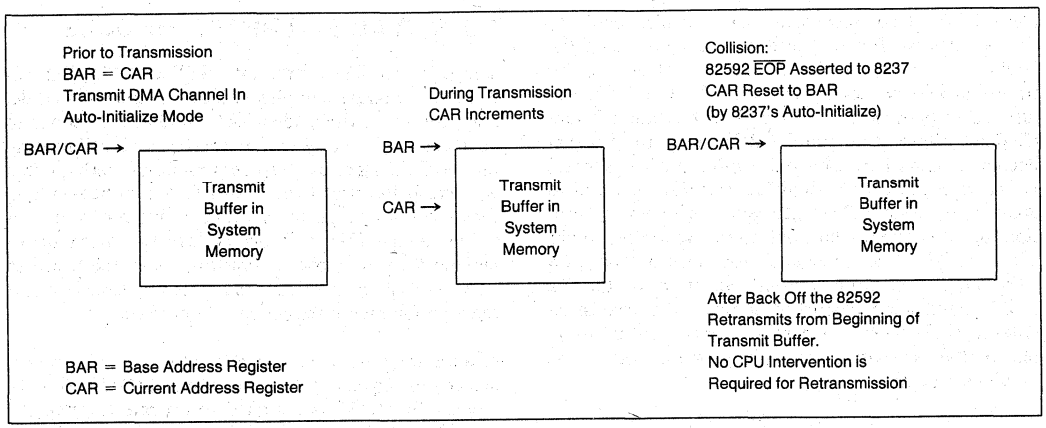


Figure 7. Automatic Retransmission on Collision

4.4 Target Systems for Integration

The 82592 Embedded LAN Module is designed to be implemented on an Intel SYP301 motherboard; thereby demonstrating a low-cost LAN connection for a workstation. The SYP301 has an IBM PC AT style bus architecture with a 32-bit Intel 80386 as the main processor. The interface between the 82592 LAN Module and the SYP301 is based on standard interface signals (DRQ, $\overline{\text{DACK}}$, $\overline{\text{EOP}}$, IRQ, $\overline{\text{IOR}}$, $\overline{\text{IOW}}$, etc.) so the basic architecture of the module can be implemented on PC AT based systems. This design has been successfully tested in PC AT style systems produced by several manufacturers. For some PC AT based systems, and PS/2 Micro Channel systems, the module's design may require some modification. IBM PC and PC XT based systems do not have sufficient DMA bandwidth to support the non-buffered architecture of this module.

4.4.1 PC AT BASED DESIGNS

High-integration chip sets replace a large number of discrete VLSI, LSI, and TTL components with several integrated VLSI devices that duplicate a large portion of the PC's functionality. PC AT compatible systems using such chip sets may lack support for the automatic retransmission feature of the 82592 LAN Module. This is because many manufacturers of such chip sets have integrated the $\overline{\text{EOP}}$ function but eliminated the $\overline{\text{EOP}}$ input. This lack of an $\overline{\text{EOP}}$ input disables auto-initialization of the DMA controller for retransmission. In this case retransmission can be performed in one of two ways.

- Should a collision occur while transmitting the preamble, the 82592 (when configured to automatic retransmission mode) will automatically retransmit without CPU intervention or auto-initialization of the DMA. This is effective for shorter network topologies where collisions are normally detected early in the frame.
- Should a collision occur after the preamble, the 82592 will interrupt the CPU and the CPU will initiate the retransmission.

For a PC AT style architecture, logic must be implemented to accommodate DRAM refresh. DRAM refresh cycles typically occur at 15 μs intervals. In a standard PC AT, any DMA user should limit the time of a DMA burst to 15 μs; this is to ensure that the system bus is free for the refresh to take place. Any designer using burst mode DMA must consider this requirement when implementing a design.

4.4.2 PS/2 MICRO CHANNEL ARCHITECTURE DESIGNS

The IBM PS/2 and other compatibles using the Micro Channel architecture have a different host interface to the 82592 Embedded LAN Module; however, the basic architecture of the module is still applicable. As in the SYP301 solution, the TCI between the 82592 and a

control PAL loads the address latch with a pointer to the last receive frame. Based on the contents of the latch and the 82592 receive memory structure, the frames are recovered for processing by the CPU. The differences between a PC AT architecture and a Micro Channel architecture require different control signal decoding. The Micro Channel requires a 24-bit address latch, as opposed to a 16-bit latch in the 301, and to acquire the system DMA it requires different arbitration logic to drive a 4-bit arbitration level on the Micro Channel. The Micro Channel also does not have an $\overline{EOP}$ input; therefore, auto-initialization of the Tx DMA channel and support of automatic retransmission without CPU intervention must be provided by using one of the alternative methods recommended in the previous section.

4.4.3 EMBEDDED CONTROL DESIGNS

The 82592 Embedded LAN Module architecture can also be applied to an embedded control application that contains some DMA functions. For an embedded application using an 8237, 82380 or 82370 DMA controller, the basic architecture of the 82592 Embedded LAN Module can be used. For an interface to DMA devices that do not feature the $\overline{EOP}$ signal as an input (for example, DMA units on board a CPU), the alternative methods for retransmission given earlier can be used.

5.0 SERIAL INTERFACE MODULE

The serial interface for the Intel SYP301 82592 Embedded LAN Module is implemented as a separate module. Since the 82592 Embedded LAN Module is intended to be integrated into a system motherboard, implementing the serial interface as a separate module—perhaps as a very small PC board that plugs into a socket—allows for easy interchangeability between different serial interface media. This modularity allows the system board manufacturer to avoid committing his motherboard to only one type of medium, and thus requiring a major redesign for each different serial interface.

Modularity in the data communications field is encouraged by the Open Systems Interconnect (OSI) reference model. The 82592 is designed to operate through the lower half of the Data Link Layer (see Figure 8), implementing CSMA/CD Medium Access Control and interfacing directly with the Physical layer below it. By interfacing the 82592's standard CSMA/CD interface signals to a serial module (TxD, RxD, $\overline{TxC}$, $\overline{RxC}$, $\overline{CDT}$, $\overline{CRS}$, and others) different Physical Link modules can be implemented without any change to software. Examples of serial interface modules that could be interchanged by simply plugging a new module into the motherboard are Ethernet/Cheapernet, Twisted Pair Ethernet (TPE), StarLAN, Broadband Ethernet, and many proprietary CSMA serial media. Figure 9 shows the schematics of an Ethernet module; and Figure 10 those of an Ethernet/Cheapernet module.

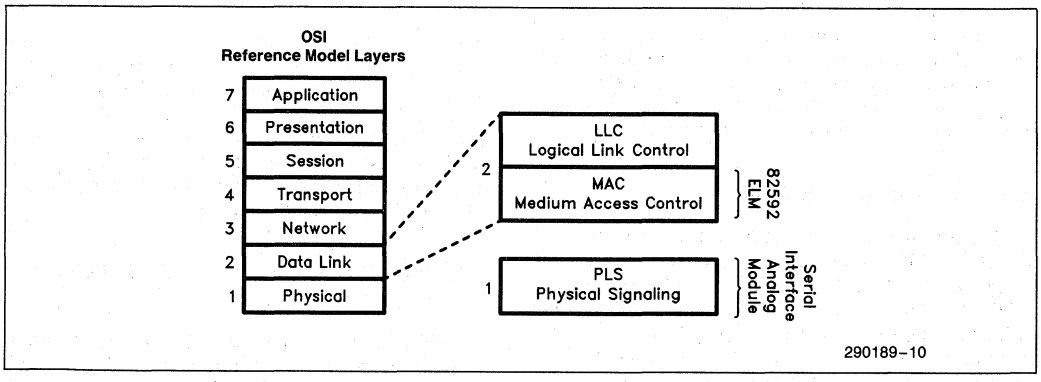


Figure 8. The 82592 Embedded LAN Module Relationship to the OSI Reference Model

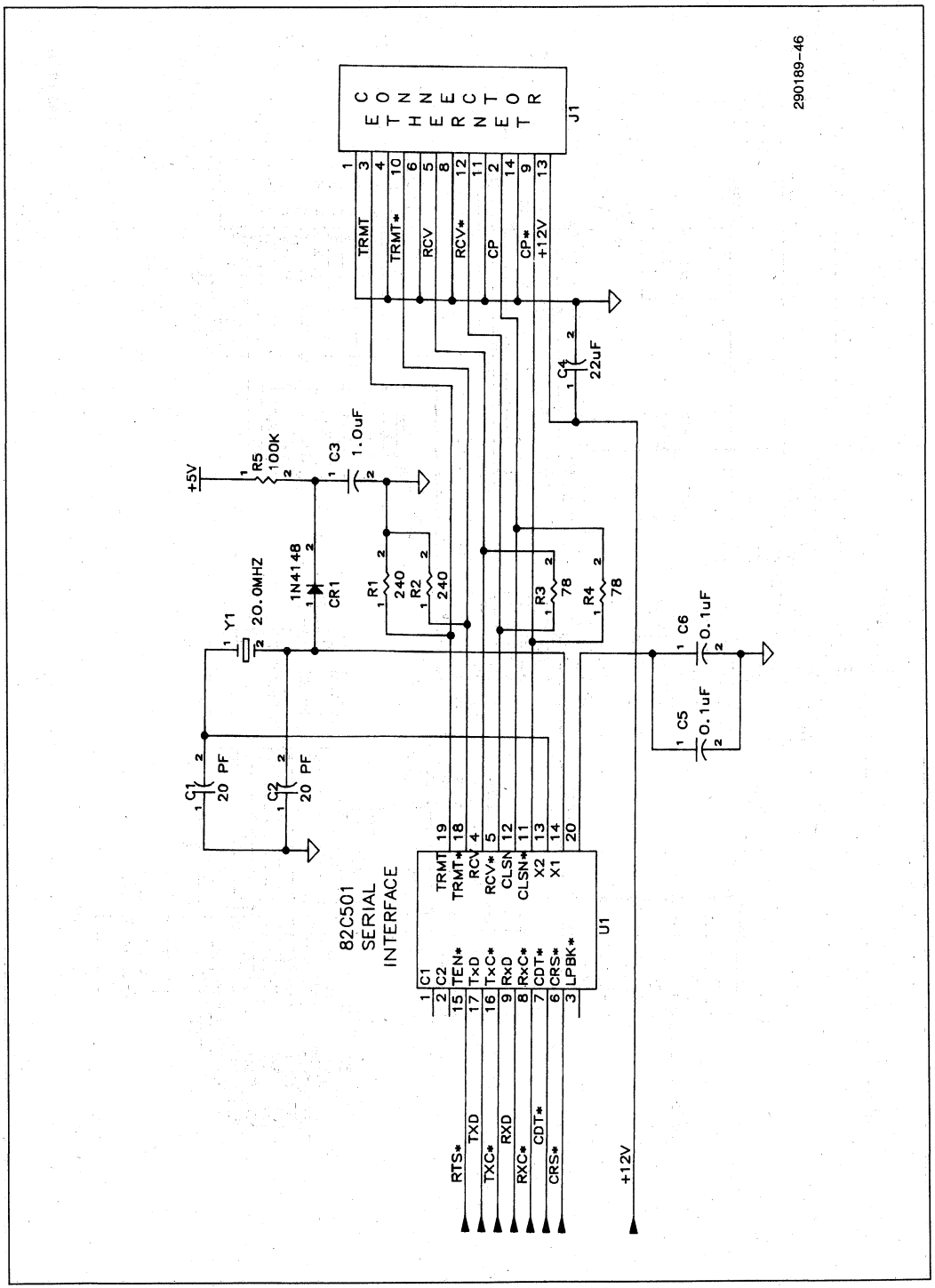


Figure 9. Ethernet Analog Module

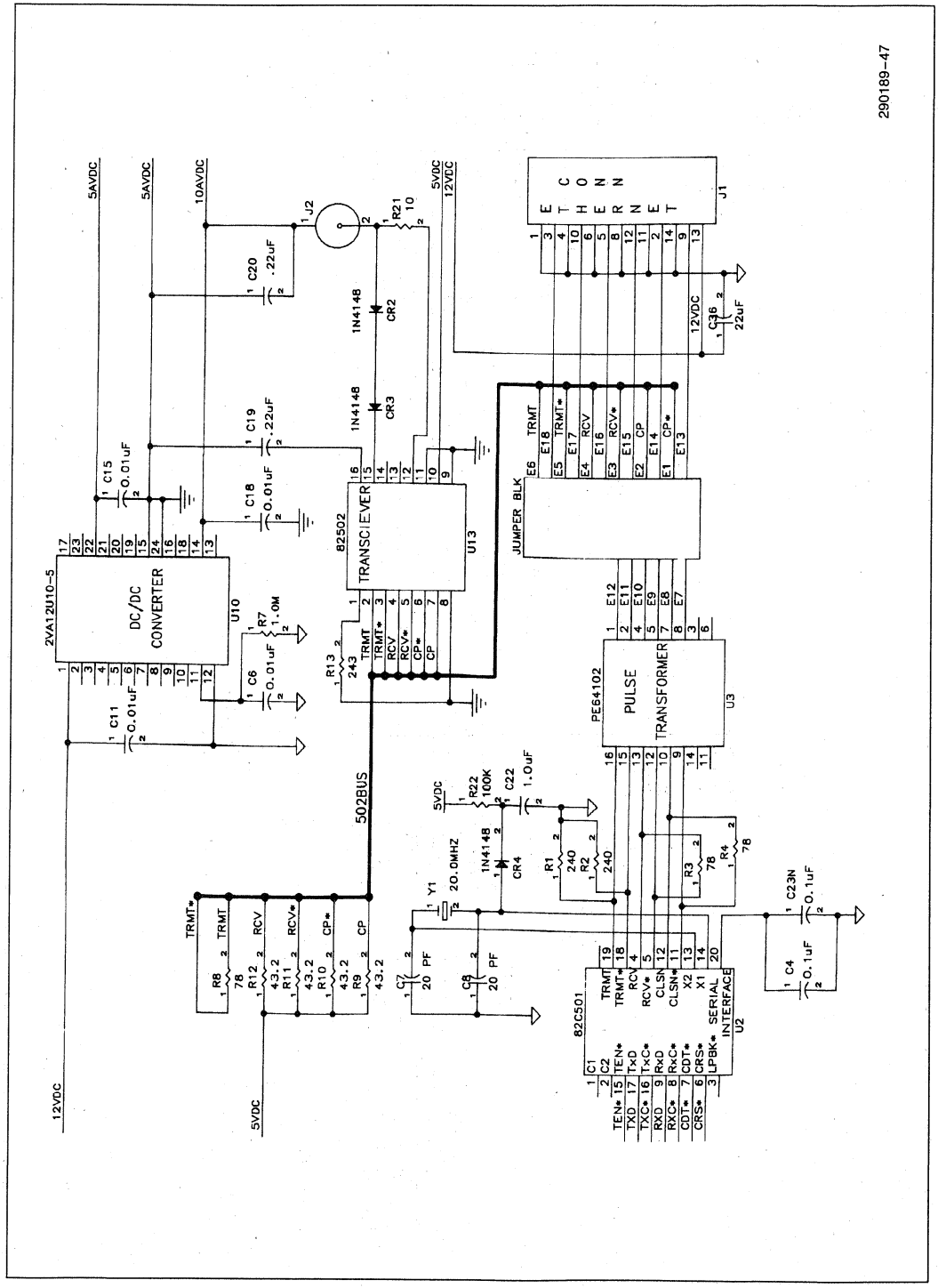


Figure 10. Ethernet/Cheapernet Analog Module

6.0 PERFORMANCE COMPARISON

Figure 11 compares the performance of the 82592 Embedded LAN Module with the PC586E nonintelligent, buffered adapter. The PC586E is an Intel evaluation board based on the Intel 82586 LAN Coprocessor. It contains 16 kB of local memory, has a 16-bit bus interface, and has a high-performance arbitration scheme providing both the CPU and the 82586 LAN controller zero wait state access to local memory. The PC586 has been characterized in the industry as one of the highest performance nonintelligent, buffered adapters available.

A performance comparison, using Novell's *Perform* 2 utility, shows that the 82592 Embedded LAN Module, operating as a workstation accessing a file server, outperforms the PC586E. For all tests the host system was an Intel SYP301. The SYP301 was run in both standard mode, a nominal 16 MHz*, and in its reduced speed mode, 6 MHz. In all cases the SYP301 system DMA operates at 4 clocks per cycle at 4 MHz. The file server was a Novell 286A, an 8 MHz, zero wait state system, using a PC586E as the LAN adapter. The tests recorded are for one node on the network (the workstation under test). For write tests to the file server's hard disk, the performance numbers are generally the same. This is due to limitations in accessing the file server's hard disk. This slow access causes a bottleneck. For the read tests the workstations are accessing files stored in cache memory, thus removing the bottleneck for this test. Without this limitation, the 82592 Embedded LAN Module accesses the file server at a higher rate than the PC586E: at full speed, 318 kB/s vs 282.3 kB/s; and at reduced speed, 202.8 kB/s vs 195.2 kB/s.

7.0 SOFTWARE EXAMPLES

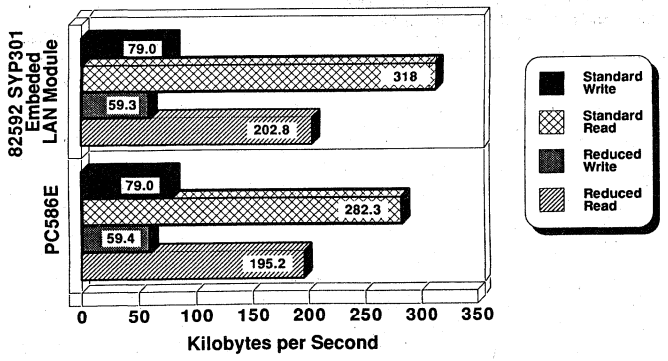
The following examples are from a driver written for an 82592 Embedded LAN Module operating in an Intel SYP301. The driver was written by Joe Dragony, Intel Data Communications Technical Marketing Engineer. The excerpts will cover (1) declarations of program constants and variables, (2) initializing the Embedded LAN Module hardware and buffer space, (3) assembly and transmission of a frame, and (4) processing received frames. A brief description of each of these processes is followed by excerpts from the code. The driver uses the Xerox Internetwork Packet Exchange (IPX) protocol and serves as a software interface between the 82592 Embedded LAN Module hardware and the IPX.

Exerciser Software for the 82592 Embedded LAN Module is also available from Intel. Detailed documentation for both the exerciser program and the network driver are available upon request from Intel.

7.1 Declarations

Table 2 shows declarations of program variables and equates of program constants. This section is included to help the reader understand the following program excerpts.

*NOTE:
The benchmark program *Landmark CPU Speed Test*, © 1986 by Landmark Software, shows an effective throughput of 14.3 MHz for a SYP301 in standard mode; and 5.4 MHz in reduced speed mode.



290189-11

NOTES:
Novell *Perform 2 Version 2.3*
File Server: 286A. 8 MHz, Zero-Wait-State with PC586E LAN Adapter
Node System: SYP301 (One Node on Network)
Reduced Speed Mode: Equivalent to 5.4 MHz AT
Standard Mode: Equivalent to 14.3 MHz AT
301 System DMA: 4 MHz, Four Clocks per Transfer

Figure 11. 82592 SYP301 Embedded LAN Module vs PC586E Buffered Adapter

Table 2. Declarations

```
$%*define(slow) local label (
    jmp     short %label
%label:
)

%*define(fastcopy) local label (
    shr cx, 1
    rep movsw
    jnc %label
    movsb
%label:
)

%*define(inc32 m) (
    add word ptr %m[0], 1
    adc word ptr %m[2], 0
)

    name    LANOnMotherboardModule

CGroup      group    Code, mombo_init

    assume    cs: CGroup, ds: CGroup

Code    segment word public 'CODE'

    public    DriverSendPacket
    public    DriverBroadcastPacket
    public    DriverPoll

    public    LANOptionName

    extrn     IPXGetECB: NEAR
    extrn     IPXReturnECB: NEAR
    extrn     IPXReceivePacket: NEAR
    extrn     IPXReceivePacketEnabled: NEAR
    extrn     IPXHoldEvent: NEAR
    extrn     IPXServiceEvents: NEAR
    extrn     IPXIntervalMarker: word
    extrn     MaxPhysPacketSize: word
    extrn     ReadWriteCycles: byte
    extrn     IPXStartCriticalSection: NEAR
    extrn     IPXEndCriticalSection: NEAR
```

290189-16

Table 2. Declarations (Continued)

```
;;;;;;;;;;;;;;;;;;;;
;     Equates
;;;;;;;;;;;;;;;;;;;;

CR                          equ  0Dh
LF                          equ  0Ah
BAD                         equ  0FFh
BPORT                       equ  0
IRQLOC                      equ  19
DMA0LOC                     equ  23
DMA6LOC                     equ  25
TransmitHardwareFailure     equ  0FFh
PacketUnDeliverable         equ  0FEh
PacketOverflow              equ  0FDh
ECBProcessing               equ  0FAh
TxTimeOutTicks              equ  20


;  Latch definitions
TenCentLo    equ  301h
TenCentHi    equ  302h

;  Enables for 10cent
EnLAN     equ  303h
DisLAN    equ  304h

;  8259  definitions

InterruptControlPort        equ    020h
InterruptMaskPort           equ    0A1h  ;for secondary 8259A
ExtraInterruptControlPort   equ    0A0h
EOI                         equ    020h

;  8237 definitions

DMAcmdstat     equ  0D0h
DMAreq         equ  0D2h
DMAsnglmsk     equ  0D4h
DMAmode        equ  0D6h
DMAff          equ  0D8h
DMAtmpclr      equ  0DAh
DMAclrmsk      equ  0DCh
DMAallmsk      equ  0DEh
DMA6page       equ  089h
DMA6addr       equ  0C8h
DMA6wdcount    equ  0CAh
DMA7page       equ  08Ah
DMA7addr       equ  0CCh
DMA7wdcount    equ  0CEh
DMAtx6         equ  01Ah      ;demand mode, autoinit, read transfer
DMAtx7         equ  01Bh      ;demand mode, autoinit, read transfer
DMArx6         equ  006h      ;demand mode, no autoinit, write transfer
DMArx7         equ  007h      ;demand mode, no autoinit, write transfer
DMA6msk        equ  006h
DMA6unmsk      equ  002h
DMA7msk        equ  007h
DMA7unmsk      equ  003h
DMAena         equ  0h
```

290189-17

Table 2. Declarations (Continued)

```
NetWareType   equ    1111h


; 82592 Commands

C_NOP      equ  00h
C_SWP1     equ  10h
C_SELRST   equ  0Fh
C_SWP0     equ  01h
C_IASET    equ  01h
C_CONFIG   equ  02h
C_MCSET    equ  03h
C_TX       equ  04h
C_TDR      equ  05h
C_DUMP     equ  16h
C_DIAG     equ  07h
C_RXENB    equ  18h
C_ALTBUF   equ  09h
C_RXDISB   equ  1Ah
C_STPRX    equ  1Bh
C_RETX     equ  0Ch
C_ABORT    equ  0Dh
C_RST      equ  0Eh
C_RLSPTR   equ  0Fh
C_FIXPTR   equ  1Fh
C_INTACK   equ  80h


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Data Structures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
even
hardware_structure    struc
    io_addr1          dw     ?
    io_range1         dw     ?
    io_addr2          dw     ?
    decode_range2     dw     ?
    mem_addr1         dw     ?
    mem_range1        dw     ?
    mem_addr2         dw     ?
    mem_range2        dw     ?
    int_used1         db     ?
    int_line1         db     ?
    int_used2         db     ?
    int_line2         db     ?
    dma_used1         db     ?
    dma_chan1         db     ?
    dma_used2         db     ?
    dma_chan2         db     ?
hardware_structure    ends

ecb_structure     struc
    link                      dd   0
    esr_address               dd   0
    in_use                    db   0
    completion_code           db   0
```

290189-18

Table 2. Declarations (Continued)

```
    socket_number               dw   0
    ipx_workspace               db   4   dup (0)
    driver_workspace            db   12  dup (0)
    immediate_address           db   6   dup (0)
    fragment_count              dw   1
    fragment_descriptor_list    db   6   dup (?)
ecb_structure     ends

fragment_descriptor    struc
    fragment_address  dd   ?
    fragment_length   dw   ?
fragment_descriptor    ends

rx_buf_structure    struc
    rx_dest_addr         db   6 dup (?)
    rx_source_addr       db   6 dup (?)
    rx_physical_length   dw   ?
    rx_checksum          dw   ?
    rx_length            dw   ?
    rx_tran_control      db   ?
    rx_hdr_type          db   ?
    rx_dest_net          db   4 dup (?)
    rx_dest_node         db   6 dup (?)
    rx_dest_socket       dw   ?
    rx_source_net        db   4 dup (?)
    rx_source_node       db   6 dup (?)
    rx_source_socket     dw   ?
rx_buf_structure    ends

tci_status    struc
    status0     db    ?
    dead1       db    ?
    status1     db    ?
    dead2       db    ?
    bc_lo       db    ?
    dead3       db    ?
    bc_hi       db    ?
tci_status    ends

ipx_header_structure    struc
    checksum              dw   ?
    packet_length         dw   ?
    transport_control     db   ?
    packet_type           db   ?
    destination_network   db   4 dup (?)
    destination_node      db   6 dup (?)
    destination_socket    dw   ?
    source_network        db   4 dup (?)
    source_node           db   6 dup (?)
    source_socket         dw   ?
ipx_header_structure    ends


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    even
```

290189-19

Table 2. Declarations (Continued)

```
tx_start_time   dw  0
adapter_io      dw  ?
config          dw  ?
send_list       dd  0          ;points to list of ECBs to be sent
buffer_segment  dw  ?
rx_ecb          dd  ?
tx_ecb          dd  ?

config_block    db
0Fh,00h,48h,80h,26h,00h,60h,00h,0F2h,00h,00h,40h,0F5h,00h,3Fh,87h,0F0h,0DFh

temp_flag           db    0
int_mask_register   dw    ?
old_irq_vector      dd    ?
int_vector_addr     dw    ?
int_bit             db    ?
int_mask            db    ?
command_reg         dw    300h        ;82592 port 0 address
read_in_length      dw    ?
config_dma0_loc     db    ?
config_dma1_loc     db    ?
config_irq_loc      db    ?
config_bport        dw    ?
tx_active_flag      db    0
frame_status        db    0
status10            db    0
status11            db    0
status20            db    0
status21            db    0

even

gp_buf              dw   5000 dup (0)   ;twice the required size
gp_length           dw   1388h
gp_buf_offset       dw   cgroup:gp_buf
gp_offset_adjust    dw   0
gp_buf_start        dw   0      ;A1-A16 of General Purpose Buffer EA
gp_buf_page         dw   0      ;A17-A23 of General Purpose Buffer EA
tx_byte_cnt         dw   0      ;IPX packet length plus header length
rx_buf_start        dw   0      ;A1-A16 of General Purpose Buffer EA
rx_buf_page         dw   0      ;A17-A23 of General Purpose Buffer EA
rx_buf_head         dw   0      ;current rx head, buffer has been flushed to
here
rx_buf_tail         dw   0      ;value read from 10 cent latches
rx_buf_ptr          dw   0      ;used during rx list generation
rx_buf_stop         dw   0      ;point to reset the DMA controller
rx_buf_length       dw   0
rx_buf_segment      dw   0      ;calculated at init for use by IPXReceivePacket
curr_rx_length      dw   0
rx_list             dw   180 dup (0)
num_of_frames       dw   0
reset_rx_buf        dw   0
padding             dw   0

;
;    Define Hardware Configuration
;
```

290189-20

Table 2. Declarations (Continued)

```
ConfigurationID         db    'NetWareDriverLAN WS

SDriverConfiguration    LABEL       byte

reserved1               db    4 dup (0)
node_addr               db    6 dup (0)
reserved2               db    0     ;non-zero means is a real driver.
node_addr_type          db    0     ;address is determined at initialization
max_data_size           dw    1024 ;largest read data request will handle
(512, 1024, 2048, 4096)
lan_desc_offset         dw    LANOptionName
lan_hardware_id         db    0AAh     ;Bogus Type Code
transport_time          dw    1     ;transport time
reserved_3              db    11 dup (0)
major_version           db    01h ;Bogus version number
minor_version           db    00h
flag_bits               db    0
selected_configuration  db    0     ;board configuration (interrupts, IO
addresses, etc.)
number_of_configs       db    01
config_pointers         dw    configuration0

LANOptionName       db    'Intel LAN-On-Motherboard Module',0,'$'

configuration0      dw    300h, 16, 0, 0          ;IO ports and ranges
        db      0
        dw      0 , 0
        db      0
        dw      0 , 0         ;memory decode
        db      0FFh, 10, 0, 0     ;interrupt level 10
        db      0FFh, 6, 0FFh, 7     ;DMA channels 6 and 7
        db      0,0
        db      'IRQ 10, IO Addr = 300h, DMA 6 and 7, For Evaluation Only', 0

;*******************************************************
;
;    Error Counters
;
;*******************************************************
    Public DriverDiagnosticTable,DriverDiagnosticText

DriverDiagnosticTable  LABEL  byte

DriverDebugCount          dw    DriverDebugEnd-DriverDiagnosticTable
DriverVersion             db    01,00
StatisticsVersion         db    01,00
TotalTxPacketCount        dw    0,0
TotalRxPacketCount        dw    0,0
NoECBAvailableCount       dw    0
PacketTxTooBigCount       dw    -1        ;not used
PacketTxTooSmallCount     dw    -1        ;not used
PacketRxOverflowCount     dw    0
PacketRxTooBigCount       dw    0
PacketRxTooSmallCount     dw    0
PacketTxMiscErrorCount    dw    -1        ;not used
PacketRxMiscErrorCount    dw    -1        ;not used
RetryTxCount              dw    0
ChecksumErrorCount        dw    -1        ;not used
```

290189-21

Table 2. Declarations (Continued)

```
HardwareRxMismatchCount  dw   0
NumberOfCustomVariables  dw   (DriverDiagnosticText-DriverDebugEnd1)/2

DriverDebugEnd1  LABEL    byte


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Driver Specific Error counts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


rx_errors              dw  0
underruns              dw  0
no_cts                 dw  0
no_crs                 dw  0
rx_aborts              dw  0
no_590_int             dw  0
false_590_int          dw  0
lost_rx                dw  0
stop_tx                dw  0
ten_cent_latch_crash   dw  0
rx_disb_failure        dw  0
tx_abort_failure       dw  0
rx_buff_ovflw          dw  0
tx_timeout             dw  0

DriverDiagnosticText        LABEL    byte


    db      'RxErrorCount',0
    db      'UnderrunCount',0
    db      'LostCTSCount',0
    db      'LostCRSCount',0
    db      'RxAbortCount',0
    db      'No590InterruptCount',0
    db      'False590InterruptCount',0
    db      'LostOurReceiverCount',0
    db      'QuitTransmittingCount',0
    db      'TencentLatchCrashCount',0
    db      'RxDisableFailureCount',0
    db      'TxWontAbort',0
    db      'ReceiveBufferOverflow',0
    db      'TxTimeoutErrorCount',0

    db      0,0


DriverDebugEnd    LABEL  word
```

290189-22

7.2 Initialization Routine

This routine, Driver Initialize, initializes the Embedded LAN Module hardware and the system hardware needed to support the module. It also sets up the system memory structure to support the module.

7.2.1 HARDWARE INITIALIZATION AND 82592 CONFIGURATION

Initialization of the Embedded LAN Module hardware begins with generating an individual address for the station, initializing the interrupt line and interrupt vector, and enabling the module by writing to port address 303h. After initializing the memory structure, the 82592 is directly programmed. This programming includes configuring the 82592 and initializing it with the station's individual address. The 82592 is configured in two steps. The first specifies a 16-bit-wide system bus interface by issuing a Configure command to the 82592, with 00h as the byte count; i.e., no parameters passed to the device. Then a second Configure command is issued; it does the following.

- The 82592 is put in High Speed Mode to support Ethernet serial bit rates.
- It is placed in TCI mode for interface to the Embedded LAN Module architecture.
- All network parameters (e.g., Frame Length, Slot Time, and Preamble Length) are set up for default Ethernet values.

Following this initialization and configuration of the module's hardware, the 8259A Programmable Interrupt Controller's interrupt line for the module is enabled, allowing the interrupt-driven events frame reception and completed transmission. Then a Receive Enable command is issued to the 82592. Table 3 contains the code for hardware initialization.

Table 3. Hardware Initialization

```
mombo_init    segment 'CODE'

    public    DriverInitialize, DriverUnHook
no_card_message           db  CR,LF,'No adapter installed in PC$'
config_failure_message    db  CR,LF,'Configuration Failure$'
iaset_failure_message     db  CR,LF,'IA Setup Failure$'
ConfigDataUnderrunMess    db  CR,LF,'Configuration underrun$'

;
;   Driver Initialize
;
;   assumes:
;     DS, ES are set to CGroup (== CS)
;     DI points to where to stuff node address
;     Interrupts are ENABLED
;     The Real Time Ticks variable is being set, and the
;     entire AES system is initialized.
;
;   returns:
;     If initialization is done OK:
;          AX has a 0
;     If board malfunction:
;          AX gets offset (in CGroup) of '$'-terminated error string
;

DriverInitialize PROC    NEAR
    mov    MaxPhysPacketSize, 1024
    cli
    cld
    mov  ax, cs
    mov  ds, ax
    mov  es, ax
;    get DOS time and use for address.
    mov    ah,02Ch
    int    21h
    mov    bx, OFFSET CGroup: node_addr
    mov    byte ptr cgroup:[bx], 00h
    mov    byte ptr cgroup:[bx+1], 0AAh
    mov    byte ptr cgroup:[bx+2], ch
    mov    byte ptr cgroup:[bx+3], dl
    mov    byte ptr cgroup:[bx+4], dh
    mov    byte ptr cgroup:[bx+5], 7Eh
    mov    si, bx
    movsw           ;stuff address at point IPX indicated
    movsw
    movsw
    sti

; initialize the configuration table
    mov    al,selected_configuration
    cbw
    shl    ax,1             ; multiply by two
    add    ax,OFFSET CGROUP:config_pointers    ;ax contains the offset value
```

290189-23

Table 3. Hardware Initialization (Continued)

```
    mov     bx,ax                   ;of the default configuration
    mov     bx,[bx]                   ;list
    mov     Config,bx
    mov     al,[bx+DMA0LOC]
    mov     config_dma0_loc,al
    mov     al,[bx+DMA6LOC]
    mov     config_dma1_loc,al
    mov     al,[bx+IRQLOC]
    mov     config_irq_loc,al
    mov     ax,[bx+BPORT]
    mov     command_reg, 300h

SetTheInterruptVector:
;
;    SET UP THE INTERRUPT VECTORS
;
    push    di
    mov     al, config_irq_loc
    mov     bx, OFFSET CGroup: DriverISR
    call    SetInterruptVector
    pop     di
    mov     dx, EnLAN
    out     dx, al          ;enable LAN on MB module
%slow
    mov     dx, command_reg
    mov     al, C_RST
    out     dx, al      ; reset the 82592 controller

;generate 20 bit address for DMA controller from configure block location
;this is necessary to accomodate the page register used in the PC DMA

    call    set_up_buffers

;set up DMA channel for configure command
    xor     ax, ax
    out     DMAff, al           ;data is don't care
%slow
    mov     al, DMAena
    out     DMAcmdstat, al
    mov     ax, gp_buf_start
%slow
    out     DMA6addr, al
    mov     al, ah
%slow
    out     DMA6addr, al
    mov     ax, gp_buf_page
%slow
    out     DMA6page, al       ;DMA page value
    mov     ax, 1
%slow
    out     DMA6wdcount, al     ;make two transfers
    mov     al, ah
%slow
    out     DMA6wdcount, al
    mov     al, DMAtx6          ;setup channel 6 for tx mode
%slow
    out     DMAmode, al
    mov     al, DMA6unmsk
```

290189-24

Table 3. Hardware Initialization (Continued)

```
%slow
    out     DMAsnglmsk, al
    xor     ax, ax

the
    mov     di, gp_buf_offset  ;mov zeroes into the byte count field of the
    stosw                      ;buffer to put the 82592 into 16 bit mode
    stosw
%slow
    mov     dx, command_reg
    mov     al, C_CONFIG          ;configure the 82592 for 16 bit mode
    out     dx, al              ;issue configure command
%slow

wide_mode_wait_loop:
    xor     al, al
%slow
    out     dx, al      ;point to register 0
%slow
    in        al, dx      ;read register 0
    and     al,0DFh     ;disregard exec bit
    cmp     al, 82h     ; is configure finished?
    jz        do_config
    loop    wide_mode_wait_loop
    mov     ax, OFFSET CGroup: no_card_message
    jmp     init_exit

do_config:
    mov     al, C_INTACK
    out     dx, al            ;clear interrupt
    xor     ax, ax
%slow
    out     DMAff, al           ;data is don't care
    mov     ax, gp_buf_start
%slow
    out     DMA6addr, al
    mov     al, ah
%slow
    out     DMA6addr, al
    mov     ax, gp_buf_page
%slow
    out     DMA6page, al       ;DMA page value
%slow
    mov     al, DMAtx6          ;setup channel 1 for tx mode
    out     DMAmode, al
%slow
    mov     ax, 8
    out     DMA6wdcount, al
%slow
    mov     al, ah
    out     DMA6wdcount, al
%slow
    mov     al, DMA6unmsk
    out     DMAsnglmsk, al
    mov     ax, ds
    mov     es, ax
    mov     si, offset cgroup:config_block
    mov     di, gp_buf_offset
```

290189-25

Table 3. Hardware Initialization (Continued)

```
    mov     cx, 18
rep movsb
    mov     dx, command_reg
    mov     al, C_CONFIG          ; configure the 82592
    out     dx, al
%slow
    xor     cx, cx

config_wait_loop:
%slow
    xor     al, al
%slow
    out     dx, al      ;point to register 0
%slow
    in        al, dx      ;read register 0
    and     al, 0DFh    ;discard extraneous bits
    cmp     al, 82h     ; is configure finished?
    jz        config_done
    loop    config_wait_loop
    mov     ax, OFFSET CGroup: config_failure_message
    jmp     init_exit

config_done:
; clear interrupt caused by configuration
    mov     al, C_INTACK
    out     dx, al

; do an IA_setup
    mov     di, gp_buf_offset
    mov     al, 06h     ;address byte count
    stosb
    mov     al, 00h
    stosb
    mov     si, OFFSET CGROUP:node_addr
    mov     cx, SIZE node_addr
rep movsb
    out     DMAff, al           ;data is don't care
%slow
    mov     ax, gp_buf_start
    out     DMA6addr, al
    mov     al, ah
%slow
    out     DMA6addr, al
    mov     ax, gp_buf_page
%slow
    out     DMA6page, al       ;DMA page value
%slow
    mov     al, DMAtx6          ;setup channel 1 for tx mode
    out     DMAmode, al
%slow
    mov     ax, 3
    out     DMA6wdcount, al
%slow
    mov     al, ah
    out     DMA6wdcount, al
%slow
    mov     al, DMA6unmsk
    out     DMAsnglmsk, al
```

290189-26

Table 3. Hardware Initialization (Continued)

```
    mov     dx, command_reg
    mov     al, C_IASET             ;set up the 82592 individual address
    out     dx, al
    xor     cx, cx     ;cx is used by the loop instruction below. this
              ;causes the loop to be executed 64k times max
ia_wait_loop:
    xor     al, al
    out     dx, al
%slow
    in        al, dx
    and     al, 0DFh     ;discard extraneous bits
    cmp     al, 81h      ; is command finished?
    jz        ia_done
    loop    ia_wait_loop
    mov     ax, OFFSET CGroup: iaset_failure_message
    jmp     init_exit

ia_done:
    mov     al, C_INTACK
    out     dx, al          ;clear interrupt from iaset
;initialize the receive DMA channel
    xor     al, al
    out     DMAff, al
    mov     ax, rx_buf_start  ;set dma up to point to the beginning of rx buf
%slow
    out     DMA7addr, al
    mov    al, ah
%slow
    out     DMA7addr, al
    mov     ax, rx_buf_page   ;set rx page register
%slow
    out     DMA7page, al
    mov     al, DMArx7
%slow
    out     DMAmode, al
    mov     ax, rx_buf_length   ;set wordcount to proper value
%slow
    out     DMA7wdcount, al
    mov    al, ah
%slow
    out     DMA7wdcount, al
    mov     al, dma7unmsk       ;unmask receive DMA channel
%slow
    out     DMAsnglmsk, al

;unmask our interrupt channel
    in       al, InterruptMaskPort
    mov    bl, 0FBh
    and    al, bl
%slow
    out     InterruptMaskPort, al

;enable the receiver
    mov     dx, command_reg     ;enable receives
    mov     al, C_RXENB
    out     dx, al
    xor     ax, ax
```

290189-27

Table 3. Hardware Initialization (Continued)

```
    mov     cx, 1

init_exit:
    ret


ConfigDataUnderrun:
    mov     ax, OFFSET CGroup: ConfigDataUnderrunMess
    jmp     init_exit

IASetupDataUnderrun:
    mov     ax, OFFSET CGroup: IASetupDataUnderrun
    jmp     init_exit


DriverInitialize    endp


;
;    SetInterruptVector
;
;    Set the interrupt vector to the interrupt procedure's address
;    save the old vector for the unhook procedure
;
;    assumes: bx has the ISR offset
;      al has the IRQ level
;      interrupts are disabled
;

SetInterruptVector    PROC    NEAR
;    mask on the appropriate interrupt mask
    push    ax
    xchg    ax, cx
    mov     dl, 1
    shl     dl, cl          ;get the appropriate bit location
    mov     int_bit, dl       ;set the interrupt bit variable
    not     dl
    mov     int_mask, dl    ;set the interrupt mask variable
    mov     ax, InterruptMaskPort
    mov     int_mask_register, ax
    pop     ax
    cld
    cbw
    xor     cx, cx
    mov     es, cx
    add     al, 68h     ;adding 8 converts int number to int type, i.e.,
              ;int 4 = type 12, int 5 = type 13 etc.
    shl     ax, 1
    shl     ax, 1     ;two shifts = mul by 4 to create offset of vector
    xchg    ax, di
    mov     int_vector_addr, di    ;save this address for unhook
    mov     ax, es: [di]          ;save old interrupt vector
    mov     word ptr old_irq_vector, ax
    mov     ax, es: [di] + 2
    mov     word ptr old_irq_vector + 2, ax
    xchg    ax, bx      ;bx has the ISR offset
    stosw
    mov     ax, cs
```

290189-28

```
    stosw
    ret

SetInterruptVector    endp
```

290189-29

7.2.2 INITIALIZING SYSTEM MEMORY

A buffer is constructed in system memory to support the Embedded LAN Module architecture. This buffer is divided into a receive buffer area and a transmit/general-purpose buffer area. This buffer (Tx/GP) is used as the transmit buffer and as the parameter block for 82592 commands that require parameters.

The combined size of the buffer areas requested by the program is 10 kB. The Tx/GP buffer should be at least 1200 bytes long. The Rx buffer should be at least 5 kB long. The amount of memory requested is twice the size of the minimum Rx buffer length because of the possibility of a DMA page break occurring at some point in the 10 kB buffer area. A page break can occur because the SYP301 (or any PC AT based architecture) uses a static page register to supply the upper address bits (A_{17}–A_{23} for a 16-bit DMA channel) during a DMA cycle. These upper bits of the address cannot be incremented. The software checks for a page break and adjusts the buffer size if one is found. There are three possible page break scenarios.

- No page break occurs. The buffer size is not adjusted, the Tx/GP buffer area will be in the first 1200 bytes of the 10 kB buffer, and the Rx area will use the remainder.
- A page break occurs, and the buffer is divided so that one fragment is smaller than 1200 bytes. This fragment is too small to be used and both the Tx/GP and Rx areas will be placed in the larger segment.
- A page break occurs that divides the 10 kB buffer into two segments both larger than 1200 bytes. The software then places the Tx/GP area in the smaller segment, and the Rx area in the larger.

These three scenarios are shown in Figure 12. In no case is the Rx area less than 5 kB—half the total buffer size. Once these calculations are made, the transmit and receive DMA channels, along with their page registers, are programmed to point to their respective areas in the buffer (Tx/GP and Rx). With the memory now initialized, configuration and initialization of the 82592 can begin.

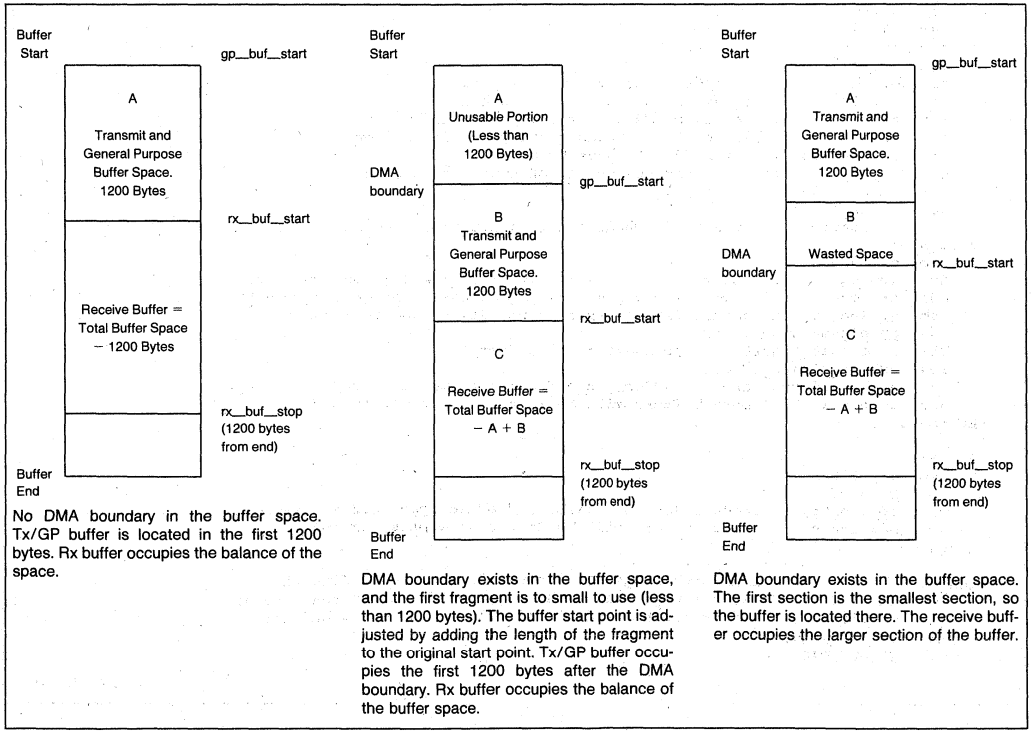


Figure 12. DMA Page Break Affect on Buffer Size

The Rx buffer area is implemented as a restartable linear buffer. As frames are stored in this buffer they are processed by the IPX routine IPXReceivePacket. A variable called RX_BUF_STOP points to a location 1200 bytes from the end of the Rx area. On reaching (or passing) this location in the Rx area, frame reception is temporarily disabled, and the remainder of the receive frames are processed. After the last frames have been processed, the receive area is reinitialized, the receive channel DMA is initialized to point back to the beginning of the receive area, and frame reception is reenabled. Table 4 contains the code that initializes the buffer memory. Section 7.3 gives further information on receive frame processing.

Table 4. Buffer Memory Initialization

```
;    Set up Buffers:
;    This routine generates the page and offset addresses for the 16 bit
;    DMA. It checks for a page crossing and uses the smaller half of the
;    buffer area for Tx and general purpose if a crossing is detected. If
;    no crossing is detected the general purpose/transmit buffer is placed
;    at the beginning of the buffer area. This routine also generates a
;    segment address for the receive buffer which allows the value read
;    from the "10 cent" latches to be used as read for the offset passed
;    to IPXReceivePacket. This saves some arithmetic steps when tracing
;    back through the rx buffer chain.
;

set_up_buffers      proc      near

    mov  ax, offset cgroup: gp_buf
    mov  gp_buf_offset, ax
    mov  bx, cs
    mov  dx, cs
    shr  ax, 1
    mov  cx, 3
    shl  bx, cl
    rol  dx, cl       ;get upper 3 bits for page register
    and  dx, 0007h      ;clear all but the lowest 3 bits
    add  ax, bx       ;ax contains EA of first location in buffer
    adc  dx, 0          ;if addition caused a carry add it to page
    mov  cx, 0FFFFh      ;of buffer to page break
    sub  cx, ax       ;cx contains the number of bytes to page break
    cmp  cx, 01388h
    jb     intel_hop
    jmp  copacetic      ;it's cool, whole buffer space is in one page
intel_hop:
    cmp  cx, 0258h
    ja   low_ok         ;low fragment is a usable size, check upper fragment
    add  ax, cx         ;move pointer past the page break to discard fragment
    sub  gp_length, cx ;adjust length variable to reflect shorter length
    mov  gp_offset_adjust, cx
    shl  gp_offset_adjust, 1   ;convert to byte format
    mov  cx, gp_offset_adjust
    add  gp_buf_offset, cx     ;adjust gp_buf starting point to reflect change
    jmp  copacetic      ;both buffers will be in the same page, rx buf
shortened

low_ok:
    cmp  cx, 1130h
    jb   high_ok
    mov  gp_length, cx ;adjust length variable, discard upper buffer fragment
    jmp  copacetic      ;both buffers will be in the same page, rx buf
shortened

high_ok:          ;now since both fragments are usable we have to find the
    cmp  cx, 09C4h      ;actual page break. the large half will be the receive
    ja   rx_first      ;buffer and the small half will be the gp-tx buffer.
    mov  gp_buf_page, dx
    shl  gp_buf_page, 1
```

290189-30

Table 4. Buffer Memory Initialization (Continued)

```
    mov  gp_buf_start, ax
    mov  rx_buf_start, 0000h
    mov  rx_buf_head, 0000h
    add  dx, 1                  ;next page
    mov  rx_buf_page, dx
    shl  rx_buf_page, 1
    shl  ax, 1
    adc  dx, 0
    mov  bx, cx     ;save number of bytes to page break
    mov  cx, 12
    shl  dx, cl
    mov  rx_buf_segment,dx
    sub  gp_length, bx
    mov  cx, gp_length
    mov  rx_buf_length, cx
    sub  cx, 258h
    shl  cx, 1
    add  cx, ax
    mov  rx_buf_stop, cx
    jmp  buffers_set

rx_first:
    mov  rx_buf_page, dx
    shl  rx_buf_page, 1
    mov  rx_buf_start, ax
    mov  rx_buf_head, ax
    shl  rx_buf_head, 1
    mov  rx_buf_length, cx
    mov  rx_buf_stop, 0FB9Eh   ;1200 bytes from end of buffer
    mov  gp_buf_start, 0000h
    add  dx, 1                 ;next page
    mov  gp_buf_page, dx
    shl  gp_buf_page, 1
    add  cx, 1
    shl  cx, 1
    mov  gp_offset_adjust, cx
    add  gp_buf_offset, cx
    sub  dx, 1
    shl  dx, 1
    shl  ax, 1
    adc  dx, 0
    mov  cx, 12
    shl  dx, cl
    mov  rx_buf_segment,dx
    jmp  buffers_set

copacetic:
    mov  gp_buf_start, ax  ;A1-A16 of gp buffer, gp buffer is first
    add  ax, 258h          ;1200 bytes for gp buffer at front of buffer space
    mov  rx_buf_start, ax  ;rx buffer starts 1200 bytes in
    mov  rx_buf_head, ax
    shl  rx_buf_head, 1
    sub  gp_length, 258h
    mov  cx, gp_length
    mov  rx_buf_length, cx
    shl  dx, 1             ;convert segment to byte address
    mov  rx_buf_page, dx
    mov  gp_buf_page, dx
```

290189-31

```
    shl  ax, 1             ;convert offset to byte address
    adc  dx, 0             ;adjust segment for shift
    mov  cx, 12
    shl  dx, cl
    mov  rx_buf_segment, dx   ;load variable for transfers to IPX
    mov  cx, rx_buf_length
    sub  cx, 258h          ;setup marker for low rx buffer space, >600 words
    shl  cx, 1
    add  ax, cx
    mov  rx_buf_stop, ax

buffers_set:
    ret

set_up_buffers      endp
```

290189-32

7.3 Assembly and Transmission of Frames

Frame assembly and transmission is accomplished by the interaction of the software driver and IPX through the use of IPX Event Control Blocks (ECBs). To transmit a frame, a transmit ECB is prepared that contains address information and a list of fragments in memory containing the frame to be transmitted. This ECB is placed in a queue for assembly and transmission of the frame. If the queue is empty, or when the ECB reaches the front of the queue, a routine is called that processes the ECB for transmission. This routine determines the length of the frame (padding the frame if necessary) and then constructs the frame in the Tx/GP buffer area. The construction of the frame is based on the ECB's address information and fragment list. The transmit DMA channel is now initialized to point to the beginning of the transmit frame in the Tx/GP area, and the byte count for that channel is also initialized. A Transmit command is now issued to the 82592. A separate routine monitors the transmission for a time-out error. When an interrupt from the 82592 indicates that the transmission attempt is complete (whether successful or unsuccessful), or if a time-out error has occurred, the proper completion code is inserted into the frame's ECB, and the ECB is passed back to IPX. If additional ECBs remain in the transmit queue the processing of the next ECB will begin. Table 5 contains the code used for assembly and transmission of frames.

Table 5. Assembly and Transmission of Frames

```
;
;   Driver Send Packet
;
;   Assumes
;     ES:SI points to a fully prepared Event Control Block
;     DS = CS
;     Interrupts are DISABLED but may be reenabled temporarily if necessary
;
;     don't need to save any registers
;
DriverBroadcastPacket:
DriverSendPacket    PROC    NEAR
    cli                                 ; disable the interrupts
    mov  cx, word ptr send_list + 2
    jcxz AddToFrontOfList
;    search to the end of the list, and add there.
    mov  di, word ptr send_list

AddToListLoop:
    mov  ds, cx
    mov  cx, ds: word ptr [di].link + 2
    jcxz AddListEndFound
    mov  di, ds: word ptr [di].link
    jmp  AddToListLoop

AddListEndFound:
    mov  es: word ptr [si].link, cx        ;move null pointer to newest SCB's
    mov  es: word ptr [si].link + 2, cx    ;link field
    mov  ds: word ptr [di].link, si
    mov  ds: word ptr [di].link + 2, es
    mov  ax, cs
    mov  ds, ax    ;set ds back to entry condition
    ret

AddToFrontOfList:
    mov  es:word ptr[si].link, cx
    mov  es:word ptr[si].link + 2, cx
    mov  word ptr send_list, si
    mov  word ptr send_list + 2, es
; drop through to Start Send

DriverSendPacket    endp


;
;    Start Send
;
;    assumes:
;     ES: SI    points to the ECB to be sent.
;     interrupts are disabled
;

start_send    PROC    NEAR
public      start_send
    cli             ; disable the interrupts
```

290189-33

Table 5. Assembly and Transmission of Frames (Continued)

```
    cld
;    save SCB address in variable tx_ecb to liberate registers
    mov  word ptr tx_ecb, si
    mov  word ptr tx_ecb + 2, es
    push ds        ;save ds for future use
;    get IPX packet length out of the first fragment (IPX header)
    lds  bx, es: dword ptr [si].fragment_descriptor_list
    mov  ax, ds: [bx].packet_length
    pop  ds           ;restore ds to CGROUP
    push ax           ;save length for later use in 590 length field
    xchg al, ah       ;byte swap for 592 length field calculation
    add  ax, 18       ;add in the overhead bytes DA,SA,CRC,length

    mov  padding, 0
    cmp  ax, 64
    ja   long_enough
    mov  padding, 64       ;minimum length frame
    sub  padding, ax       ;pad length
    mov  ax, 64
long_enough:
    sub  ax, 10       ;SA and CRC are done automatically
    inc  ax
    and  al, 0FEh      ;frame must be even
    mov  tx_byte_cnt, ax
    mov  di,gp_buf_offset
    mov  bx, cs
    mov  es, bx
; move the byte count into the transmit buffer
    stosw
; move the destination address from the tx ECB to the tx buffer
    mov  bx, si
    lea  si, [bx].immediate_address
    mov  ds,word ptr tx_ecb + 2
    movsw
    movsw
    movsw
    mov  ax,cs         ; get back to the code (Dgroup) section
    mov  ds,ax

; now the 590 length field
    pop  ax
    xchg ah, al
    inc  ax
    and  al, 0FEh         ;make sure E-Net length field is even
    xchg ah, al
    stosw
    lds  si, tx_ecb
    mov  ax, ds: [si].fragment_count
    lea  bx, [si].fragment_descriptor_list
move_frag_loop:
    push ds               ; save the segment
    mov  cx, ds: [bx].fragment_length
    lds  si, ds: [bx].fragment_address
    %fastcopy
    pop  ds               ; get the segment back
    add  bx, 6
    dec  ax
    jnz  move_frag_loop
```

290189-34

Table 5. Assembly and Transmission of Frames (Continued)

```
;start transmitting
    mov  cx, cs
    mov  ds, cx
;add any required padding
    mov  cx, 4          ;make sure frame ends with a NOP
    add  cx, padding
    shr  cx, 1
    rep  stosw
    mov  tx_active_flag, 1
    xor  ax, ax
    out  DMAff, al          ;data is don't care, AX has been zeroed
    mov  ax, gp_buf_start
%slow
    out  DMA6addr, al
    mov  al, ah
%slow
    out  DMA6addr, al
    mov  ax, gp_buf_page
%slow
    out  DMA6page, al       ;DMA page value
%slow
    mov  al, DMAtx6         ;setup channel 1 for tx mode
    out  DMAmode, al
    mov  ax, tx_byte_cnt
    add  ax, 4          ;add two for byte count, two for tx chain fetch
    shr  ax, 1          ;convert to word value and account for odd
    adc  ax, 0          ;byte DMA transfer
    out  DMA6wdcount, al
%slow
    mov  al, ah
    out  DMA6wdcount, al
%slow
    mov  al, DMA6unmsk
    out  DMAsnglmsk, al
    mov  dx, command_reg
    mov  al, C_TX
    out  dx, al
    mov  ax, IPXIntervalMarker
    mov  tx_start_time, ax
    %inc32  TotalTxPacketCount
    ret

start_send    endp

;;*****************************************************************
;
;    Driverpoll
;
;    Poll the driver to see if there is anything to do
;
;    Is there a transmit timeout? If so, abort transmission and return
;    ECB with bad completion code. Check to see if frames are queued.
;    If they are set up ES:SI and call DriverSendPacket.
;
;*****************************************************************************

DriverPoll    PROC    NEAR
    cli
```

290189-35

Table 5. Assembly and Transmission of Frames (Continued)

```
    cmp  tx_active_flag, 0
    jz   NotWaitingOnTx
    mov  dx, IPXIntervalMarker
    sub  dx, tx_start_time
    cmp  dx, TxTimeOutTicks
    jb   NotTimedOutYet

; This transmit is taking too long so let's terminate it now
;
; Issue an abort to the 82592

    mov  dx, command_reg
    mov  al, C_ABORT          ;abort transmit
    out  dx, al

    inc  tx_timeout
    les  si, tx_ecb
    mov  es: [si].completion_code, TransmitHardwareFailure   ;stuff completion
code of a failed tx
    mov  ax, es: word ptr [si].link
    mov  word ptr send_list, ax
    mov  ax, es: word ptr [si].link + 2
    mov  word ptr send_list + 2, ax

; Finish the transmit

    mov  es: [si].in_use, 0
    call IPXHoldEvent

;make sure that execution unit didn't lock up because of abort errata
;
    mov  dx, command_reg
    mov  al, C_SWP1
    out  dx, al
    mov  al, C_SELRST
%slow
    out  dx, al
    mov  al, C_SWP0
%slow
    out  dx, al
    mov  al, C_RXENB
%slow
    out  dx, al
    mov  tx_active_flag, 0

; See if any frames are queued

    mov  cx, word ptr send_list + 2
    jcxz queue_empty
    mov  es, cx
    mov  si, word ptr send_list
    call start_send

queue_empty:
NotWaitingOnTx:
NotTimedOutYet:
    ret
```

290189-36

Table 5. Assembly and Transmission of Frames (Continued)

```
DriverPoll    endp

;*****************************************************************
;
;    Interrupt Procedure
;
;*****************************************************************

even

RxErrorTypeCheck:


BufferOverflow:
    inc   rx_buff_ovflw
    jmp   int_exit


not_590_int:
    inc  no_590_int
    jmp  int_exit

DriverISR        PROC    far
public      DriverISR


    push ax
    push bx
    push cx
    push dx
    push si
    push di
    push bp


    push ds
    push es
    cld

int_poll_loop:
    cli
    call IPXStartCriticalSection  ;tell AES we're busy
    mov  al, EOI
    out  InterruptControlPort, al
    out  ExtraInterruptControlPort, al
    mov  ax, cs
    mov  ds, ax    ;DS points to C/DGroup
    mov  dx, command_reg
    mov  al, 0
    out  dx, al    ;set status reg to point to reg 0
%slow
    in   al, dx
    test al, 80h
    jz   not_590_int

    and  al, NOT 20h         ;ignore the EXEC bit
    mov  ah, al        ;save the status in AH
    cmp  ah, 0D8h        ;did I receive a frame?
```

290189-37

Table 5. Assembly and Transmission of Frames (Continued)

```
    jz   rcvd_packet
    cmp  ah, 84h        ;did I finish a transmit?
    jz   sent_packet_jmp
    cmp  ah, 8Ch        ;did I finish a retransmit?
    jz   sent_packet_jmp
    inc  false_590_int  ;unwanted interrupt
    jmp  int_exit

sent_packet_jmp:
    jmp  sent_packet

sent_packet:
    cli
    cmp  tx_active_flag, 0
    jz   false_tx_int     ;shouldn't have been transmitting
    in   al, dx
    mov  status10, al
%slow
    in   al, dx
    mov  status11, al
    test status11, 20h
    jz   tx_error
    mov  al, status10       ;extract the total number of retries from
    and  ax, 0Fh          ;the status register and add to retry count
    add  RetryTxCount, ax
    xor  ax, ax          ;status = 0, good transmit

FinishUpTransmit:
    les  si, tx_ecb
    mov  es: [si].completion_code, al
    mov  ax, es: word ptr [si].link
    mov  word ptr send_list, ax
    mov  ax, es: word ptr [si].link + 2
    mov  word ptr send_list + 2, ax
    mov  es: [si].in_use, 0
    call IPXHoldEvent
    push cs
    pop  ds
    mov  cx, word ptr send_list + 2
    mov  tx_active_flag, cl
    jcxz int_exit_jmp1
    mov  es, cx            ;segment of next SCB in list
    mov  si, word ptr send_list    ;offset of next SCB in list
    call start_send
    jmp  finish_exit

int_exit_jmp1:
    jmp  int_exit

false_tx_int:
    jmp  int_exit

tx_error:
    test status10, 20h        ;Max collisions??
    jnz  QuitTransmitting
    test status11, 01h        ;Tx underrun??
    jz   lost_cts
    inc  underruns
```

290189-38

```
lost_cts:
    test status11, 02h        ;did we lose clear to send??
    jz   lost_crs
    inc  no_cts
lost_crs:
    test status11, 04h        ;did we lose carrier sense??
    jz   hmmm
    inc  no_crs
hmmm:
    les  si,tx_ecb
    call start_send
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit

QuitTransmitting:
    mov  al, status10
    and  ax, 0Fh
    add  RetryTxCount, ax
    inc  stop_tx
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit

DriverISR    endp
```

290189-39

7.4 Receive Frame Processing

Receive frame processing is triggered by an interrupt from the 82592. If the status read from the 82592 by the Interrupt Service Routine (ISR) indicates that a frame has been received, a jump is made to the beginning of the code that services frames. The receive buffer area is managed by using several variables. These variables are listed below. Please refer to Section 4.1 and 4.2 for a review of receive frame processing.

- RX__BUF__TAIL. Contains the contents of the 16-bit latch. They point to the byte count of the last frame written into memory.
- RX__BUF__PTR. Keeps track of the current position in the buffer while the CPU recovers locations of the received frames in the buffer processing.
- RX__BUF__HEAD. Contains the pointer to the byte count of the last frame that was processed by the CPU. (This differs from rx__buf__tail, which points to the byte count of a frame not yet processed.)
- RX__BUF__STOP. Points to a location that is 1200 bytes from the end of the receive buffer (slightly more than the maximum size of a frame).

After servicing a receive frame, the contents of the 16-bit latch are loaded into RX__BUF__TAIL and RX__BUF__PTR. This value is compared with the value stored in RX__BUF__STOP to determine if most of the buffer has been used and if the buffer must be reinitialized after the current receive frames have been processed (In this case a flag called RESET__RX__BUF is set to indicate that the buffer variables and receive DMA channel must be reinitialized before the Interrupt Service Routine is exited). To process the frame or frames received, both the byte count and status bytes of the frame are used. If the status indicates a receive error the frame is not passed up to IPX. The byte count is used to index back through the chain of received frames, using RX__BUF__PTR to keep track of the current position in the buffer. The frames are checked for length (maximum and minimum), and a check is also made to verify that the Ethernet and IPX length fields agree (including provisions for padding the Ethernet length field). If these checks pass, the frames are added to the list of received frames by storing their location, length, and source address in an array of structures called RX__LIST. When the RX__BUF__PTR contains the same value as RX__BUF__HEAD, all currently received frames have been processed, and a jump is made to a label called HAND__OFF__PACKET. In this routine the frames are handed up to IPX, in the order they were received, using calls to the IPX routine IPXReceivePacket. The value stored in RX__BUF__TAIL is loaded into the RX__BUF__HEAD variable, which now holds the address of the last location in the receive area that was processed, and the execution of the ISR falls through to a routine to exit the ISR. Before exiting the ISR an Interrupt Acknowledge is issued to the 82592; a check for additional pending interrupts is made, if one is found the ISR process is repeated; and the flag RESET__RX__BUF is checked, if it is set the receive buffer is reinitialized. The machine states of the previous routines are restored to their original states, and the ISR is exited. Table 6 contains the code used for receive frame processing.

Table 6. Receive Frame Processing

```
;*****************************************************************
;
;    Interrupt Procedure
;
;*****************************************************************

even

RxErrorTypeCheck:


BufferOverflow:
    inc   rx_buff_ovflw
    jmp   int_exit


not_590_int:
    inc  no_590_int
    jmp  int_exit

DriverISR        PROC    far
public      DriverISR


    push ax
    push bx
    push cx
    push dx
    push si
    push di
    push bp


    push ds
    push es
    cld

int_poll_loop:
    cli
    call IPXStartCriticalSection  ;tell AES we're busy
    mov  al, EOI
    out  InterruptControlPort, al
    out  ExtraInterruptControlPort, al
    mov  ax, cs
    mov  ds, ax    ;DS points to C/DGroup
    mov  dx, command_reg
    mov  al, 0
    out  dx, al    ;set status reg to point to reg 0
%slow
    in   al, dx
    test al, 80h
    jz   not_590_int
```

290189-40

Table 6. Receive Frame Processing (Continued)

```
   int_poll_loop:
    and  al, NOT 20h          ;ignore the EXEC bit
    mov  ah, al          ;save the status in AH
    cmp  ah, 0D8h         ;did I receive a frame?
    jz   rcvd_packet
    cmp  ah, 84h          ;did I finish a transmit?
    jz   sent_packet_jmp
    cmp  ah, 8Ch          ;did I finish a retransmit?
    jz   sent_packet_jmp
    inc  false_590_int  ;unwanted interrupt
    jmp  int_exit

sent_packet_jmp:
    jmp  sent_packet
bad_rcv:
    inc  rx_errors
    jmp  RxErrorTypeCheck

int_exit_jmp:
    jmp  int_exit

;When the address bytes are being read it is possible that another frame
;could come in and cause a coherency problem with the ten-cent latches.
;I am dealing with this possibility by reading TenCentHi twice and making
;sure the values match. If they don't the read is redone.

rcvd_packet:
    cli
    mov  dx, TenCentHi     ;read high address byte of last frame received
    in   al, dx
    mov  ah, al         ;save it in ah
    mov  dx, TenCentLo     ;read low address byte of last frame received
    in   al, dx
    mov  rx_buf_tail, ax   ;this is the last location containing rx data
;Read TenCentHi again to make sure it hasn't changed.......
    mov  dx, TenCentHi       ;read high address byte again
    in   al, dx
    cmp  al, ah
    jz   addr_ok
    jmp  rcvd_packet         ;read the latches again
addr_ok:
    mov  ax, rx_buf_tail     ;this is a valid address
    mov  rx_buf_ptr, ax      ;this is the last location containing rx data
    cmp  rx_buf_stop, ax     ;is most of the buffer already used?
    ja   BufferOK
    mov  reset_rx_buf, 1
BufferOK:
    cmp  ax, rx_buf_head
    ja   process_new_frames
    inc  ten_cent_latch_crash
    jmp  int_exit

do_next_frame:
process_new_frames:
    mov  bx, rx_buf_ptr       ;end of current frame to process
    sub  bx, 6                ;set bx up to point to beginning of the status
    mov  es, rx_buf_segment   ;this is necessary because latches hold EA not
                              ;offset relative to CGROUP
```

290189-41

Table 6. Receive Frame Processing (Continued)

```
    mov  al, es:[bx].status1
    test al,20h                 ;test for good receive
    jnz  good_rx
    mov  cl, es:[bx].bc_lo
    mov  ch, es:[bx].bc_hi      ;cx has actual number of bytes read
    dec  cx                     ; toss byte count & status
    and  cl, 0feh               ; round up
    sub  bx, cx                 ;bx points to first location of frame
    cmp  rx_buf_head, bx
    je   hand_off_packet_jmp    ;this was the first frame in the sequence
    mov  rx_buf_ptr, bx
    sub  rx_buf_ptr, 2
to_do_next_frame:
    jmp  do_next_frame
hand_off_packet_jmp:
    jmp  hand_off_packet

good_rx:
    mov  cl, es:[bx].bc_lo
    mov  ch, es:[bx].bc_hi      ;cx has actual number of bytes read
    mov  curr_rx_length, cx
    dec  cx                      ; toss byte count & status
    and  cl, 0feh                ; round up
    sub  bx, cx                 ;bx points to first location of frame
    mov  rx_buf_ptr, bx
    sub  rx_buf_ptr, 2           ;rx_buf_ptr = last location of n-1 frame
    sub  cx, 14                   ; sub length of 802.3 header
    cmp  cx, 1024 + 64
    jbe  not_too_big
    inc  PacketRxTooBigCount
    jmp  do_next_frame
not_too_big:
    cmp  cx, 30
    jae  not_too_small
    inc  PacketRxTooSmallCount
    jmp  do_next_frame
not_too_small:

    mov  ax, es:[bx].rx_length  ; get IPX length
    xchg al, ah
    inc  ax
    and  al, 0feh
    xchg al, ah
    cmp  ax, es:[bx].rx_physical_length     ; same as 802.3 length ?
    jne  to_do_next_frame
    xchg al, ah
    cmp  ax, 60 - 14              ; at least min length minus header
    ja   len_ok                    ; yes, continue
    mov  ax, 60 - 14              ; no, round up
len_ok:
    cmp  ax, cx         ; match physical length
    jz   not_inconsistent     ; yes, continue
    inc  HardwareRxMismatchCount
    jmp  do_next_frame
not_inconsistent:
    %inc32  TotalRxPacketCount       ; Double Word Increment
    mov  ax, 12
    mul  num_of_frames
```

290189-42

Table 6. Receive Frame Processing (Continued)

```
    mov  di, ax
    mov  rx_list [di], bx     ;first location of ethernet frame
    add  rx_list [di], 14     ;first location of ipx packet
    mov  ax, rx_buf_segment
    mov  rx_list [di + 2], ax
    mov  ax, word ptr es:[bx].rx_length
    xchg al,ah
    mov  rx_list [di + 4], ax
    mov  ax, word ptr es:[bx].rx_source_addr + 0
    mov  word ptr rx_list [di + 6], ax
    mov  ax, word ptr es:[bx].rx_source_addr + 2
    mov  word ptr rx_list [di + 8], ax
    mov  ax, word ptr es:[bx].rx_source_addr + 4
    mov  word ptr rx_list [di + 10], ax
    add  num_of_frames, 1
    cmp  rx_buf_head, bx
    je   hand_off_packet
    cmp  num_of_frames, 50
    je   hand_off_packet
    jmp  do_next_frame

hand_off_packet:
    mov  si, rx_list[di]
    mov  es, rx_list[di + 2]
    mov  cx, rx_list[di + 4]
    lea  bx, rx_list[di + 6]
    cli
    push ds
    call IPXReceivePacket
    pop  ds
    sub  num_of_frames, 1
    jz   adjust_rx_head
    sub  di, 12
    jmp  hand_off_packet
adjust_rx_head:
    mov  ax, rx_buf_tail
    add  ax, 2
    mov  rx_buf_head, ax      ;set rx_buf_head to new value for next receive
                 ;interrupt
int_exit:
    push cs
    pop  ds
    cmp  tx_active_flag, 0
    jnz  finish_exit

     verify that our receiver is still going.

    mov  dx, command_reg
    mov  al, 60h          ;point to status byte 3
    out  dx, al
%slow
    in   al, dx
    test al, 20h
    jnz  finish_exit
    jmp  LostOurReceiver

finish_exit:
    cli
```

290189-43

Table 6. Receive Frame Processing (Continued)

```
    call IPXEndCriticalSection
    mov  dx, command_reg
    mov  al, C_INTACK
    out  dx, al          ;issue interrupt acknowledge to the 590


%slow
    xor  al, al
    out  dx, al      ;set status reg to point to reg 0
%slow
    in   al, dx
    test al, 80h
    jnz  int_pending
    cmp  reset_rx_buf, 1
    jnz  no_rx_buf_reset
    mov  al, dma7msk        ;mask receive DMA channel
    out  DMAsnglmsk, al
%slow
    out  DMAff, al         ;data is don't care
    mov  ax, rx_buf_start  ;set dma up to point to the beginning of rx buf
    mov  rx_buf_head, ax
    shl  rx_buf_head, 1
    out  DMA7addr, al
    mov  al, ah
%slow
    out  DMA7addr, al
    mov  al, DMArx7
%slow
    out  DMAmode, al
    mov  ax, rx_buf_length  ;set up  rx buf
%slow
    out  DMA7wdcount, al
    mov  al, ah
%slow
    out  DMA7wdcount, al
    mov  dx, DMAsnglmsk
    mov  al, DMA7unmsk
%slow
    out  dx, al
    mov  dx, command_reg
    mov  al, C_RXENB
    out  dx, al
    mov  reset_rx_buf, 0

no_rx_buf_reset:
    cli
    call IPXServiceEvents
    pop  es
    pop  ds


    pop  bp
    pop  di
    pop  si
    pop  dx
    pop  cx
    pop  bx
```

290189-44

```
    pop  ax

    sti
    iret

LostOurReceiver:
    inc  lost_rx
    mov  al, C_RXENB
    mov  dx, command_reg
    out  dx, al
    jmp  finish_exit

too_big:
    inc  PacketRxOverflowCount
    jmp  int_exit

int_pending:
    jmp  int_poll_loop
```

290189-45

APPENDIX A

Expanding the 82592 Embedded LAN Module Architecture to a Low-Cost Non-Buffered Adapter

The basic architecture of the 82592 Embedded LAN Module can be expanded and applied to a low-cost, non-buffered adapter. This requires adding a DMA unit and some logic for a bus master handshake. Such an adapter would contain no local buffer memory. Its cost advantage would come from using existing system memory, as the embedded module does. This adapter is less complex than most existing designs because it does not require arbitration logic for access to local memory. This adapter becomes a bus master when data transfers take place, either to the 82592 (Tx) from system memory or from the 82592 (Rx) into system memory.

The same features of the 82592 that make it successful in embedded applications make it well-suited for non-buffered adapters. As with the embedded module, there is no intermediate buffering of data in a local memory, therefore data transfers to and from system memory take place in real time. The 82592's large FIFO area allows it to tolerate long system bus latencies during memory access. The 82592's high-performance, 16-bit bus interface allows the adapter to efficiently transfer data to and from system memory when it gains access to the system bus. The TCI of the 82592 will interface with the adapter's control logic and DMA unit to provide back-to-back frame reception and automatic retransmission on collision (both without CPU intervention). Figure 13 is a block diagram of the basic architecture of the embedded module modified for a non-buffered adapter application. The block titled "Control PALs and Latch" together with the 82592 is the core of the embedded module architecture. One additional PAL (PAL C) has been added to the basic architecture to offer more logic for decoding additional components added to the adapter. The address latch has also been expanded to 24 bits. The three shaded blocks (DMA Machine, Master Logic, and Control PALs and Latch) show the most likely path for integration on this adapter, providing a three-chip solution of ASIC, 82592, and 82C501. The 82C37 is common in many ASIC cell libraries, offering a migration path for this integration.

ADAPTER BLOCK DESCRIPTIONS

DMA Machine

- **8237 DMA Controller.** Serves as the core for the DMA machine. Performs addressing and control for data transfers between the 82592 and host system memory.
- **8-Bit Page Counter.** Provides the addressing bits for the upper bits of address (A_{17}–A_{23}).
- **8-Bit Register.** Serves as the base register for the upper bits of the Tx DMA channel for reinitialization for automatic retransmission.
- **8-Bit Multiplexer.** Selects between the upper bits of Rx- or Tx-channel DMA.
- **8-Bit Latch.** Latches the upper bits of address from the 8237 (A_8–A_{15}).

Master Logic

- **Master PAL.** Implements a "master" handshake with the host system bus to gain access to the bus as a bus master.
- **Timers (2).** Controls the maximum time the adapter can hold the bus, and the minimum time it must wait before attempting to regain bus access.

Control PALs and Latch (Together with 82592 and 82C501)

The basic architecture of the 82592 Embedded LAN Module.

Transceivers

Used to buffer the adapter logic from the host system bus, for drive purposes. Address consists of 24 bits; and Data, 16 bits.

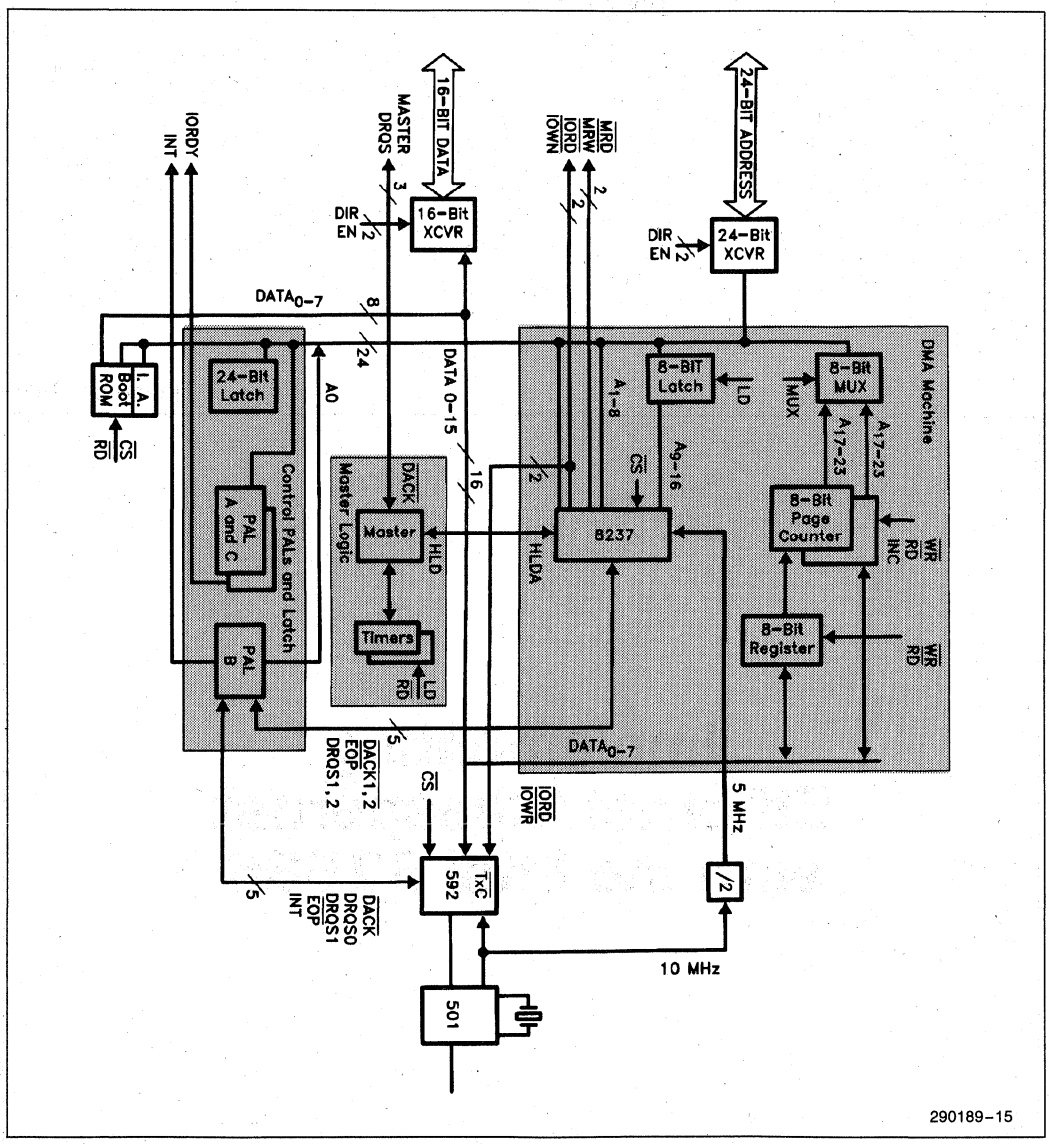


Figure 13. 82592 Non-Buffered Adapter Block Diagram (PC AT Version)

intel®

APPLICATION NOTE

AP-274

November 1986

Implementing Ethernet/Cheapernet with the Intel 82586

KIYOSHI NISHIDE
APPLICATIONS ENGINEER

Order Number: 292010-002

PREFACE

Intel's three VLSI chip set, the 82586, 82501, and 82502, is a complete solution for IEEE 802.3 10M bps LAN standards—10BASE5 (Ethernet) and 10BASE2 (Cheapernet). The 82586 is an intelligent peripheral which completely manages the processes of transmitting and receiving frames over a network under the CSMS/CD protocol. The 82586 with its on-chip four DMA channels offloads the host CPU of the tasks related to managing communication activities. The chip, for example, does not depend on the host CPU for time critical functions, such as transmissions/retransmissions and receptions of frames. The 82501 is a 10 MHz serial interface chip specially designed for the 82586. The primary function of the 82501 is to perform Manchester encoding/decoding, provide 10 MHz transmit and receive clocks to the 82586, and drive the transceiver (AUI) cable in Ethernet applications. In addition, the 82501 provides a loopback function and on-chip watchdog timer. The 82502 is a CMOS transceiver chip. The 82502 is the chip which actually drives the coaxial cable used for Ethernet or Cheapernet.

This Ap Note presents a design example of a simple but general Ethernet/Cheapernet board based on the three chip set. The board is called LANHIB (LAN High Integration Board) and uses an 80186 microprocessor as the host CPU. The LANHIB is an independent single board computer and requires only a power supply and ASCII terminal. Demo software, called TSMS (Traffic Simulator and Monitor Station) is also included in this Ap Note. The TSMS program is a network debugger and exercise tool used to exercise the 82586. In addition, flowcharts for troubleshooting are provided in order to minimize debugging time of the LANHIB board.

1.0 INTRODUCTION

A brief overview of the CSMA/CD protocol is described in Section 2. Ethernet and Cheapernet are also compared in this section.

Section 3 discusses hardware of the LANHIB in detail. This section should be helpful not only to understand the LANHIB, but also to learn in general how a system based on the three chip set can be put together. Since the 82502 involves analog circuitry, an explanation on proper layout is provided.

Demo software is presented in Section 4.0. It covers EPROM programming procedures and three sample sessions. Step by step operations at a terminal are illustrated in the figures.

Section 5 describes LANHIB troubleshooting procedures. Flowcharts are used to guide troubleshooting.

Complete LANHIB schematics and parts list are found in Appendix A. If a LANHIB is to be built, the schematics and Section 5 can be submitted to an available wire wrap facility. In parallel to board construction, Sections 3 and 4 can be studied. A factory wire wrap board for the LANHIB is offered at a discount price by Augat Corporation. Please return the enclosed card for more information.

Listing of the TSMS program and LANHIB Initialization Routine are in Appendix B. The source codes and related files are available on a diskette by returning the card enclosed in this design kit or through Insite (Intel's Software Index and Technology Exchange Library).

2.0 ETHERNET/CHEAPERNET OVERVIEW

2.1 CSMA/CD

Carrier Sense Multiple Access with Collision Detection (CSMA/CD) is a simple and efficient means of determining how a station transmits information over common medium that is shared with other stations. CSMA/CD is the access method defined by the IEEE 802.3 standard.

Carrier Sense (CS) means that any station wishing to transmit "listens" first. When the channel is busy (i.e., some other station is transmitting) the station waits (defers) until the channel is clear before transmitting ("listen before talk").

Multiple Access (MA) means that any stations wishing to transmit can do so. No central controller is needed to decide who is able to transmit and in what order.

Collision Detection (CD) means that when the channel is idle (no other station is transmitting) a station can start transmitting. It is, however, possible for two or more stations to start transmitting simultaneously causing a "collision". In the event of a collision, the transmitting stations will continue transmitting for a fixed time to ensure that all transmitting stations detect the collision. This is known as jamming. After the jam, the stations stop transmitting and wait a random period of time before retrying. The range of random wait times increases with the number of successive collisions such that collisions can be resolved even if a large number of stations are colliding.

There are three significant advantages to the CSMA/CD protocol. The first and foremost is that CSMA/CD is a proven technology. One CSMA/CD network, Ethernet, has been used by Xerox since 1975. Ethernet is so well understood and accepted that IEEE adopted

it (with minor changes) as the IEEE 802.3 10Base5 (10 Mbps, Baseband, 500 meters per segment) standard. Reliability is the second advantage to the 802.3 protocol. This media access method enables the network to operate without central control or switching. Thus, if a single station malfunctions, the rest of the network can continue operation. Finally, since CSMA/CD networks are passive and distributed in nature, they allow for easy expansion. New nodes can be added at any time without reinitializing the entire network.

2.2 Ethernet and Cheapernet

The IEEE 802.3 Type 10BASE5 standard (Ethernet) has gained wide acceptance by both large and small corporations as a high speed (10 Mbps) Local Area Network. The Ethernet channel is a low noise, shielded 50Ω coaxial cable over which information is transmitted at 10 million bits per second. Each segment of cable can be up to 500 meters in length and can be connected to longer network lengths using repeaters. Repeaters regenerate the signal from one cable segment onto another. At each end of a cable segment a terminator is attached. This passive device provides the proper electrical termination to eliminate reflections. The transceiver transmits and receives signals on the coaxial cable. In addition, it isolates the node from the channel so that a failure within the node will not affect the rest of the network. The transceiver is also responsible for detecting collisions—simultaneous transmissions by two or more stations. Ethernet transceivers are connected to the network coaxial cable using a simple tap, and to the station it serves via the transceiver cable which can be

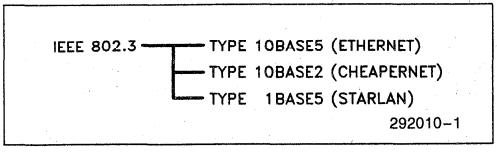


Figure 1. Different Implementations of IEEE 802.3 (Note: "10BASE5", for example, implies 10 Mbps, Baseband, and 500 meters span.)

up to 50 meters in length. The transceiver cable is made of four individually shielded twisted pairs of wires. An Ethernet interface at a computer (DTE), which includes a serial interface and data link controller, provides the connection to the user or server station. It also performs frame manipulation, addressing, detecting transmission errors, network link management, and encoding and decoding of the data to and from the transceiver.

The IEEE 802.3 Type 10BASE2 (Cheapernet) has the same functional and electrical specifications as Type 10BASE5 (Ethernet) with only two exceptions in physical (or rather mechanical) characteristics. Cheapernet is as shown in Figure 1 just a different implementation of the IEEE standard. Ethernet and Cheapernet are both 10 million bits/second CSMA/CD LANs and use the identical network parameters, such as slot time = 51.2 μs. Ethernet and Cheapernet can, therefore, be built by the same VLSI components with the same software (Figure 2).

The two physical differences attribute to the cost reduction purpose of Cheapernet—cheaper implementation of Ethernet. First, the cable used in Cheapernet may be a lower cost 50Ω coaxial cable than the one for Ethernet. The most common coaxial cable for Cheapernet is RG58 which cost about $0.15/ft. A typical Ethernet cable costs about $0.83/ft.

Second, the transceiver is integrated into the DTE in Cheapernet. The coaxial cable physically comes to the DTE, connects to the transceiver within the DTE, and goes to the next DTE (see Figure 3). The kind of connector used at the DTE is an off-the-shelf BNC "T" connector. Topology is, therefore, a simple daisy chaining. This cabling scheme contributes to further cost reduction due to omission of the Transceiver (AUI) Cable, cheaper connectors, and easier installation. The Ethernet transceiver cable costs about $1.49/ft. More flexible thin coaxial cables and familiar BNC "T" connectors are making Cheapernet a user installable Ethernet compatible network.

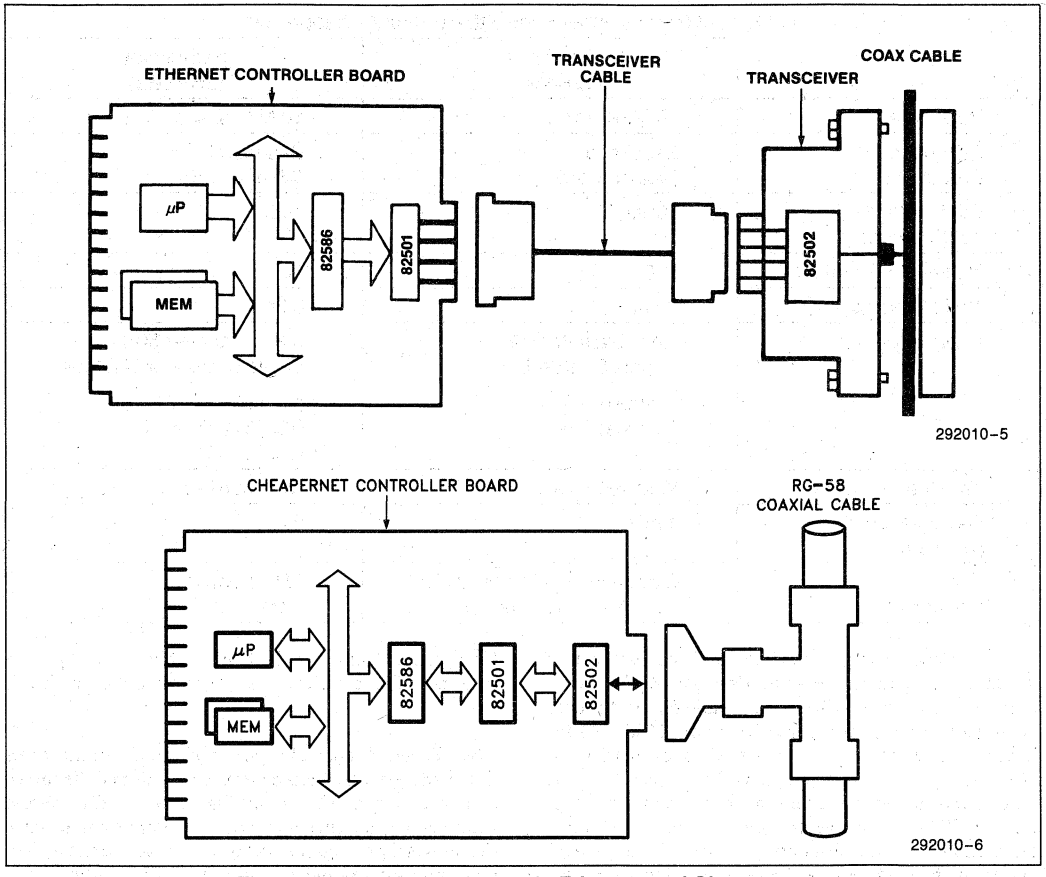


Figure 2. 82586/82501/82502 in Ethernet and Cheapernet

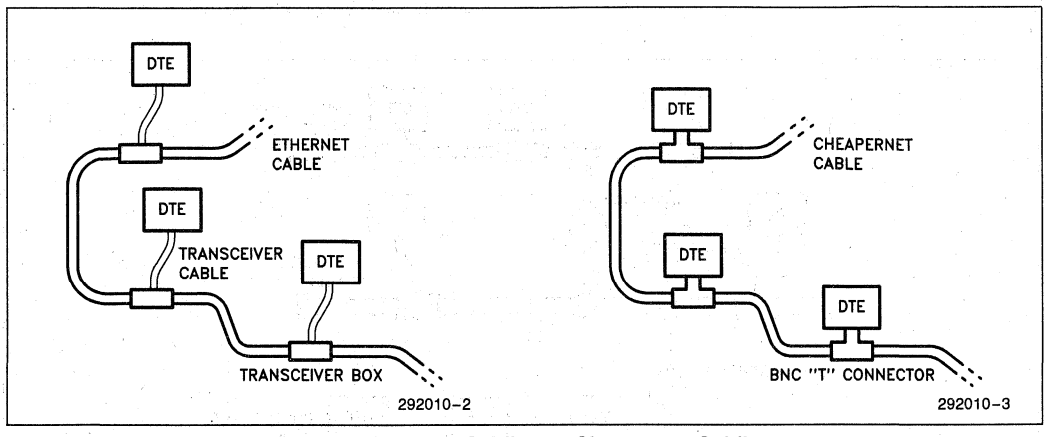


Figure 3. Ethernet Cabling vs Cheapernet Cabling

Table 1. Differences between Ethernet and Cheapernet

	Ethernet (10BASE5)	Cheapernet (10BASE2)
Data Rate	10 M bits/sec.	10 M bits/sec.
Baseband or Broadband	Baseband (Manchester)	Baseband (Manchester)
Cable Length per Segment	500m	185m
Nodes per Segment	100	30
Node Spacing	2.5m	0.5m
Cable Type	0.4 in diameter 50Ω Double Shielded example: Ethernet Coax.	0.2 in diameter 50Ω Single or Double Shielded example: RG 58 A/U or RG 58 C/U
Transceiver Cable	Yes, up to 50m	No, not needed
Capacitance per node	4 pF	8 pF
Typical Connector	Clamp-on Tap Connector or Type N Plug Connector	BNC Female Connector

Because of the lower quality cables and connectors used in Cheapernet, there are some drawbacks. The maximum distance for one Cheapernet cable segment is only 185m (600 feet), as compared to 500m (1640 feet) for Ethernet. The maximum number of nodes allowed for one Cheapernet cable segment is 30. Ethernet on the other hand allows a maximum of 100 nodes per segment. A BNC "T" connector used in Cheapernet introduces more electrical discontinuity on the transmission line than the clamp-on tap connector widely used for Ethernet. The maximum capacitance load allowed at a Cheapernet connection is 8 pF, while it is 4 pF for Ethernet. These differences are summarized in Table 1.0.

Since Ethernet and Cheapernet share the same functional and electrical characteristics, both may be mixed in a network as shown in Figure 4. In this hybrid Ethernet/Cheapernet network, it is important to keep the network propagation delay within 46.4 μs. The network may be expanded as required within this round trip propagation delay limit. Ethernet, for example, may serve as a backbone for Cheapernet in a hybrid Ethernet/Cheapernet network.

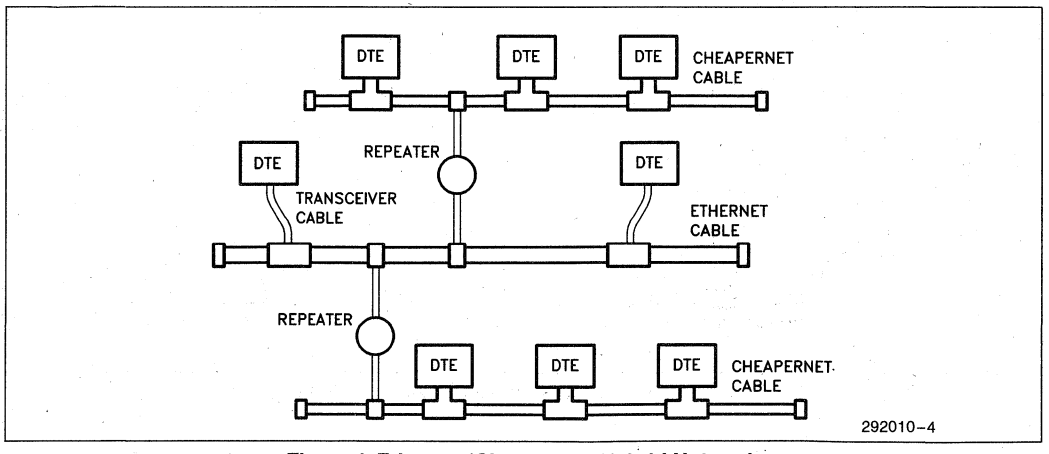


Figure 4. Ethernet/Cheapernet Hybrid Network

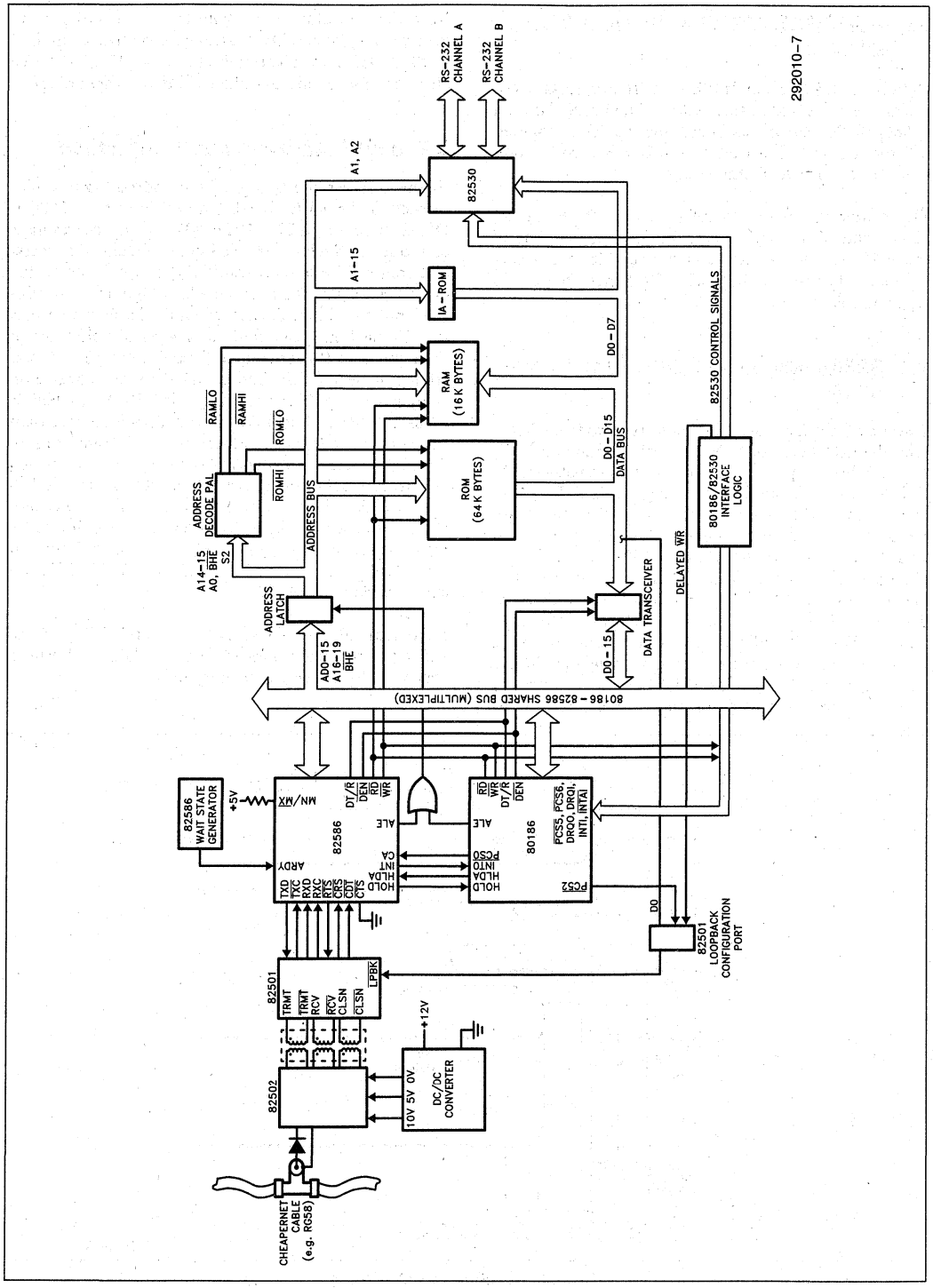


Figure 5. LANHIB Block Diagram

3.0 ETHERNET/CHEAPERNET NODE DESIGN

Details on LAN High Integration Board (LANHIB) design are presented in this section. The LANHIB is an 82586/80186 shared bus board and can be configured to Ethernet or Cheapernet. The 82586 is used in minimum mode to reduce chip count.

The reader is advised to refer to the 80186, 82586, 82501, and 82502 data sheets. Basic understanding of the 80186 microprocessor is assumed. Figure 5 shows the block diagram of the LANHIB. Schematics are in Appendix A.

3.1 82586 (Min Mode) Interface to the 80186

The 82586 can be placed in minimum mode by strapping the MN/$\overline{MX}$ pin to V_{CC}. In the minimum mode, the chip directly provides all bus control signals—ALE, $\overline{RD}$, $\overline{WR}$, DT/$\overline{R}$, and $\overline{DEN}$, saving the 8288 Bus Controller. The 80186, which is the only other bus master on the shared bus, also generates these bus control signals directly. The HOLDs and HLDAs of these two chips are connected together so that only one of the two bus masters can exclusively drive the bus at a time under the HOLD/HLDA protocol. Except for the ALE, all bus signals including address and data lines float when the chip does not have control of the bus. In this design example, $\overline{RD}$s, $\overline{WR}$s, DT/$\overline{R}$ and $\overline{DEN}$ from the two chips are connected together respectively. ALEs from the two chips are connected to an OR-gate to generate a system ALE. Multiplexed address data lines AD0–AD15 and address lines A15–A19 of the two chips are also connected line by line correspondingly.

3.2 82586 Address Latch Interface

Figure 6 shows the timing of the address signals with respect to the ALE signal. The ALE of the 82586 is OR-ed with the ALE of the 80186 and the result is connected to the latch enable inputs of Octal Transceiver Latches. The latches transfer the input data to the output as long as the latch enable is high, and captures the input data into the latch when the latch enable goes low. In this timing diagram, the setup and hold times of the input data (82586 address) required by the address latch can be verified. Estimating 7 ns of propagation delay in the 74S32, the setup time is T38 + 7, which is 32 ns at 8 MHz. The hold time for A19 is shorter than the other address lines because it is valid only during T1. The hold time for the A19 is T4 − T36 − 7, which is 3 ns. The hold time for the other address lines is T39 − 7, which is 38 ns. In this design, a 74F373 was chosen to latch address lines A16–A19 and two 74LS373s were used to latch address lines AD0–AD15. Required setup and hold times of the 74F and 74LS 373s are summerized in Table 2.

Note that address lines A16–A18 and $\overline{BHE}$ of the 82586 are not really needed to be latched. These lines stay valid for an entire memory cycle.

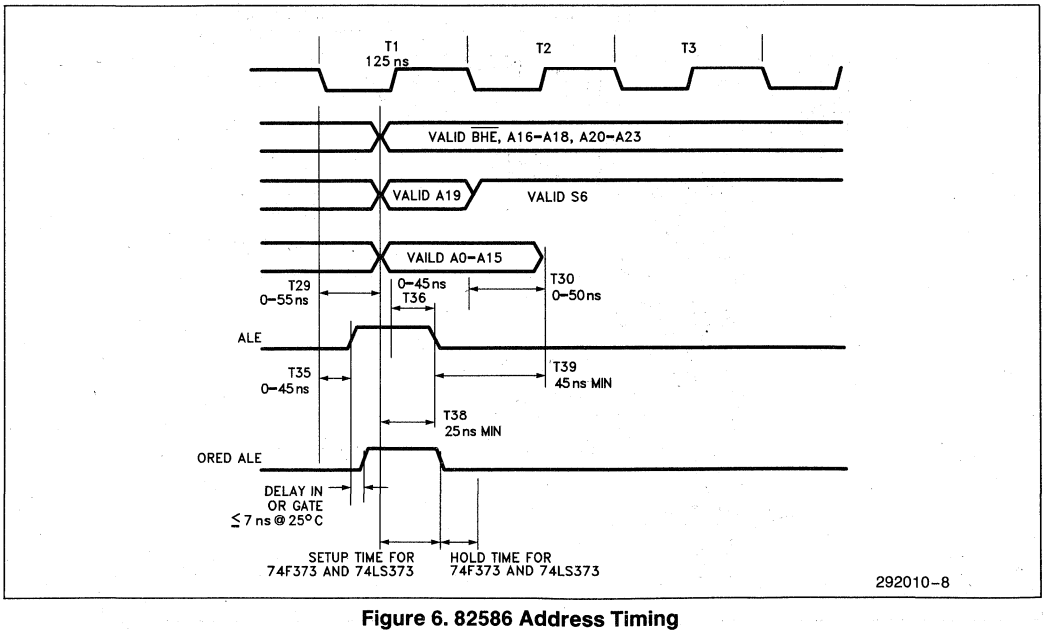


Figure 6. 82586 Address Timing

Table 2. 74F and 74LS Data Setup and Hold Time Specifications at 25°C

	74F373			74LS373			Unit
	Min	Nom	Max	Min	Nom	Max	
Data Setup Time	2 ↓			5 ↓			ns
Data Hold Time	3 ↓			20 ↓			ns

3.3 80186 Address Latch Interface

The address latch used by the 82586 is shared by the 80186. Figure 7 shows the 80186 address line timing with respect to the ALE. Again estimating 7 ns delay in the 74S32, the setup time for the latch is TAVAL + 7 and the hold time is TLLAX − 7. These are 37 ns and 23 ns respectively at 8 MHz. Comparing to the required values shown in Table 2, it is quite obvious that the setup and hold times of the latch are met by wide margins. Note that the 80186's address lines A16–A18 and $\overline{BHE}$ are not valid for an entire memory cycle; therefore, they have to be latched.

3.4 82586 Memory Interface

The 74LS373 has a delay of 18 ns for input data to reach the output assuming the latch enable is high. A demultiplexed valid address (output of the address latch), therefore, becomes available after T29 + 18 measuring from the beginning of T1 (Figure 8). The demultiplexed address remains valid until the ALE of the next memory access becomes active. Upper address lines, A14 through A20, are connected to a 16L8 PAL, which provides address decode logic for all memory devices. The PAL truth table is in Appendix A. The PAL has a maximum of 35 ns propagation delay, so chip selects will become active after 55 + 18 + 35 ns (max.) from the beginning of T1 as indicated in Figure 8. Since address decode logic is implemented by a PAL, any memory expansion would only require a reprogramming of this PAL.

Two 74LS245 bus transceiver chips are controlled by the DT/$\overline{R}$ and $\overline{DEN}$. Output enable and disable times of the 74LS245 are 40 and 25 ns respectively. The maximum propagation delay when the output enable is active is 12 ns.

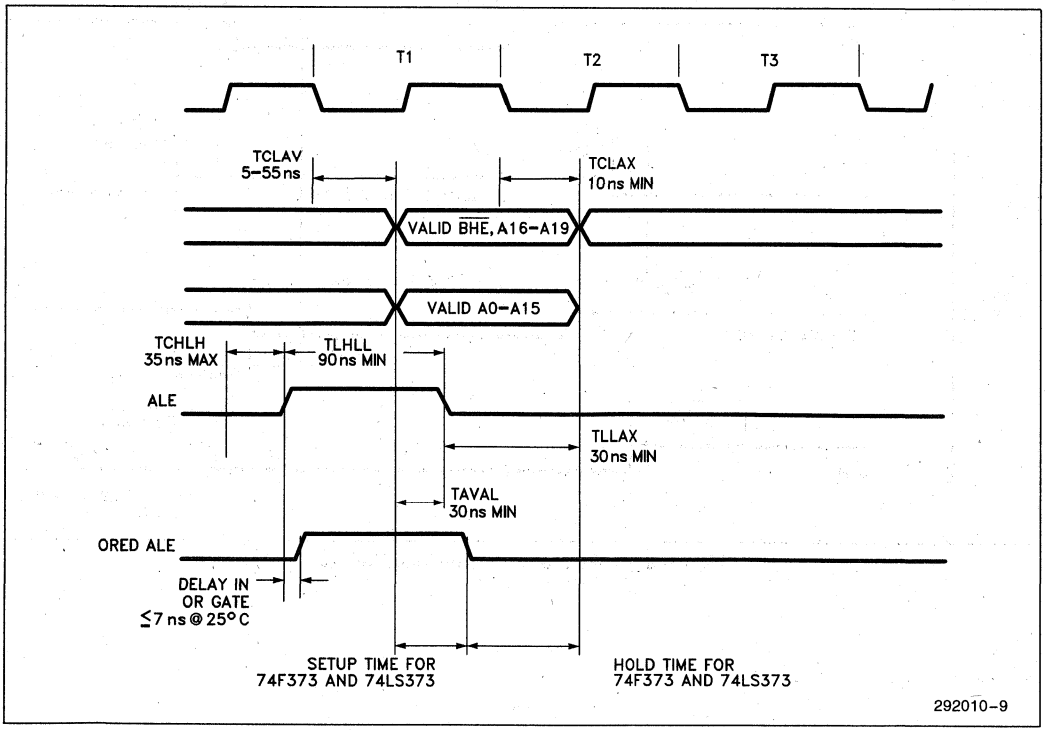


Figure 7. 80186 Address Timing

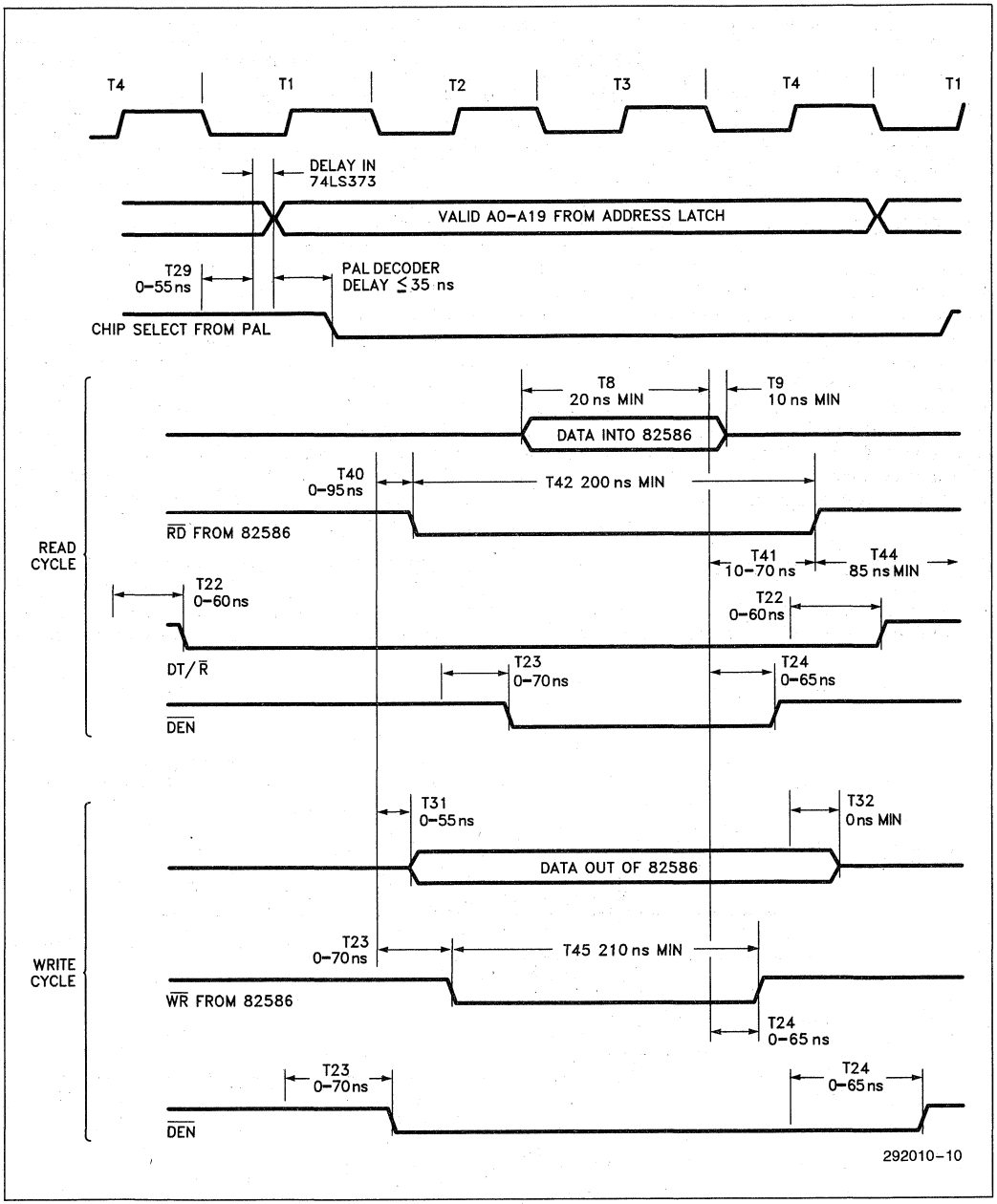


Figure 8. 82586 Memory Interface Timing

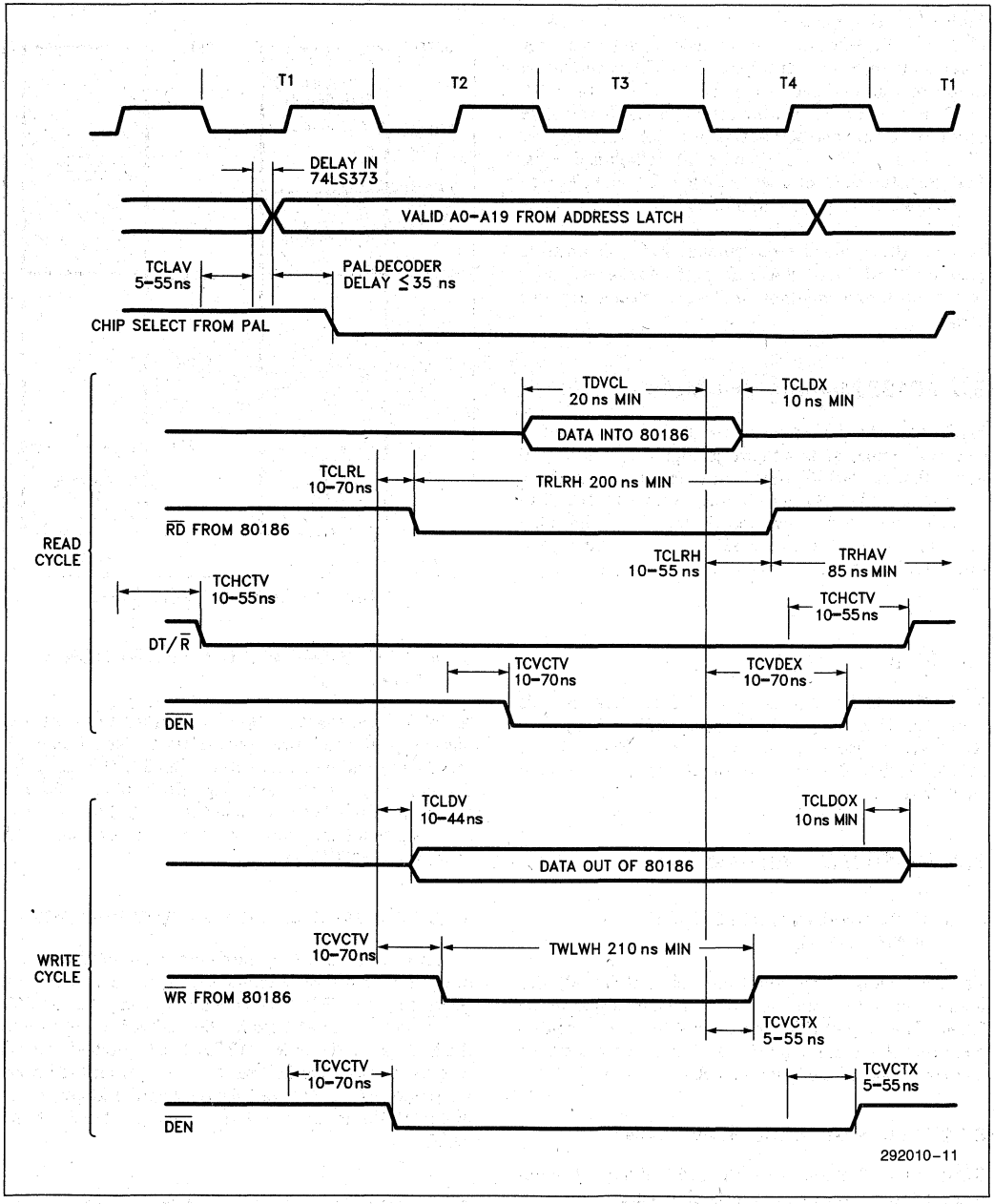


Figure 9. 80186 Memory Interface Timing

Address access time is 3 × T1 − T29 − 18 − T8 − 12 + n × T1, where n is the number of wait states. For 0 wait states operation at 8 MHz, it is 270 ns minimum. Chip select access time is 3 × T1 − T29 − 18 − T8 − 12 + n × T1 − 35, which is 235 ns for 0 wait state operation. Command access time for a read cycle is 2 × T1 − T40 − T8 − 12 + n × T1, which is 123 ns. Address setup time for a write cycle is T1 − T29 − 18 + T23, which is 52 ns minimum.

To meet these timing requirements, 2764-20s must be used for ROM. Static RAM chips, HM6264P-15, offer very wide timing margins and were selected for this design.

3.5 80186 Memory Interface

Figure 9 shows the timing of the 80186 memory interface. By comparing this figure to Figure 7, it is easy to notice that the 80186 offers a little faster bus interface. For example, TCLRL which is equivalent to T40 (0 to 95 ns) of the 82586 is specified as 10 to 70 ns. Since the memory choice satisfies the 82586 memory timing parameters, it also satisfies the 80186 memory timing parameters.

3.6 Memory Map

With 2764-20 EPROMs and 6264P-15 SRAMs, this board has 32 K bytes of ROM space and 16 K bytes of RAM space. Memory map is given in Figure 10. If 27128-20 EPROMs are used, the ROM space becomes 64 K bytes.

3.7 80186 I/O Interface

3.7.1 82586 CHANNEL ATTENTION GENERATION

The active low Peripheral Chip Select 0 ($\overline{PCS0}$) was used to generate a channel attention (CA) signal to the 82586. This way of CA generation satisfies the requirement that the width of a CA which must be wider than a clock period of the system clock.

3.7.2 82586 HARDWARE RESET PORT

$\overline{PCS1}$ of the 80186 will reset the 82586 if any I/O command is executed using this I/O chip select.

3.7.3 82530 INTERFACE

82530 interface to the 80186 was derived from the design example presented in the 82530 SCC-80186 Interface Ap Brief. This document is attached to this Ap Note as Appendix C.

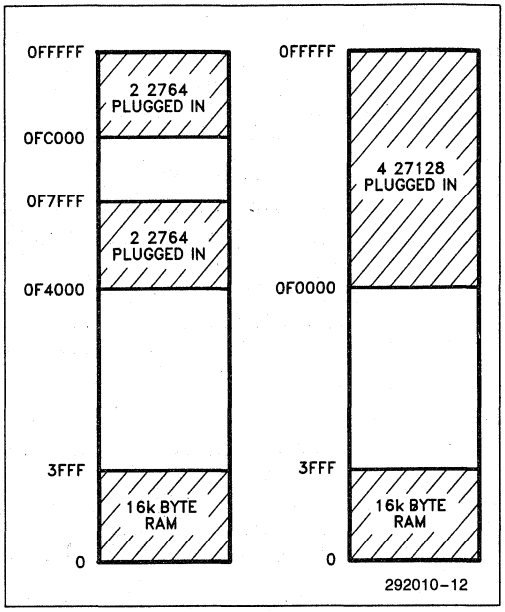


Figure 10. LANHIB Memory Map

3.7.4 82501 LOOPBACK CONFIGURATION PORT

A 74LS74 D-type flip flop was used for this port. On power up, it configures the 82501 to Non-Loopback mode by providing a high level to pin 3 ($\overline{LOOPBACK}$). The chip select is generated from the 80186's $\overline{PCS2}$ and the sychronized $\overline{WR}$ command of the 82530 interface. The least significant bit of I/O output data becomes the state of the 82501's pin 3.

3.7.5 ON-BOARD INDIVIDUAL ADDRESS PORT

To provide the 82586 a hardware configured host address, a 32x8 ROM is connected to the bus. The chip select for this ROM is generated from the 80186's $\overline{PCS3}$, so that the address for the ROM is mapped into the I/O space. Six or two (IEEE 802.3 specified address lengths) consecutive I/O reads starting from the lowest address of ROM will transfer the board address stored in the ROM to an IA-Setup command block of the 82586.

3.8 82586 Ready Signal Generation

82586 asynchronous ready (ARDY) signal is generated from a shift register. The shift register provides the 82586 a "normally ready" signal. When a wait state is needed, the ready signal is dropped to the low state. As shown in Table 3, the 82586 can be programmed to have 0 to 8 wait states by setting the DIP switch properly. Even though the on-board memory devices are

Table 3. DIP Switch Settings for Various Numbers of 82586 Wait States

Dip Switch Setting								Number of Wait States the 82586 Inserts
7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	0	0	2
1	1	1	1	1	0	0	0	3
1	1	1	1	0	0	0	0	4
1	1	1	0	0	0	0	0	5
1	1	0	0	0	0	0	0	6
1	0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	0	8

1 = Switch Open
0 = Switch Closed

fast enough for 0 wait states operation, this programmable wait state capability was added so that the effect of wait states on the 82586 performance could be evaluated.

3.9 82501 Circuits

Since the 82501 is designed to work with the 82586, no interfacing circuits are required.

The transceiver cable side of the 82501 requires some passive components. The receive and collision differential inputs must be terminated by 78Ω ±5% resistors. Common mode voltages on these differential inputs are established internally. 240Ω ±5% pull down resistors must be connected on the TRMT and $\overline{\text{TRMT}}$ output pins.

A 0.022 μF ±10% capacitor connected between pin 1 and 2 of the 82501 is for the analog phase-locked loop.

Connected between the X1 and X2 pins is a 20 MHz parallel resonant quartz crystal (antiresonant with 20 pF load fundamental mode). An internal divide-by-two counter generates the 10 MHz clock. Since both Ethernet and Cheapernet tolerate an error of only ±0.01% in bit rate, a high quality crystal is recommended. The accuracy of a crystal should be equal to or better than ±0.002% @ 25°C and ±0.005% for 0–70°C. A 30–35 pF capacitor is connected from each crystal pin (X1 and X2) to ground in order to adjust effective capacitance load for the crystal, which should be about 20 pF including stray capacitance.

3.10 82502 Circuits

3.10.1 ISOLATION AND POWER REQUIREMENTS

The IEEE 802.3 standard requires an electrical isolation within the transceiver (MAU). Cheapernet (10BASE2) requires the isolation means to withstand 500V ac, rms for one minute. Ethernet (10BASE5) requires 250 Vrms. This electrical isolation is normally accomplished by transformer coupling of each signal pair. The kind of transformers recommended for the 82502 are the pulse transformers which have a 1:1 turn ratio and at least 50 microhenry inductance. PE64102 and PE64107 manufactured by Pulse Engineering are found to be good selections for this purpose. The PE 64102 offers 500 Vrms isolation. The PE64107 offers 2000 Vrms isolation. Both products provide three transformers in one package. Even though the current Type 10BASE5 specification requires only 250 Vrms, it is very common to have a higher isolation, at least 500 Vrms, in transceivers.

The standard specifies the voltage input level and maximum current allowed on the power pair of the transceiver cable. The voltage level may be between +11.28V dc and +15.75V dc. The maximum current is limited to 500 mA. Since the 82502 requires +10V ±10% and +5V ±10% as power, there has to be a DC/DC converter. In addition the DC/DC converter must be isolated due to the requirement described above. The DC/DC converter should be able to supply about 100 mA on the +10V line and 60 mA on the 5V line. The efficiency required in the converter is, therefore, ((11V × 100 mA + 5.5V × 60 mA) / ((11.28V − 0.5A × 4Ω) × 500 mA)) × 100 = 31% worst case. 4Ω is the maximum round trip resistance the power pair may have. 82502's CMOS process is the major contributor to this low DC/DC efficiency requirement.

Since the DC/DC converter has an isolation transformer inside, the output voltages are all floating voltages. The 0V output of the converter, for example, has no voltage relationship with the DTE's ground. The V_{SS} and AV_{SS} pins of the 82502 should be connected to the 0V output of the DC/DC converter which is the 82502's ground (reference voltage).

Both Pulse Engineering and Reliability Incorporated produce DC/DC converters that meet the 82502's requirements. The Pulse Engineering's part number is PE64369 (enclosed in this design kit). The device measures about 1.5" x 1.5" x 0.5" and provides 2000 Vrms breakdown. The Reliability's part number is 2E12R10-5. Preliminary data sheets are available from Reliability.

3.10.2 OTHER PASSIVE AND ACTIVE DEVICES FOR THE 82502

A 78Ω ±5% resistor is required to terminate the transmit pair of the Transceiver cable. The chip has an internal circuit that establishes a common mode voltage, thus no voltage divider is required. The receive and collision pair drivers need pull up resistors. A 43.2 ±1% resistor must be connected from each output pin to +5V.

A 243Ω ±0.5% precision resistor is required on the REXT pin to the ground. The accuracy of this resistor is very important since this resistor is a part of current and voltage reference circuits in the analog sections of the 82502.

Grounding the HBD (Heartbeat Disable) pin will allow the chip to perform Signal Quality Error check (Heartbeat) as required by the IEEE 802.3. The chip will transmit the collision presence signal after each transmission during Interframe Spacing (IFS) time. In a repeater application, this feature is disabled (HBD = +5V).

Diodes connected on the CXTD pin are to reduce the capacitive loading onto the coaxial cable. One diode is sufficient, but two will provide a protection in case one burns out (Short Circuit). The diode should have about 2 pF shunt capacitance at Vd = 0V and be able to handle at least 100 mA when biased in forward direction. A few candidates are 1N5282, 1N3600, and 1N4150.

A 100Ω fusible resistor connected on the CXRD pin is purely for protection. It is there as a fuse, not as a resistor. The 82502 works without this resistor. The IEEE 802.3, however, states that "component failures within the MAU (Media Attachment Unit or Transceiver) electronics should not prevent communication among other MAUs on the coaxial cable." It is recommending a transceiver design that minimizes the probability of total network failure. The fusible resistor will provide an open circuit in an event of excess current. A short circuit from the CXRD pin to ground will not bring down the network due to the blown fuse.

A 1 MΩ resistor connected between the coaxial cable shield and the Transceiver cable shield will provide a static discharge path. The Ethernet coaxial cable should also have an effective earth ground at one point in a network as required by the standard. A 0.01 μF in parallel to the 1 MΩ resistor provides ground for RF signals.

3.10.3 LAYOUT CONSIDERATION FOR THE 82502 CIRCUITS

It is strongly recommended that the board have a special ground plane for the 82502 (see Figure 11). The 0V (reference) output of the isolated DC/DC converter should be connected to the ground plane. The V_{SS} and AV_{SS} pins of the 82502 should be connected to the ground plane with minimum lead wires.

There should be a 0.22 μF capacitor connected between the coaxial cable shield and ground. The signal path from the coax. shield through the 0.22 μF capacitor to

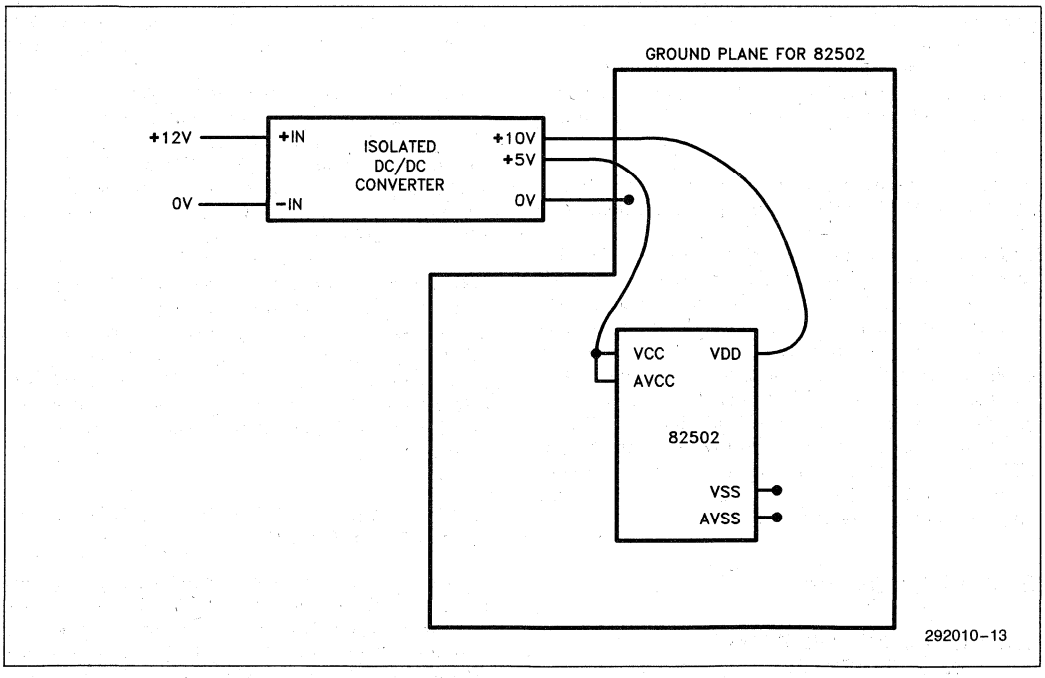


Figure 11. Ground Plane for the 82502

the ground should be kept as short as possible—leads of the 0.22 μF capacitor should be as short as possible.

The path length from the CXTD pin through two diodes to the center conductor of the coax should also be minimized.

These are recommendations which will produce a more reliable circuit if followed carefully. Remember that the 82502 has analog circuits in it.

4.0 DEMONSTRATION SOFTWARE

The demonstration software included in this Ap Note is called "Traffic Simulator and Monitor Station" (TSMS) program. The TSMS program is written in PL/M and has the following features:

1. Programmable network load generation
2. Network statistical monitoring capabilities
3. Interactive command execution of all 82586 commands
4. Interactive buffer monitoring

The environment created with the TSMS program was found to be very useful for network debugging and other individual station's hardware and software debugging. The TSMS software listing is found in Appendix B.

NOTE:
The 82586 Date Link Driver presented in AP Note 235 also runs on the LANHIB. Please refer to the Ap Note for detailed operations of the software.

4.1 Programming PROMs to Run the TSMS Program

By returning the card enclosed in this kit or by contacting Insite, the TSMS program and related batch files can be obtained on a diskette. TSMS related files that are on the diskette are:

READ.ME
TSMS.PLM
IO.PLM
INI186.PLM
LANHIB.BAT
SBC.BAT
IUPHIB.BAT
IUPSBC.BAT
HI.BYT
LO.BYT
ROM.BAT

The READ.ME file contains instructions for programming PROMs. HI.BYT and LO.BYT are the files which can be downloaded to PROMs directly. These files are already configured for the LANHIB. The batch file ROM.BAT invokes the Intel PROM Programming Software (iPPS) under the DOS operation system and programs two 2764 EPROMs. The Intel Universal Programmer must be placed in ON-LINE mode.

Other files contained in the diskette are for compiling and locating the original TSMS program. Using these files, the original TSMS program can be changed or can be compiled for an iSBC 186/51. 'TSMS.PLM' is the original TSMS source program. 'IO.PLM' contains the IO driver needed when the TSMS program is run on the iSBC 186/51. INI186.PLM is the LANHIB initialization routine. LANHIB.BAT is the batch file that compiles, links, and locates the TSMS program and the LANHIB initialization routine. SBC.BAT compiles, links, and locates the TSMS program and the IO driver for the iSBC 186/51. IUPHIB.BAT programs two 2764s for the LANHIB. IUPSBC.BAT programs two 2764s for the iSBC 186/51.

Therefore, if the TSMS program is to be run on the LANHIB (Demo board), steps required are:

1. C:>LANHIB
2. C:>IUPHIB

If the TSMS program is to be run on the iSBC 186/51, steps required are:

1. C:>SBC
2. C:>IUPSBC

4.2 Capabilities and Limits of the TSMS Program

The TSMS program initializes the LANHIB Ethernet/Cheapernet station by executing 82586's Diagnose, Configure, IA-Setup, and MC-Setup commands. The program asks a series of questions in order to set up a linked list of these 82586 commands. After initialization is completed, the program automatically starts the 82586's Receive Unit (monitoring capability). Transmissions are optional (traffic simulation capability).

The TSMS program has two modes of operation: Continuous mode and Interactive Command Execution mode. The program automatically gets into the Continuous mode after initialization. The Interactive Command Execution mode can be entered from the Continuous mode. Once entered in the Continuous mode, the software uses the format shown in Figure 12 to display information. Detailed description of each of these fields is as follows:

Host Address: host (station) address used in the most recently prepared IA-Setup command. The software simply writes the address stored in the IA-Setup command block with its least significant bit being in the most right position. Note that if the IA-Setup com-

```
*************************** Station Configuration ************************

 Host Address: 00  AA  00  00  18  6D
 Multicast Address(es): No Multicast Addresses Defined
 Destination Address: FF  FF  FF  FF  FF  FF
 Frame Length: 118 bytes
 Time Interval between Transmit Frames:  159.4 microseconds
 Network Percent Load generated by this station: 35.7 %
 Transmit Frame Terminal Count: Not Defined
 82586 Configuration Block: 08  00  26  00  60  00  F2  00  00  40

***************************** Station Activities *************************

 # of Good     # of Good     CRC          Alignment    No           Receive
 Frames        Frames        Errors       Errors       Resource     Overrun
 Transmitted   Received                                 Errors       Errors
 10130         0             0            0            0            0
                                                                    292010-14
```

Figure 12. Continuous Mode Display

mand was just set up and not executed, the address displayed in this field may not be the address stored in the 82586.

Multicast Address(es): multicast addresses used in the most recently prepared MC-Setup command. As in the case of host address, the software simply writes the addresses stored in the MC-Setup command block. Note that if the MC-Setup command was just set up and not executed, the addresses displayed in this field may not be the addresses stored in the 82586.

Destination Address: destination address stored in the transmit command block if AL-LOC=0. If AL-LOC=1, destination address is picked up from the transmit buffer. The least significant bit is in the most right position.

Frame Length: transmit frame byte count including destination address, source address, length, data, and CRC field.

Time Interval Between Transmit Frames: approximate time interval obtainable between transmit frames (Figure 13). The number is correct if there are no other stations transmitting on the network.

Network Percent Load Generated by This Station: approximate network percent load that is generated by this station (Figure 13). The number is correct if there are no other stations transmitting on the network.

Transmit Frame Terminal Count: number of frames this station will transmit before it stops network traffic load generation. If this station is transmitting indefinitely, this field will be 'Not Defined'.

82586 Configuration Block: configuration parameters used in the most recently prepared Configure command. As in the case of IA-Setup command, the software simply writes the parameters from the Configure command block. The least significant byte (FIFO Limit) of the configuration parameters is printed in the most left position.

# of Good Frames Transmitted: number of good frames transmitted. This is a snap shot of the 32-bit transmit frame counter. It is incremented only when both C and OK bits of the transmit command status are set after an execution. The counter is 32-bit wide.

# of Good Frames Received: number of good frames received. This is a snap shot of the 32-bit receive frame counter. It is incremented only when both C and OK bits of a receive frame descriptor status are set after a reception. The counter is 32-bit wide.

CRC Errors: number of frames that had a CRC error. This is a snap shot of the 16-bit CRC counter maintained by the 82586 in the SCB.

Alignment Errors: number of frames that had an alignment error. This is a snap shot of the 16-bit alignment counter maintained by the 82586 in the SCB.

No Resource Errors: number of frames that had a no resource error. This is a snap shot of the 16-bit no resource counter maintained by the 82586 in the SCB.

Receive Overrun Errors: number of frames that had a receive overrun error. This is a snap shot of the 16-bit receive overrun error counter maintained by the 82586 in the SCB.

If the station is actively transmitting, # of good frames transmitted should be incrementing. If the station is actively receiving frames, # of good frames received should be incrementing. In this continuous mode, a user can see the activities of the network.

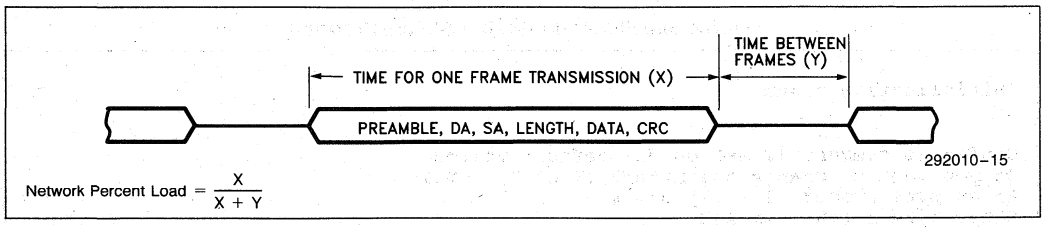


Figure 13. Network Percent Load

Hitting any key on the keyboard while the program is running in the Continuous mode will exit the mode. The program will respond with a message 'Enter Command (H for Help) → '. In this Interactive Command Execution mode, a user can set up any one of the 82586 action commands and/or execute any one of the 82586 SCB control commands. Setting up a Dump command and executing a SCB Command Unit Start command will, for example, execute the Dump command. Display commands are also available to see the contents of the 82586's data structure blocks. A display command will enable a user to see the contents of the 82586's dump (see Section 6.3).

Typing 'E' after 'Enter command (H for help) → ', executing a SCB Command Unit Start command with a transmit command, or executing a SCB Receive Unit Start command will exit the Interactive Command Execution mode. The program will be back in the Continuous mode. Using this Interactive Command Execution mode, one can, for example, reconfigure the station and come back to the Continuous mode. Section 6 lists actual example executions of the TSMS program.

The TSMS program should be run in an 8 MHz system. The software running at 8 MHz with a maximum of 2 wait states has been tested and verified to be able to receive back-to-back frames separated by 9.6 microseconds and still keep track of the correct number of frames received. This capability, for example, can be used to find out exactly how many frames a new station in the network had transmitted.

The software does not perform extensive loopback tests and hardware diagnostics during the initialization. A loopback operation can be performed interactively in the Interactive Command Execution mode.

The software allows a user to set up only 8 multicast addresses maximum. It is not possible with this program to set up more than 8 multicast addresses.

The command chaining feature of the 82586 is not used in the Interactive Command Execution mode. Each command setup performed by a 'S' command after 'Enter command (H for help) → ' sets up a command with its EL bit set, I bit reset, and S bit reset. Diagnose, Configure, IA-Setup, and MC-Setup commands are chained together during the initialization routine and executed at once with only one CA.

The software sets up 5 Receive Frame Descriptors linked in a circular list. Therefore, a user can see only the last 5 frames the station has received. It also sets up 5 receive buffers, each being 1514 bytes long, linked in circle. Therefore, the 82586 never goes into the NO RESOURCES state.

4.3 Example Executions of the TSMS Program

This section presents three example executions of the TSMS program. When the TSMS program needs a command to be typed, it asks a question with ' → '. Anything after ' → ' is what a user needs to type in on the keyboard. To switch from the continuous mode to the interactive command execution mode, type any key on the keyboard.

4.3.1 EXAMPLE 1: EXTERNAL LOOPBACK EXECUTION

In this example, 500 external loopback transmissions and receptions are executed (Figure 14). In order for the software to process each loopback properly, a large delay was given between transmissions.

4.3.2 EXAMPLE 2: FRAME RECEPTION IN PROMISCUOUS MODE

The 82586 is configured to receive any frame that exists in the network (Figure 15). In this example, the station received 100 frames.

4.3.3 EXAMPLE 3: 35.7% NETWORK TRAFFIC LOAD GENERATION

The station is programmed to transmit 118 byte long frames with a time interval of 159.4 microseconds in between (Figure 16). The network load is about 35.7 percent if no other stations are transmitting in the network.

A key was hit to enter the Interactive Command Execution mode. In that mode, a Dump command was executed and the result was displayed. After the Dump execution, a transmit command was set up again and the station was put in the Continuous mode.

Traffic Simulator and Monitor Station Program

```
Initialization begun


Configure command is set up for default values.
Do you want to change any bytes? (Y or N) ==> Y
Enter byte number (1 - 11) ==> 4
Enter byte 4 (4H) ==> A6H
Any more bytes? (Y or N) ==> Y
Enter byte number (1 - 11) ==> 11
Enter byte 11 (BH) ==> 6
Any more bytes? (Y or N) ==> N
Configure the 586 with the prewired board address ==> N
Enter this station's address in Hex ==> 000000002200
You can enter up to 8 Multicast Addresses.
Would you like to enter a Multicast Address? (Y or N) ==> N
You entered 0 Multicast Address(es).

Would you like to transmit?
Enter a Y or N ==> Y
Enter a destination address in Hex ==> 000000002200

Enter TYPE ==> 0
How many bytes of transmit data?
Enter a number ==> 2
Transmit Data is continuous numbers (0, 1, 2, 3, ... )
Change any data bytes? (Y or N) ==> N
Enter a delay count ==> 10000000000
The number is too big.
It has to be less than or equal to 65535 (FFFFH).
Enter a number ==> 60000

Setup a transmit terminal count? (Y or N) ==> Y
Enter a transmit terminal count ==> 500

Destination Address: 00  00  00  00  22  00
Frame Length: 20 bytes
Time Interval between Transmit Frames: 30.18 miliseconds
Network Percent Load generated by this station:    .0 %
Transmit Frame Terminal Count: 500

Good enough? (Y or N) ==> Y

Receive Unit is active.
```

292010-16

Figure 14. External Loopback Execution

```
---Transmit Command Block---
0000 at 033E
8004
FFFF
034E
2200
0000
0000
0000


Hit <CR> to countinue

transmission started!




**************************** Station Configuration ************************

 Host Address: 00  00  00  00  22  00
 Multicast Address(es): No Multicast Addresses Defined
 Destination Address: 00  00  00  00  22  00
 Frame Length: 20 bytes
 Time Interval between Transmit Frames: 30.18 miliseconds
 Network Percent Load generated by this station:    .0 %
 Transmit Frame Terminal Count: 500
 82586 Configuration Block: 08  00  A6  00  60  00  F2  00  00  06

**************************** Station Activities ***************************

 # of Good     # of Good     CRC          Alignment    No           Receive
 Frames       Frames        Errors       Errors       Resource     Overrun
 Transmitted  Received                                 Errors       Errors
 500          500           0            0            0            0
```

292010-17

Figure 14. External Loopback Execution (Continued)

Traffic Simulator and Monitor Station Program

```
Initialization begun


Configure command is set up for default values.
Do you want to change any bytes? (Y or N) ==> Y
Enter byte number (1 - 11) ==> 9
Enter byte 9 (9H) ==> 1
Any more bytes? (Y or N) ==> N
Configure the 586 with the prewired board address ==> Y
You can enter up to 8 Multicast Addresses.
Would you like to enter a Multicast Address? (Y or N) ==> N
You entered 0 Multicast Address(es).

Would you like to transmit?
Enter a Y or N ==> N

Receive Unit is active.




**************************** Station Configuration ************************

Host Address: 00  AA  00  00  18  6D
Multicast Address(es): No Multicast Addresses Defined
82586 Configuration Block: 08  00  26  00  60  00  F2  01  00  40

**************************** Station Activities ***************************

# of Good     # of Good     CRC          Alignment     No           Receive
Frames        Frames        Errors       Errors        Resource     Overrun
Transmitted   Received                                 Errors       Errors
0             100           0            0             0            0
Enter command (H for help) ==> D

Command Block or Receive Area? (R or C) ==> R
Frame Descriptors:
4000 at 036C  A000 at 0382  A000 at 0398  A000 at 03AE  A000 at 03C4
0000          0000          0000          0000          0000
0382          0398          03AE          03C4          036C
03DA          03E4          03EE          03F8          0402
2200          2200          2200          2200          2200
2200          2200          2200          2200          2200
0000          0000          0000          0000          0000
```

292010-18

Figure 15. Frame Reception in Promiscuous Mode

```
0000          0000          0000          0000          0000
0000          0000          0000          0000          0000
0000          0000          0000          0000          0000
0000          0000          0000          0000          0000

 Receive Buffer Descriptors:
C064 at 03DA  C064 at 03E4  C064 at 03EE  C064 at 03F8  C064 at 0402
03E4          03EE          03F8          0402          03DA
040C          09F6          0FE0          15CA          1BB4
0000          0000          0000          0000          0000
05DC          05DC          05DC          05DC          05DC

 Display the receive buffers? (Y or N) ==> Y
 Receive Buffers:

 Receive Buffer 0 :
002C:014C 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
002C:015C 10  11  12  13  14  15  16  17  18  19  1A  1B  1C  1D  1E  1F
002C:016C 20  21  22  23  24  25  26  27  28  29  2A  2B  2C  2D  2E  2F
002C:017C 30  31  32  33  34  35  36  37  38  39  3A  3B  3C  3D  3E  3F
002C:018C 40  41  42  43  44  45  46  47  48  49  4A  4B  4C  4D  4E  4F
002C:019C 50  51  52  53  54  55  56  57  58  59  5A  5B  5C  5D  5E  5F
002C:01AC 60  61  62  63

Hit <CR> to countinue

 Receive Buffer 1 :
002C:0736 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
002C:0746 10  11  12  13  14  15  16  17  18  19  1A  1B  1C  1D  1E  1F
002C:0756 20  21  22  23  24  25  26  27  28  29  2A  2B  2C  2D  2E  2F
002C:0766 30  31  32  33  34  35  36  37  38  39  3A  3B  3C  3D  3E  3F
002C:0776 40  41  42  43  44  45  46  47  48  49  4A  4B  4C  4D  4E  4F
002C:0786 50  51  52  53  54  55  56  57  58  59  5A  5B  5C  5D  5E  5F
002C:0796 60  61  62  63

Hit <CR> to countinue

 Receive Buffer 2 :
002C:0D20 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
002C:0D30 10  11  12  13  14  15  16  17  18  19  1A  1B  1C  1D  1E  1F
002C:0D40 20  21  22  23  24  25  26  27  28  29  2A  2B  2C  2D  2E  2F
002C:0D50 30  31  32  33  34  35  36  37  38  39  3A  3B  3C  3D  3E  3F
002C:0D60 40  41  42  43  44  45  46  47  48  49  4A  4B  4C  4D  4E  4F
002C:0D70 50  51  52  53  54  55  56  57  58  59  5A  5B  5C  5D  5E  5F
002C:0D80 60  61  62  63

Hit <CR> to countinue

 Receive Buffer 3 :
002C:130A 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
002C:131A 10  11  12  13  14  15  16  17  18  19  1A  1B  1C  1D  1E  1F
002C:132A 20  21  22  23  24  25  26  27  28  29  2A  2B  2C  2D  2E  2F
002C:133A 30  31  32  33  34  35  36  37  38  39  3A  3B  3C  3D  3E  3F
002C:134A 40  41  42  43  44  45  46  47  48  49  4A  4B  4C  4D  4E  4F
002C:135A 50  51  52  53  54  55  56  57  58  59  5A  5B  5C  5D  5E  5F
002C:136A 60  61  62  63

Hit <CR> to countinue
```

292010-19

Figure 15. Frame Reception in Promiscuous Mode (Continued)

```
 Receive Buffer 4 :
002C:18F4 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
002C:1904 10  11  12  13  14  15  16  17  18  19  1A  1B  1C  1D  1E  1F
002C:1914 20  21  22  23  24  25  26  27  28  29  2A  2B  2C  2D  2E  2F
002C:1924 30  31  32  33  34  35  36  37  38  39  3A  3B  3C  3D  3E  3F
002C:1934 40  41  42  43  44  45  46  47  48  49  4A  4B  4C  4D  4E  4F
002C:1944 50  51  52  53  54  55  56  57  58  59  5A  5B  5C  5D  5E  5F
002C:1954 60  61  62  63

Hit <CR> to countinue


 Enter command (H for help) ==> E




**************************** Station Cofiguration *************************

 Host Address: 00  AA  00  00  18  6D
 Multicast Address(es): No Multicast Addresses Defined
 82586 Configuration Block: 08  00  26  00  60  00  F2  01  00  40

**************************** Station Activities ***************************

 # of Good     # of Good     CRC          Alignment    No           Receive
 Frames        Frames        Errors       Errors       Resource     Overrun
 Transmitted   Received                                 Errors       Errors
 0             100           0            0            0            0
```

292010-20

Figure 15. Frame Reception in Promiscuous Mode (Continued)

Traffic Simulator and Monitor Station Program

```
 Initialization begun


 Configure command is set up for default values.
 Do you want to change any bytes? (Y or N) ==> N
 Configure the 586 with the prewired board address ==> Y
 You can enter up to 8 Multicast Addresses.
 Would you like to enter a Multicast Address? (Y or N) ==> N
 You entered 0 Multicast Address(es).

 Would you like to transmit?
 Enter a Y or N ==> Y
 Enter a destination address in Hex ==> FFFFFFFFFFFF

 Enter TYPE ==> 0
 How many bytes of transmit data?
 Enter a number ==> 100
 Transmit Data is continuous numbers (0, 1, 2, 3, ... )
 Change any data bytes? (Y or N) ==> N

 Enter a delay count ==> 0

 Setup a transmit terminal count? (Y or N) ==> N

 Destination Address: FF  FF  FF  FF  FF  FF
 Frame Length: 118 bytes
 Time Interval between Transmit Frames:  159.4 microseconds
 Network Percent Load generated by this station: 35.7 %
 Transmit Frame Terminal Count: Not Defined

 Good enough? (Y or N) ==> Y

 Receive Unit is active.

---Transmit Command Block---
0000 at 033E
8004
FFFF
034E
FFFF
FFFF
FFFF
0000


Hit <CR> to countinue
```

292010-21

Figure 16. 35.7% Network Load Generation

```
transmission started!


*************************** Station Configuration ************************

 Host Address: 00  AA  00  00  18  6D
 Multicast Address(es): No Multicast Addresses Defined
 Destination Address: FF  FF  FF  FF  FF  FF
 Frame Length: 118 bytes
 Time Interval between Transmit Frames:  159.4 microseconds
 Network Percent Load generated by this station: 35.7 %
 Transmit Frame Terminal Count: Not Defined
 82586 Configuration Block: 08  00  26  00  60  00  F2  00  00  40

*************************** Station Activities ***************************

 # of Good     # of Good     CRC          Alignment    No           Receive
 Frames       Frames        Errors       Errors       Resource     Overrun
 Transmitted  Received                                 Errors       Errors
 10459        0             0            0            0            0
 Enter command (H for help) ==> H

 Commands are:
 S - Setup CB              D - Display RFD/CB
 P - Print SCB             C - SCB Control CMD
 L - ESI Loopback On       N - ESI Loopback Off
 A - Toggle Number Base
 Z - Clear Tx Frame Counter
 Y - Clear Rx Frame Counter
 E - Exit to Continuous Mode


 Enter command (H for help) ==> S

 Enter command block type (H for help) ==> H

Command block type:
 N - Nop                   I - IA Setup
 C - Configure             M - MA Setup
 T - Transmit              R - TDR
 D - Diagnose              S - Dump Status
 H - Print this message

 Enter command block type (H for help) ==> S

 Enter command (H for help) ==> C

 Do you want to enter any SCB commands? (Y or N) ==> Y
 Enter CUC ==> 1
 Enter RES bit ==> 0
 Enter RUC ==> 0
 Issued Channel Attention



 Enter command (H for help) ==> D
```

292010-22

Figure 16. 35.7% Network Load Generation (Continued)

```
 Command Block or Receive Area? (R or C) ==> C
---Dump Status Command Block---
A000 at 0364
8006
FFFF
27D6

 Dump Status Results
 at 27D6
00  E8  3F  26  08  60  00  FA  00  00  40  FF  6D  18  00  00
AA  00  40  20  00  00  00  00  FF  FF  FF  FF  B5  9E  EE  CF
62  63  3F  B0  00  00  00  00  00  00  00  00  FF  85  08  FC
00  00  00  00  00  00  00  00  00  00  00  00  70  03  06  00
DC  05  00  00  0C  04  DC  05  E4  03  DA  03  DA  03  78  05
82  03  6C  03  F8  03  64  80  D6  27  E8  21  FF  FF  4E  03
06  80  FF  FF  64  03  00  00  D2  02  00  00  00  00  00  00
00  00  D6  27  00  01  00  28  00  00  00  00  30  26  00  00
20  00  40  06  30  01  00  00  90  00  10  01  00  00  6C  03
00  00  6A  03  0E  00  6C  28  00  00  74  03  00  00  00  00
00  00  00  00  00  C0  00  00  00  00

 Enter command (H for help) ==> S

 Enter command block type (H for help) ==> T
 Enter a destination address in Hex ==> FFFFFFFFFFFF

 Enter TYPE ==> 0
 How many bytes of transmit data?
 Enter a number ==> 100
 Transmit Data is continuous numbers (0, 1, 2, 3, ... )
 Change any data bytes? (Y or N) ==> N

 Enter a delay count ==> 0

 Setup a transmit terminal count? (Y or N) ==> N

 Destination Address: FF  FF  FF  FF  FF  FF
 Frame Length: 118 bytes
 Time Interval between Transmit Frames:  159.4 microseconds
 Network Percent Load generated by this station: 35.7 %
 Transmit Frame Terminal Count: Not Defined

 Good enough? (Y or N) ==> Y

 Enter command (H for help) ==> C

 Do you want to enter any SCB commands? (Y or N) ==> Y
 Enter CUC ==> 1
 Enter RES bit ==> 0
 Enter RUC ==> 0
 Issued Channel Attention
```

292010-23

Figure 16. 35.7% Network Load Generation (Continued)

```
**************************** Station Configuration ************************

 Host Address: 00  AA  00  00  18  6D
 Multicast Address(es): No Multicast Addresses Defined
 Destination Address: FF  FF  FF  FF  FF  FF
 Frame Length: 118 bytes
 Time Interval between Transmit Frames:  159.4 microseconds
 Network Percent Load generated by this station: 35.7 %
 Transmit Frame Terminal Count: Not Defined
 82586 Configuration Block: 08  00  26  00  60  00  F2  00  00  40

**************************** Station Activities ***************************

 # of Good     # of Good     CRC          Alignment    No           Receive
 Frames       Frames        Errors       Errors       Resource     Overrun
 Transmitted  Received                                 Errors       Errors
 106020       0             0            0            0            0
```

292010-24

Figure 16. 35.7% Network Load Generation (Continued)

5.0 IN CASE OF DIFFICULTY

This section presents methods of troubleshooting ("debugging") a LANHIB board. When a LANHIB board is powered up with the TSMS program stored in EPROMs, it should display "TRAFFIC SIMULATOR AND MONITOR STATION PROGRAM" message on a terminal screen. If the message is not displayed, the board has to be debugged. Section 5.1 describes basic 80186/82586 system troubleshooting procedures. Section 5.2 is for troubleshooting 82501 and 82502 circuits. After the 80186/82586 system is debugged, the 82501/82502 circuits have to be tested.

5.1 Troubleshooting 80186/82586 System

Shown in Figure 17 is a flow chart for troubleshooting 80186/82586 system. The procedure requires an oscilloscope. A logic analyzer is needed if problems appear to be serious. The procedures will debug the board to the point where the 82530 is initialized properly. If the 82530 can be initialized properly, ROM and RAM interfaces must be functioning. Board initialization routines (INI186.PLM) linked to the TSMS program requires ROM and RAM accesses. Since the 82586 shares most of the system with the 80186, no special debugging is required for the 82586. Wiring of all 82586 parallel signal pins should, however, be checked.

The flow chart branches to two major paths after the first decision box. One path debugs the RS-232 channel and the other debugs the 80186/82586 system. The waveform of the $\overline{\text{TRXCB}}$ output of the 82530 determines which path to be taken. If the 82530 is getting programmed properly, there should be 153.6 KHz ($1/f = 6.51\ \mu s$) clock on this output pin. If there is a clock, the problem is probably in the RS-232 interface. If there is no clock, then the system has to be debugged using a logic analyzer.

5.2 Troubleshooting 82501/82502 Circuits

If the TSMS program runs on the LANHIB but the 82586 is not able to transmit or receive, there must be a problem in 82501/82502 circuits. The flow chart in Figure 19 will guide troubleshooting in these circuits. An oscilloscope is required.

The board should be configured to Cheapernet and disconnected from the network. Two terminators will be required to terminate a "T" BNC connector providing an effective load resistance of 25Ω to the 82502.

The 82586 must have the system and transmit clocks running upon reset. Since the transmit clock is generated by the 82501, the 82501 transmit clock output pin (pin 16) should be checked. The TSMS program executes 82586's Diagnose, Configure, IA-Setup, and MC-Setup commands during initialization. If the 82586 has active $\overline{\text{CRS}}$ (Carrier Sense) signal, it cannot complete execution of these commands. The 82501 should, therefore, be checked if it is generating inactive $\overline{\text{CRS}}$ signal to the 82586 after power up. The LANHIB powers up the 82501 in non-loopback mode.

After making sure that the 82501 is generating proper signals to the 82586, the TSMS program is restarted with an initialization shown in Figure 20. The 82586 is configured to EXT-LPBK = 1, TONO-CRS = 1, and MIN-FRM-LEN = 6. The chip is also loaded with a destination address identical to the source address. If there are no problems in the 82501/82502 circuits, the station will be receiving its own transmitted frames. If problems exist, the station will only be transmitting. Since the 82586 is configured to TONO-CRS (Transmission On NO Carrier Sense), the chip will keep transmitting regardless of the state of carrier sense. The 82501/82502 circuits can then be probed with an oscilloscope at the locations indicated in Figure 21. Probing will catch problems like wiring mistakes, missing load resistors, etc.

Once the station is debugged, it can be connected to the network. If there is a problem in the network, the 82586's TDR command can be used to find the location and nature of the problem.

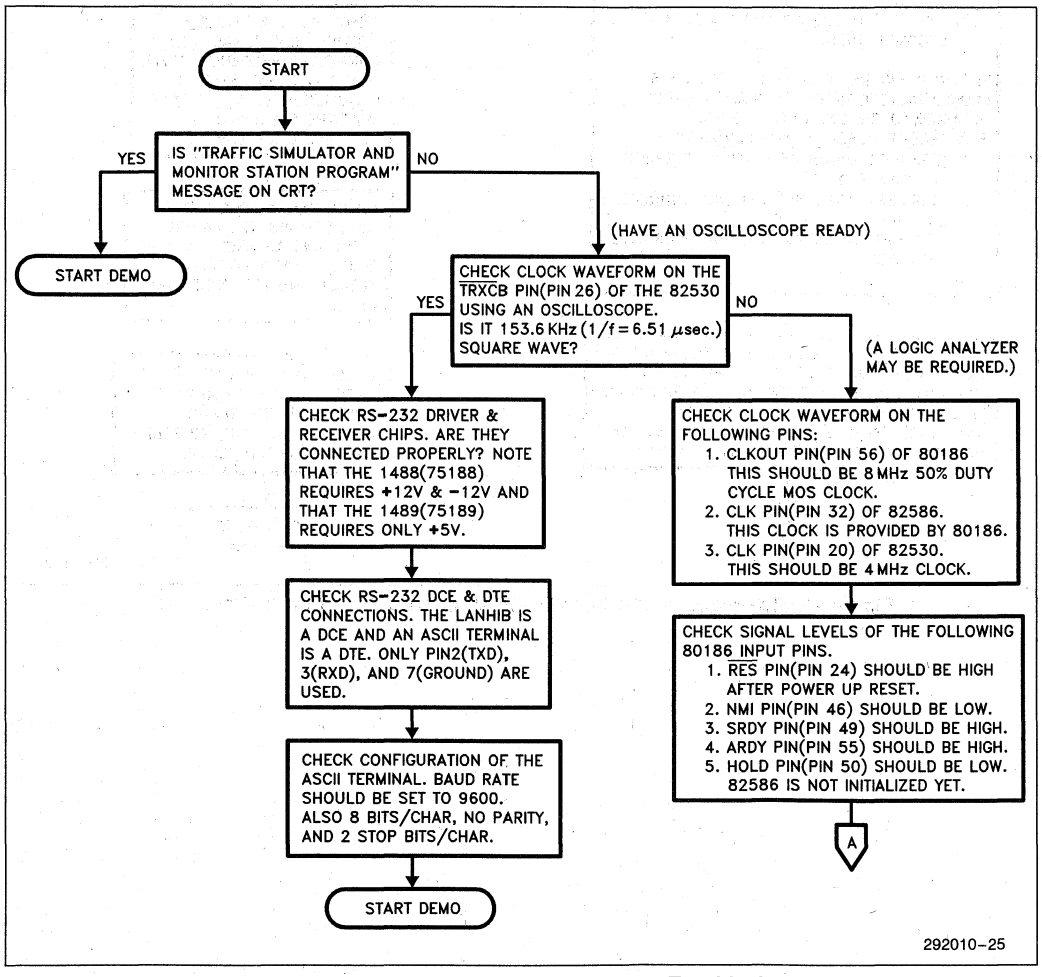


Figure 17. Flowchart for 80186/82586 System Troubleshooting

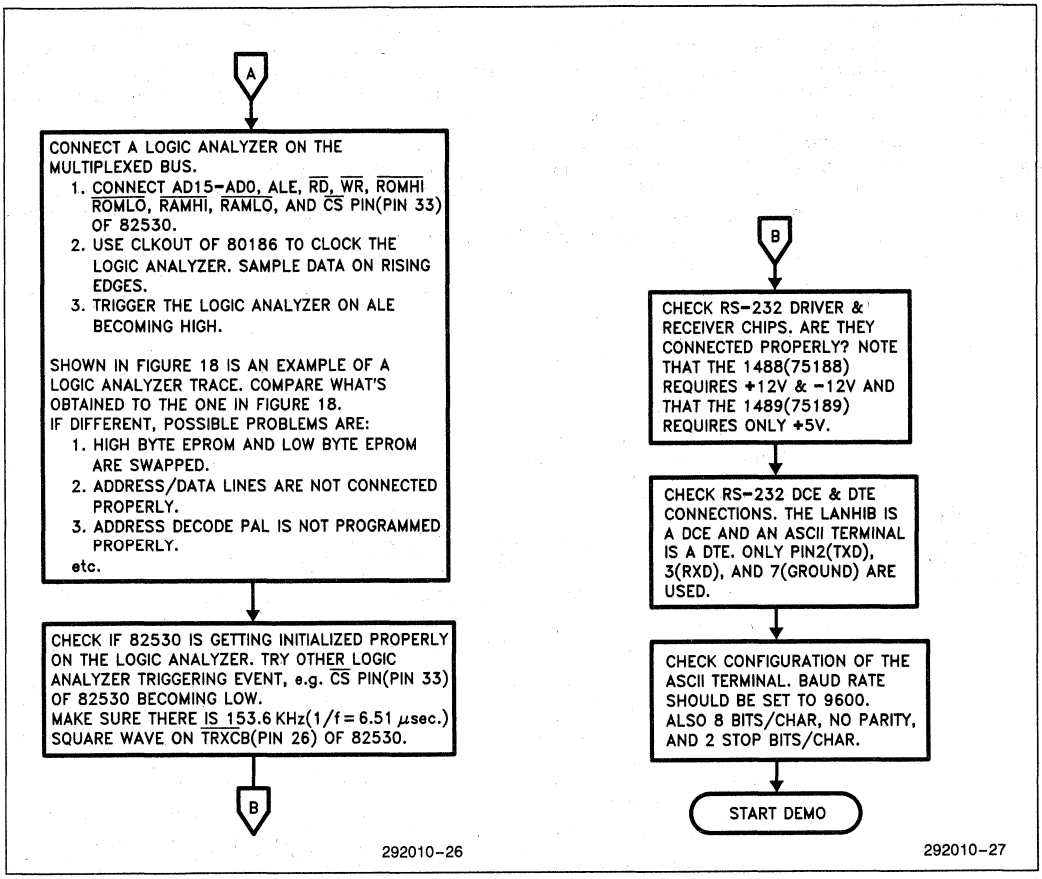


Figure 17. Flowchart for 80186/82586 System Troubleshooting (Continued)

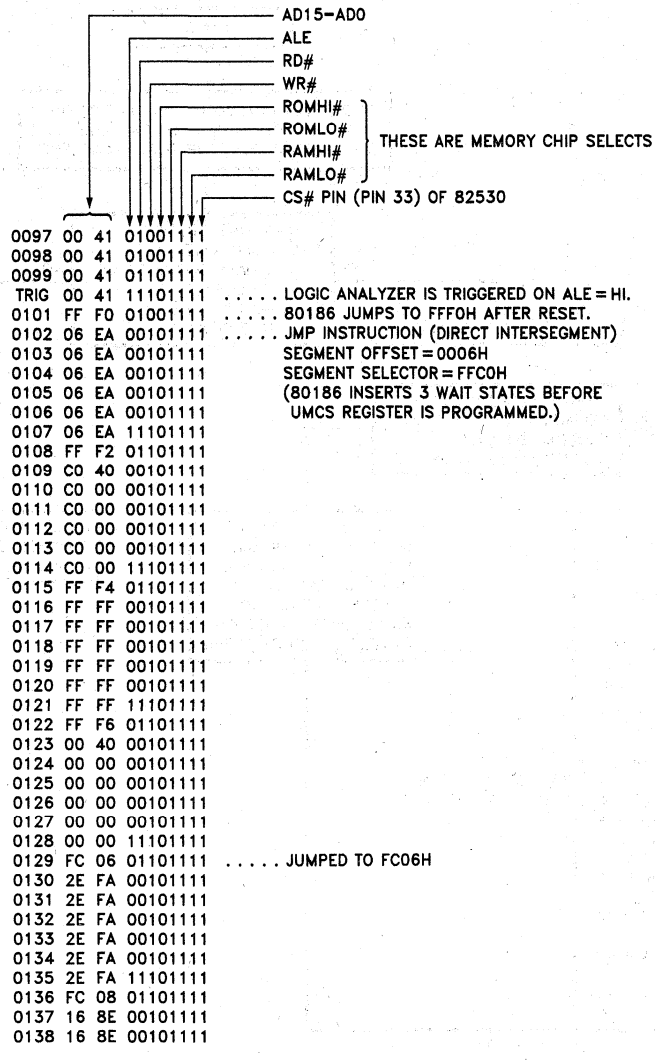


Figure 18. Example of Logic Analyzer Trace

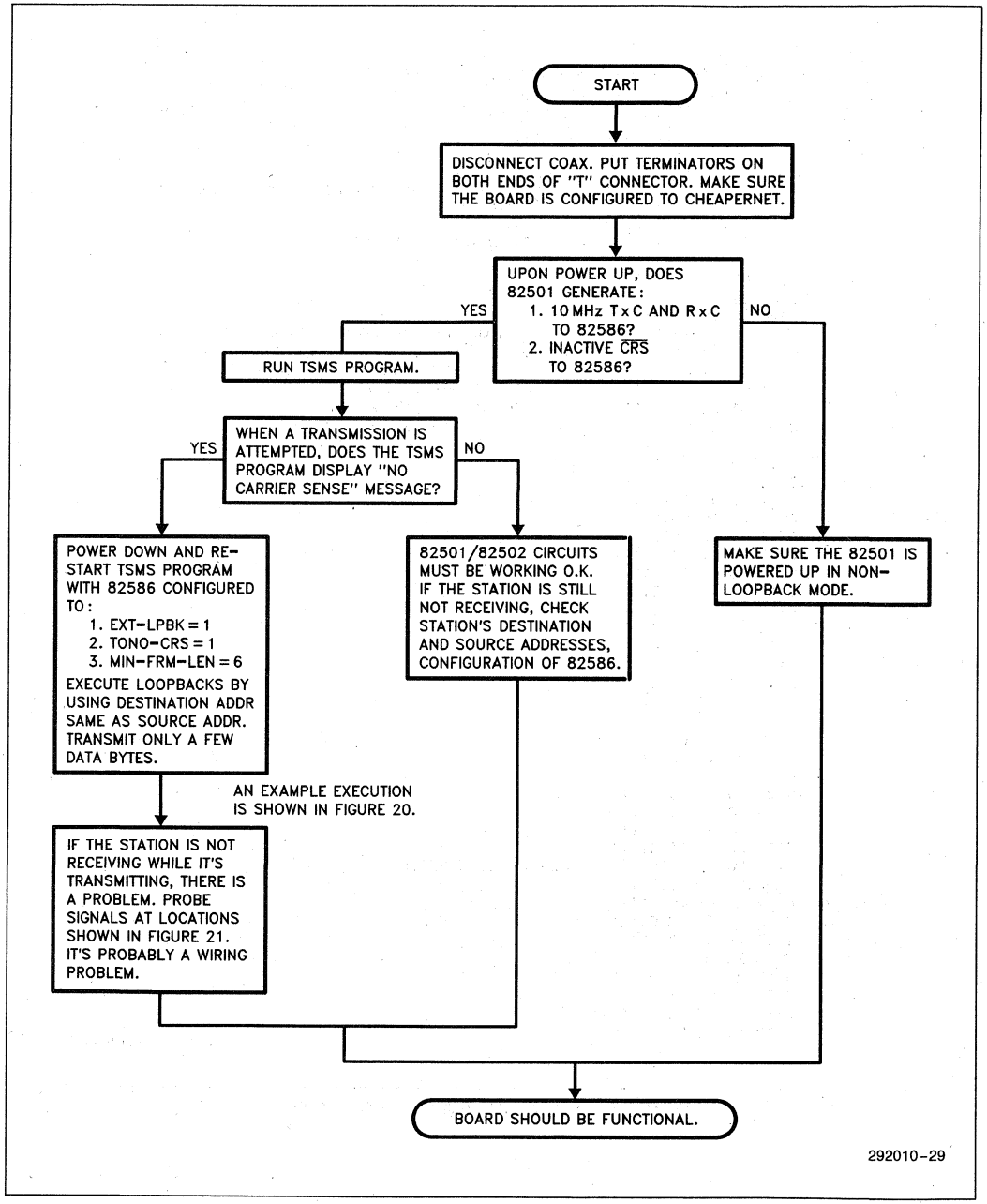


Figure 19. Flowchart for 82501/82502 Circuits Troubleshooting

Traffic Simulator and Monitor Station Program

```
Initialization begun


Configure command is set up for default values.
Do you want to change any bytes? (Y or N) ==> Y
Enter byte number (1 - 11) ==> 4
Enter byte 4 (4H) ==> A6H
Any more bytes? (Y or N) ==> Y
Enter byte number (1 - 11) ==> 9
Enter byte 9 (9H) ==> 08H
Any more bytes? (Y or N) ==> Y
Enter byte number (1 - 11) ==> 11
Enter byte 11 (BH) ==> 6
Any more bytes? (Y or N) ==> N
Configure the 586 with the prewired board address ==> N
Enter this station's address in Hex ==> 000000002200
You can enter up to 8 Multicast Addresses.
Would you like to enter a Multicast Address? (Y or N) ==> N
You entered 0 Multicast Address(es).

Would you like to transmit?
Enter a Y or N ==> Y
Enter a destination address in Hex ==> 000000002200

Enter TYPE ==> 0
How many bytes of transmit data?
Enter a number ==> 2
Transmit Data is continuous numbers (0, 1, 2, 3, ... )
Change any data bytes? (Y or N) ==> N
Enter a delay count ==> 0
Setup a transmit terminal count? (Y or N) ==> N

Destination Address: 00  00  00  00  22  00
Frame Length: 20 bytes
Time Interval between Transmit Frames: 159.4 seconds
Network Percent Load generated by this station: 11.0 %
Transmit Frame Terminal Count: Not Defined

Good enough? (Y or N) ==> Y
```

292010-77

Figure 20. TSMS Initialization for 82501/82502 Circuits Troubleshooting

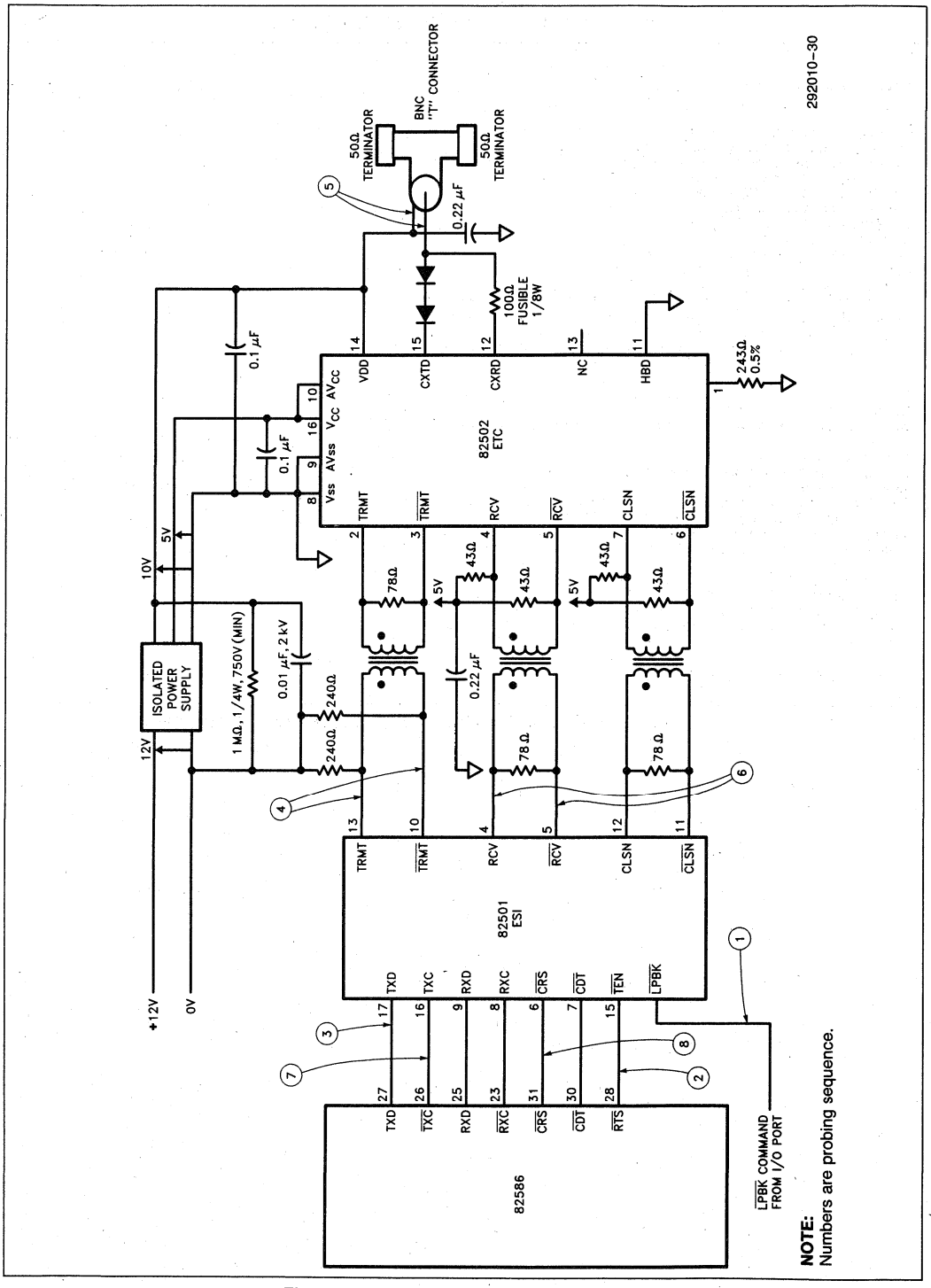
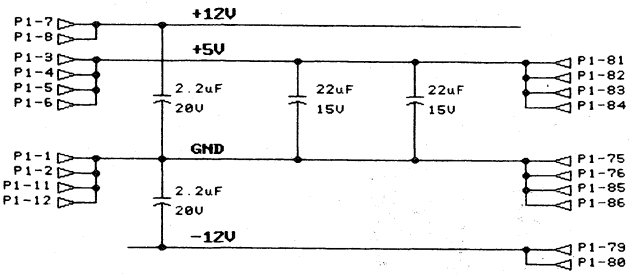


Figure 21. Probing 82501/82502 Circuits

APPENDIX A
LANHIB SCHEMATICS
PARTS LIST
PAL EQUATIONS
DIP SWITCH SETTINGS
WIRE WRAP SERVICES

PARTS LIST

REFERENCES	DESCRIPTION	MFR. PART NO.	MFR. CODE	QTY.
U1	IC	74S32	OBD	1
U2	IC	74LS04	OBD	1
U3, U4	IC	74LS245	OBD	2
U5	IC	74F373	OBD	1
U6,U7	IC	74LS373	OBD	2
U8	IC	80186	INT	1
U9	IC	82586	INT	1
U10	IC	16L8	OBD	1
U11	IC	74LS02	OBD	1
U12, U27	IC	74LS74	OBD	2
U28	IC	74AS74	OBD	1
U13	IC	74LS165	OBD	1
U14	IC	82501	INT	1
U15	Pulse Transformer Pack	PE64102	PE	1
U16	IC	82502	INT	1
U17	DC/DC Converter	PE64369	PE	1
U18, U20	IC, 64K-Bit EPROM	2764-20	INT	2
U26	IC	74AS08	OBD	1
U22, U23	IC, SRAM	HM6264-15	HIT	2
U24	IC, 256-Bit PROM	TBP18S030	TI	1
U25	IC	74AS04	OBD	1
U29	IC	82530	INT	1
U30	IC	1489	OBD	1
U31	IC	1488	OBD	1
U32	IC, 1M-Bit EPROM (Optional)	27210	INT	1
R1-R3, R6 R19, R20	Resistor, 10K ohm, 1/4W, 5%	COML	OBD	6
R4	Resistor, 100K ohm, 1/4W, 5%	COML	OBD	1
R5	Resistor, 2.2K ohm, 1/4W, 5%	COML	OBD	1
R7, R8, R12	Resistor, 78.7 ohm, 1/8W, 1%	COML	OBD	3
R9, R10	Resistor, 240 ohm, 1/4W, 5%	COML	OBD	2
R11	Resistor, 1M ohm, 1/4W, 750Vdc (min), 5%	COML	OBD	1
R13-R16	Resistor, 43.2ohm 1/8W, 1%	COML	OBD	4
R17	Resistor, 100 ohm, Fusible 1/8W, 5%	COML	RCD	1
R18	Resistor, 243 ohm, 1/8W, 0.5%	COML	OBD	1
R21, R22	Resistor, 5K ohm, 1/4W, 5%	COML	OBD	2
RP1	Resistor Pack, 1K ohm, 16 pin	COML	OBD	1
R23-R26	Resistor, 1K ohm, 1/4W, 5%	COML	OBD	4
C1, C2	Capacitor, 20pF, 100V, 5%	COML	OBD	2
C3	Capacitor, 10uF, 20V	COML	OBD	1
C4, C5	Capacitor, 30pF, 100V, 5%	COML	OBD	2
C6	Capacitor, 0.022uF, 50V	COML	OBD	1
C7	Capacitor, 1.0uF, 50V	COML	OBD	1
C11,C12	Capacitor, 0.01uF, 50V	COML	OBD	2
C10	Capacitor, 0.01uF, 2KV	COML	OBD	1
C8, C9	Capacitor, 0.22uF, 50V	COML	OBD	2
CR1	Diode	1N914	OBD	1
CR2, CR3	Diode	1N5282	OBD	2
Y1	Parallel Resonant Crystal, 16M Hz	COML	OBD	1
Y2	Parallel Resonant Crystal, 20M Hz	COML	OBD	1



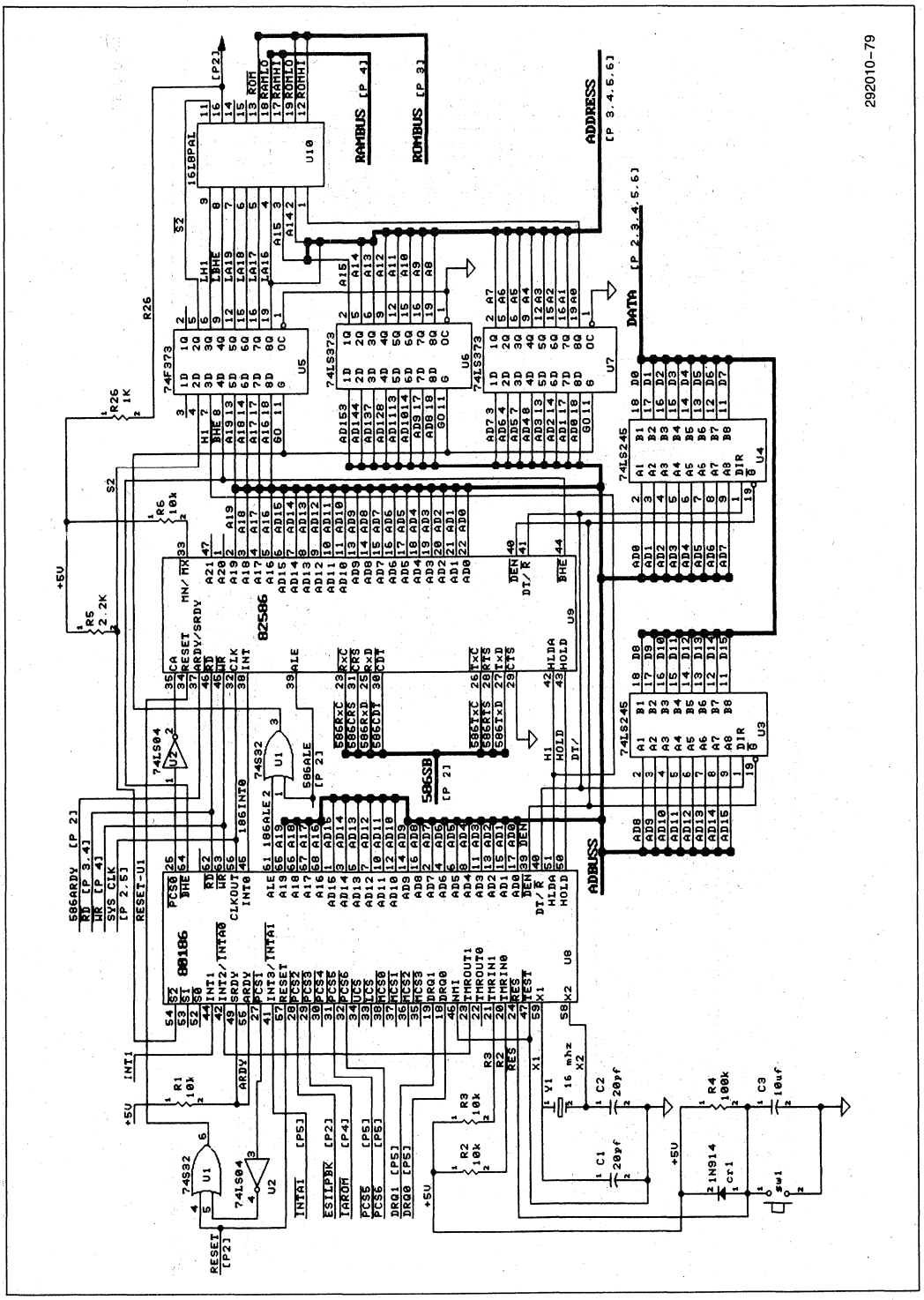
POWER SUPPLY CONNECTIONS

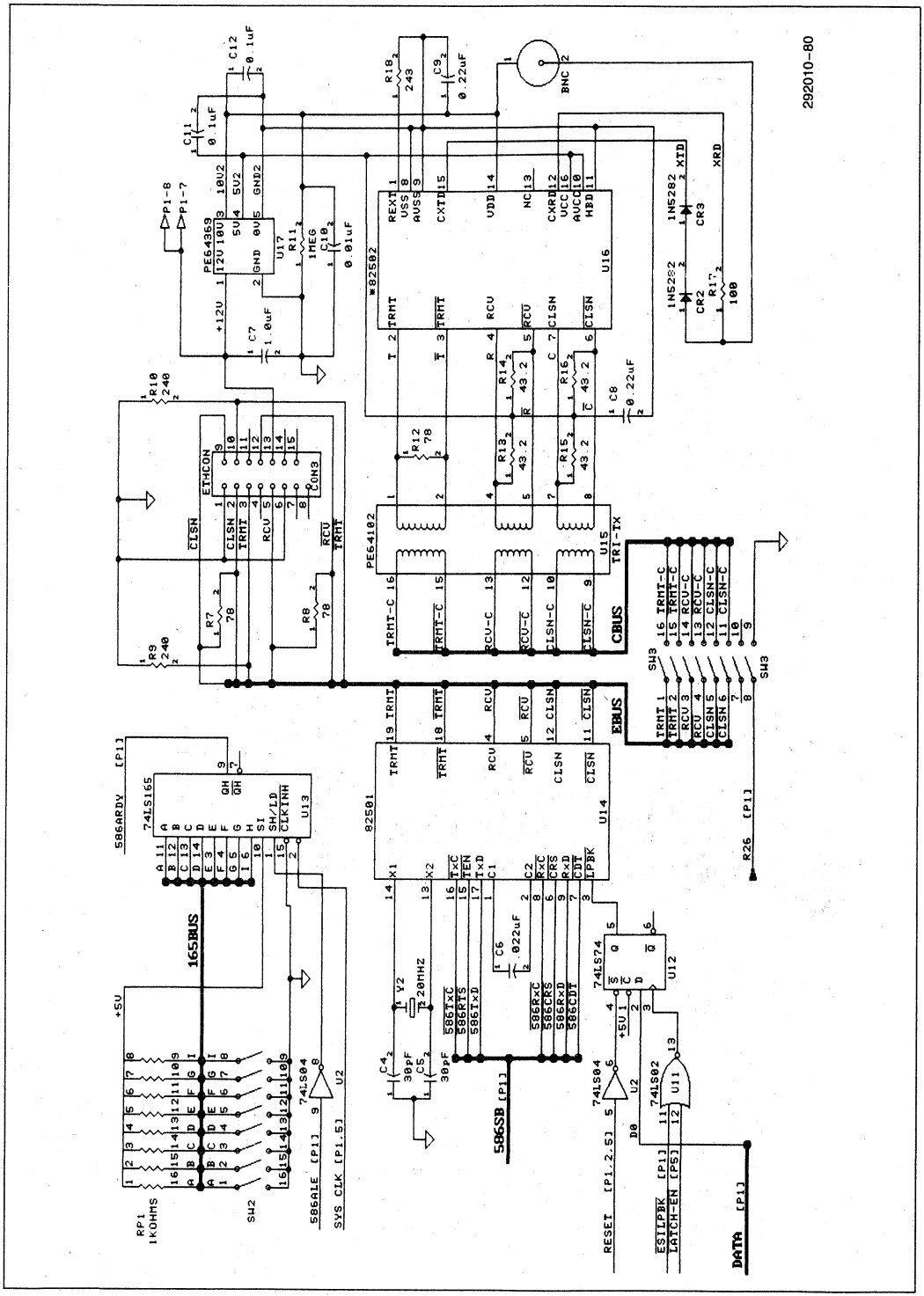
NOTES:

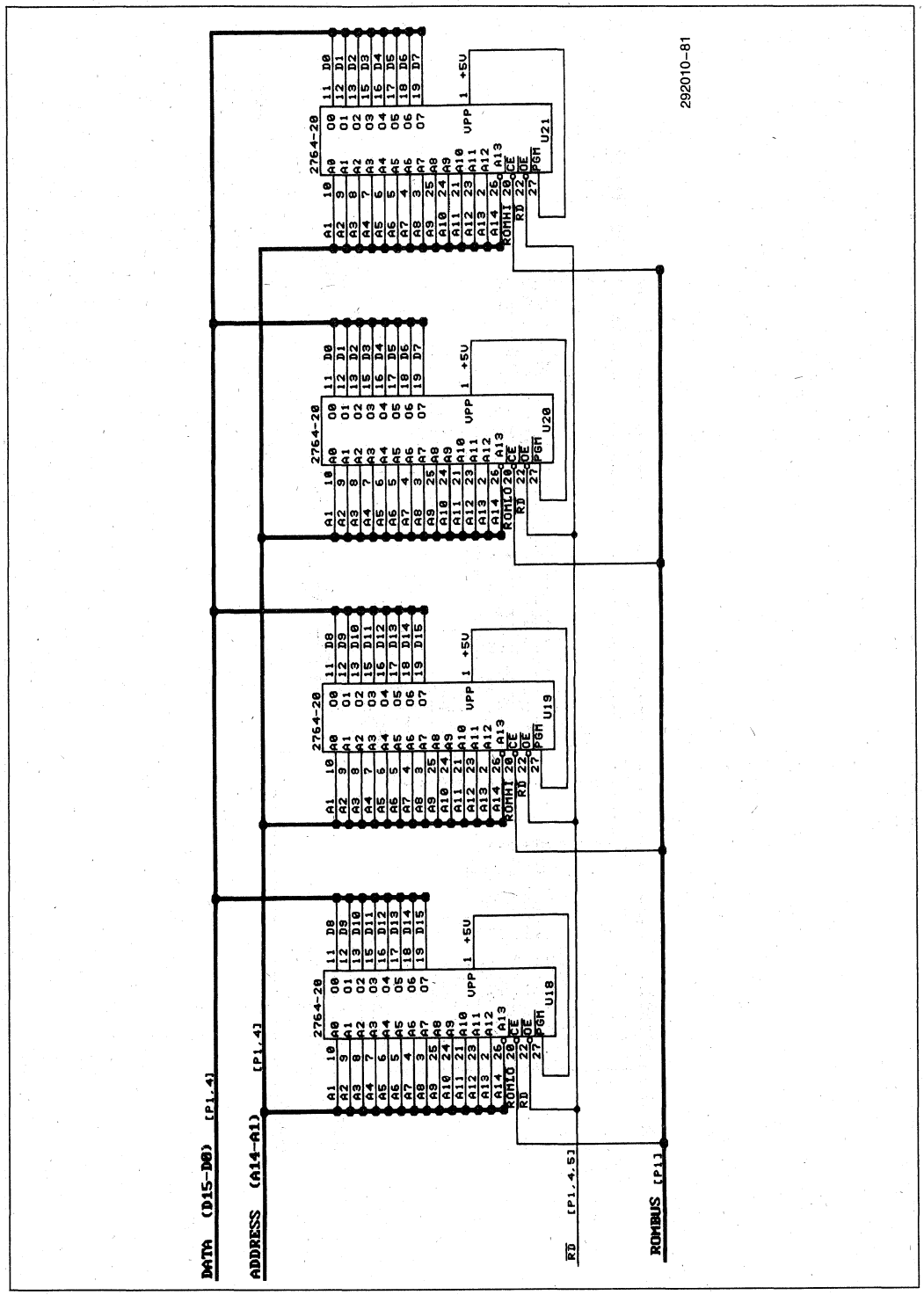
1. THE BOARD REQUIRES +5V, +12V, AND -12V. MULTIBUS POWER PINS FOR THESE VOLTAGES AND GROUND ARE SHOWN ABOVE.
2. EACH IC SHOULD HAVE A 0.1uF CAPACITOR BETWEEN POWER PIN AND GROUND PIN. PARTS LIST DOES NOT INCLUDE DECOUPLING CAPACITORS.
3.

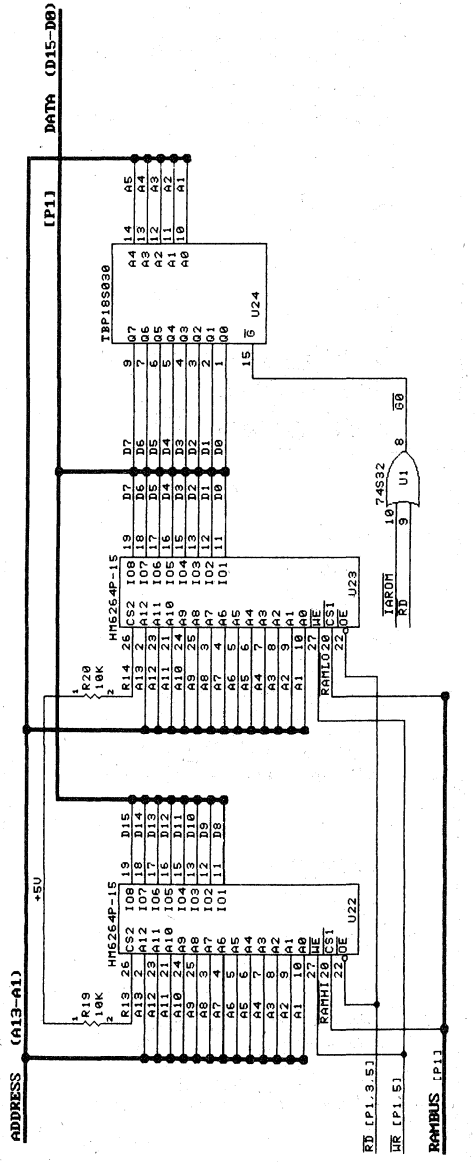
MFR. CODE	MANUFACTURE	LOCATION
INT	INTEL CORPORATION	SANTA CLARA, CA
HIT	HITACH AMERICA LTD.	SAN JOSE, CA
OBD	ORDER BY DESCRIPTION (ANY COMMERCIAL (COML) SOURCE)	
PE	PULSE ENGINEERING	SAN DIEGO, CA
RCD	RCD COMPONENTS INC.	MANCHESTER, NH
TI	TEXAS INSTRUMENTS	DALLAS, TX

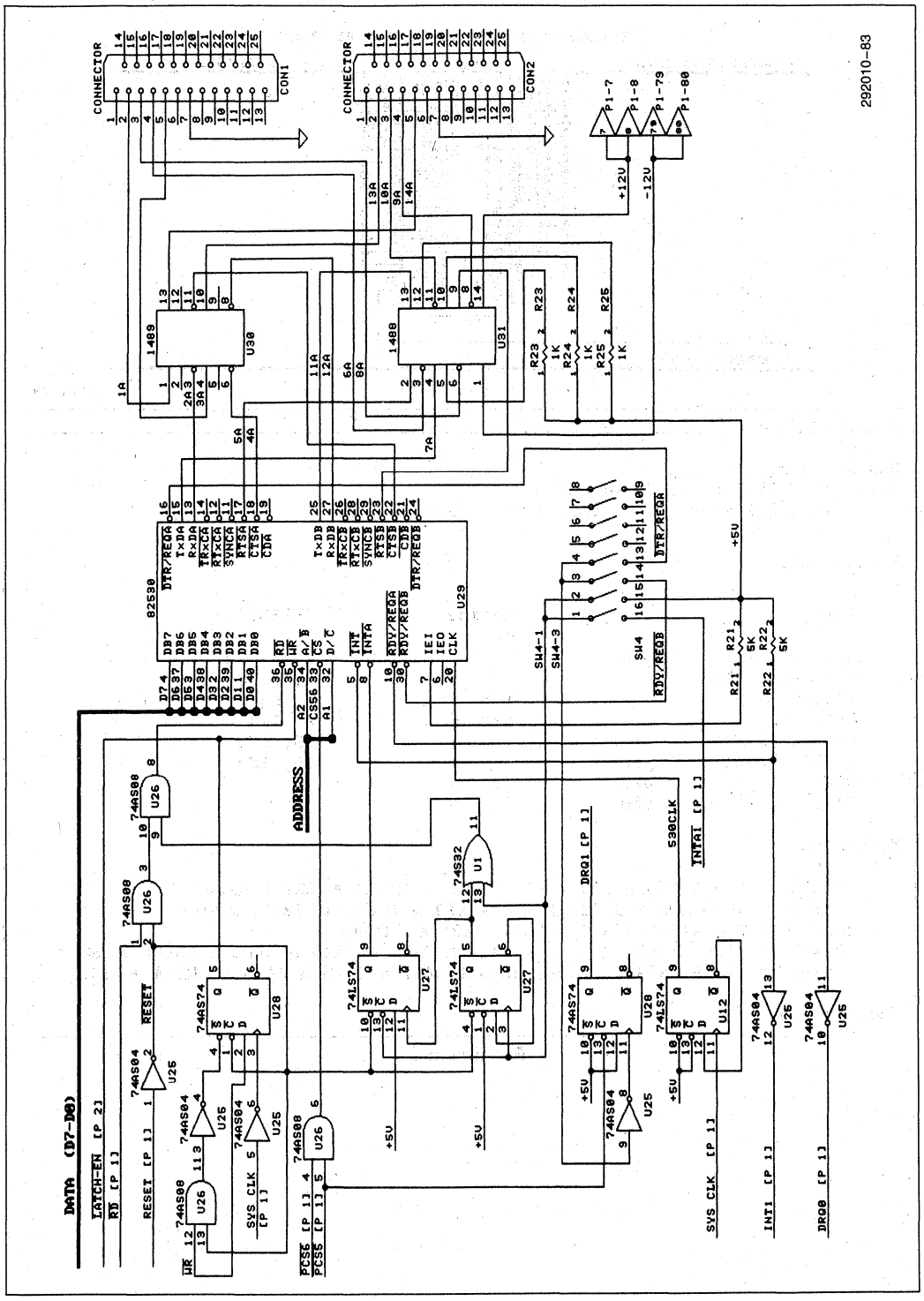
292010-78

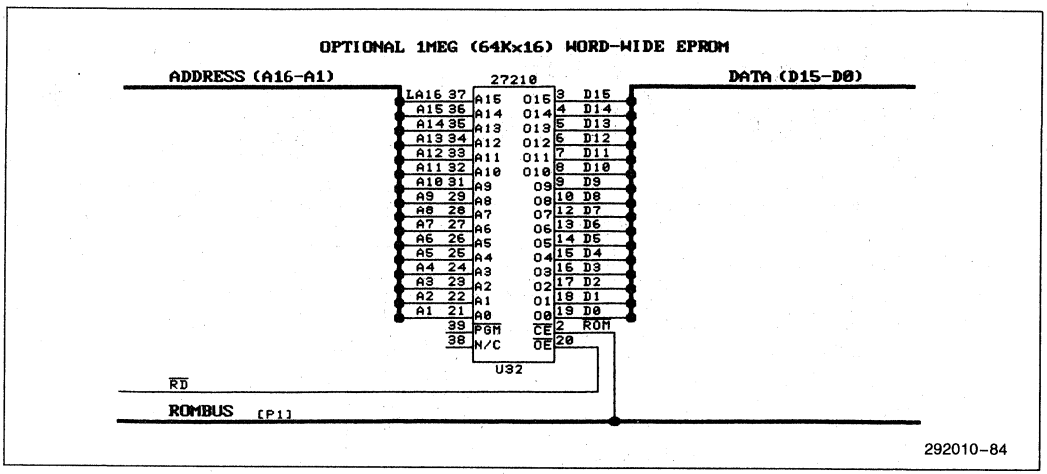

292010-79
82586
80186
82288
82289
74F373
74LS373
74LS245
74S32
74LS04
16L8PAL
RAMBUS [P 4]
ROMBUS [P 3]
ADDRESS [P 3.4.5.6]
DATA [P 2.3.4.5.6]
ADBUSS
+5V
RESET [P2]
INT1 [P5]
ESICPBK [P2]
IAROM [P4]
PCS5 [P5]
PCS6 [P5]
DRQ1 [P5]
DRQ0 [P5]
586ARDY [P 2]
RD [P 3.4]
WR [P 4]
SYS CLK [P 2.5]
RESET-U1
586ALE [P 2]
586SSB [P 2]
R1 10k
R2 10k
R3 10k
R4 100k
R5 2.2K
R6 10k
R26 1K
C1 20pf
C2 20pf
C3 10uf
V1 16 mhz
1N914
sw1


292010-80
82501
82502
PE64102
PE64369
74LS165
74LS74
74LS04
74LS02
ETHCON
CONN3
CBUS
EBUS
TRI-TX
165BUS
586SB [P1]
586RDY [P1]
586ALE [P1]
SYS CLK [P1.5]
RESET [P1.2.5]
ESILPBK [P1]
LATCH-EN [P6]
DATA [P1]
RP1
1KOHMS
SW2
SW3
BNC
CR2
CR3
1N5282
XTD
XRD
20MHZ
0.1uF
0.22uF
0.01uF
1.0uF
022uF
30pF
+5V
+12V


DATA (D15-D8) [P1.4]
ADDRESS (A14-A1) [P1.4]
RD [P1.4.5]
ROMBUS [P1]
2764-28
U18
U19
U20
U21
ROM0
ROM1
+5V
VPP
PGM
CE
OE
292010-81


292010-82
DATA (D15-D0)
[P1]
ADDRESS (A13-A1)
+5U
R19
10K
R20
10K
HM6264P-15
U22
U23
RAMHI
RAMLO
TBP18S030
U24
74S32
U1
IAROM
RD
WR
RAMBUS [P1]
RD [P1.3.5]
WR [P1.5]


CONNECTOR
CON1
CON2
U30
U31
1489
1488
82S90
U29
R23
R24
R25
1K
+12V
-12V
P1-7
P1-8
P1-79
P1-80
SW4
SW4-1
SW4-3
DTR/REQA
RDY/REQB
+5V
5K
R21
R22
ADDRESS
DATA (D7-D0)
LATCH-EN [P 2]
RD [P 1]
RESET [P 1]
WR
PCS6 [P 1]
PCS5 [P 1]
SYS CLK [P 1]
INT1 [P 1]
DRQ0 [P 1]
DRQ1 [P 1]
INTA1 [P 1]
530CLK
74S08
74S04
74S32
74LS74
74AS74
U26
U25
U27
U28
U1
U12
292010-83




```
Module Addr_dec
Title 'LANHIB Address Decode Logic
      Kiyoshi Nishide    Intel Corp.    March, 1986'

"Declarations

    PAL1                            device          'P16L8';

    A0, A14, A15        pin                         1, 2, 3;
    A16, A17, A18       pin                         4, 5, 6;
    A19, BHE                        pin             7, 8;
    HLDA, S2                        pin             9, 11;
    RAML0, RAMHI        pin                         18, 17;
    ROML0, ROMHI        pin                         19, 12;
    ROM                 pin         13;
    R104                pin         16;

Equations

    !ROMHI = A15 & A16 & A17 & A18 & A19 & (HLDA # S2) & R104;
    !ROML0 = !A15 & A16 & A17 & A18 & A19 & (HLDA # S2) & R104;
    !ROM = A17 & A18 & A19 & (HLDA # S2) & !R104;
    !RAMHI = !A14 & !A15 & !A16 & !A17 & !A18 & !A19 & !BHE & (HLDA # S2);
    !RAML0 = !A0 & !A14 & !A15 & !A16 & !A17 & !A18 & !A19 & (HLDA # S2);

End Addr_dec
```

PAL Equations

DIP SWITCH SETTINGS FOR VARIOUS OPERATIONS

"1" indicates ON (Switch is closed).
"0" indicates OFF (Switch is open).
"X" indicates Don't Care.

1. To configure the board to Ethernet or Cheapernet:

	SW3 87654321	Comment
Ethernet	XX000000	
Cheapernet	XX111111	Transceiver Cable should not be connected.

2. To run the TSMS program or the Data Link Driver program:

	SW4 87654321	Comment
TSMS Program or Data Link Driver Program	XXXX0001	TSMS program uses the 82530 in Asynchronous Polling mode. Data Link Driver program uses the 825830 in Asynchronous Polling and Vectored Interrupt modes.

3. To select the 2764-20 EPROMs or 27210 EPROM:

	SW3 87654321
2764-20 EPROMs	0XXXXXXX
27210 EPROM	1XXXXXXX

4. Dip Switch Setting Examples:

	SW3 87654321	SW4 87654321
1) To run the TSMS Program from the 2764-20 EPROMs in Cheapernet Configuration	0X111111	XXXX0010
2) To run the TSMS Program from the 2764-20 EPROMs in Ethernet Configuration	0X000000	XXXX0010
3) To run the TSMS Program or the Data Link Driver program from the 27210 EPROM in Cheapernet Configuration	1X111111	XXXX0001
4) To run the TSMS Program or the Data Link Driver program from the 27210 EPROM in Ethernet Configuration	1X000000	XXXX0001

5. Dip Switch SW2 programs the number of wait states for the 82586 (see Table 3).

COMPANIES OFFERING WIRE WRAP SERVICES

AUGAT
Interconnection Systems Division

40 Perry Avenue
P.O. Box 1037
Attleboro, MA 02703
(617) 222-2202

100935 South Wilcrest Drive
Houston, TX 77099
(713) 495-3100

Automation Delectronics Corporation

1650 Locust Avenue
Bohemia, NY 11716
(516) 567-7007

dataCon, Inc.

Eastern Division
60 Blanchard Road
Burlington, MA 01803
(617) 273-5800

Mid-Western Division
502 Morse Avenue
Schaumburg, IL 60193
(312) 529-7690

Western Division
20150 Sunburst Street
Chatsworth, CA 91311-6280
(818) 700-0600

South-Western Division
1829 Monetary Lane
Carrollton, TX 75006
(214) 245-6161

European Division
In der Klinge 5
D-7100 Heilbronn, West Germany
(01731) 217 12

DATAWRAP

37 Water Street
Wakefield, MA 01880
(617) 938-8911

Elma/EMS
A Division of Sandberg Industries

Berkshire Industrial Park
Bethel, CT 06801
(203) 797-9711

1851 Reynolds Avenue
Irvine, CA 92714
(714) 261-9473

3042 Scott Boulevard
Santa Clara, CA 95054
(408) 970-8874

WRAPEX Corporation

96 Mill Street
Woonsocket, RI 02895
(401) 769-3805

APPENDIX B
SOFTWARE LISTINGS—TSMS PROGRAM AND LANHIB INITIALIZATION ROUTINE

```
/*****************************************************************************/
/*                                                                           */
/*                  Traffic Simulator/Monitor Station Program                */
/*                  for 186/586 High Integration Board and                   */
/*                  iSBC 186/51                                              */
/*                                                                           */
/*                  Ver. 1.0              December 17, 1984                  */
/*                                                                           */
/*                  Kiyoshi Nishide       Intel Corporation                  */
/*                                                                           */
/*****************************************************************************/


        /* This software can be conditionally compiled to work on the iSBC 186/51 or
           on the LANHIB.  If 'set(SBC18651)' is added to the compiler call statement,
           this source program will be compiled for the iSBC18651.  */

1       tsms:

        do;

2   1   declare main label public;

        /* literals */

        $IF SBC18651

        declare lit             literally 'literally',
                true                    lit '1',
                false                   lit '0',
                forever                 lit 'while 1',
                ISCP$LOC$LO             lit '0FFF0H',
                ISCP$LOC$HI             lit '0',
                SCB$BASE$LO             lit '0',
                SCB$BASE$HI             lit '0',
                CA$PORT                 lit '0C8H',
                BOARD$ADDRESS$BASE      lit '0F0H',
                INT$TYPE$586            lit '20H',
                INT$TYPE$TIMER0         lit '30H',
                INT$CTL$TIMER0          lit '0FF32H',
                INT$7                   lit '27H',
                PIC$MASK$130            lit '0E2H',
                PIC$MASK$186            lit '0FF28H',
                ENABLE$586              lit '0FEH',
                ENABLE$586$186          lit '0EEH',
                PIC$EOI$130             lit '0E0H',
                EOI$CMD0$130            lit '60H',
                EOI$CMD4$130            lit '64H',
                PIC$EOI$186             lit '0FF22H',
                EOI$CMD0$186            lit '0',
                PIC$VTR$186             lit '0FF20H',
```

292010-31

Traffic Simulator/Monitor Station Program

```
                TIMER0$CTL              lit 'OFF56H',
                TIMER0$COUNT            lit 'OFF50H',
                MAX$COUNT$A             lit 'OFF52H',
                CA                      lit '0',
                ESI$PORT                lit '0CBH',
                NO$LOOPBACK             lit '8',
                LOOPBACK                lit '0';

        $ELSE

3   1   declare lit             literally 'literally',
                true                    lit '1',
                false                   lit '0',
                forever                 lit 'while 1',
                ISCP$LOC$LO             lit '03FF8H',
                ISCP$LOC$HI             lit '0',
                SCB$BASE$LO             lit '0',
                SCB$BASE$HI             lit '0',
                CA$PORT                 lit '8000H',
                BOARD$ADDRESS$BASE      lit '8180H',
                INT$TYPE$586            lit '12',
                INT$TYPE$TIMER0         lit '8',
                INT$CTL$TIMER0          lit 'OFF32H',
                PIC$MASK$186            lit 'OFF28H',
                ENABLE$586              lit 'OEFH',
                ENABLE$586$186          lit 'OEEH',
                PIC$EOI$186             lit 'OFF22H',
                EOI$CMD0$186            lit '12',
                EOI$CMD4$186            lit '8',
                TIMER0$CTL              lit 'OFF56H',
                TIMER0$COUNT            lit 'OFF50H',
                MAX$COUNT$A             lit 'OFF52H',
                CA                      lit '0',
                ESI$PORT                lit '8100H',
                NO$LOOPBACK             lit '1',
                LOOPBACK                lit '0';

        $ENDIF

        $IF NOT SBC18651

        /* System Configuration Pointer */

4   1   declare scp structure
                (
                sysbus byte,
                unused (5) byte,
                iscp$addr$lo word,
                iscp$addr$hi word
                )
                at (OFFFF6H) data (0, 0, 0, 0, 0, 0, ISCP$LOC$LO, ISCP$LOC$HI);

        $ENDIF


        /* Intermediate System Configuration Pointer */
```

292010-32

Traffic Simulator/Monitor Station Program (Continued)

```
5   1       declare iscp$ptr pointer,
                    iscp based iscp$ptr structure
                    (
                    busy byte,      /* set to 1 by CPU before its first CA to 586,
                                       cleared by 586 after reading info from it */
                    unused byte,    /* unused */
                    scb$o word,     /* offset of system control block */
                    scb$b (2) word /* base of system control block */
                    );

                /* System Control Block */

6   1       declare scb structure
                    (
                    status word,      /* cause(s) of interrupt, CU state, RU state */
                    cmd word,         /* int acks, CU cmd, RESET bit, RU cmd */
                    cbl$offset word, /* offset of first command block in CBL */
                    rpa$offset word, /* offset of first packet descriptor in RPA */
                    crc$errs word,    /* crc error encounterd so far */
                    aln$errs word,    /* alignment errors */
                    rsc$errs word,    /* no resources */
                    ovrn$errs word    /* overrun errors */
                    );

                /* 82586 Action Commands */

                /* NOP */

7   1       declare nop structure
                    (
                    status word,
                    cmd word,
                    link$offset word
                    );


                /* Individual Address Setup */

8   1       declare ia$setup structure
                    (
                    status word,
                    cmd word,
                    link$offset word,
                    ia$address (6) byte
                    );

                /* Configure */

9   1       declare configure structure
                    (
                    status word,
                    cmd word,
                    link$offset word,
                    byte$cnt byte,
                    info (11) byte
                    );
```

292010-33

Traffic Simulator/Monitor Station Program (Continued)

```
                /* Multicast Address Setup */

10    1         declare mc$setup structure
                        (
                        status word,
                        cmd word,
                        link$offset word,
                        mc$byte$count word,
                        mc$address (48) byte  /* only 8 MC addresses are allowed */
                        );


                /* Transmit */

                /* This transmit command is made of one transmit buffer descriptor and one
                   1518 bytes long buffer. */

11    1         declare transmit structure
                        (
                        status word,
                        cmd word,
                        link$offset word,
                        bd$offset word,
                        dest$adr (6) byte,
                        type word
                        );

                /* Transmit Buffer Descriptor */

12    1         declare tbd structure
                        (
                        act$count word,
                        link$offset word,
                        ad0 word,
                        ad1 word
                        );

                /* Transmit Buffer */

13    1         declare tx$buffer (1518) byte;


                /* TDR */

14    1         declare tdr structure
                        (
                        status word,
                        cmd word,
                        link$offset word,
                        result word
                        );

                /* Diagnose */

15    1         declare diagnose structure
                        (
```

292010-34

Traffic Simulator/Monitor Station Program (Continued)

```
                    status word,
                    cmd word,
                    link$offset word
                    );


            /* Dump Status */

16   1      declare dump structure
                    (
                    status word,
                    cmd word,
                    link$offset word,
                    buff$ptr word
                    );

            /* Dump Area */

17   1      declare dump$area (170) byte;


            /* Frame Descriptor */

            /* Receive frame area is made of 5 RFDs, 5 RBDs, and 5 1514 bytes long
               buffers. */

18   1      declare rfd (5) structure
                    (
                    status word,
                    el$s word,
                    link$offset word,
                    bd$offset word,
                    dest$adr (3) word,
                    src$adr (3) word,
                    type word
                    );

            /* Receive Buffer Descriptor */

19   1      declare rbd (5) structure
                    (
                    act$count word,
                    next$bd$link word,
                    ad0 word,
                    ad1 word,
                    size word
                    );

            /* Receive Buffer */

20   1      declare rbuf (5) structure
                    (buffer (1514) byte);


            /* global variables */

21   1      declare status word,              /* UART status */
```

292010-35

Traffic Simulator/Monitor Station Program (Continued)

```
                        actual word,                    /* actual number of chars UART transferred */
                        c$buf (80) byte,                /* buffer for a line of chars */
                        dhex byte,                      /* number base switch */
                        ch byte at (@c$buf),
                        char$count byte,
                        receive$count dword,            /* counter for received frames */
                        count dword,                    /* counter for transmitted frames */
                        preamble word,                  /* preamble length in word */
                        address$length byte,            /* address length in byte */
                        ad$loc byte,                    /* address location control of 82586 */
                        crc byte,                       /* crc length */
                        goback byte,                    /* if set, go back to Continuous Mode */
                        reset byte,                     /* reset flag */
                        delay word,                     /* delay conunt for tranmission delay */
                        cur$cb$offset word,             /* offset of current command block */
                        current$frame byte,             /* offset of frame descriptor just used */
                        no$transmission byte,
                        stop$count dword,               /* transmit terminal frame count */
                        stop byte,
                        mc$count byte,
                        z byte,
                        y byte;

                /* external procedures */

22   1          read: procedure (a, b, c, d, e) external;
23   2          declare (a, c) word,
                        (b, d, e) pointer;
24   2          end read;


25   1          write: procedure (a, b, c, d) external;
26   2          declare (a, c) word,
                        (b, d) pointer;
27   2          end write;


28   1          csts: procedure byte external;
29   2          end csts;


                /* utility procedures */

30   1          offset: procedure (ptr) word;

                /* This procedure takes a pointer variable (selector:offset), caluculates an
                   absolute address, subtracts the 82586 SCB offset from the absolute address,
                   and then returns the result as an offset value for the 82586. */

31   2          declare (ptr, ptr$loc) pointer,
                        base586 dword,
                        w based ptr$loc (2) word;

32   2             ptr$loc = @ptr;

                   /* 82586 SCB Base Address (20-bit wide in this 186 based system) */
```

292010-36

Traffic Simulator/Monitor Station Program (Continued)

```
33   2          base586 = (shl(double (iscp.scb$b(1)), 16) and 000F0000H) + iscp.scb$b(0);
34   2          return low((shl(double (w(1)), 4) + w(0)) - base586);

35   2      end offset;


36   1      writeln: procedure (a, b, c, d);

            /* This procedure writes a line and put a CR/LF at the end. */

37   2      declare (a, c) word,
                     (b, d) pointer;

38   2          call write(a, b, c, d);
39   2          call write(0, @(0DH, 0AH), 2, @status);

40   2      end writeln;


41   1      cr$lf: procedure;

            /* This procedure writes a CR/LF. */

42   2          call write (0, @(0DH, 0AH), 2, @status);

43   2      end cr$lf;


44   1      pause: procedure;

            /* This procedure breaks a program flow, and waits for a char to be typed. */

45   2          call write(0, @(0DH, 0AH, 'Hit <CR> to countinue'), 23, @status);
46   2          call read(1, @c$buf, 80, @actual, @status);
47   2          call cr$lf;

48   2      end pause;


49   1      skip: procedure byte;

            /* This procedure skips all leading blank characters and returns the first
               non-blank character.  */

50   2      declare i byte;

51   2          i = 0;
52   2          do while (c$buf(i) = ' ');
53   3              i = i + 1;
54   3          end;
55   2          return i;

56   2      end skip;


57   1      read$char: procedure byte;
```

292010-37

Traffic Simulator/Monitor Station Program (Continued)

```
                /* This procedure reads a line and returns ther first non-blank character.  */

58    2         declare i word;

59    2             call read(1, @c$buf, 80, @actual, @status);
60    2             i = skip;
61    2             return(c$buf(i));

62    2         end read$char;


63    1         read$bit: procedure byte;

                /* This procedure reads a bit and returns the value.  */

64    2         declare b byte;

65    2             do forever;
66    3                 b = read$char;
67    3                     if b = '1' then return 1;
69    3                 else
70    3                     if b = '0' then return 0;
71    3                 else
                          call write(0, @(' Enter a 0 or 1 ==> '), 20, @status);
72    3             end;

73    2         end read$bit;


74    1         yes: procedure byte;

                /* This procedure reads a character and determines if it is a Y(y) or N(n).   */

75    2         declare b byte;

76    2             do forever;
77    3                 b = read$char;
78    3                 if (b = 'Y') or (b = 'y') then return true;
80    3                 else
81    3                     if (b = 'N') or (b = 'n') then return false;
82    3                 else
                          call write(0, @(0DH, 0AH, ' Enter a Y or N ==> '), 22, @status);
83    3             end;

84    2         end yes;


85    1         char$to$int: procedure (c) byte;

                /* This procedure converts a byte of ASCII integer to an integer.   */

86    2         declare c byte;

87    2             if ('0' <= c) and (c <= '9') then return (c - 30H);
89    2             else
90    2                 if('A' <= c) and (c <= 'F') then return (c - 37H);
91    2                 else
```

292010-38

Traffic Simulator/Monitor Station Program (Continued)

```
 92   2                  if ('a' <= c) and (c <= 'f') then return (c - 57H);
 93   2                  else return OFFH;

 94   2      end char$to$int;


 95   1      int$to$asci: procedure (value, base, ld, bufadr, width);

              /* This procedure converts an interger < OFFFFFFFFH to an array of ASCII
                 codes.
                 Input variables are:  value = integer to be converted,
                                       base = number base to be used for conversion,
                                       ld = leading character to be filled in,
                                       bufadr = buffer address of the array,
                                       width = size of array.  */

 96   2      declare value dword,
                      bufadr pointer,
                      (i, j, base, ld, width) byte,
                      chars based bufadr (1) byte;

 97   2          do i = 1 to width;
 98   3              j = value mod base;
 99   3              if j < 10 then chars (width - i) = j + 30H;
101   3              else chars (width - i) = j + 37H;
102   3              value = value / base;
103   3          end;
104   2          i = 0;
105   2          do while chars (i) = '0' and i < width - 1;
106   3              chars (i) = ld;
107   3              i = i + 1;
108   3          end;
109   2          char$count = width - i;

110   2      end int$to$asci;


111   1      out$word: procedure (w$ptr, distance);

              /* An integer at (selector of w$ptr):(offset of w$ptr + distance) is printed
                 as a 4 digit hexadecimal number.   */

112   2      declare chars(4) byte,
                      w$ptr pointer,
                      distance byte,
                      w based w$ptr (1) word;

113   2          call int$to$asci(w(distance), 16, '0', @chars(0), 4);
114   2          call write(0, @chars(0), 4, @status);

115   2      end out$word;


116   1      write$int: procedure(dw, t);

              /* An integer (dw) is printed in hexadecimal (t = 1) or in decimal (t = 0).  */
```

292010-39

Traffic Simulator/Monitor Station Program (Continued)

```
117   2       declare dw dword,
                      chars (10) byte,
                      t byte;

118   2           if t then
119   2           do;
120   3               call int$to$asci(dw, 16, 0, @chars(0), 8);
121   3               call write(0, @chars(8-char$count), char$count, @status);
122   3           end;
123   2           else
              do;
124   3               call int$to$asci(dw, 10, 0, @chars(0), 10);
125   3               call write(0, @chars(10-char$count), char$count, @status);
126   3           end;

127   2       end write$int;


128   1       out$dec$hex: procedure(dw);

              /* This procedure prints an integer in decimal and hexadecimal. */

129   2       declare dw dword;

130   2           call write$int(dw, 0);
131   2           call write(0, @(' ('), 2, @status);
132   2           call write$int(dw, 1);
133   2           call write(0, @('H)'), 2, @status);

134   2       end out$dec$hex;


135   1       write$offset: procedure(w$ptr);

              /* This procedure takes a pointer variable, converts it to a 82586 type offset,
                 and prints it in hexadecimal.  */

136   2       declare w$ptr pointer,
                      w word;

137   2           call write(0, @(' at '), 4, @status);
138   2           w = offset(w$ptr);
139   2           call out$word(@w, 0);
140   2           call write(0, @('  '), 2, @status);

141   2       end write$offset;


142   1       write$address: procedure (ptr);

              /* This procedure takes a pointer variable and prints it in the
                 'selector:offset' format.  */

143   2       declare (ptr, ptr$loc)  pointer,
                      w based ptr$loc (2) word;

144   2           ptr$loc = @ptr;
```

292010-40

Traffic Simulator/Monitor Station Program (Continued)

```
145   2           call out$word(@w(1), 0);
146   2           call write(0, @(': '), 1, @status);
147   2           call out$word(@w(0), 0);
148   2           call write(0, @(' '), 1, @status);

149   2       end write$address;


150   1       print$wds: procedure(w$ptr, no$words);

                /* This procedure prints no$words number of words starting at w$ptr.   */

151   2       declare w$ptr pointer,
                        (i, no$words) byte;

152   2           if no$words <> 0 then
153   2           do;
154   3               call cr$lf;
155   3               do i = 0 to no$words - 1;
156   4                   call out$word(w$ptr, i);
157   4                   if i = 0 then
158   4                       call write$offset(w$ptr);
159   4                   call cr$lf;
160   4               end;
161   3           end;

162   2       end print$wds;


163   1       print$str: procedure (str$ptr, len);

                /* This procedure prints len number of bytes starting at str$ptr. */

164   2       declare (len, i) byte,
                        chars (2) byte,
                        str$ptr pointer,
                        str based str$ptr (1) byte;

165   2           if len <> 0 then
166   2           do i = 0 to (len - 1);
167   3               call int$to$asci(str(i), 16, '0', @chars(0), 2);
168   3               call write(0, @chars(0), 2, @status);
169   3               call write(0, @('  '), 2, @status);
170   3           end;
171   2           call cr$lf;

172   2       end print$str;


173   1       print$buff: procedure (ptr, cnt);

                /* This procedure prints cnt number of buffer contents starting at ptr. */

174   2       declare ptr pointer,
                        bt based ptr (1) byte,
                        (i, j) byte,
                        cnt word;
```

292010-41

Traffic Simulator/Monitor Station Program (Continued)

```
175   2          if cnt > 16 then
176   2          do;
177   3              i = shr(cnt, 4) - 1;
178   3              do j = 0 to i;
179   4                  call write$address(@bt(16*j));
180   4                  call print$str(@bt(16*j), 16);
181   4                  if (j = 20) or (j = 40) or (j = 60) or (j = 80) then
182   4                  call pause;
183   4              end;
184   3              i = i + 1;
185   3              if cnt-16*i <> 0 then call write$address(@bt(16*i));
187   3              call print$str(@bt(16*i), cnt-16*i);
188   3          end;
189   2          else
              do;
190   3              call write$address(@bt(0));
191   3              call print$str(@bt(0), cnt);
192   3          end;

193   2      end print$buff;



194   1      read$int: procedure (limit) dword;

              /* This procedure reads integer characters and forms an integer.  If the
                 integer is bigger than 'limit' or an overflow error is encounterred, then
                 an error message is printed.  */

195   2      declare (wd, wh, limit) dword,
                      (i, j, k, done, hex, dover, hover) byte;

196   2          do forever;
197   3              call read(1, @c$buf, 80, @actual, @status);
198   3              i, k = skip;
199   3              hex, done, dover, hover = false;
200   3              wd, wh = 0;
201   3              j = char$to$int(c$buf(i));
202   3              do while j <= 15;
203   4                  if j > 9 then hex = true;
205   4                  if not dover then
206   4                      if wd > 429496729 then dover = true;
208   4                      else if (wd = 429496729) and (j > 5) then dover = true;
210   4                  wd = wd*10 + j;
211   4                  if not hover then if wh > OFFFFFFFH then hover = true;
214   4                  wh = wh*16 + j;
215   4                  i = i + 1;
216   4                  j = char$to$int(c$buf(i));
217   4              end;
218   3              if ((c$buf(i) <> 'H') and (c$buf(i) <> 'h') and (c$buf(i) <> ODH) and
                           (c$buf(i) <> OAH) and (c$buf(i) <> ' ')) or (i = k) then
219   3              call writeln(0, @(ODH, OAH, ' Illegal character'), 20, @status);
220   3              else
                      do;
221   4                  if (c$buf(i) = 'H') or (c$buf(i) = 'h') then hex = true;
223   4                  if hex then
```

292010-42

Traffic Simulator/Monitor Station Program (Continued)

```
224   4                 do;
225   5                     if not hover and (wh <= limit) then return wh;
227   5                 end;
228   4                 else
229   4                     if not dover and (wd <= limit) then return wd;
230   4                 call writeln(0, @(0DH, 0AH, ' The number is too big. '), 25,
                                                                          @status);
231   4                 call write(0, @(' It has to be less than or equal to '), 36,
                                                                          @status);
232   4                 call out$dec$hex(limit);
233   4                 call writeln(0, @('.'), 1, @status);
234   4             end;
235   3             call write(0, @(' Enter a number ==> '), 20, @status);
236   3         end;

237   2     end read$int;


238   1     put$address: procedure(where);

            /* This procedure puts an address typed in hexadecimal to the specified
               location 'where'. */

239   2     declare where pointer,
                    (i, j, m, err) byte,
                    addr based where (1) byte;

240   2         do forever;
241   3             err = false;
242   3             call read(1, @c$buf, 80, @actual, @status);
243   3             i = skip;
244   3             m = address$length;
245   3             do while (m <> 0) and not err;
246   4                 j = char$to$int(c$buf(i));
247   4                 if j = 0FFH then err = true;
249   4                 else
                        do;
250   5                     addr(m-1) = shl(j, 4);
251   5                     j = char$to$int(c$buf(i+1));
252   5                     if j = 0FFH then err = true;
254   5                     else addr(m-1) = addr(m-1) or j;
255   5                 end;
256   4                 i = i + 2;
257   4                 m = m - 1;
258   4             end;
259   3             if not err then
260   3             do;
261   4                 m = c$buf(i);
262   4                 if (m = 0DH) or (m = 0AH) or (m = 'h') or (m = 'H') or (m = ' ')
263   4                     then return;
264   4             end;
265   3             call writeln(0, @(0DH, 0AH, ' Illegal character'), 20, @status);
266   3             call write(0, @(' Enter an address in Hex ==> '), 29, @status);
267   3         end;

268   2     end put$address;
```

292010-43

Traffic Simulator/Monitor Station Program (Continued)

```
269   1     percent: procedure;

            /* This procedure calculates and prints a network percent load generated
               by this station.  The equation used in this procedure was obtained
               from actual measurements. */

270   2     declare i word,
                    (j, k) dword,
                    pcent (3) byte;

271   2         j = (tbd.act$count and 3FFFH)*8;
272   2         if not ad$loc then k = (2*address$length + 2 + crc + preamble)*8;
274   2         else k = (crc + preamble)*8;
275   2         if delay <> 0 then

            $IF NOT SBC18651

276   2             i = low((1000*(j + k))/(1805 + k + 5*double(delay) + j));

            $ELSE

                  i = low((1000*(j + k))/(2021 + k + 5*double(delay) + j));

            $ENDIF

277   2             else

            $IF NOT SBC18651

                  i = low((1000*(j + k))/(1810 + k + j));

            $ELSE

                  i = low((1000*(j + k))/(2026 + k + j));

            $ENDIF

278   2             call int$to$asci(i, 10, 0, @pcent(0), 3);
279   2             call write(0, @pcent(0), 2, @status);
280   2             call write(0, @('. '), 1, @status);
281   2             call write(0, @pcent(2), 1, @status);
282   2             call writeln(0, @(' %'), 2, @status);

283   2     end percent;


284   1     print$network$addr: procedure (ptr);

            /* This station's address is printed with its least significant bit
               in the most right position.  */

285   2     declare ptr pointer,
                    addr based ptr (1) byte,
                    char (6) byte,
                    i byte;
```

292010-44

Traffic Simulator/Monitor Station Program (Continued)

```
286   2           do i = 1 to address$length;
287   3               char(i-1) = addr(address$length-i);
288   3           end;
289   2           call print$str(@char(0), address$length);

290   2       end print$network$addr;



291   1       print$parameters: procedure;

              /* This procedure prints transmission parameters. */

292   2       declare w dword,
                      stgs (6) byte;

293   2           call write(0, @(' Destination Address: '), 22, @status);
294   2           if not ad$loc then
295   2               call print$network$addr(@transmit.dest$adr(0));
296   2           else
                      call print$network$addr(@tx$buffer(0));
297   2           if not ad$loc then
298   2           w =  (tbd.act$count and 3FFFH) + address$length * 2 + 2 + crc;
299   2           else w = (tbd.act$count and 3FFFH) + crc;
300   2           call write(0, @(' Frame Length: '), 15, @status);
301   2           call write$int(w, 0);
302   2           call writeln(0, @(' bytes'), 6, @status);
303   2           call write(0, @(' Time Interval between Transmit Frames: '), 40, @status);
304   2           if delay <> 0 then
305   2           do;
              $IF NOT SBC18651

306   3                 w = 1810 + (double(delay) - 1) * 5;

              $ELSE

                      w = 2026 + (double(delay) - 1) * 5;

              $ENDIF

307   3                 call int$to$asci(w, 10, 0, @stgs, 6);
308   3                 if w >= 10000 then
309   3                 do;
310   4                     call write(0, @stgs(0), 2, @status);
311   4                     call write(0, @('.'), 1, @status);
312   4                     call write(0, @stgs(2), 2, @status);
313   4                     call writeln(0, @(' miliseconds'), 12, @status);
314   4                 end;
315   3                 else
                          do;
316   4                     call write(0, @stgs(0), 5, @status);
317   4                     call write(0, @('.'), 1, @status);
318   4                     call write(0, @stgs(5), 1, @status);
319   4                     call writeln(0, @(' microseconds'), 13, @status);
320   4                 end;
321   3           end;
322   2           else
```

292010-45

Traffic Simulator/Monitor Station Program (Continued)

```
            $IF NOT SBC18651

                call writeln(0, @(' 159.4 microseconds'), 19, @status);

            $ELSE

                call writeln(0, @(' 172.8 microseconds'), 19, @status);

            $ENDIF

323   2         call write(0, @(' Network Percent Load generated by this station: '), 49,
                                                                              @status);
324   2         call percent;
325   2         call write(0, @(' Transmit Frame Terminal Count: '), 32, @status);
326   2         if stop then call write$int(stop$count, dhex);
328   2             else call write(0, @('Not Defined'), 11, @status);
329   2         call cr$lf;

330   2     end print$parameters;


331   1     print$scb: procedure;

            /* prints the SCB */

332   2         call writeln(0, @(0DH, 0AH, '*** System Control Block ***'), 30, @status);
333   2         call print$wds(@scb.status, 8);

334   2     end print$scb;



335   1     wait$scb: procedure;

            /* This procedure provids a wait loop for the SCB command word to
               become cleared.  */

336   2     declare i word;

337   2         i = 0;
338   2         do while (scb.cmd <> 0) and (i < 8000H);
339   3             i = i + 1;
340   3         end;
341   2         if scb.cmd <> 0 then
342   2             do;
343   3                 call write(0, @(0DH, 0AH, ' Wait Time = '), 15, @status);
344   3                 call write$int(i, 0);
345   3                 call cr$lf;
346   3             end;

347   2     end wait$scb;


348   1     start$timer0: procedure;

            /* 80186 timer0 is started. */
```

292010-46

Traffic Simulator/Monitor Station Program (Continued)

```
349   2           output(TIMER0$CTL) = 0E000H;
350   2       end start$timer0;


351   1       isr: procedure interrupt INT$TYPE$586 reentrant;
                /* interrupt service routine for 82586 interrupt */
352   2       declare i byte;
                    /* Enable 82586 Interrupt */
                $IF SBC18651
                    output (PIC$EOI$130) = EOI$CMD0$130;
                    enable;
                $ELSE
353   2             output (PIC$EOI$186) = EOI$CMD0$186;
354   2             enable;
                $ENDIF
                    /* Frame Received Interrupt has the highest priority */
355   2             if (scb.status and 4000H) = 4000H then
356   2             do;
357   3                 disable;
358   3                 scb.cmd = 4000H;
359   3                 output (CA$PORT) = CA;
360   3                 call wait$scb;
361   3                 if rfd(current$frame).status = 0A000H then
362   3                 do;
363   4                     receive$count = receive$count + 1;
364   4                     current$frame = current$frame + 1;
365   4                     if current$frame = 5 then current$frame = 0;
367   4                 end;
368   3                 return;
369   3             end;

370   2             if (scb.status and 2000H) = 2000H then
371   2             do;
372   3                 disable;
373   3                 scb.cmd = 2000H;
374   3                 output(CA$PORT) = CA;
375   3                 call wait$scb;
376   3                 enable;
377   3                 if (transmit.status and 0A000H) = 0A000H then
378   3                 do;
379   4                     count = count + 1;
380   4                     if (stop and (count = stop$count)) then return;
382   4                     else
                            do;
```

292010-47

Traffic Simulator/Monitor Station Program (Continued)

```
383   5                     transmit.status = 0;
384   5                     if delay = 0 then
385   5                     do;
386   6                         disable;
387   6                         scb.cmd = 0100H;
388   6                         output(CA$PORT) = CA;
389   6                         call wait$scb;
390   6                         return;
391   6                     end;
392   5                     else
                            do;
393   6                         call start$timer0;
394   6                         return;
395   6                     end;
396   5                 end;
397   4             end;
398   3             if (transmit.status and 0020H) = 0020H then
399   3             do;
400   4                 transmit.status = 0;
401   4                 disable;
402   4                 scb.cmd = 0100H;
403   4                 output (CA$PORT) = CA;
404   4                 call wait$scb;
405   4                 return;
406   4             end;
407   3             if (transmit.status and 0400H) = 0400H then
408   3             do;
409   4                 call write(0, @(0DH, ' No Carrier Sense!', 0DH), 20, @status);
410   4                 transmit.status = 0;
411   4                 disable;
412   4                 scb.cmd = 0100H;
413   4                 output (CA$PORT) = CA;
414   4                 call wait$scb;
415   4                 return;
416   4             end;
417   3             if (transmit.status and 0200H) = 0200H then
418   3             do;
419   4                 call write(0, @(0DH, ' Lost Clear to Send!', 0DH), 22, @status);
420   4                 transmit.status = 0;
421   4                 disable;
422   4                 scb.cmd = 0100H;
423   4                 output (CA$PORT) = CA;
424   4                 call wait$scb;
425   4                 return;
426   4             end;
427   3             if (transmit.status and 0100H) = 0100H then
428   3             do;
429   4                 call write(0, @(0DH, ' DMA Underrun!', 0DH), 16, @status);
430   4                 transmit.status = 0;
431   4                 disable;
432   4                 scb.cmd = 0100H;
433   4                 output (CA$PORT) = CA;
434   4                 call wait$scb;
435   4                 return;
436   4             end;
437   3         end;
438   2         if (scb.status and 8000H) = 8000H then
```

292010-48

Traffic Simulator/Monitor Station Program (Continued)

```
439   2           do;
440   3               disable;
441   3               scb.cmd = 8000H;
442   3               output (CA$PORT) = CA;
443   3               call wait$scb;
444   3           end;
445   2           if (scb.status and 1000H) = 1000H then
446   2           do;
447   3               disable;
448   3               scb.cmd = 1000H;
449   3               output (CA$PORT) = CA;
450   3               call wait$scb;
451   3               call write(0, @(0DH, ' Receive Unit became not ready.', 0DH), 33,
                                                                                @status);
452   3           end;

453   2           if reset then
454   2           do;
455   3               if iscp.busy then
456   3               do;
457   4                   call writeln(0, @(0DH, 0AH, ' Reset failed.'), 16, @status);
458   4                   disable;
459   4                   scb.cmd = 0080H;
460   4                   output (CA$PORT) = CA;
461   4                   call wait$scb;
462   4                   output (CA$PORT) = CA;
463   4                   call writeln(0, @(' Software Reset Executed!'), 25, @status);
464   4               end;
465   3               else reset = false;
466   3           end;

467   2       end isr;



468   1       tx$isr: procedure interrupt INT$TYPE$TIMER0;

                /* interrupt service routine for 80186 timer interrupt*/

469   2           scb.cmd = 0100H;
470   2           output(CA$PORT) = CA;
471   2           call wait$scb;

                $IF SBC18651

                    output(PIC$EOI$130) = EOI$CMD4$130;
                    enable;
                    output(PIC$EOI$186) = EOI$CMD0$186;

                $ELSE

472   2               output(PIC$EOI$186) = EOI$CMD4$186;

                $ENDIF

473   2       end tx$isr;
```

292010-49

Traffic Simulator/Monitor Station Program (Continued)

```
            $IF SBC18651

            isr$7: procedure interrupt INT$7;

            /* The 80130 generates an interrupt 7 if the original interrupt is not
               active any more when the first interrupt acknowledge is received. */

                call write(0, @(0DH, 'Interrupt 7', 0DH), 13, @status);

            end isr$7;

            $ENDIF


474   1     read$byte: procedure (k) byte;
475   2     declare k word;

476   2         call write(0, @(0DH, 0AH, ' Enter byte '), 14, @status);
477   2         call out$dec$hex(k);
478   2         call write(0, @(' ==> '), 5, @status);
479   2         return read$int(0FFH);

480   2     end read$byte;


481   1     init$186$timer0: procedure;

            /* This procedure initializes the 80186 timer 0.  */

482   2     declare i byte;

            $IF SBC18651

                output(INT$CTL$TIMER0) = 8;
                call write(0, @(0DH, 0AH, ' Enter a delay count ==> '), 27, @status);
                delay = read$int(0FFFFH);
                if (delay < 100) and (delay <> 0) then
                do;
                    call cr$lf;
                    call cr$lf;
                    call loop$char(35, '*');
                    call write(0, @(' WARNING '), 9, @status);
                    call loop$char(35, '*');
                    call writeln(0, @(0DH, 0AH, 'A delay count between 0 and 100 may be very ',
                            'dangerous when this station starts'), 80, @status);
                    call writeln(0, @('to receive many frames separated only by the ',
                            'IFS period (9.6 microseconds). '), 75, @status);
                    call writeln(0, @('If this station never receives a frame, then ',
                                'ignore this warning. '), 65, @status);
                    call loop$char(79, '*');
                end;
                output(MAX$COUNT$A) = delay;
                call cr$lf;
                output(PIC$MASK$186) = 3EH;
```

292010-50

Traffic Simulator/Monitor Station Program (Continued)

```
            $ELSE

483   2         output(INT$CTL$TIMER0) = 0CH;
484   2         call write(0, @(0DH, 0AH, ' Enter a delay count ==> '), 27, @status);
485   2         delay = read$int(0FFFFH);
486   2         output(MAX$COUNT$A) = delay;
487   2         call cr$lf;
488   2         output(PIC$MASK$186) = ENABLE$586$186;

            $ENDIF

489   2       end init$186$timer0;


490   1       setup$ia$parameters: procedure;
491   2       declare i byte;

492   2           call write(0, @(0DH, 0AH, ' Configure the 586 with the prewired'
                                          ,' board address ==> '), 57, @status);
493   2           if yes then
494   2           do i = 0 to address$length - 1;
495   3               ia$setup.ia$address(i) = input(BOARD$ADDRESS$BASE + 10 - 2 * i);
496   3           end;
497   2           else
                 do;
498   3               call write(0, @(0DH, 0AH, ' Enter this station''s address',
                                                ' in Hex ==> '),43, @status);
499   3               call put$address(@ia$setup.ia$address(0));
500   3           end;

501   2       end setup$ia$parameters;


502   1       setup$mc$parameters: procedure;
503   2       declare (j, k, done) byte;

504   2           j = 0;
505   2           call writeln(0, @(0DH, 0AH, ' You can enter up to 8 Multicast Addresses. '),
                                                                          45, @status);
506   2           done = false;
507   2           call write(0, @(' Would you like to enter a Multicast Address?',
                                                    ' (Y or N) ==> '), 59, @status);
508   2           do while not done;
509   3               if yes then
510   3               do;
511   4                   k = j * address$length;
512   4                   j = j + 1;
513   4                   call cr$lf;
514   4                   if j = 9 then
515   4                   do;
516   5                       call write(0, @(' You already entered 8 Multicast addresses. '),
                                                                  43, @status);
517   5                       done = true;
518   5                   end;
519   4                   else
                         do;
520   5                       call write(0, @(' Enter a Multicast Address ==> '), 31, @status);
```

292010-51

Traffic Simulator/Monitor Station Program (Continued)

```
521   5                    call put$address(@mc$setup.mc$address(k));
522   5                    call write(0, @(0DH, 0AH, ' More Multicast Addresses?',
                                                    ' (Y or N) ==> '), 42, @status);
523   5                end;
524   4            end;
525   3            else done = true;
526   3        end;
527   2        if j = 9 then j = j - 1;
529   2        mc$count = address$length * j;
530   2        mc$setup.mc$byte$count = mc$count;
531   2        call write(0, @(0DH, 0AH, ' You entered '), 15, @status);
532   2        call write$int(j, 0);
533   2        call writeln(0, @(' Multicast Address(es).'), 23, @status);

534   2    end setup$mc$parameters;


535   1    setup$configure$parameters: procedure;
536   2    declare (k, j) byte;

537   2        configure.byte$cnt = 11;
538   2        configure.info(0) = 8;
539   2        configure.info(1) = 0;
540   2        configure.info(2) = 26H;
541   2        configure.info(3) = 0;
542   2        configure.info(4) = 96;
543   2        configure.info(5) = 0;
544   2        configure.info(6) = 0F2H;
545   2        configure.info(7) = 0;
546   2        configure.info(8) = 0;
547   2        configure.info(9) = 64;
548   2        j = 0;
549   2        call write(0, @(0DH, 0AH, ' Configure command is set up for default',
                ' values.', 0DH, 0AH, ' Do you want to change any bytes?',
                                                    ' (Y or N) ==> '), 99, @status);
550   2        do while yes;
551   3            do while j = 0;
552   4                call write(0, @(0DH, 0AH, ' Enter byte number (1 - 11) ==> '), 34,
                                                                        @status);
553   4                j = read$int(11);
554   4                if j = 0 then
555   4                call write(0, @(0DH, 0AH, ' Illegal byte number'), 22, @status);
556   4            end;
557   3            if j = 1 then configure.byte$cnt = read$byte(j);
559   3                else configure.info(j - 2) = read$byte(j);
560   3            j = 0;
561   3            call write(0, @(0DH, 0AH, ' Any more bytes? (Y or N) ==> '), 32,
                                                                        @status);
562   3        end;
563   2        preamble = shl(1, shr((configure.info (2) and 30H), 4)+1);
564   2        address$length = configure.info(2) and 07H;
565   2        if address$length = 7 then address$length = 0;
567   2        ad$loc = shr((configure.info(2) and 08H), 3);
568   2        if shr((configure.info(7) and 20H), 5) then crc = 2; else crc = 4;
571   2        if shr((configure.info(7) and 10H), 4) then crc = 0;

573   2    end setup$configure$parameters;
```

292010-52

Traffic Simulator/Monitor Station Program (Continued)

```
574   1       setup$tx$parameters: procedure;
575   2       declare (size, i) word;

576   2           do forever;
577   3               no$transmission = false;
578   3               transmit.bd$offset = offset (@tbd.act$count);
579   3               if not ad$loc then
580   3               do;
581   4                   call write(0, @(0DH, 0AH,
                            ' Enter a destination address in Hex ==> '), 42, @status);
582   4                   call put$address(@transmit.dest$adr(0));
583   4               end;
584   3               else call writeln(0, @(' 82586 is configured to pick up DA, IA, ',
                                ' and TYPE from TX buffer. '), 64, @status);
585   3               call cr$lf;
586   3               if not ad$loc then
587   3               do;
588   4                   call write(0, @(0DH, 0AH, ' Enter TYPE ==> '), 18, @status);
589   4                   transmit.type = read$int(0FFFFH);
590   4               end;
591   3               call writeln(0, @(0DH, 0AH, ' How many bytes of transmit data?'), 35,
                                                                          @status);
592   3               call write(0, @(' Enter a number ==> '), 20, @status);
593   3               size = read$int(1518);
594   3               tbd.act$count = size or 8000H;
595   3               if size <> 0 then
596   3               do;
597   4                   tbd.link$offset = 0FFFFH;
598   4                   tbd.ad0 = offset (@tx$buffer(0));
599   4                   tbd.ad1 = 0;
600   4                   do i = 0 to 1517;
601   5                       tx$buffer(i) = i;
602   5                   end;
603   4                   call writeln(0,
                            @(0DH, 0AH, ' Transmit Data is continuous numbers (0, 1, 2, 3, ',
                                                                  ' ... )'), 57, @status);
604   4                   call write(0, @(' Change any data bytes? (Y or N) ==> '), 37,
                                                                          @status);
605   4                   do while yes;
606   5                       call write(0, @(0DH, 0AH, ' Enter a byte number ==> '),
                                                                      27, @status);
607   5                       i = read$int(size);
608   5                       call write(0, @(0DH, 0AH, ' Byte '), 8, @status);
609   5                       call out$dec$hex(i);
610   5                       call write(0, @(' currently contains '), 20, @status);
611   5                       call out$dec$hex(tx$buffer(i));
612   5                       call write(0, @('. '), 1, @status);
613   5                       tx$buffer(i) = read$byte(i);
614   5                       call write(0, @(0DH, 0AH, ' Any more bytes? (Y or N) ==> '),
                                                                      32, @status);
615   5                   end;
616   4               end;
617   3               else
                        transmit.bd$offset = 0FFFFH;
618   3               call cr$lf;
```

292010-53

Traffic Simulator/Monitor Station Program (Continued)

```
619   3             call init$186$timer0;
620   3             call write(0, @(0DH, 0AH, ' Setup a transmit terminal count?',
                                                ' (Y or N) ==> '), 49, @status);
621   3             if yes then
622   3             do;
623   4                 stop = true;
624   4                 call write (0, @(0DH, 0AH, ' Enter a transmit',
                                                 ' terminal count ==> '), 39, @status);
625   4                 stop$count = read$int(0FFFFFFFFH);
626   4             end;
627   3             else stop = false;
628   3             call cr$lf;
629   3             call cr$lf;
630   3             call print$parameters;
631   3             call write(0, @(0DH, 0AH, ' Good enough? (Y or N) ==> '), 29,
                                                                        @status);
632   3             if yes then return;
634   3         end;

635   2     end setup$tx$parameters;


636   1     loop$char: procedure (i, j);
637   2     declare (i, j, k) byte;

638   2         do k = 1 to i;
639   3             call write(0, @j, 1, @status);
640   3         end;

641   2     end loop$char;


642   1     init: procedure;

643   2     declare  i byte;

644   2         call cr$lf;
645   2         call loop$char(13, 0AH);
646   2         call loop$char(15, ' ');
647   2         call writeln(0, @('TRAFFIC SIMULATOR AND MONITOR',
                                                ' STATION PROGRAM '), 46, @status);
648   2         call loop$char(7, 0AH);
649   2         call writeln(0, @(0DH, 0AH, ' Initialization begun'), 23, @status);
650   2         call cr$lf;
651   2         reset = true;
652   2         cur$cb$offset = 0FFFFH;
653   2         output(ESI$PORT) = NO$LOOPBACK;
654   2         output(ESI$PORT) = LOOPBACK;
655   2         dhex = false;

              /* set up interrupt logic */

656   2         call set$interrupt(INT$TYPE$586, isr);
657   2         call set$interrupt(INT$TYPE$TIMER0, tx$isr);

          $IF SBC18651
```

292010-54

Traffic Simulator/Monitor Station Program (Continued)

```
                    call set$interrupt(INT$7, isr7);
                    output (PIC$MASK$130) = ENABLE$586$186;
                    output (PIC$EOI$130) = EOI$CMD0$130;
                    output (PIC$EOI$130) = EOI$CMD4$130;
                    output (PIC$EOI$186) = EOI$CMD0$186;
                    output (PIC$VTR$186) = 30H;

                $ELSE

658   2             output (PIC$EOI$186) = EOI$CMD0$186;
659   2             output (PIC$EOI$186) = EOI$CMD4$186;
660   2             output (PIC$MASK$186) = ENABLE$586;

                $ENDIF

                    /* locate iscp */

661   2             iscp$ptr = ISCP$LOC$LO;

                    /* set up fields in ISCP */

662   2             iscp.busy = 1;
663   2             iscp.scb$b(0) = SCB$BASE$LO;
664   2             iscp.scb$b(1) = SCB$BASE$HI;
665   2             iscp.scb$o = offset (@scb.status);

                    /* set up SCB */

666   2             scb.status = 0;
667   2             scb.cbl$offset = offset (@diagnose.status);
668   2             scb.rpa$offset = offset (@rfd(0).status);
669   2             scb.crc$errs = 0;
670   2             scb.aln$errs = 0;
671   2             scb.rsc$errs = 0;
672   2             scb.ovrn$errs = 0;

                    /* set up Diagnose command */

673   2             diagnose.status = 0;
674   2             diagnose.cmd = 7;
675   2             diagnose.link$offset = offset (@configure.status);

                    /* set up CONFIGURE command */

676   2             configure.status = 0;
677   2             configure.cmd = 2;
678   2             configure.link$offset = offset (@ia$setup.status);;
680   2             call setup$configure$parameters;


                    /* set up IA command */

681   2             ia$setup.status = 0;
682   2             ia$setup.cmd = 1;
683   2             ia$setup.link$offset = offset (@mc$setup.status);
684   2             call setup$ia$parameters;
```

292010-55

Traffic Simulator/Monitor Station Program (Continued)

```
                    /* set up MC command */

685   2             mc$setup.status = 0;
686   2             mc$setup.cmd = 8003H;
687   2             mc$setup.link$offset = OFFFFH;
688   2             call setup$mc$parameters;



                    /* set up one transmit cb linked to itself */

689   2             transmit.status = 0;
690   2             call writeln(0, @(0DH, 0AH, ' Would you like to transmit?'), 30, @status);
691   2             call write(0, @(' Enter a Y or N ==> '), 20, @status);
692   2             if yes then
693   2             do;
694   3                 transmit.cmd = 8004H;
695   3                 transmit.link$offset = OFFFFH;
696   3                 transmit.bd$offset = offset (@tbd.act$count);
697   3                 call setup$tx$parameters;
698   3             end;
699   2             else no$transmission = true;


                    /* initialize receive packet area */

700   2             do i = 0 to 3;
701   3                 rfd(i).status = 0;
702   3                 rfd(i).el$s = 0;
703   3                 rfd(i).link$offset = offset (@rfd(i+1).status);
704   3                 rfd(i).bd$offset = OFFFFH;
705   3                 rbd(i).act$count = 0;
706   3                 rbd(i).next$bd$link = offset (@rbd(i+1).act$count);
707   3                 rbd(i).ad0 = offset (@rbuf(i).buffer(0));
708   3                 rbd(i).ad1 = 0;
709   3                 rbd(i).size = 1500;
710   3             end;
711   2             rfd(0).bd$offset = offset (@rbd(0).act$count);
712   2             rfd(4).status = 0;
713   2             rfd(4).el$s = 0;
714   2             rfd(4).link$offset = offset (@rfd(0).status);
715   2             rfd(4).bd$offset = OFFFFH;
716   2             rbd(4).act$count = 0;
717   2             rbd(4).next$bd$link = offset (@rbd(0).act$count);
718   2             rbd(4).ad0 = offset (@rbuf(4).buffer(0));
719   2             rbd(4).ad1 = 0;
720   2             rbd(4).size = 1500;



                    /* initialize counters */

721   2             count = 0;
722   2             receive$count = 0;
723   2             current$frame = 0;

                    /* issue the first CA */
```

292010-56

Traffic Simulator/Monitor Station Program (Continued)

```
724   2          output(CA$PORT) = CA;

725   2      end init;


726   1      print$help: procedure;

727   2          call writeln(0, @(0DH, 0AH, ' Commands are: '), 16, @status);
728   2          call writeln(0, @(0DH, 0AH, ' S - Setup CB              D - Display RFD/CB'),
                                                                        45, @status);
729   2          call writeln(0, @(' P - Print SCB            C - SCB Control CMD'), 44,
                                                                        @status);
730   2          call writeln(0, @(' L - ESI Loopback On      N - ESI Loopback Off'), 45,
                                                                        @status);
731   2          call writeln(0, @(' A - Toggle Number Base'), 23,
                                                                        @status);
732   2          call writeln(0, @(' Z - Clear Tx Frame Counter'), 27, @status);
733   2          call writeln(0, @(' Y - Clear Rx Frame Counter'), 27, @status);
734   2          call writeln(0, @(' E - Exit to Continuous Mode'), 28, @status);

735   2      end print$help;


736   1      enter$scb$cmd: procedure;
737   2      declare i byte;

              /* enter a command into the SCB */

738   2          call cr$lf;
739   2          if scb.cmd <> 0 then
740   2          do;
741   3              call writeln(0, @(' SCB command word is not cleared'), 32, @status);
742   3              call write(0, @(' Try a Channel Attention? (Y or N) ==> '),
                                                                        39, @status);
743   3              if yes then
744   3              do;
745   4                  output(CA$PORT) = CA;
746   4                  call writeln(0, @(' Issued channel attention'), 25, @status);
747   4                  call cr$lf;
748   4                  return;
749   4              end;
750   3          end;
751   2          call write(0, @(' Do you want to enter any SCB commands? (Y or N) ==> '),
                                                                        53, @status);
752   2          if not yes then return;
754   2          call write(0, @(0DH, 0AH, ' Enter CUC ==> '), 17, @status);
755   2          i = read$int(4);
756   2          scb.cmd = scb.cmd or shl(double(i), 8);
757   2          if i = 1 then scb.cbl$offset = cur$cb$offset;
759   2          call write(0, @(0DH, 0AH, ' Enter RES bit ==> '), 21, @status);
760   2          i = read$bit;
761   2          scb.cmd = scb.cmd or shl(i, 7);
762   2          call write(0, @(0DH, 0AH, ' Enter RUC ==> '), 17, @status);
763   2          i = read$int(4);
764   2          scb.cmd = scb.cmd or shl(i, 4);
```

292010-57

Traffic Simulator/Monitor Station Program (Continued)

```
765   2          if (((scb.cbl$offset = offset (@transmit.status))
                     and ((scb.cmd and 0100H) = 0100H)) or ((scb.cmd and 0010H) = 0010H))
                     and not ((scb.cmd and 0080H) = 0080H)
766   2              then goback = 1;
767   2          call writeln(0, @(0DH, 0AH, ' Issued Channel Attention'), 27, @status);
768   2          call cr$lf;
769   2          output(CA$PORT) = CA;

770   2      end enter$scb$cmd;


771   1      print$type$help: procedure;

772   2          call writeln(0, @(0DH, 0AH, 0AH, 'Command block type: '), 22, @status);
773   2          call writeln(0, @(' N - Nop                I - IA Setup'), 35, @status);
774   2          call writeln(0, @(' C - Configure          M - MA Setup'), 35, @status);
775   2          call writeln(0, @(' T - Transmit           R - TDR'), 30, @status);
776   2          call writeln(0, @(' D - Diagnose           S - Dump Status'), 38, @status);
777   2          call writeln(0, @(' H - Print this message'), 23, @status);

778   2      end print$type$help;


779   1      setup$cb: procedure;
780   2      declare (t, valid) byte;

781   2          valid = false;
782   2          do while not valid;
783   3              call write(0, @(0DH, 0AH, ' Enter command block type (H for',
                                                ' help) ==> '), 45, @status);
784   3              t = read$char;
785   3              if (t <> 'H') and (t <> 'h') and (t <> 'T') and (t <> 't') and
                       (t <> 'N') and (t <> 'n') and (t <> 'R') and (t <> 'r') and
                       (t <> 'D') and (t <> 'd') and (t <> 'C') and (t <> 'c') and
                       (t <> 'I') and (t <> 'i') and (t <> 'M') and (t <> 'm') and
                       (t <> 'S') and (t <> 's') then
786   3              call write(0, @(0DH, 0AH, ' Illegal command block type'), 29,
                                                                        @status);
787   3              else
788   3              if (t = 'H') or (t = 'h') then call print$type$help;
789   3              else valid = true;
790   3          end;
791   2          if (t = 'N') or (t = 'n') then
792   2          do;
793   3              cur$cb$offset = offset (@nop.status);
794   3              nop.status = 0;
795   3              nop.cmd = 8000H;
796   3              nop.link$offset = 0FFFFH;
797   3          end;
798   2          if (t = 'I') or (t = 'i') then
799   2          do;
800   3              cur$cb$offset = offset (@ia$setup.status);
801   3              ia$setup.status = 0;
802   3              ia$setup.cmd = 8001H;
803   3              ia$setup.link$offset = 0FFFFH;
804   3              call setup$ia$parameters;
805   3          end;
```

292010-58

Traffic Simulator/Monitor Station Program (Continued)

```
806   2           if (t = 'C') or (t = 'c') then
807   2           do;
808   3               cur$cb$offset = offset (@configure.status);
809   3               configure.status = 0;
810   3               configure.cmd = 8002H;
811   3               configure.link$offset = OFFFFH;
812   3               call setup$configure$parameters;
813   3           end;
814   2           if (t = 'M') or (t = 'm') then
815   2           do;
816   3               cur$cb$offset = offset (@mc$setup.status);
817   3               mc$setup.status = 0;
818   3               mc$setup.cmd = 8003H;
819   3               mc$setup.link$offset = OFFFFH;
820   3               call setup$mc$parameters;
821   3           end;
822   2           if (t = 'T') or (t = 't') then
823   2           do;
824   3               cur$cb$offset = offset (@transmit.status);
825   3               transmit.status = 0;
826   3               transmit.cmd = 8004H;
827   3               transmit.link$offset = OFFFFH;
828   3               call setup$tx$parameters;
829   3           end;
830   2           if (t = 'R') or (t = 'r') then
831   2           do;
832   3               cur$cb$offset = offset (@tdr.status);
833   3               tdr.status = 0;
834   3               tdr.cmd = 8005H;
835   3               tdr.link$offset = OFFFFH;
836   3               tdr.result = 0;
837   3           end;
838   2           if (t = 'S') or (t = 's') then
839   2           do;
840   3               cur$cb$offset = offset (@dump.status);
841   3               dump.status = 0;
842   3               dump.cmd = 8006H;
843   3               dump.link$offset = OFFFFH;
844   3               dump.buff$ptr = offset (@dump$area(0));
845   3           end;
846   2           if (t = 'D') or (t = 'd') then
847   2           do;
848   3               cur$cb$offset = offset (@diagnose.status);
849   3               diagnose.status = 0;
850   3               diagnose.cmd = 8007H;
851   3               diagnose.link$offset = OFFFFH;
852   3           end;

853   2       end setup$cb;



854   1       display$command$block: procedure;
855   2       declare (i, j) byte,
                     wh pointer,
                     sel selector,
                     w word;
```

292010-59

Traffic Simulator/Monitor Station Program (Continued)

```
856   2          call cr$lf;
857   2          if cur$cb$offset = OFFFFH then
858   2              call write(0, @(' No Command Block to display'), 28, @status);
859   2          if cur$cb$offset = offset (@nop.status) then
860   2          do;
861   3              call write(0, @('---NOP Command Block---'), 23, @status);
862   3              call print$wds(@nop.status, 3);
863   3          end;
864   2          if cur$cb$offset = offset (@tdr.status) then
865   2          do;
866   3              call write(0, @('---TDR Command Block---'), 23, @status);
867   3              call print$wds(@tdr.status, 4);
868   3          end;
869   2          if cur$cb$offset = offset (@diagnose.status) then
870   2          do;
871   3              call write(0, @('---Diagnose Command Block---'), 28, @status);
872   3              call print$wds(@diagnose.status, 3);
873   3          end;
874   2          if cur$cb$offset = offset (@transmit.status) then
875   2          do;
876   3              call write(0, @('---Transmit Command Block---'), 28, @status);
877   3              if not address$length then i = address$length;
879   3              else i = address$length + 1;
880   3              if ad$loc then call print$wds(@transmit.status, 4);
882   3              else call print$wds(@transmit.status, i/2+1);
883   3              call cr$lf;
884   3              call cr$lf;
885   3              if transmit.bd$offset <> OFFFFH then
886   3              do;
887   4                  call write(0, @('---Transmit Buffer Descriptor---'), 33, @status);
888   4                  call print$wds(@tbd.act$count, 4);
889   4                  call write(0, @(ODH, OAH, OAH,
                                ' Display the transmit buffer? (Y or N) ==> '), 46, @status);
890   4                  if yes then
891   4                  do;
892   5                      call cr$lf;
893   5                      call writeln(0, @(' Transmit Buffer: '), 17, @status);
894   5                      w = tbd.act$count and 3FFFH;
895   5                      call print$buff(@tx$buffer(0), w);
896   5                  end;
897   4              end;
898   3          end;
899   2          if cur$cb$offset = offset (@ia$setup.status) then
900   2          do;
901   3              call write(0, @('---IA Setup Command Block---'), 28, @status);
902   3              call print$wds(@ia$setup.status, 6);
903   3          end;
904   2          if cur$cb$offset = offset (@configure.status) then
905   2          do;
906   3              call write(0, @('---Configure Command Block---'), 29, @status);
907   3              call print$wds(@configure.status, 9);
908   3          end;
909   2          if cur$cb$offset = offset (@mc$setup.status) then
910   2          do;
911   3              call write(0, @('---MC Setup Command Block---'), 28, @status);
912   3              i = 4 + mc$count/2;
```

292010-60

Traffic Simulator/Monitor Station Program (Continued)

```
913   3              if mc$count > 24 then
914   3                  do;
915   4                      call print$wds(@mc$setup.status, 16);
916   4                      call pause;
917   4                      i = i - 16;
918   4                      call print$wds(@mc$setup.mc$address(8), i);
919   4                  end;
920   3              else call print$wds(@mc$setup.status, i);
921   3          end;
922   2          if cur$cb$offset = offset (@dump.status) then
923   2          do;
924   3              call write(0, @('---Dump Status Command Block---'), 31, @status);
925   3              call print$wds(@dump.status, 4);
926   3              if dump.status = 0A000H then
927   3              do;
928   4                  call writeln(0, @(0DH, 0AH, ' Dump Status Results'), 22, @status)
929   4                  call write$offset(@dump$area(0));
930   4                  call cr$lf;
931   4                  do i = 0 to 9;
932   5                      call print$str(@dump$area(16*i), 16);
933   5                  end;
934   4                  call print$str(@dump$area(160), 10);
935   4                  call cr$lf;
936   4              end;
937   3          end;

938   2      end display$command$block;


939   1      display$receive$area: procedure;
940   2      declare (i, k, j, l) byte,
                  chars(4) byte;

941   2          call writeln(0, @(0DH, 0AH, ' Frame Descriptors:'), 21, @status);
942   2          if ad$loc then
943   2          do;
944   3              call writeln(0, @(0DH, 0AH, ' DA, SA, and TYPE are in buffer.', 0DH,
                                                                        0AH), 36, @status);
945   3              j = 3;
946   3          end;
947   2          else j = address$length + 4;
948   2          do k = 0 to j;
949   3              do i = 0 to 4;
950   4                  call out$word(@rfd(i).status, k);
951   4                  if k = 0 then call write$offset(@rfd(i).status);
953   4                      else call loop$char(10, ' ');
954   4              end;
955   3              call cr$lf;
956   3          end;
957   2          call writeln(0, @(0DH, 0AH, 0AH, ' Receive Buffer Descriptors:'), 31,
                                                                            @status);
958   2          do k = 0 to 4;
959   3              do i = 0 to 4;
960   4                  call out$word(@rbd(i).act$count, k);
961   4                  if k = 0 then call write$offset(@rbd(i).act$count);
963   4                      else call loop$char(10, ' ');
964   4              end;
```

292010-61

Traffic Simulator/Monitor Station Program (Continued)

```
 965   3              call cr$lf;
 966   3          end;
 967   2          call write(0, @(ODH, OAH, OAH, ' Display the receive
                                          ' buffers (Y or N) ==> '), 46, @status);
 968   2          if not yes then return;
 970   2          call writeln(0, @(ODH, OAH, ' Receive Buffers.'), 19, @status);
 971   2          do i = 0 to 4;
 972   3              call write(0, @(ODH, OAH, '  Receive Buffer '), 18, @status);
 973   3              call write$int(i, 0);
 974   3              call writeln(0, @(' :'), 2, @status);
 975   3              k = rbd(i).act$count and 3FFFH;
 976   3              call print$buff(@rbuf(i).buffer(0), k);
 977   3              call pause;
 978   3          end;

 979   2      end display$receive$area;


 980   1      display$cb$rpa: procedure;
 981   2      declare i byte;

 982   2          call write(0, @(ODH, OAH, ' Command Block or Receive Area? (R or C) ==> '),
                                                                          47, @status);
 983   2          i = read$char;
 984   2          do while (i <> 'R') and (i <> 'r') and (i <> 'C') and (i <> 'c');
 985   3              call writeln(0, @(ODH, OAH, ' Illegal command'), 18, @status);
 986   3              call write(0, @(' Enter R or C ==> '), 18, @status);
 987   3              i = read$char;
 988   3          end;
 989   2          if (i = 'R') or (i = 'r') then call display$receive$area;
 991   2          else call display$command$block;

 992   2      end display$cb$rpa;


 993   1      process$cmd: procedure;
 994   2      declare (b, i) byte;

 995   2          goback = 0;
 996   2          b = read$char;
 997   2          call cr$lf;
 998   2          if (b <> 'H') and (b <> 'h') and (b <> 'S') and (b <> 's') and
                     (b <> 'D') and (b <> 'd') and (b <> 'P') and (b <> 'p') and
                     (b <> 'C') and (b <> 'c') and (b <> 'E') and (b <> 'e') and
                     (b <> 'L') and (b <> 'l') and (b <> 'N') and (b <> 'n') and
                     (b <> 'Z') and (b <> 'z') and (b <> 'Y') and (b <> 'y') and
                     (b <> 'A') and (b <> 'a') then
 999   2          call write(0, @(' Illegal command'), 16, @status);
1000   2          if (b = 'H') or (b = 'h') then call print$help;
1002   2          if (b = 'A') or (b = 'a') then
1003   2              if dhex then
1004   2              do;
1005   3                  dhex = false;
1006   3                  call write(0, @(' Counters are displayed in decimal.'), 35,
                                                                          @status);
1007   3              end;
1008   2              else
```

292010-62

Traffic Simulator/Monitor Station Program (Continued)

```
                    do;
1009    3                   dhex = true;
1010    3                   call write(0, @(' Counters are displayed in hexadecimal. '), 39,
                                                                                    @status);
1011    3               end;
1012    2           if (b = 'L') or (b = 'l') then
1013    2           do;
1014    3               output(ESI$PORT) = LOOPBACK;
1015    3               call write(0, @(' ESI is in Loopback Mode. '), 25, @status);
1016    3           end;
1017    2           if (b = 'N') or (b = 'n') then
1018    2           do;
1019    3               output(ESI$PORT) = NO$LOOPBACK;
1020    3               call write(0, @(' ESI is NOT in Loopback Mode. '), 29, @status);
1021    3           end;
1022    2           if (b = 'Z') or (b = 'z') then
1023    2           do;
1024    3               count = 0;
1025    3               call write(0, @(' Transmit Frame Counter is cleared. '), 35, @status);
1026    3           end;
1027    2           if (b = 'Y') or (b = 'y') then
1028    2           do;
1029    3               receive$count = 0;
1030    3               scb.crc$errs, scb.aln$errs, scb.rsc$errs, scb.ovrn$errs = 0;
1031    3               call write(0, @(' Receive Frame Counter is cleared. '), 34, @status);
1032    3           end;
1033    2           if (b = 'C') or (b = 'c') then call enter$scb$cmd;
1035    2           if (b = 'S') or (b = 's') then call setup$cb;
1037    2           if (b = 'P') or (b = 'p') then call print$scb;
1039    2           if (b = 'D') or (b = 'd') then call display$cb$rpa;
1041    2           if (b = 'E') or (b = 'e') then goback = 1;
1043    2           call cr$lf;

1044    2       end process$cmd;


1045    1       getout: procedure;
1046    2       declare b byte;

1047    2           b = read$char;
1048    2           goback = 0;
1049    2           call write(0, @(0DH, 0AH, ' Enter command (H for help) ==> '), 34,
                                                                                    @status);
1050    2           do forever;
1051    3               if csts then
1052    3               do;
1053    4                   disable;
1054    4                   call process$cmd;
1055    4                   enable;
1056    4                   if goback then return;
1058    4                   call write(0, @(0DH, 0AH, ' Enter command (H for help) ==> '), 34,
                                                                                    @status);
1059    4               end;
1060    3           end;

1061    2       end getout;
```

292010-63

Traffic Simulator/Monitor Station Program (Continued)

```
1062   1      update: procedure;
1063   2      declare i byte;

1064   2          call cr$lf;
1065   2          call loop$char(10, 0AH);
1066   2          call loop$char(28, '*');
1067   2          call write(0, @(' Station Configuration '), 23, @status);
1068   2          call loop$char(27, '*');
1069   2          call cr$lf;
1070   2          call cr$lf;
1071   2          call write(0, @(' Host Address: '), 15, @status);
1072   2          call print$network$addr(@ia$setup.ia$address(0));
1073   2          i = 0;
1074   2          call write(0, @(' Multicast Address(es): '), 24, @status);
1075   2          if mc$setup.mc$byte$count = 0
1076   2          then call writeln(0, @('No Multicast Addresses Defined'), 30, @status);
1077   2          else
                       do while i < mc$setup.mc$byte$count;
1078   3                      call print$network$addr(@mc$setup.mc$address(i));
1079   3                      call loop$char(24, ' ');
1080   3                      i = i + 6;
1081   3                  end;
1082   2          call write(0, @(0DH), 1, @status);
1083   2          if not no$transmission then call print$parameters;
1085   2          call write(0, @(' 82586 Configuration Block: '), 28, @status);
1086   2          call print$str(@configure.info(0), 10);
1087   2          call cr$lf;
1088   2          call loop$char(29, '*');
1089   2          call write(0, @(' Station Activities '), 20, @status);
1090   2          call loop$char(29, '*');
1091   2          call cr$lf;
1092   2          call cr$lf;
1093   2          call writeln(0,
                   @(' # of Good    # of Good    CRC          Alignment    No           Receive'),
                                                                                  73, @status);
1094   2          call writeln(0,
                   @(' Frames       Frames       Errors       Errors       Resource     Overrun'),
                                                                                  73, @status);
1095   2          call writeln(0,
                   @(' Transmitted  Received                                 Errors       Errors'),
                                                                                  72, @status);

1096   2      end update;


1097   1      main:

                   call init;
1098   1          enable;
1099   1          do while reset;
1100   2          end;
1101   1          disable;
1102   1          scb.cmd = 0100H;
1103   1          output(CA$PORT) = CA;
1104   1          call wait$scb;
1105   1          enable;
```

292010-64

Traffic Simulator/Monitor Station Program (Continued)

```
1106   1          do while (diagnose.status and 8000H) <> 8000H;
1107   2          end;
1108   1          call cr$lf;
1109   1          if diagnose.status <> 0A000H
1110   1          then call writeln(0, @(' Diagnose failed!'), 17 , @status);
1111   1          if configure.status <> 0A000H
1112   1          then call writeln(0, @(' Configure failed!'), 18 , @status);
1113   1          if ia$setup.status <> 0A000H
1114   1          then call writeln(0, @(' IA Setup failed!'), 17 , @status);
1115   1          if mc$setup.status <> 0A000H
1116   1          then call writeln(0, @(' MC Setup failed!'), 17 , @status);
1117   1          scb.cbl$offset = offset (@transmit.status);
1118   1          call writeln(0, @(0DH, 0AH, ' Receive Unit is active.'), 26, @status);
1119   1          disable;
1120   1          scb.cmd = 0010H;
1121   1          output(CA$PORT) = CA;
1122   1          call wait$scb;
1123   1          enable;
1124   1          output(ESI$PORT) = NO$LOOPBACK;
1125   1          call cr$lf;
1126   1          if not no$transmission then
1127   1          do;
1128   2              call write(0, @('---Transmit Command Block---'), 28, @status);
1129   2              call print$wds(@transmit.status, 8);
1130   2              call cr$lf;
1131   2              cur$cb$offset = offset (@transmit.status);
1132   2              call pause;
1133   2              do z = 1 to 60;
1134   3                  call time(250);
1135   3              end;
1136   2              call writeln(0, @(0DH, 0AH, 'transmission started!'), 23, @status);
1137   2              call cr$lf;
1138   2              disable;
1139   2              scb.cmd = 0100H;
1140   2              output (CA$PORT) = CA;
1141   2              call wait$scb;
1142   2              enable;
1143   2          end;
1144   1          call update;
1145   1          do forever;
1146   2              call write(0, @(0DH, ' '), 2, @status);
1147   2              do y = 0 to 5;
1148   3                  do case y;
1149   4                      call write$int(count, dhex);
1150   4                      call write$int(receive$count, dhex);
1151   4                      call write$int(scb.crc$errs, dhex);
1152   4                      call write$int(scb.aln$errs, dhex);
1153   4                      call write$int(scb.rsc$errs, dhex);
1154   4                      call write$int(scb.ovrn$errs, dhex);
1155   4                  end;
1156   3                  char$count = 13 - char$count;
1157   3                  call loop$char(char$count, ' ');
1158   3              end;
1159   2              if csts then
1160   2              do;
1161   3                  disable;
1162   3                  call getout;                                    292010-65
1163   3                 call update;
1164   3              end;
1165   2          end;

1166   1      end tsms;


MODULE INFORMATION:

     CODE AREA SIZE     = 23C3H   9155D
     CONSTANT AREA SIZE = 0F85H   3973D
     VARIABLE AREA SIZE = 265EH   9822D
     MAXIMUM STACK SIZE = 0092H    146D
     1994 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     159KB MEMORY AVAILABLE
     23KB MEMORY USED   (14%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
                                                                     292010-66
```

Traffic Simulator/Monitor Station Program (Continued)

```
/*****************************************************************************
/*                                                                           */
/*          186/586 High Integration Board Initialization Routine            */
/*          (This driver is configured for Ethernet/Cheapernet Design        */
/*           Kit Demo Board)                                                 */
/*                                                                           */
/*          Ver. 2.0                          March 14, 1986                 */
/*                                                                           */
/*          Kiyoshi Nishide                   Intel Corporation              */
/*                                                                           */
/*****************************************************************************/

/* The conditional compilation parameter 'EPROM27128' determines board ROM
   size.  If it is true, the 80186's wait state generator is programmed to
   0 wait state for upper 64K-byte memory locations.  If it is false, the
   wait state generator is programmed to 0 wait state for upper 128K-byte
   memory locations   */


1       ini186:

        do;

2   1   declare hib_ir label public;
3   1   declare main label external;
4   1   declare menu label external;

        /* literals */

5   1   declare lit       literally 'literally',
                UMCS_reg          lit '0FFA0H',
                LMCS_reg          lit '0FFA2H',
                PACS_reg          lit '0FFA4H',
                MPCS_reg          lit '0FFA8H',
                INT_MASK_reg      lit '0FF28H',
                ISCP$LOC$LO       lit '03FF8H',
                ISCP$LOC$HI       lit '0',
                SCC_CH_B_CMD      lit '8300H',
                SCC_CH_B_DATA     lit '8302H',
                SCC_CH_A_CMD      lit '8304H',
                SCC_CH_A_DATA     lit '8306H',
                NUL               lit '0',
                CR                lit '0DH',
                LF                lit '0AH',
                BS                lit '08H',
                SP                lit '20H',
                QM                lit '3FH',
                DEL               lit '07FH',
                BEL               lit '07H';
```

292010-67

186/586 High Integration Board Initialization Routine

```
                /* System Configuration Pointer */

 6    1         declare scp structure
                        (
                        sysbus byte,
                        unused (5) byte,
                        iscp$addr$lo word,
                        iscp$addr$hi word
                        )
                        at (OFFFF6H) data (0, 0, 0, 0, 0, 0, ISCP$LOC$LO, ISCP$LOC$HI);


 7    1         init$int$clt: procedure;

 8    2                 output(INT_mask_reg) = OFFH; /* mask all interrupts */

 9    2         end init$int$clt;


10    1         rra: procedure (reg_no) byte;
11    2         declare reg_no byte;

12    2             if (reg_no and OFH) <> 0 then output(SCC_CH_A_CMD) = reg_no and OFH;
14    2             return input(SCC_CH_A_CMD);

15    2         end rra;


16    1         rrb: procedure (reg_no) byte;
17    2         declare reg_no byte;

18    2             if (reg_no and OFH) <>0 then output (SCC_CH_B_CMD) = reg_no and OFH;
20    2             return input(SCC_CH_B_CMD);

21    2         end rrb;


22    1         wra: procedure (reg_no, value);
23    2         declare (reg_no, value) byte;

24    2             if (reg_no and OFH) <> 0 then output (SCC_CH_A_CMD) = reg_no and OFH;
26    2             output (SCC_CH_A_CMD) = value;

27    2         end wra;

28    1         wrb: procedure (reg_no, value);
29    2         declare (reg_no, value) byte;

30    2             if (reg_no and OFH) <> 0 then output (SCC_CH_B_CMD) = reg_no and OFH;
32    2             output (SCC_CH_B_CMD) = value;

33    2         end wrb;

34    1         init$SCC$B: procedure;

35    2             call wrb(09, 01000000b);  /* channel B reset */
```

292010-68

186/586 High Integration Board Initialization Routine (Continued)

```
36   2          call wrb(04, 01001110b);   /* 2 stop, no parity, brf = 16x */
37   2          call wrb(03, 11000000b);   /* rx 8 bits/char, no auto-enable */
38   2          call wrb(05, 01100000b);   /* tx 8 bits/char */
39   2          call wrb(10, 00000000b);
40   2          call wrb(11, 01010110b);   /* rxc = txc = BRG, trxc = BRG out */
41   2          call wrb(12, 00001011b);   /* baud rate = 9600 */
42   2          call wrb(13, 00000000b);
43   2          call wrb(14, 00000011b);   /* BRG source = SYS CLK, enable BRG */
44   2          call wrb(15, 00000000b);   /* all ext status interrupts off */

45   2          call wrb(03, 11000001b);   /* scc-b receive enable */
46   2          call wrb(05, 11101010b);   /* scc-b transmit enable, dtr on, rts on */

47   2      end init$SCC$B;


48   1      c$in: procedure byte public;

49   3          do while (input(SCC_CH_B_CMD) and 1) = 0; end;
51   2          return (input(SCC_CH_B_DATA));

52   2      end c$in;


53   1      c$out: procedure (char) public;
54   2      declare char byte;

55   3          do while (input(SCC_CH_B_CMD) and 4) = 0; end;
57   2          output(SCC_CH_B_DATA) = char;

58   2      end c$out;



59   1      read: procedure (file$id, msg$ptr, count, actual$ptr, status$ptr) public;
60   2      declare file$id word,
                    msg$ptr pointer,
                    count word,
                    actual$ptr pointer,
                    status$ptr pointer,
                    msg based msg$ptr (1) byte,
                    buf (200) byte,
                    actual based actual$ptr word,
                    status based status$ptr word,
                    i word,
                    ch byte;

              /* This procedure implements the ISIS read procedure. All control characters */
              /* except LF, BS, and DEL are ignored. If BS or DEL is encountered, a        */
              /* backspace is done.                                                        */

61   2          status = 0;
62   2          i, ch = 0;
63   2          do while (ch <> CR) and (ch <> LF) and (i < 198);
64   3              ch = c$in and 07FH;
65   3              if (ch = BS) or (ch = DEL) then
66   3              do;
```

292010-69

186/586 High Integration Board Initialization Routine (Continued)

```
 67   4                  if i > 0 then
 68   4                  do;
 69   5                      i = i - 1;
 70   5                      call c$out(DEL);
 71   5                      call c$out(BS);
 72   5                      call c$out(SP);
 73   5                      call c$out(DEL);
 74   5                      call c$out(BS);
 75   5                  end;
 76   4                  else
                             call c$out(BEL);
 77   4              end;
 78   3              else
                         if ch >= SP then
 79   3                      do;
 80   4                          call c$out(ch);
 81   4                          buf(i) = ch;
 82   4                          i = i + 1;
 83   4                      end;
 84   3                  else
                             if (ch = CR) or (ch = LF) then
 85   3                          do;
 86   4                              buf(i) = CR;
 87   4                              buf(i + 1) = LF;
 88   4                              i = i + 2;
 89   4                          end;
 90   3                  else
                             call c$out(BEL);
 91   3          end;
 92   2          call c$out(CR);
 93   2          if i > count then i = count;
 95   2          actual = i;
 96   2          do i = 0 to actual - 1;
 97   3              msg(i) = buf(i);
 98   3          end;

 99   2      end read;


100   1      csts: procedure byte public;

101   2          return ((input(SCC_CH_B_CMD) and 1) <> 0);

102   2      end csts;


103   1      write: procedure (file$id, msg$ptr, count, status$ptr) public;
104   2      declare (file$id, count) word,
                 (msg$ptr, status$ptr) pointer,
                 msg based msg$ptr (1) byte,
                 status based status$ptr word,
                 ch byte,
                 i word;

              /* This procedure implements the ISIS write */

105   2          status = 0;
```

292010-70

186/586 High Integration Board Initialization Routine (Continued)

```
106   2           i = 0;
107   2           do while i < count;
108   3               ch = msg(i);
109   3               if ((ch >= SP) and (ch < DEL)) or (ch = CR) or (ch = LF) or (ch = NUL)
                      then
110   3                   call c$out(ch);
111   3               else
                          call c$out(@M);
112   3               i = i + 1;
113   3           end;

114   2       end write;


115   1       hib_ir:

                $IF EPROM27128

                        output(UMCS_reg) = 0F038H;    /* Starting Address = 0F0000H,
                                                         no wait state */

                $ELSE

                        output(UMCS_reg) = 0E038H;    /* Starting Address = 0E0000H,
                                                         no wait state */

                $ENDIF

116   1                 output(LMCS_reg) = 03FCH;     /* 16K, no wait state */
117   1                 output(PACS_reg) = 083CH;     /* PBA = 8000H, no wait state for
                                                         PSC0-3 */
118   1                 output(MPCS_reg) = 0BFH;      /* Peripherals in I/O space, no A1 & A2
                                                         provided, 3 wait states for PSC4-6 */

119   1                 call init$int$clt;
120   1                 call init$SCC$B;
121   1                 go to main;

122   1       end ini186;
```

292010-71

186/586 High Integration Board Initialization Routine (Continued)

APPENDIX C
THE 82530 SCC - 80186 INTERFACE AP BRIEF

INTRODUCTION

The object of this document is to give the 82530 system designer an in-depth worst case design analysis of the typical interface to a 80186 based system. This document has been revised to include the new specifications for the 6 MHz 82530. The new specifications yield better margins and a 1 wait state interface to the CPU (2 wait states are required for DMA cycles). These new specifications will appear in the 1987 data sheet and advanced specification information can be obtained from your local Intel sales office. The following analysis includes a discussion of how the interface TTL is utilized to meet the timing requirements of the 80186 and the 82530. In addition, several optional interface configurations are also considered.

INTERFACE OVERVIEW

The 82530 - 80186 interface requires the TTL circuitry illustrated in Figure 1. Using five 14 pin TTL packages, 74LS74, 74AS74, 74AS08, 74AS04, and 74LS32, the following operational modes are supported:

- Polled
- Interrupt in vectored mode
- Interrupt in non-vectored mode
- Half-duplex DMA on both channels
- Full-duplex DMA on channel A

A brief description of the interface functional requirements during the five possible BUS operations follows below.

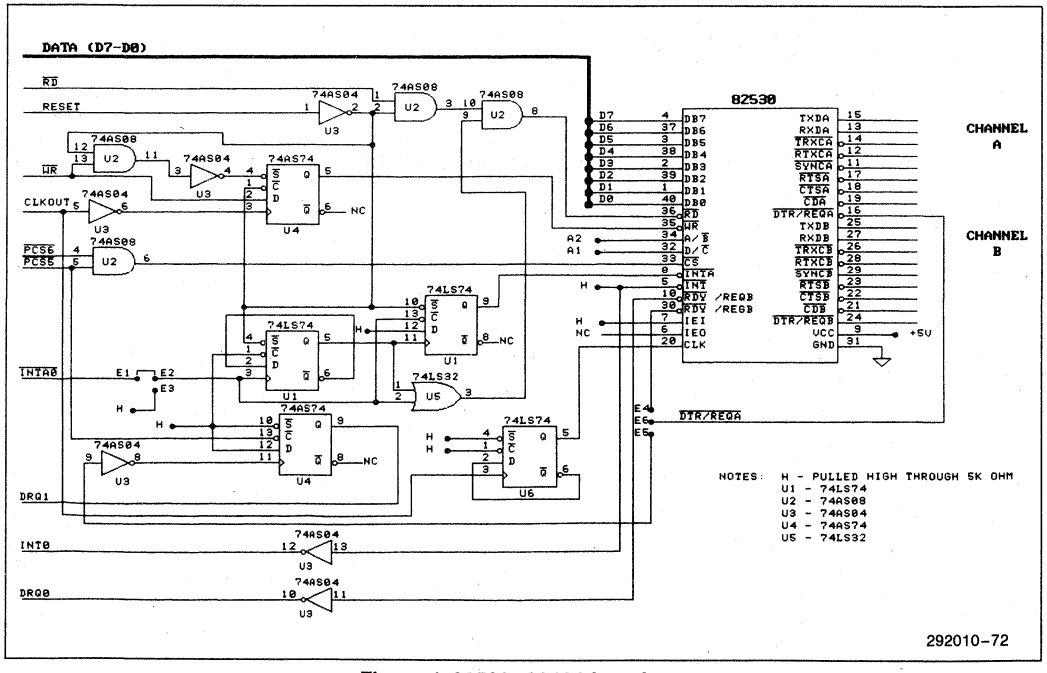


Figure 1. 82530–80186 Interface

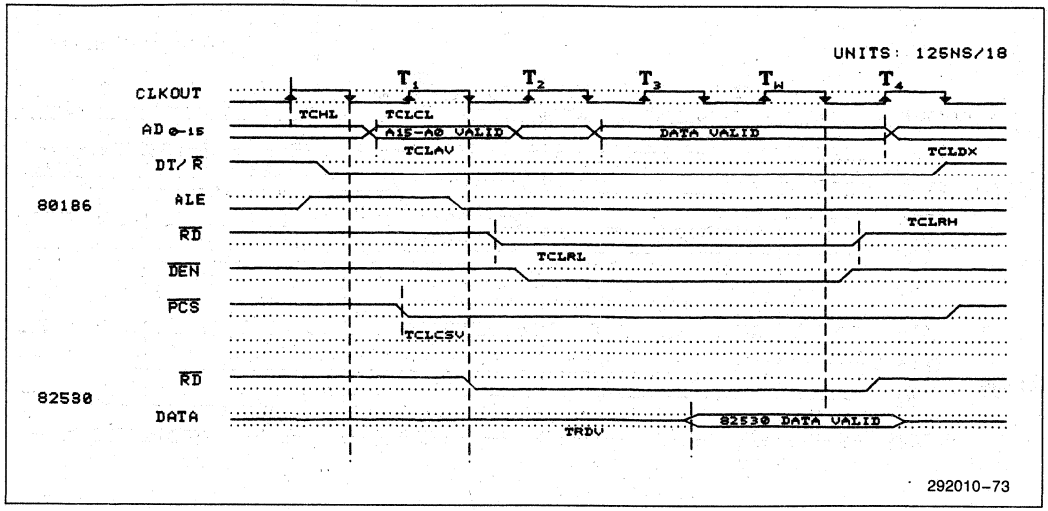


Figure 2. 80186–82530 Interface Read Cycle

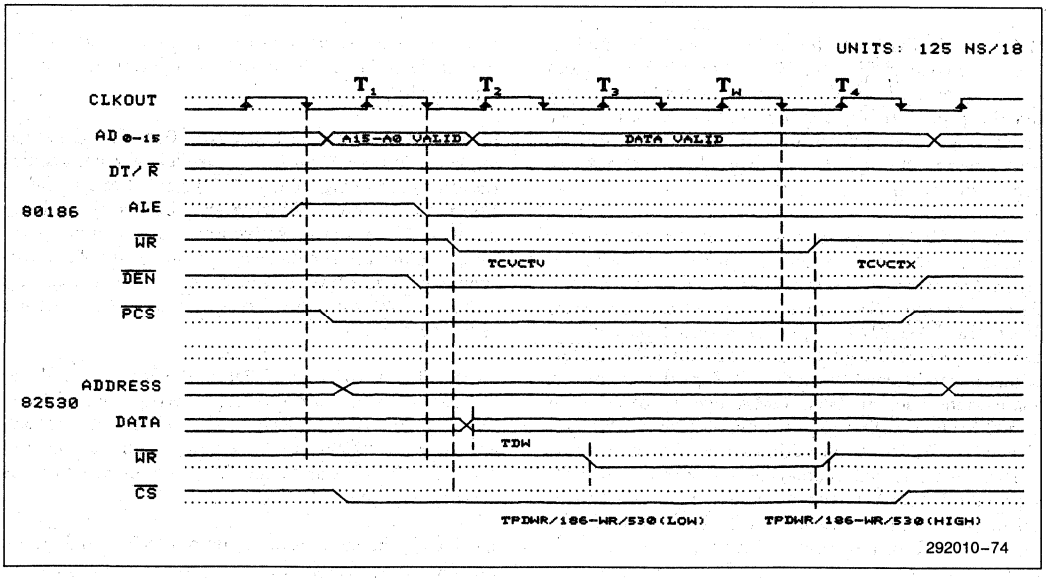


Figure 3. 80186–82530 Interface Write Cycle

READ CYCLE: The 80186 read cycle requirements are met without any additional logic, Figure 2. At least one wait state is required to meet the 82530 tAD access time.

WRITE CYCLE: The 82530 requires that data must be valid while the $\overline{WR}$ pulse is low, Figure 3. A D Flip-Flop delays the leading edge of $\overline{WR}$ until the falling edge of CLOCKOUT when data is guaranteed valid and $\overline{WR}$ is guaranteed active. The CLOCKOUT signal is inverted to assure that $\overline{WR}$ is active low before the D Flip-Flop is clocked. No wait states are necessary to meet the 82530's $\overline{WR}$ cycle requirements, but one is assumed from the $\overline{RD}$ cycle.

INTA CYCLE: During an interrupt acknowledge cycle, the 80186 provides two $\overline{INTA}$ pulses, one per bus cycle, separated by two idle states. The 82530 expects only one long $\overline{INTA}$ pulse with a $\overline{RD}$ pulse occurring only after the 82530 $\overline{IEI}$/$\overline{IEO}$ daisy chain settles. As

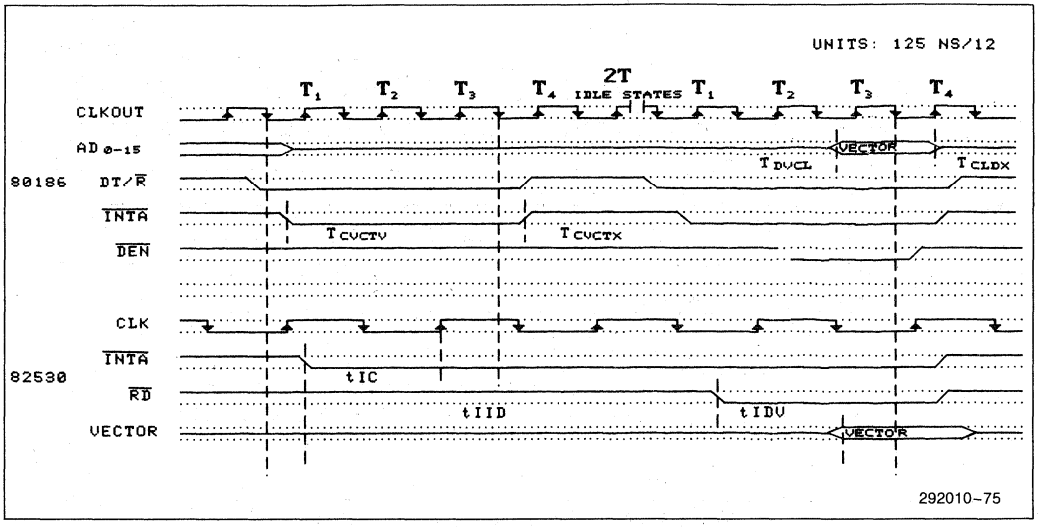


Figure 4. 82530–80186 INTA Cycle

illustrated in Figure 4, the INTA signal is sampled on the rising edge of CLK (82530). Two D Flip-Flops and two TTL gates, U2 and U5, are implemented to generate the proper INTA and RD pulses. Also, the INT signal is passively pulled high, through a 1 k resistor, and inverted through U3 to meet the 80186's active high requirement.

DMA CYCLE: Conveniently, the 80186 DMA cycle timings are the same as generic read and write operations. Therefore, with two wait states, only two modifications to the DMA request signals are necessary. First, the RDY/REQA signal is inverted through U3 similar to the INT signal, and second the DTR/REQA signal is conditioned through a D Flip-Flop to prevent inadvertent back to back DMA cycles. Because the 82530 DTR/REQA signal remains active low for over five CLK (82530)'s, an additional DMA cycle could occur. This uncertain condition is corrected when U4 resets the DTR/REQ signal inactive high. Full Duplex on both DMA channels can easily be supported with one extra D Flip-Flop and an inverter.

RESET: The 82530 does not have a dedicated RESET input. Instead, the simultaneous assertion of both RD and WR causes a hardware reset. This hardware reset is implemented through U2, U3, and U4.

ALTERNATIVE INTERFACE CONFIGURATIONS

Due to its wide range of applications, the 82530 interface can have many varying configurations. In most of these applications the supported modes of operation need not be as extensive as the typical interface used in this analysis. Two alternative configurations are discussed below.

8288 BUS CONTROLLER: An 80186 based system implementing an 8288 bus controller will not require the preconditioning of the WR signal through the D Flip-Flop U4. When utilizing an 8288, the control signal IOWC does not go active until data is valid, therefore, meeting the timing requirements of the 82530. In such a configuration, it will be necessary to logically OR the IOWC with reset to accommodate a hardware reset operation.

NON-VECTORED INTERRUPTS: If the 82530 is to be operated in the non-vectored interrupt mode (B step only), the interface will not require U1 or U5. Instead, INTA on the 82530 should be pulled high, and pin 3 of U2 (RD AND RESET) should be fed directly into the RD input of the SCC.

Obviously, the amount of required interface logic is application dependent and in many cases can be considerably less than required by the typical configuration, supporting all modes of SCC operation.

DESIGN ANALYSIS

This design analysis is for a typical microprocessor system, pictured in Figure 5. The Timing analysis assumes an 8 MHz 80186 and a 4 MHz 82530. Also, included in the analysis are bus loading, and TTL-MOS compatibility considerations.

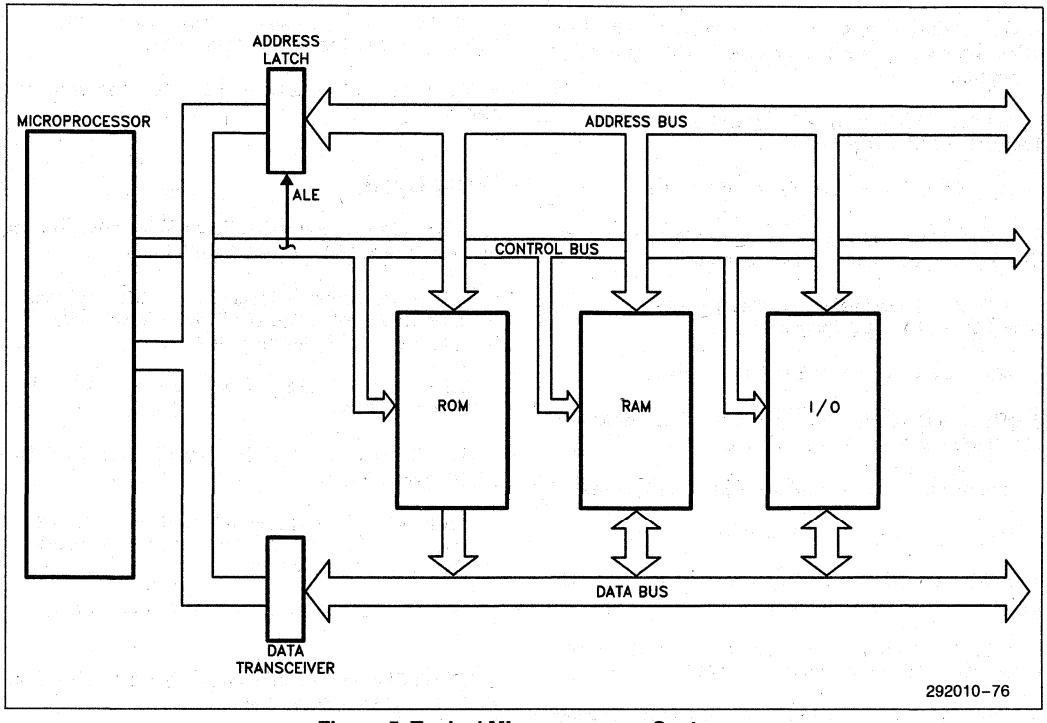


Figure 5. Typical Microprocessor System

Bus Loading and Voltage Level Compatabilities

The data and address lines do not exceed the drive capability of either 80186 or the 82530. There are several control lines that drive more than one TTL equivalent input. The drive capability of these lines are detailed below.

$\overline{WR}$: The $\overline{WR}$ signal drives U3 and U4.

* Iol (2.0 mA) > Iil (−0.4 mA + −0.5 mA)
Ioh (−400 μA) > Iih (20 μA + 20 μA)

$\overline{PCS5}$: The $\overline{PCS5}$ signal drives U2 and U4.

* Iol (2.0 mA) > Iil (−0.5 mA + −0.5 mA)
Ioh (−400 μA) > Iih (20 μA + 20 μA)

$\overline{INTA}$: The $\overline{INTA}$ signal drives 2(U1) and U5.

* Iol (2.0 mA) > Iil (–0.4 mA + −0.8 mA + −0.4 mA)
Ioh (−400 μA) > Iih (20 μA + 40 μA + 20 μA)

All the 82530 I/O pins are TTL voltage level compatible.

TIMING ANALYSIS

Certain symbolic conventions are adhered to throughout the analysis below and are introduced for clarity.

1. All timing variables with a lower case first letter are 82530 timing requirements or responses (i.e., tRR).
2. All timing variables with Upper case first letters are 80186 timing responses or requirements unless preceded by another device's alpha-numeric code (i.e., Tclcl or '373 Tpd).
3. In the write cycle analysis, the timing variable Tpd$\overline{WR}$186-$\overline{WR}$530 represents the propagation delay between the leading or trailing edge of the $\overline{WR}$ signal leaving the 80186 and the $\overline{WR}$ edge arrival at the 82530 $\overline{WR}$ input.

Read Cycle

1. **tAR:** Address valid to $\overline{RD}$ active set up time for the 82530. Since the propagation delay is the worst case path in the assumed typical system, the margin is calculated only for a propagation delay constrained and not an ALE limited path. The spec value is 0 ns minimum.

* 1 Tclcl − Tclav(max) − '245 Tpd(max) + Tclrl(min) + 2(U2) Tpd(min) − tAR(min)

= 125 − 55 − 20.8 + 10 + 2(2) − 0 = 63.2 ns margin

2. **tRA:** Address to $\overline{RD}$ inactive hold time. The ALE delay is the worst case path and the 82530 requires 0 ns minimum.

* 1 Tclcl − Tclrh (max) + Tchlh(min) + '373 LE Tpd(min) − 2(U2) Tpd(max)

= 55 − 55 + 5 + 8 − 2(5.5) = 2 ns margin

3. **tCLR:** $\overline{CS}$ active low to $\overline{RD}$ active low set up time. The 82530 spec value is 0 ns minimum.

* 1 Tclcl − Tclcsv(max) − Tclrl(min) − U2 skew($\overline{RD}$ − $\overline{CS}$) + U2 Tpd(min)

= 125 − 66 − 10 − 1 + 2 = 50 ns margin

4. **tRCS:** $\overline{RD}$ inactive to $\overline{CS}$ inactive hold time. The 82530 spec calls for 0 ns minimum.

* Tcscsx(min) − U2 skew($\overline{RD}$ − $\overline{CS}$) − U2 Tpd(max)

− 35 − 1 − 5.5 = 28.5 ns margin

5. **tCHR:** $\overline{CS}$ inactive to $\overline{RD}$ active set up time. The 82530 requires 5 ns minimum.

* 1 Tclcl + 1 Tchcl − Tchcsx(max) + Tclrl(min) − U2 skew ($\overline{RD}$ − $\overline{CS}$) + U2 Tpd(min) − tCHR

= 125 + 55 − 35 − 10 − 1 + 2 − 5 = 131 ns margin

6. **tRR:** $\overline{RD}$ pulse active low time. One 80186 wait state is included to meet the 150 ns minimum timing requirements of the 82530.

* Trlrh(min) + 1($\overline{Tclcl}$wait state) − 2(U2 skew) − tRR

= (250−50) + 1(125) − 2(1) − 150 = 173 ns margin

7. tRDV: $\overline{RD}$ active low to data valid maximum delay for 80186 read data set up time (Tdvcl = 20 ns). The margin is calculated on the Propagation delay path (worst case).

* 2 Tclcl + 1($\overline{Tclcl}$wait state) − Tclrl(max) − Tdvcl(min) − '245 Tpd(max) − 82530 tRDV(max) − 2(U2) Tpd(max)

= 2(125) + 1(125) − 70 − 20 − 14.2 − 105 − 2(5.5) = 154 ns margin

8. **tDF:** $\overline{RD}$ inactive to data output float delay. The margin is calculated to $\overline{DEN}$ active low of next cycle.

* 2 Tclcl + Tclch(min) − Tclrh(max) + Tchctv(min) − 2(U2) Tpd(max) − 82530 tDF(max)

= 250 + 55 −55 + 10 − 11 − 70 = 179 ns margin

9. **tAD:** Address required valid to read data valid maximum delay. The 82530 spec value is 325 ns maximum.

* 3 Tclcl + 1($\overline{Tclcl}$wait state) − Tclav(max) − '373 Tpd(max) − '245 Tpd − Tdvcl(min) − tAD

= 375 + 125 − 55 − 20.8 −14.2 − 20 −325 = 65 ns margin

Write Cycle

1. **tAW:** Address required valid to $\overline{WR}$ active low set up time. The 82530 spec is 0 ns minimum.

* Tclcl − Tclav(max) − Tcvctv(min) − '373 Tpd(max) + Tpd$\overline{WR}$186 − $\overline{WR}$530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)] − tAW

= 125 − 55 − 5 − 20.8 + [125 − 5 + 1 + 4.4] − 0 = 170.6 ns margin

2. **tWA:** $\overline{WR}$ inactive to address invalid hold time. The 82530 spec is 0 ns.

* Tclch(min) − Tcvctx(max) + Tchlh(min) + '373 LE Tpd(min) − Tpd$\overline{WR}$186=$\overline{WR}$530(HIGH) [U2 Tpd(max) + U3 Tpd(max) + U4 Tpd(max)]

= 55 − 55 + 5 + 8 − [5.5 + 3 + 7.1] = −2.6 ns margin

3. **tCLW:** Chip select active low to $\overline{WR}$ active low hold time. The 82530 spec is 0 ns.

* 1 Tclcl − Tclcsv(max) + Tcvctv(min) − U2 Tpd(max) + Tpd$\overline{WR}$186=$\overline{WR}$530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)]

= 125 − 66 + 5 − 5.5 + [125 − 5 + 1 + 4.4] = 183.9 ns margin

4. **tWCS:** $\overline{WR}$ invalid to Chip Select invalid hold time. 82530 spec is 0 ns.

* Tcxcsx(min) − U2 Tpd(max) − Tpd$\overline{WR}$186=$\overline{WR}$530(HIGH) [U2 Tpd(max) + U3 Tpd(max) + U4 Tpd(max)]

= 35 + 1.5 − [5.5 + 3 + 7.1] = 20.9 ns margin

5. **tCHW:** Chip Select inactive high to $\overline{WR}$ active low set up time. The 82530 spec is 5 ns.

* 1 Tclcl + Tchcl(min) + Tcvctv(min) − Tchcsx(max) − U2 Tpd(max) + Tpd$\overline{WR}$186=$\overline{WR}$530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)] − tCHW

= 125 + 55 + 5 − 35 − 5.5 + [125 −5 + 1 + 4.4] − 5 = 264 ns margin

6. **tWW:** $\overline{WR}$ active low pulse. 82530 requires a minimum of 60 ns from the falling to the rising edge of $\overline{WR}$. This includes one wait state.

* Twlwh [2Tclcl − 40] + 1 (Tclclwait state) − TpdWR/186−WR530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(max) + U4 Tpd(max)] + TpdWR/186=WR/530(HIGH) [U2 Tpd(min) U3 Tpd(min) + U4 Tpd(min)] − tWW

= 210 + 1(125) − [125 − 5 + 4.5 + 9.2] − [1.5 + 1 + 3.2] − 60 = 135.6 ns margin

7. **tDW:** Data valid to WR active low setup time. The 82530 spec requires 0 ns.

* Tcvctv(min) − Tcldv(max) − '245 Tpd(max) + TpdWR186−WR530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)]

= 5 − 44 − 14.2 + 125 − 5 + 1.0 + 4.4 = 72.2 ns margin

8. **tWD:** Data valid to WR inactive high hold time. The 82530 requires a hold time of 0 ns.

* Tclch − skew {Tcvctx(max) + Tcvctx(min)} + '245 OE Tpd(min) − TpdWR186−WR530(HIGH) [U2 Tpd(max) + U3 Tpd(max) + U4 Tpd(max)]

= 55 − 5 + 11.25 − [5.5 + 3.0 + 7.1] = −50.6 ns margin

INTA Cycle:

1. **tIC:** This 82530 spec implies that the INTA signal is latched internally on the rising edge of CLK (82530). Therefore the maximum delay between the 80186 asserting INTA active low or inactive high and the 82530 internally recognizing the new state of INTA is the propagation delay through U1 plus the 82530 CLK period.

* U1 Tpd(max) + 82530 CLK period

= 45 + 250 = 295 ns

2. **tCI:** rising edge of CLK to INTA hold time. This spec requires that the state of INTA remains constant for 100 ns after the rising edge of CLK. If this spec is violated any change in the state of INTA may not be internally latched in the 82530. tCI becomes critical at the end of an INTA cycle when INTA goes inactive. When calculating margins with tCI, an extra 82530 CLK period must be added to the INTA inactive delay.

3. **tIW:** INTA inactive high to WR active low minimum setup time. The spec pertains only to 82530 WR cycle and has a value of 55 ns. The margin is calculated assuming an 82530 WR cycle occurs immediately after an INTA cycle. Since the CPU cycles following an 82530 INTA cycle are devoted to locating and executing the proper interrupt service routine, this condition should never exist. 82530 drivers should insure that at least one CPU cycle separates INTA and WR or RD cycles.

4. **tWI:** WR inactive high to INTA active low minimum hold time. The spec is 0 ns and the margin assumes CLK coincident with INTA.

* Tclcl − Tcvctx(max) − TpdWR186 − WR530(HIGH) [U3 Tpd(max) + U4 Tpd(max)] + Tcvctv(min) + U1 Tpd(min)

= 125 − 55 − [5.5 + 3 + 7.1] + 5 + 10 = 69.4 ns margin

5. **tIR:** INTA inactive high to RD active low minimum setup time. This spec pertains only to 82530 RD cycles and has a value of 55 ns. The margin is calculated in the same manner as tIW.

6. **tRI:** RD inactive high to INTA active low minimum hold time. The spec is 0 ns and the margin assumes CLK coincident with INTA.

* Tclcl − Tclrh(max) − 2 U2 Tpd(max) + Tcvctv(min) + U1 Tpd(min)

= 125 − 55 − 2(5.5) + 5 + 10 = 74 ns margin

7. **tIID:** INTA active low to RD active low minimum setup time. This parameter is system dependent. For any SCC in the daisy chain, tIID must be greater than the sum of tCEQ for the highest priority device in the daisy chain, tEI for this particular SCC, and tEIEO for each device separating them in the daisy chain. The typical system with only 1 SCC requires tIID to be greater than tCEQ. Since tEI occurs coincidently with tCEQ and it is smaller it can be neglected. Additionally, tEIEO does not have any relevance to a system with only one SCC. Therefore tIID > tCEQ = 250 ns.

* 4 Tclcl + 2 Tidle states − Tcvctv(max) − tIC [U1 Tpd(max) + 82530 CLK period] + Tcvctv(min) + U5 Tpd(min) + U2 Tpd(min) − tIID

= 500 + 250 − 70 − [45 + 250] + 5 + 6 + 2 − 250 = 148 ns margin

8. **tIDV:** RD active low to interrupt vector valid delay. The 80186 expects the interrupt vector to be valid on the data bus a minimum of 20 ns before T4 of the second acknowledge cycle (Tdvcl). tIDV spec is 100 ns maximum.

* 3 Tclcl − Tcvctv(max) − U5 Tpd(max) − U2 Tpd(max) − tIDV(max) − '245 Tpd(max) − Tdvcl(min)

= 375 − 70 − 25 − 5.5 − 100 − 14.2 − 20 = 140.3 ns margin

9. **tII:** $\overline{RD}$ pulse low time. The 82530 requires a minimum of 125 ns.

* 3 TclcI − Tcvctv(max) − U5 Tpd(max) − U2 Tpd(max) + Tcvctx(min) + U5 Tpd(min) + U2 Tpd(min) − tII(min)

= 375 − 70 − 25 − 5.5 + 5 + 6 + 1.5 − 125 = 162 ns margin

DMA Cycle

Fortunately, the 80186 DMA controller emulates CPU read and write cycle operation during DMA transfers. The DMA transfer timings are satisfied using the above analysis. Because of the 80186 DMA request input requirements, two wait states are necessary to prevent inadvertent DMA cycles. There are also CPUDMA intracycle timing considerations that need to be addressed.

1. **tDRD:** $\overline{RD}$ inactive high to $\overline{DTR}$REQ (REQUEST) inactive high delay. Unlike the $\overline{READY}$REQ signal, $\overline{DTR}$REQ does not immediately go inactive after the requested DMA transfer begins. Instead, the $\overline{DTR}$REQ remains active for a maximum of 5 tCY + 300 ns. This delayed request pulse could trigger a second DMA transfer. To avoid this undesirable condition, a D Flip Flop is implemented to reset the $\overline{DTR}$REQ signal inactive low following the initiation of the requested DMA transfer. To determine if back to back DMA transfers are required in a source synchronized configuration, the 80186 DMA controller samples the service request line 25 ns before T1 of the deposit cycle, the second cycle of the transfer.

* 4 TclcI − Tclcsv(max) − U4Tpd(max) − Tdrqcl(min)

= 500 − 66 − 10.5 − 25 = 398.5 ns margin

2. tRRI: 82530 $\overline{RD}$ active low to $\overline{REQ}$ inactive high delay. Assuming source synchronized DMA transfer, the 80186 requires only one wait state to meet the tRRI spec of 200 ns. Two are included for consistency with tWRI.

* 2 TclcI + 2($\overline{Tclcl}$wait state) − TclrI(max) − 2(U2) Tpd(max) − Tdrqcl − tRRI

=2(125) + 2(125) − 70 − 2(5.5) − 200 = 219 ns margin

3. tWRI: 82530 $\overline{WR}$ active low to $\overline{REQ}$ inactive high delay. Assuming destination synchronized DMA transfers, the 80186 needs two wait states to meet the tWRI spec. This is because the 80186 DMA controller samples requests two clocks before the end of the deposit cycle. This leaves only 1 TclcI + n(wait states) minus $\overline{WR}$ active delay for the 82530 to inactivate its $\overline{REQ}$ signal.

* TclcI + 2($\overline{Tclcl}$wait state) − Tcvctv(min) − Tpd$\overline{WR}$186 − $\overline{WR}$530(LOW) [TclcI − Tcvctv(min) + U3 Tpd(max) + U4 Tpd(max)] − Tdrqcl − tWRI

=375 − 5 − [125 − 5 + 4.5 + 9.2] − 25 − 200 = 11.3 ns margin

NOTE:
If one wait state DMA interface is required, external logic, like that used on the $\overline{DTR}$REQ signal, can be used to force the 82530 $\overline{REQ}$ signal inactive.

4. **tREC:** CLK recovery time. Due to the internal data path, a recovery period is required between SCC bus transactions to resolve metastable conditions internal to the SCC. The DMA request lines are marked from requesting service until after the tREC has elapsed. In addition, the CPU should not be allowed to violate this recovery period when interleaving DMA transfers and CPU bus cycles. Software drivers or external logic should orchestrate the CPU and DMA controller operation to prevent tREC violation.

Reset Operation

During hardware reset, the system RESET signal is asserted high for a minimum of four 80186 clock cycles (1000 ns). The 82530 requires $\overline{WR}$ and $\overline{RD}$ to be simultaneously asserted low for a minimum of 250 ns.

* 4 TclcI − U3 Tpd(max) − 2(U2) Tpd(max) + U4 Tpd(min) − tREC

= 1000 − 17.5 − 2(5.5) + 3.5 − 250 ns = 725 ns margin

intel®

APPLICATION NOTE

AP-324

June 1989

Implementing Twisted Pair Ethernet with the Intel 82504TA, 82505TA, and 82521TA

WILLIAM WAGER
TECHNICAL MARKETING ENGINEER

Order Number: 292057-001

ABSTRACT

The market for Local Area Networks (LANs) has been growing rapidly for several years, and LANs based on the ANSI/IEEE 802.3—1985 standard have proven to be the most popular. These networks are called CSMA/CD LANs because of their Medium Access Control method (MAC)—Carrier Sense Multiple Access with Collision Detection. Intel has been a contributor to both the standardization and the widespread acceptance of CSMA/CD LANs since their conception.

The two most prevalent types of CSMA/CD LANs are called, in IEEE terminology, 10BASE5 (aka Ethernet, Yellow Cable, or Thick Wire) and 10BASE2 (Cheapernet or Thin Wire Ethernet). Ethernet operates over a customized coaxial cable configured as a bus and restricted to a maximum length of 500 meters—point-to-point. Ethernet transmits data at 10 Mb/s on a baseband network. Cheapernet uses the more common RG-58 cable and has a maximum point-to-point distance of 185 meters; its data transmission rate is also 10 Mb/s. Other types of CSMA/CD networks are 10BROAD36 (10-Mb/s Broadband, 3600m on Coax) and 1BASE5 (1 Mb/s Baseband, 500m on standard telephone wire).

The cost of the cable and its installation and reconfiguration has been a factor in the acceptance of CSMA/CD LANs. The members of the IEEE 802.3 Working Group, including Intel, have recognized this, and we are addressing this issue. We are preparing a new CSMA/CD standard (10BASE-T) that operates at 10 Mb/s with a 100m point-to-point range and uses unshielded, twisted-pair wiring—the common telephone wire already installed in most buildings. Besides using a less expensive wire type, 10BASE-10 (TPE) uses a star topology that can operate concurrently with normal telephone traffic, and other services, in a parallel cable plant.

Besides its active participation in the 10BASE-T Task Force, Intel is now marketing products based on the work of the task force. With these products—the 82504TA Transceiver Serial Interface, the 82505TA Multiport Repeater controller, and the 82521 Serial Supercomponent—our customers can design high-speed LANs that operate over unshielded twisted pair wiring, which is usually already installed. These networks can coexist with existing CSMA/CD networks; that is, they can be integrated into a single network interfacing with already installed Ethernet or Cheapernet networks. Furthermore, Intel is committed to maintaining compatibility and conformity with the emerging standard.

1.0 INTRODUCTION

This Ap Note is intended to aid system designers who have some knowledge of IEEE 802.3 standards, but limited experience with analog design. System designers designing Twisted Pair Ethernet LANs with Intel's TPE products and Ethernet LAN controllers will find this and other Intel Ap Notes useful (see also: AP-274, *Implementing Ethernet/Cheapernet with the Intel 82586,* Kiyoshi Nishide; and AP-320, *Using the Intel 82592 to Integrate a Low-Cost Ethernet Solution into a PC Motherboard,* Michael Anzilloti).

Intel has introduced the 82504TA Transceiver Serial Interface (TSI), the 82505TA Multiport Repeater controller (MPR), and the 82521 Serial Supercomponent (SSC). These products simplify designing Twisted Pair Ethernet LANs based on the emerging 10BASE-T standard. These LANs are compatible with existing ANSI/IEEE 802.3 networks at the Physical Signaling layer and the MAC portion of the Data Link layer. This means that a Twisted Pair Ethernet LAN built with these products will be software compatible with current 802.3 networks and can connect to other 802.3 networks through the standard Attachment Unit Interface (AUI) port of a Multiport Repeater.

A Twisted Pair Ethernet LAN comprises several elements: data terminal equipment (DTE), medium attachment units (MAU), multiport repeaters (MPR), and the cable plant. More complex networks, which interconnect with existing 802.3 networks, are made possible by using the 802.3-standard AUI port of the MPR. Figure 1 illustrates a network that uses all these elements.

Figure 1 shows three types of DTE and MAU combinations. Two have embedded MAUs, the other has an external MAU connected to the DTE node by a standard AUI cable. Embedded MAU designs either use the 82504TA, and its associated circuitry, or the 82521TA SSC. The multiport repeaters are designed around the 82505TA, they also contain one 82504TA. Each of the eleven twisted pair ports contains an embedded MAU. The cable plant is standard telephone wire, 4- or 25-pair unshielded twisted pair (26 to 22 gauge). Each segment uses two twisted pairs for data, one for transmission and one for reception, and the unused pairs can carry other services as well. In a TPE design the maximum node-to-repeater distance is 100 meters.

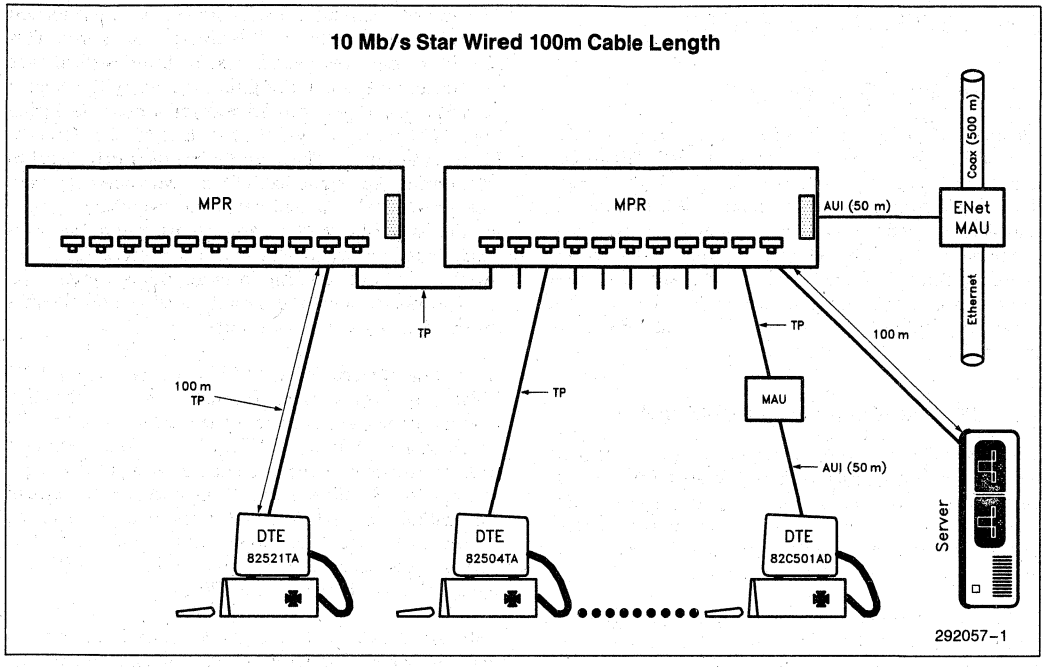


Figure 1. Typical TPE Network

2.0 SYSTEM DESCRIPTION

2.1 Network Description

The network shown in Figure 1 is a typical representation of TPE networks designed with Intel TPE products. The network follows the 10BASE-T draft standard specifications wherever possible. We recommend that network designers follow the same practice. Table 1 compares the TPE network features to the earlier 10 Mb/s standards, and Table 2 compares TPE networks based on the Intel products to those based on the most likely outcome of the 10BASE-T Task Force deliberations.

2.1.1 MEDIUM ATTACHMENT UNIT (MAU)

The MAU (i.e., the transceiver) provides the required circuitry for interfacing with the twisted pair wire. It performs several functions; e.g., line driving with predistortion, line reception, and collision detection. Multiport repeaters and DTEs can contain embedded MAUs or attach to external MAUs.

MAU Line Drivers: The transmitter is designed to drive a 96Ω properly terminated cable and must meet all its specifications under this load (unless otherwise specified). A transformer provides dc isolation from the twisted pair, and the transmitter has a matched source impedance of 96Ω ±20%. It will achieve a drive level of 2.2V to 2.8V peak differential. The power spectrum amplitude will be less than −30 dB at, or above, 30 MHz from its 10 MHz value. The signal is Manchester encoded like 10BASE5 and 10BASE2.

The transmit circuitry incorporates the predistortion algorithm adopted by the 10BASE-T Task Force. This algorithm improves overall system jitter performance by reducing the amount of jitter induced by the twisted pair. The line drivers will drive at full amplitude during "thin" (50 ns) pulses and the first half of "fat" (100 ns) Manchester pulses. They will reduce their drive level to 33% during the second half of "fat" Manchester pulses. This prevents the twisted pair from overcharging during the fat pulses. Without this predistortion, the overcharge would cause a delay in the zero crossing following the "fat" bit, resulting in more induced jitter. Figure 2 shows the idealized output waveform for the predistorted signal at the transmitter.

MAU Line Receiver: The MAU line receiver is also dc isolated by a transformer. It must have a matched differential impedance such that the return loss is at least 15 dB from 5 MHz to 10 MHz. The line receiver must operate properly in the presence of a signal having a 350 mV to 2.8V differential. It must detect the start of Idle within 1.8 bit times, and must include a squelch circuit that rejects, as noise, any signals less than 250 mV, and accepts signals greater than 350 mV having a pulse width greater than 20 ns.

Collision Detection: The MAU detects collision by noting simultaneous activity on the transmit and receive pair. No provision is made for receive-based collision detection. When a transmitting station detects a collision it begins the normal 802.3 collision sequence of jam, random backoff, and retransmit. When a repeater detects a collision it also begins a jam and it enforces the minimum frame length of 96 bits.

Table 1. Comparison of Network Features

Feature	TPE	10BASE5	10BASE2
Wire	Unshielded TP	Yellow Coax	Thin Coax
Topology	Star	Bus	Bus
Segment Length	100m	500m	185m
Software	Existing	Existing	Existing
Controller	82586/8259x	82586/8259x	82586/8259x
Data Rate	10 Mb/s	10 Mb/s	10 Mb/s
Access Method	CSMA/CD	CSMA/CD	CSMA/CD

Table 2. Differences between Current TPE and Expected 10BASE-T

Feature	Current TPE	10BASE-T
Squelch	Single Pulse	Multiple Pulse
Collision Detect	Tx and Rx Active	Tx and Rx Active for 5 Bits
Link Integrity	None	Single Linkbeat
Jabber Function	None	Watchdog Timer
DO → DI Loopback	None	Supported

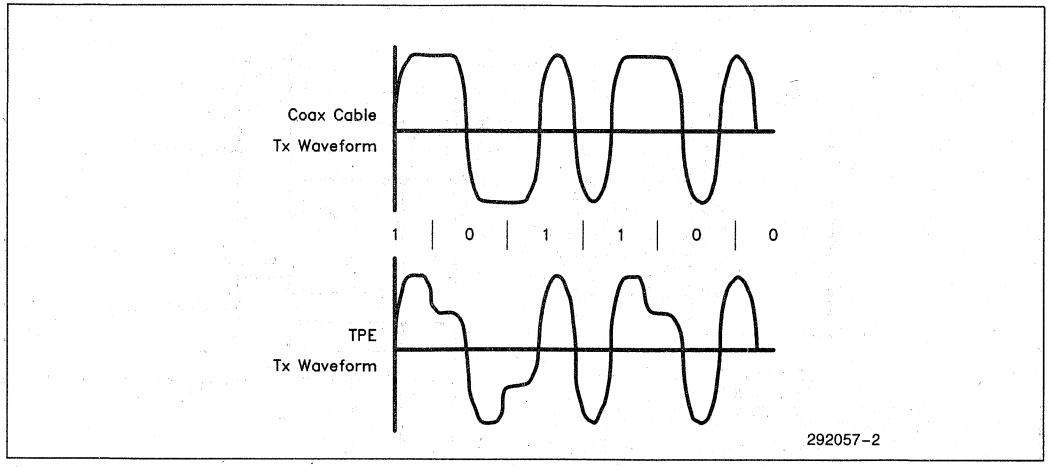


Figure 2. Predistortion Waveform

2.1.2 MULTIPORT REPEATER

The Multiport Repeater is the central point in the star-configured network. It is usually located in a telephone closet or some other central wiring point. The link segments (repeater to node wiring) can then be run using available twisted pairs in the existing telephone cable plant or a dedicated parallel cable plant. The repeater conforms to the ANSI/IEEE 802.3c—1988 standard for repeaters. It has eleven twisted-pair ports (embedded MAUs) and one AUI port.

A block diagram of an 82505TA-based repeater is shown in Figure 3. It uses one 82505TA, one 82504TA, eleven TP port processors, two 74LS529 latches, a 74LA154 decoder, and an AUI interface processor. The 82505TA handles the repeater state functions such as automatic preamble regeneration, minimum frame length enforcement, signal retiming, collision detection and jam, and control for the LED status indicators. The 82504TA handles Manchester decoding and clock recovery for the incoming data packet. The AUI interface processor contains the DO line drivers and the DI and CI line receivers as required by the ANSI/IEEE 802.3—1985 standard for AUI connectors. The 72LS154 decoder disables the transmitter on the receiving port, and the 72LS259 latches control the status LEDs.

During normal transmission without contention (i.e., no collisions) the repeater detects the transmitting port and immediately begins automatic preamble regeneration (APR) to all other ports. It routes the incoming data to the Manchester decoder and begins loading its internal FIFO. When the FIFO reaches its threshold the repeater ceases APR and begins to send data from the FIFO. This data is Manchester encoded and retimed before it is rebroadcast. When a collision occurs, the repeater stops broadcasting from the FIFO and begins transmitting a jam pattern. It continues to jam until the collision ceases and at least 96 bits have been transmitted to each port (minimum frame length enforcement).

The repeater also supports autopartitioning and jabber protection. These two features prevent faulty nodes from bringing the network down. When such a fault is detected, the port in question is removed from the network, and the remainder of the network resumes normal operation. The repeater continually monitors the faulty port, and when the fault is fixed the port is reconnected to the network.

2.1.3 DATA TERMINAL EQUIPMENT

Data Terminal Equipment (DTE) includes usernodes, file servers, and other devices that can originate and accept data packets. A TPE network uses the same controllers as other 802.3 networks. These are Intel's 82586, 82590, and 82592, as well as any future Intel Ethernet controllers. This ensures a design continuity that allows for migration from Ethernet or Cheapnet designs to Twisted Pair Ethernet. The only part of the design that requires redesign is that between the controller and the connector.

Intel's product line supports two DTE designs. Over the twisted pair they are functionally equivalent; however, they differ in the way they interface to the host

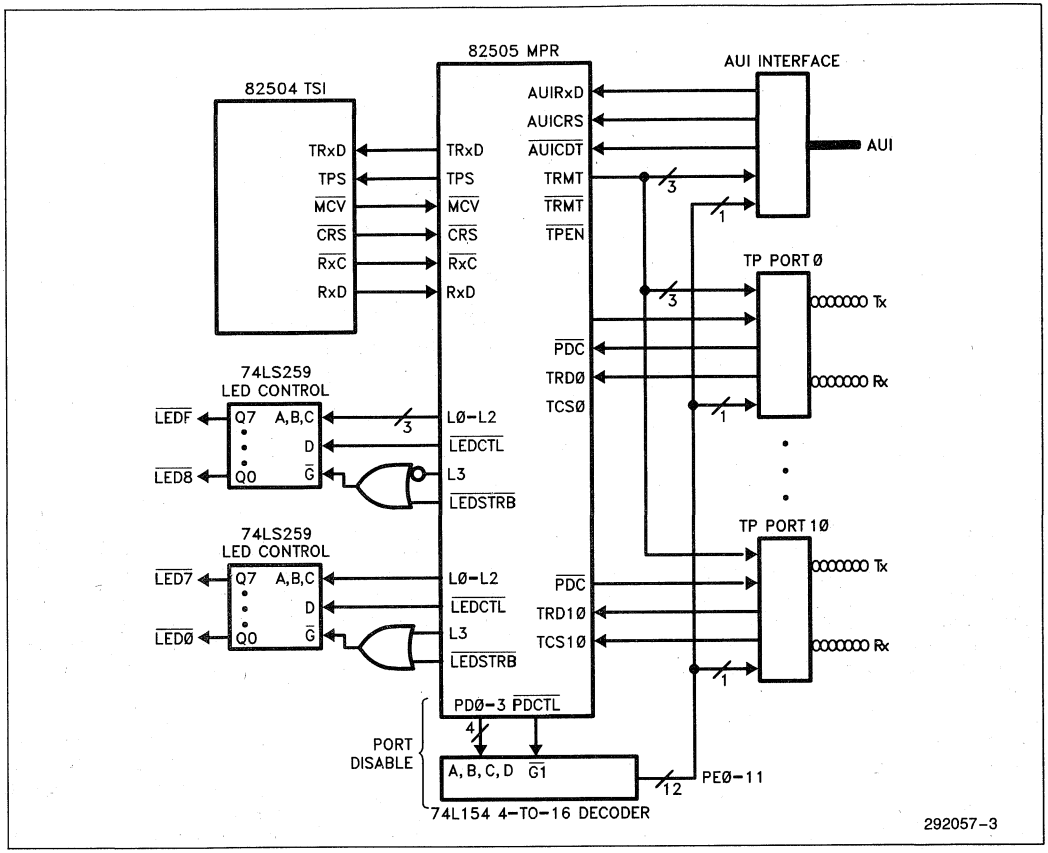


Figure 3. Repeater Block Diagram

LAN controller. Figure 4 shows the first DTE design. It is based on the 82504TA, and comprises the 82504TA, the interface logic, the twisted pair transmitters, and the twisted pair receivers. This circuit contains an embedded MAU; i.e., it connects directly to the twisted pair wire. The second DTE design, shown in Figure 5, also contains an embedded MAU. It is built around the 82521TA Serial Supercomponent and interfaces directly with the LAN controller and the twisted pair. The complete twisted pair design consists of an 82521TA and the connector. External MAUs, which interface a standard Ethernet AUI node to the twisted pair, are also allowed. External MAUs are part of Intel's future product plans.

2.1.4 LINK SEGMENT

A link segment connects two twisted pair Medium Attachment Units (MAUs); it comprises two Medium Dependent Interface connectors (RJ-45, 8-pin standard telephone connectors), two pairs of twisted pair wire (note to exceed 100m) and a crossover. The connector's pin assignments are shown in Table 3.

Table 3. Pin Assignments for MDI Connector

Pin	Signal
1	Transmit Data+ (TD+)
2	Transmit Data− (TD−)
3	Receive Data+ (RD+)
4	Not Used
5	Not Used
6	Receive Data− (RD−)
7	Not Used
8	Not Used

The crossover function connects the TD outputs of one MAU to the RD inputs of the other. This function can be implemented externally or embedded within a MAU. If the function is embedded, then the signal names on the connector refer to the remote MAU. That is Pin 1 (TD+) on a MAU with an embedded crossover is connected to the Transmit Data (+) of the remote MAU, and to its own Receive Data (+). The crossover function is defined by the following connections between MAU A and MAU B.

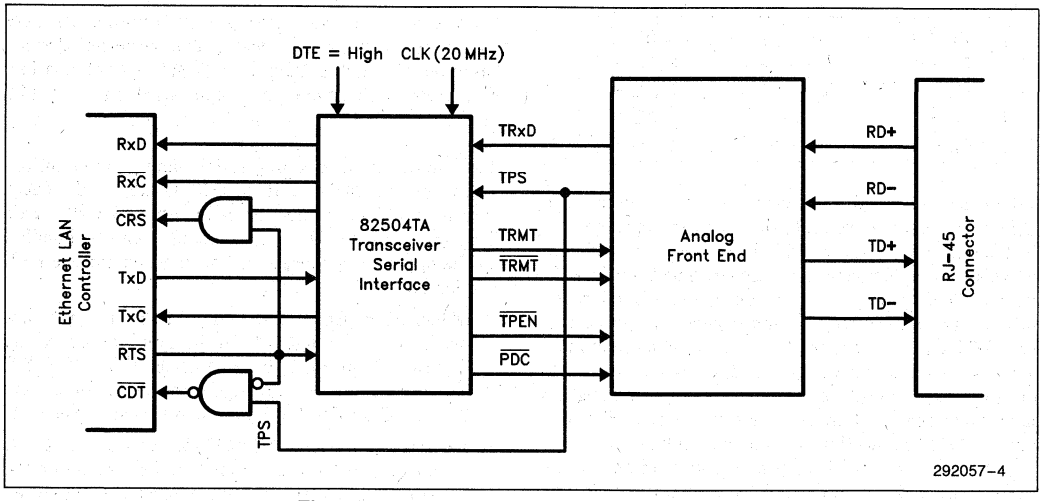


Figure 4. 82504TA Based DTE Block Diagram

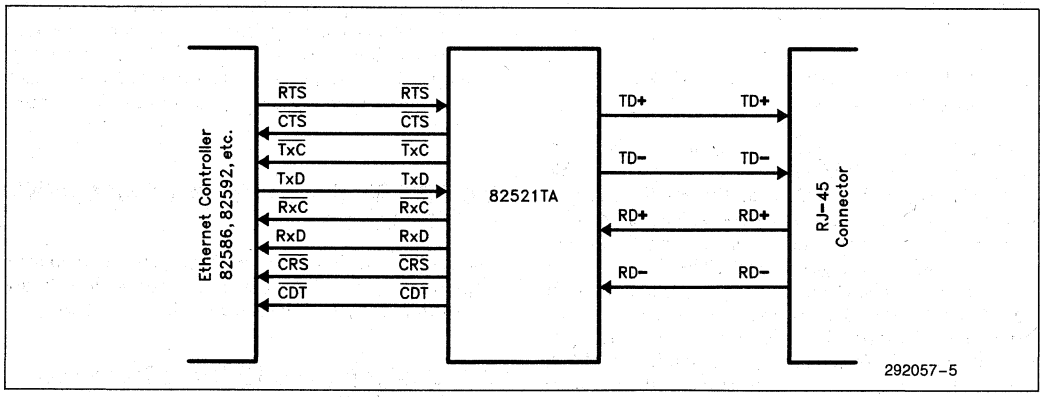


Figure 5. 82521TA Based DTE Block Diagram

MAU A		MAU B
TD+ 1	———	3 RD+
TD− 2	———	6 RD−
RD+ 3	———	1 TD+
RD− 6	———	2 TD−

When an embedded crossover function is used in a DTE to repeater connection, the crossover must be embedded in the repeater MAU. In general, repeater MAUs have an embedded crossover, and DTE MAUs do not. With proper use of the crossover function repeaters can be cascaded through twisted pair ports, and two DTEs can be connected in a point-to-point network. Repeaters can be cascaded in two ways. First, one twisted pair port on a repeater can be designed to have a switched (optional) crossover function. This enables a DTE connection on that port when the crossover is active, or a repeater connection when the crossover is disabled. Secondly, twisted pair ports with embedded crossovers can be connected by using a third external crossover.

2.2 Interoperation with Existing 802.3 Networks

Twisted Pair Ethernet networks that use Intel's Ethernet controllers and TPE products are fully compatible with existing 802.3 networks at the Medium Access Control and Physical Signaling levels. Therefore, TPE networks can be integrated with existing 802.3 networks to form one large network. The IEEE 802.3c-1988 standard allows connecting different types of 10-Mb/s networks. Because the repeater definition extends to a DTE type AUI connection on each port, the type of wiring is determined by the choice of MAU. Optionally, a repeater can have embedded MAUs on any of its ports. The only requirement is that functionality at the Medium Dependent Interface point (e.g., coax tap or twisted pair connector) be maintained.

The 82505TA Multiport Repeater provides embedded MAUs on 11 of the 12 ports, and an AUI connection on the remaining port. This allows creating local twisted pair subnetworks that are connected to an Ethernet backbone. Care must be taken not to violate the system topology rules of 802.3 networks. The most important of these are: (1) only one active signal path is allowed to exist between any two stations on the network and (2) no more than four repeaters are allowed in the signal path between any two stations on the network. There is an overall limit of 1024 stations on a network (repeaters do not count as stations).

2.3 Software Compatibility

Because the twisted pair networks use the same controller chips (82586 and 8259x) as current Ethernet and Cheapernet networks they have an implicit software compatibility. That is, Twisted Pair Ethernet designs based on the 82586 (8259x) will be software compatible with Ethernet/Cheapernet design based on the 82586 (8259x).

Two minor software configuration changes are required when the 82504TA and 82521TA are used. These changes will not be necessary in future Twisted Pair Ethernet products. Both these changes can be handled by the application-specific software driver for the TPE application.

- Manchester Encoding. The Ethernet controller needs to be configured for Manchester encoding. For the 82586, bit 2, byte 14 of the CONFIGURE command must be set to 1. For the 8259x, bit 2, byte 9 of the CONFIGURE command must be set to 1.
- External Loopback. The 82504TA and the 82521TA do not support the external loopback mode of the Ethernet controllers. Software packages that used this mode will fail without a workaround. Normally, this is only an issue for diagnostics that use this mode during self-test. This problem can be avoided by modifying the software driver to look for external loopback mode, and to do its own software loopback when appropriate (based on a destination address check).

3.0 NETWORK SYSTEM COMPONENT DESIGN

The design of various TPE network system components is presented here. DTEs with embedded MAUs are shown first, and then the repeater design is shown.

3.1 Designing a DTE Node Based on the 82504TA

Figure 4 has shown a DTE node with an embedded MAU based on the 82504TA. It showed the Ethernet LAN controller, the 82504TA, the analog front-end, and the connector. As in previous Ethernet designs, the LAN controller provides the MAC services such as transmission deferral, collision backoff and retransmission, CRC generation and checking, and address checking. It also provides the host interface. The 82504TA, in conjunction with the analog front-end, provides both the Physical Signaling and Physical Medium Attachment services. These include carrier sense, collision detect, Manchester decoding, clock recovery, line driving, and line receiving. The analog front-end handles the line driving and receiving functions from the 82504TA.

3.1.1 HOST TO ETHERNET LAN CONTROLLER

The interface of the Ethernet LAN Controller is discussed in previous Intel Ap Notes, AP-274 and AP-320.

3.1.2 82504TA TO ETHERNET LAN CONTROLLER

The 82504TA to controller interface consists of the direct connection of $\overline{TxC}$, TxD, $\overline{RxC}$, RxD, and $\overline{RTS}$. The $\overline{CRS}$ signal to the controller is generated by a logical AND (a 74F08 is used) of $\overline{CRS}$ from the 82504TA and $\overline{RTS}$. CDT is the NAND of TPS and an inverted $\overline{RTS}$; both the NAND function and invert function are done by a 74F00.

For clocking the 82504TA a clock oscillator is recommended. Many that meet the requirements of the device are available commercially. It must meet the following specifications.

Frequency Tolerance ..... ≤0.01%
Rise and Fall Times ..... ≤5 ns
Duty Cycle ..... 40/60% or better
Output ..... TTL compatible

The 82586 and the 82504TA have two specification incompatibilities. These are data sheet incompatibilities only, and will not affect performance. Work is in progress to ensure that the 82586 and 82504TA specifications are fully compatible.

- TxD setup time. The 82504TA requires a 10 ns setup time from TxD to the $\overline{TxC}$ edge. The 82586 specifies that TxD will change within 40 ns of the previous $\overline{TxC}$ edge. Therefore, if the $\overline{TxC}$ duty cycle is not exactly 50% the TxD setup time is violated. However, the 82586 actually places the TxD edge approximately 25 ns from the previous $\overline{TxC}$ edge, and it never exceeds 30 ns. The next revision of the 82586 data sheet will correct this specification problem.
- $\overline{TxC}$ duty cycle. The 82504TA does not specify the $\overline{TxC}$ duty cycle; however, the worst case would be better than 40/60% based on the high- and low-time specifications and signal rise and fall times. The 82586 data sheet requires a 45/55% duty cycle for Manchester encoding. In fact, the tight duty cycle is only important when control of the TxD duty cycle is important. The 82504TA can tolerate the worst case TxD duty cycle generated by the 82586 with a worst case $\overline{TxC}$, therefore there is no problem. The 82586 specification will become a recommendation in the next revision of the data sheet.

3.1.3 ANALOG FRONT-END

The analog front-end is shown in Figure 6. It consists of two main sections, Transmit and Receive. The transmit section contains the interface, the line drivers, the EMI filter, and the line coupling devices. The receive section consists of the line coupling devices, the EMI filter, the line receivers, and the squelch circuitry. The line coupling devices and EMI filter are similar for both the transmit and receive sections, and will be described in a common section.

The 82504TA to Line Driver Interface. The 82504TA to line driver interface consists of the four signals from the 82504TA (TRMT, $\overline{TRMT}$, $\overline{PDC}$, and $\overline{TPEN}$), a quad XOR (e.g., 74F86), and quad line drivers. The design shown here uses an octal line driver (74ACT244) with the drivers paired. The four pairs are configured using a voltage summing circuit to give two differential drivers, one at 67% power, and the other at 33%. During "thin" pulses and the first half of "fat" pulses, the two differential drivers act in unison to give 100% power. During the second half of "fat" pulses, the 33% driver is inverted to provide only 33% power as required by the predistortion algorithm.

The circuit operates as follows. The $\overline{TPEN}$ signal is the enable signal for the drivers. It is asserted by the 82504TA whenever the node is transmitting. During idle, it is deasserted, and the driver enters the tri-state mode. The Manchester data is provided by the 82504TA on the TRMT and $\overline{TRMT}$ lines, and each signal is fed into two XOR gates. One of the XORs for each signal has one input grounded, therefore it acts as a non-inverting buffer. The output of these XOR gates feeds the two line drivers composing the 67% differential driver. Therefore, the 67% driver is always driving the exact Manchester pattern. The other XOR gates are also fed by the $\overline{PDC}$ signal. This signal is low for "thin" pulses and the first half of "fat" pulses, and it is high for the second half of "fat" pulses. These XOR gates feed the 33% differential driver. The combination of the $\overline{PDC}$ signal and the XOR gates ensures that the 33% driver follows the 67% drivers when 100% power is required, and inverts when 33% power is required.

The design of this circuit is intended to present a constant driver impedance during packet transmission. This is vital since variations in the matching of driver impedance to the twisted pair cable impedance will cause reflections resulting in added jitter.

High-Voltage Protection. To prevent damage to the active devices caused by high-voltage transients from the twisted pair line, protection should be provided. We recommend placing a pair of diodes on each of the four differential signals (two transmit and two receive) as shown in Figures 7 and 8. The diodes connect to the

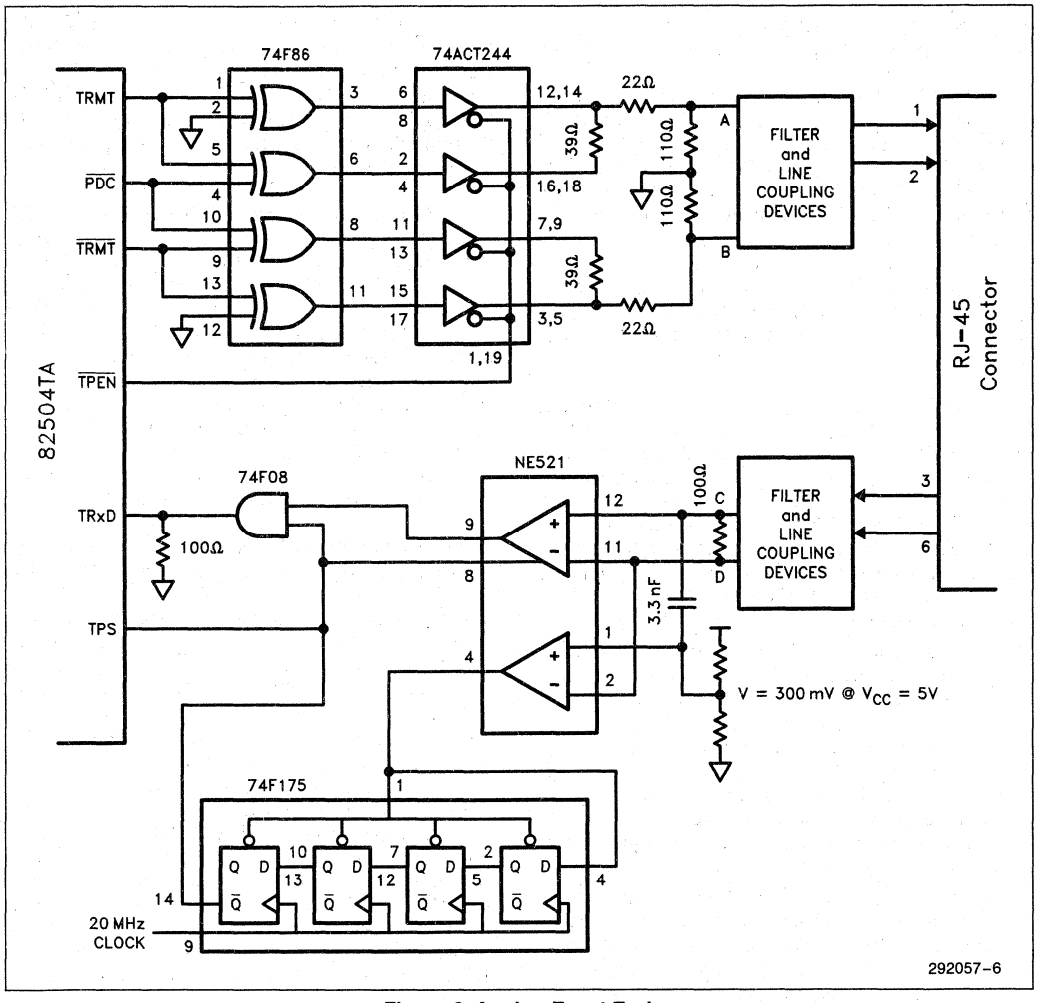


Figure 6. Analog Front End

±5V power supplies. These should be placed at the interface between the active devices and the low pass filters so the active circuits are protected and the filter attenuates the transients.

Filter Design. The main function of the low-pass filter is to remove the high-frequency components of the transmitted signal without affecting the in-band frequencies (5 MHz to 10 MHz). The high frequency components can create electromagnetic interference (EMI) above the levels permitted by FCC regulations. The design should provide minimum inband loss and minimum-in-band ripple while providing maximum attenuation of frequencies above 30 MHz with appropriate roll-off in the transition band.

The Group Delay variation is another critical factor in the design of the filter. The group delay is defined as the variation in signal phase with the frequency. The group delay variation is the derivative of the group delay. The group delay variation defines the difference in propagation delay through the filter for the frequencies of interest. These differences in propagation delay cause amplitude and phase distortions in the signal, which translate into jitter.

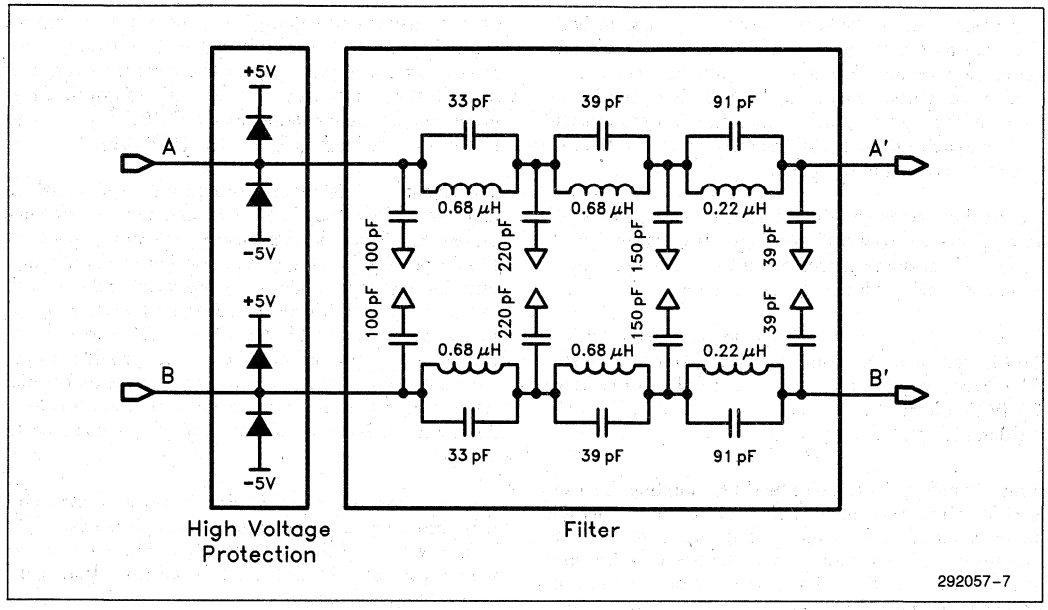


Figure 7. Tx Filter Section

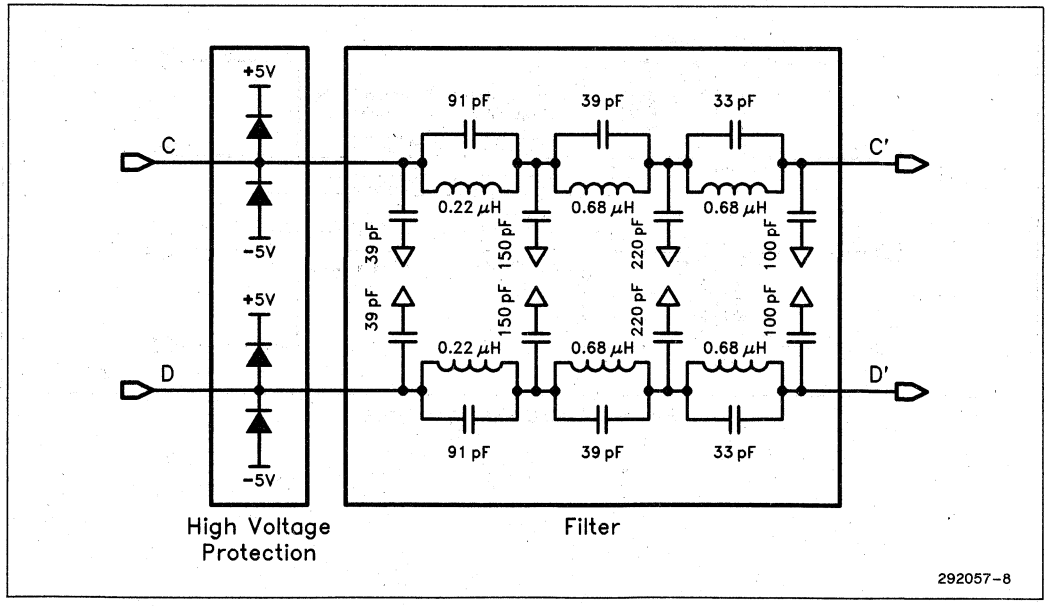


Figure 8. Rx Filter Section

The impedance of the filter must be matched to both the transmitter impedance and the line impedance. Also, balance and grounding should be tightly controlled for proper operation. Due to these considerations we recommend a differential filter built symmetrically on each line of the differential pairs with the impedance matched at each end.

Filters that provide all these characteristics are presented in Figures 7 (transmit) and 8 (receive). Their characteristics meet the requirements of the twisted pair environment. The requirements are as follows.

Type ..... 7 Pole, Balanced Elliptical
I/O Impedance ..... 96Ω ± 15% (5 MHz to 10 MHz)
3 dB Frequency ..... 17 MHz to 19 MHz
50 dB Frequency ..... ≥ 30 MHz
In-Band Ripple ..... ≤ 1 dB

Line Coupling Devices. The line coupling devices, shown in Figure 9, include the transformers, common mode chokes, and common mode noise filters. The transformers provide ac coupling between the line and the circuitry while providing dc isolation. The recommended minimum isolation is 2250 V_{dc}. To provide proper balance between the two ends of the transformers, the windings should be identical. To provide appropriate impedance matching in the frequency range of interest, the transformers should have appropriate primary and secondary inductance (200 μH typical) and minimal interwinding capacitance (<20 pF).

The common mode choke is provided to reject common mode radio frequency and electromagnetic interference picked up from the unshielded telephone lines. It should provide 1000 V_{dc} isolation between the windings. The common mode choke has four windings, each one connected with proper polarity, in series with the receive and transmit twisted pairs. The balance of the choke is very important in order to provide proper noise cancellation while passing through the differential signal unaffected. We recommend a common mode to differential balance of 30 dB at all frequencies up to 20 MHz.

The common mode noise filter removes undesirable high-frequency common mode signals picked up on the line, or generated by the transmitter. These signals are mainly generated by fast rise and fall times and signal crosstalk in the transmitter.

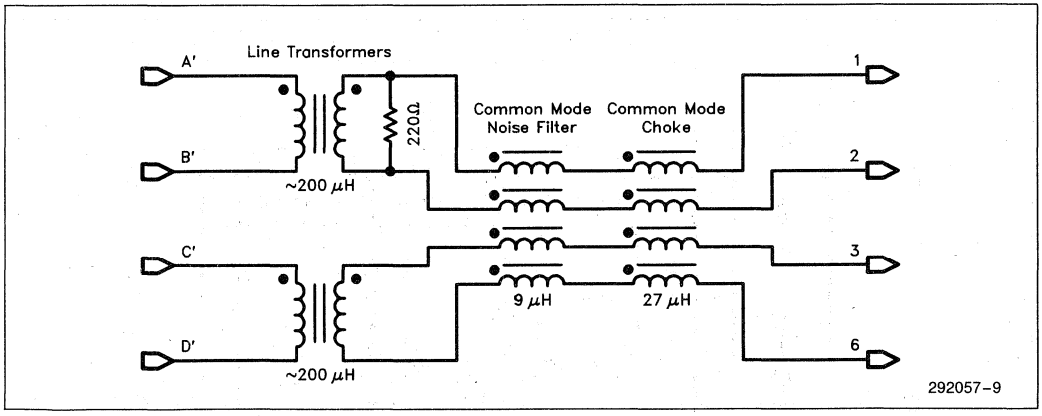


Figure 9. Line Coupling Devices

Line Receivers. The incoming receive signal passes through the line coupling devices and the low pass filter. From there it is fed into a gated line receiver controlled by the squelch circuitry. The line receiver converts the received differential signal to TTL levels and feeds it to the 82504TA. The receiver can be designed using a zero crossing detector (e.g., NE521) and gated with the TPS signal with a 74F08. A 100Ω load resistor is placed on the 74F08 output to reduce jitter induced by the difference in the threshold mismatch between the 82504TA input and the 74F08 output circuits.

Squelch Circuit. The squelch circuit differentiates noise from valid incoming data on the receive pair. It does this by detecting signals that are above a preset voltage level for a sufficient period. When there is no signal on the receive pair, the squelch circuit disables the line receiver, and deasserts the TPS signal to the 82504TA. When a signal above the threshold arrives, TPS is asserted, and the line receiver is enabled. The squelch circuit ensures that the receive circuits in the 82504TA are operating only during packet reception. The squelch circuit should meet the following specifications.

Reject ..... <250 mV
Accept ..... >350 mV and >30 ns

The circuit shown in Figure 6 uses a high-speed comparator with an offset threshold. The output of this comparator is fed to a retriggerable timing circuit that controls the TPS signal to the 82504TA. To ensure recognition of the IDL (end of packet) signal, and to prevent midpacket deassertion of TPS, the timing circuit should be set to detect positive pulses between 1.5 and 2.0 bit times (200 ns). The timing circuit can be implemented by using either a quad flip-flop (74F175) clocked from the 20 MHz clock generator or a retriggerable monostable multivibrator with an appropriate time constant. The first method provides better stability and requires fewer discrete components. If the multivibrator is used, then the selection of the timing components is critical. The timing capacitor must have very low leakage with good temperature and aging stability. The timing capacitor and resistor need to be as close as possible to the IC to minimize stray capacitance and noise injection.

Layout Considerations. The power and ground wiring should conform to good high-frequency practice and standards to minimize switching transients and parasitic interaction between various circuits. To achieve this, the following guidelines are presented.

- Place bypass capacitors (usually 0.01 μF) on each IC between V_{CC} and ground. They should be located close to the V_{CC} pins.
- Make power supply and ground traces as thick as possible. This will reduce high-frequency cross coupling caused by the inductance of thin traces.
- Separate and decouple all of the analog and digital power supply lines.
- Close signal paths to ground as close as possible to their sources to avoid ground loops and noise cross coupling.
- Connect all unused IC inputs (except as directed by the manufacturer) to ground or V_{CC} to avoid noise injection or parasitic oscillations of unused circuits.
- Use high-loss magnetic beads on power supply distribution lines.
- Group each of the receive and transmit circuits, but keep them separate from each other. Separate their grounds.
- Lay out all differential circuits symmetrically so parasitic effects are also symmetrical.
- Lay out the circuitry from the line connector to the active circuitry (especially the EMI filter) on a ground plane to prevent undesirable EMI effects.

3.2 Designing a Simplified DTE Node Based on the 82521TA Serial Supercomponent

A design for an 82521TA based DTE node within embedded MAU is shown in Figure 5. It includes all of the functions described in Section 3.1, thereby relieving the designer of those responsibilities. It is simple to use, and it does not require mastering of pole-zero diagrams. It is a direct interface from the Ethernet controller to the RJ-45 connector.

Currently, the 82521TA has the same specification incompatibilities with the 82586 as the 82504TA does, and these will be resolved concurrently. There is one added signal, Clear to Send ($\overline{CTS}$), its implementation is optional.

The layout of the 82521TA and the RJ-45 connector should keep the TD+, TD, RD+, and RD signal lines as short as possible. The power supply traces (V_{CC}, V_{EE}, V_{DD}, and ground) should be as thick as possible, and bypass capacitors should be placed between each power supply and ground. We also recommend laying out the 82521TA on a ground plane.

3.3 Designing a Multiport Repeater Using the 82505TA

Figure 3 shows the multiport repeater based on the 82505TA (with one 82504TA). The repeater contains 11 twisted pair ports with embedded MAUs and 1 AUI port. The 82505TA controls the operation of the repeater in accordance with ANSI/IEEE 802.3c—1988 repeater unit specifications; this includes signal retiming, automatic preamble generation, autopartitioning, and jam signal generation. The 82504TA performs Manchester decoding and clock recovery during an active incoming signal. Two addressable latches (74LS259) are used to control the 16 LED indicators. A 4-to-16 decoder (74LS154) is used to disable the transmitter of the receiving port during transmission without contention. The Twisted Pair port functions contain the line drivers, the line receivers, the filter, and the isolation required for a twisted pair embedded MAU. In addition, one AUI interface is present to provide access to existing (IEEE 802.3) 10 Mb/s baseband segments.

3.3.1 82505TA TO 82504TA INTERFACE AND CLOCK GENERATION

The 82505TA to 82504TA interface is straightforward. It consists of six signals directly connected between the devices. The signals are TRxD, TPS, $\overline{\text{MCV}}$, $\overline{\text{CRS}}$, $\overline{\text{RxC}}$, and RxD. The interface is shown in Figure 3.

A single clock oscillator is recommended for clocking the 82505TA and 82504TA. The requirements are identical to those shown for the DTE design using the 82504TA. They are:

Frequency Tolerance ..... ≤0.01%
Rise and Fall Times ..... ≤5 ns
Duty cycle ..... 60/40% or better
Output ..... TTL compatible

3.3.2 TWISTED PAIR PORT DESIGN

The design of the twisted pair port circuits is nearly identical to the analog front-end circuits of the DTE design based on the 82504TA. It is shown in Figure 10. The design consists of two mian sections, transmit and receive. The transmit section contains the interface circuits, the line drivers, the EMI filter, and the line coupling devices. Conversely, the receive section consists of line coupling devices, an EMI filter, line receiver, squelch circuit, and interface circuits. The line coupling devices and noise filter are similar for both the transmit and receive sections, and will be described in a common section.

The 82505TA to Line Driver Interface. The 82505TA to line driver interface consists of the four signals from the 82505TA (TRMT, $\overline{\text{TRMT}}$, $\overline{\text{PDC}}$, and $\overline{\text{TPEN}}$), the port enable (PEx) signal from the port disable control, two NAND gates, a quad XOR (e.g., 74F86), and quad line drivers. The design shown here uses an octal line driver (74ACT244) with the drivers paired. The four pairs are configured using a voltage summing circuit to give two differential drivers, one at 67% power, and the other at 33%. During "thin" pulses and the first half of "fat" pulses, the two differential drivers act in unison to give 100% power. During the second half of "fat" pulses, the 33% driver is inverted to provide only 33% power as required by the predistortion algorithm.

The circuit operates as follows. The $\overline{\text{TPEN}}$ signal is inverted and NAND'd with the individual port's Port Enable signal. This generates the enable signal for that port's drivers. It is asserted whenever the port is transmitting; i.e., when another port is receiving, or during a collision jam. During idle it is deasserted, and the drivers enter the tri-state mode. The Manchester data is provided by the 82505TA on the TRMT and $\overline{\text{TRMT}}$ lines, and each signal is fed into two XOR gates. One of the XORs for each signal has one input grounded, therefore it acts like a non-inverting buffer. The output of these XOR gates feeds the two line drivers composing the 67% differential driver. Therefore, the 67% driver is always driving the exact Manchester pattern. The other XOR gates are also fed by the $\overline{\text{PDC}}$ signal. This signal is low for "thin" pulses and the first half of "fat" pulses, and it is high for the second half of "fat" pulses. These XOR gates feed the 33% differential driver. The combination of the $\overline{\text{PDC}}$ signal and the XOR gates ensures that the 33% driver follows the 67% drivers when 100% power is required, and inverts when 33% power is required.

The design of this circuit is intended to present a constant source impedance during packet transmission. This is vital since variations in the matching of driver impedance to the twisted pair cable impedance will cause reflections resulting in added jitter.

High-Voltage Protection. To prevent damage to the active devices due to high voltage transients from the twisted pair line, high-voltage protection should be provided. We recommend placing a pair of diodes on each of the four differential signals (two transmit and two receive) as shown in Figures 7 and 8. The diodes connect to the ±5V power supplies. These should be placed at the interface between the active devices and low pass filters so that the active circuits are protected, and the filter attenuates the transients.

Filter Design. The main function of the low-pass filter is to remove the high-frequency components of the transmitted signal without affecting the in-band

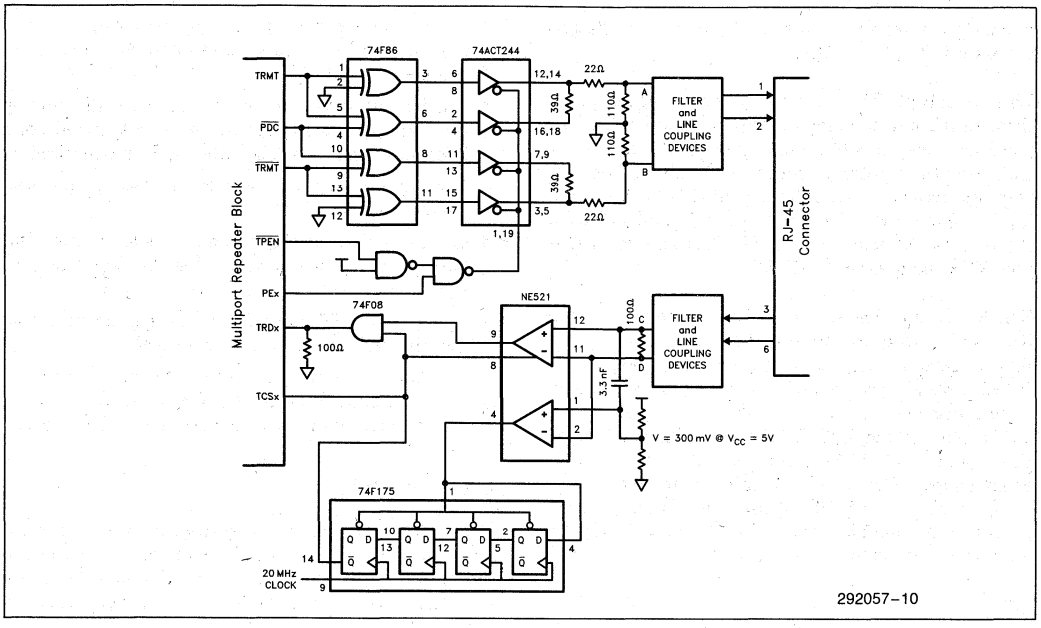


Figure 10. TP Port x

(5 MHz to 10 MHz) frequencies. The high frequency components can create electromagnetic interference (EMI) above the levels permitted by FCC regulations. The design should provide minimum in-band loss and minimum in-band ripple while providing maximum attenuation of frequencies above 30 MHz with appropriate roll-off in the transition band.

The Group Delay variation is another critical factor in the design of the filter. The group delay is defined as the variation in signal phase with the frequency. The group delay variation is the derivative of the group delay. The group delay variation defines the difference in propagation delay through the filter for the frequencies of interest. These differences in propagation delay cause amplitude and phase distortions in the signal, which translate into jitter.

The impedance of the filter must be matched to both the transmitter impedance and the line impedance. Also, balance and grounding should be tightly controlled for proper operation. Due to these considerations we recommend a differential filter built symmetrically on each line of the differential pairs with the impedance matched at each end.

Filters that provide all these characteristics are presented in Figures 7 (transmit) and 8 (receive). Their characteristics meet the requirements of the twisted pair environment. The requirements are as follows.

Type ..... 7 Pole, Balanced Elliptical
I/O Impedance ..... 96Ω ±15% (5 to 10 MHz)
3 dB Frequency ..... 17 MHz to 19 MHz
50 dB Frequency ..... ≥30 MHz
In-Band Ripple ..... ≤1 dB

Line Coupling Devices. The line coupling devices, shown in Figure 9, include the transformers, common mode chokes, and common mode noise filters. The transformers provide ac coupling between the line and the circuitry while providing dc isolation. The recommended minimum isolation is 2250 V_{dc}. To provide proper balance between the two ends of the transformers, the windings should be identical. To provide appropriate impedance matching in the frequency range of interest, the transformers should have appropriate primary and secondary inductance (200 μH typical) and minimal interwinding capacitance (<20 pF).

The common mode choke is provided to reject common mode radio frequency and electromagnetic interference picked up from the unshielded telephone lines. It should provide 1000 V_{dc} isolation between the windings. The common mode choke has four windings, each one connected with proper polarity, in series with the receive and transmit twisted pairs. The balance of the choke is very important in order to provide proper noise cancellation while passing through the differential signal unaffected. We recommend a common mode to differential balance of 30 dB at all frequencies up to 20 MHz.

The common mode noise filter removes undesirable high-frequency common mode signals picked up on the line, or generated by the transmitter. These signals are

mainly generated by fast rise and fall times and signal crosstalk in the transmitter.

Line Receiver. The incoming receive signal passes through the line coupling devices and the low pass filter. From there it is fed into a gated line receiver controlled by the squelch circuitry. The line receiver converts the received differential signal to TTL levels and feeds it to the MPR. The receiver can be designed using a zero crossing detector (e.g., NE521) and gated with the TCSx signal with a 74F08.

Squelch Circuit. The squelch circuit differentiates noise from valid incoming data on the receive pair. It does this by detecting signals above a preset voltage level. When there is no signal on the receive pair, the squelch circuit disables the line receiver, and deasserts the TCSx signal to the 82505TA. When a signal above the threshold arrives, TCSx is asserted, and the line receiver is enabled. The squelch circuit ensures that the receive circuits in the 82505TA are operating only during packet reception. The squelch circuit should meet the following specifications:

Reject ..... <250 mV
Accept ..... >300 mV and >30 ns

The circuit shown in Figure 10 uses a high-speed comparator with an offset threshold. The output of this comparator is fed to a retriggerable timing circuit that activates the TCSx pin of the 82505TA. To ensure recognition of the IDL (end of packet) signal, and to prevent midpacket deassertion of TCSx, the timing circuit should be set to detect positive pulses between 1.5 and 2.0 bit times (200 ns). The timing circuit can be implemented by using either a quad flip-flop (74F175) clocked from the 20 MHz clock generator or a retriggerable monostable multivibrator with an appropriate time constant. The first method provides better stability and requires fewer discrete components. If the multivibrator is used, then the selection of the timing components is critical. The timing capacitor must have very low leakage with good temperature and aging stability. The timing capacitor and resistor need to be as close as possible to the IC to minimize stray capacitance and noise rejection.

Layout Considerations. The power and ground wiring should conform to good high-frequency practice and standards to minimize switching transients and parasitic interaction between various circuits. To achieve this, the following guidelines are presented.

- Place bypass capacitors (usually 0.01 μF) on each IC between V_{CC} and ground. They should be located close to the V_{CC} pins.
- Make power supply and ground traces as thick as possible. This will reduce high-frequency cross coupling caused by the inductance of thin traces.
- Separate and decouple all of the analog and digital power supply lines.
- Close signal paths to ground as close as possible to their sources to avoid ground loops and noise cross coupling.
- Connect all unused IC inputs (except as directed by the manufacturer) to ground or V_{CC} to avoid noise injection or parasitic oscillations of unused circuits.
- Use high-loss magnetic beads on power supply distribution lines.
- Group each of the receive and transmit circuits, but keep them separate from each other. Separate their grounds.
- Lay out all differential circuits symmetrically so parasitic effects are also symmetrical.
- Lay out the circuitry from the line connector to the active circuitry (especially the EMI filter) on a ground plane to prevent undesirable EMI effects.

3.3.3 AUI PORT

The AUI port circuitry is shown in Figure 11. It comprises interface circuits, the DO line drivers, two quad D flip-flops (74F175), and terminated line receivers for the DI (squelch and data) and CI (squelch only) circuits.

The CI squelch line receiver feeds the D-0 and clear inputs for one of the quad D flip-flop circuits. When a signal larger than the squelch offset is seen, the flip-flops are cleared and $\overline{\text{AUICDT}}$ is asserted. This continues for as long as CI is active. During the start of idle, the squelch receiver output is held high, and the flip-flops set in sequence. After four clocks, 150 ns to 200 ns, the last flip-flop is set, and $\overline{\text{AUICDT}}$ deasserts. It remains deasserted during the entire idle period.

The DI line receivers work in much the same way, except that activity on CI, or an active transmission will inhibit AUICRS. The data channel on DI is processed without a voltage offset, and is gated by AUICRS. In this way, the least amount of jitter is added on the AUIRxD line, and the data channel is not sensitive to idle noise.

The DO line drivers are controlled by the $\overline{\text{TPEN}}$ and PE11. The drivers should activate when both are asserted. A voltage divider is provided after the drivers to achieve the proper driver levels.

3.3.4 PORT DISABLE CONTROL

The Port Disable Control, shown in Figure 12, is performed by a 74LS154 4-to-16 decoder. During transmission without contention, the address of the originating port is given to the decoder, and the control line asserted. This in turn disables the transmitter to that port. When a transmit based collision occurs, the control line to the decoder is deasserted, and jam is broadcast on all ports.

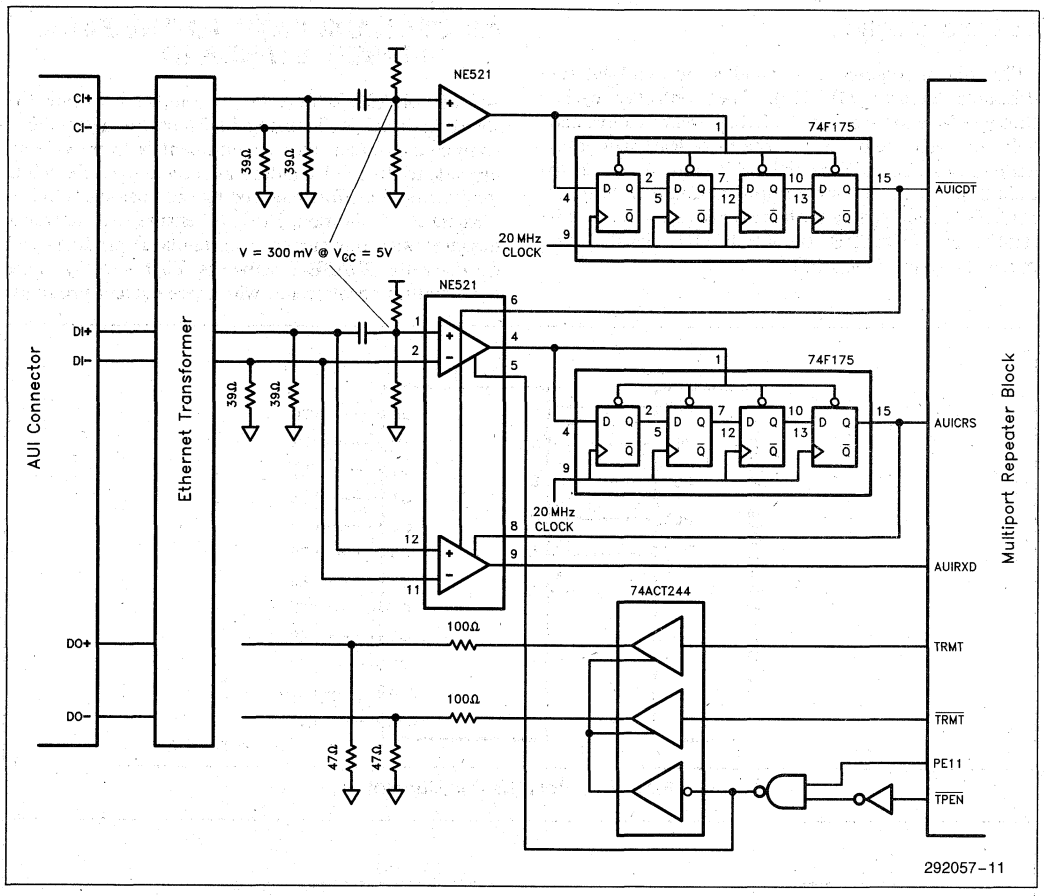


Figure 11. AUI Port

3.3.5 LED CONTROL

LED control (Figure 13) is handled by two 8-bit addressable latches (74LS259). The controller cycles through the addresses for the LEDs every 105 ms, and will turn each one on or off. The three least significant address bits (L0–L2) for the LED control are fed to each 8-bit latch. The most significant address bit (L3) controls the enable line to the two packages. When it is strobed by $\overline{\text{LEDSTRB}}$, the $\overline{\text{LEDCTRL}}$ signal determines the state of the LED.

4.0 UPGRADE PATH TO THE FINAL 10BASE-T STANDARD

As the 10BASE-T Task Force completes writing the standard, Intel is finalizing its plans for a standard-compliant product. Our commitment is to provide an upgrade to the final standard as soon as possible, while minimizing the effort required by our customers to implement it. In addition, Intel will ensure that networks designed with our current (prestandard) products will coexist with 10BASE-T networks. That is to say, there is no built-in obsolescence with these current products.

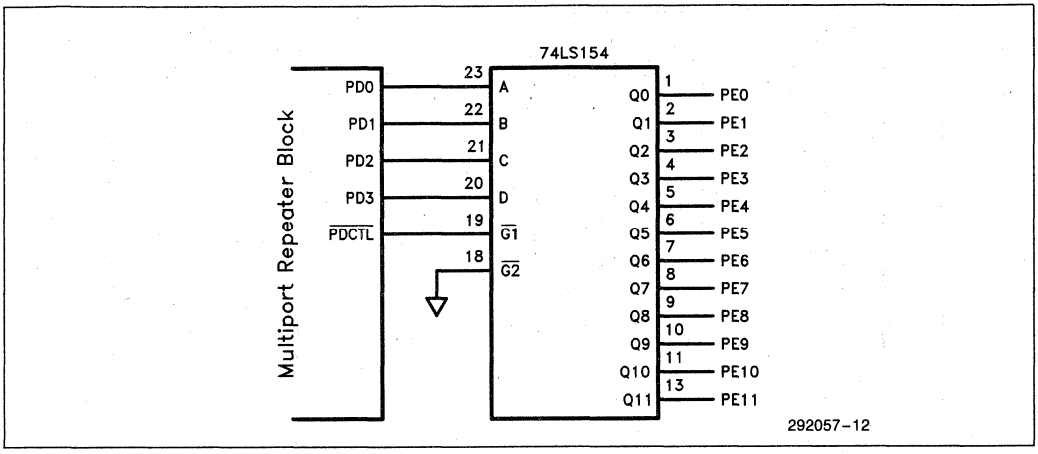


Figure 12. Port Disable Control

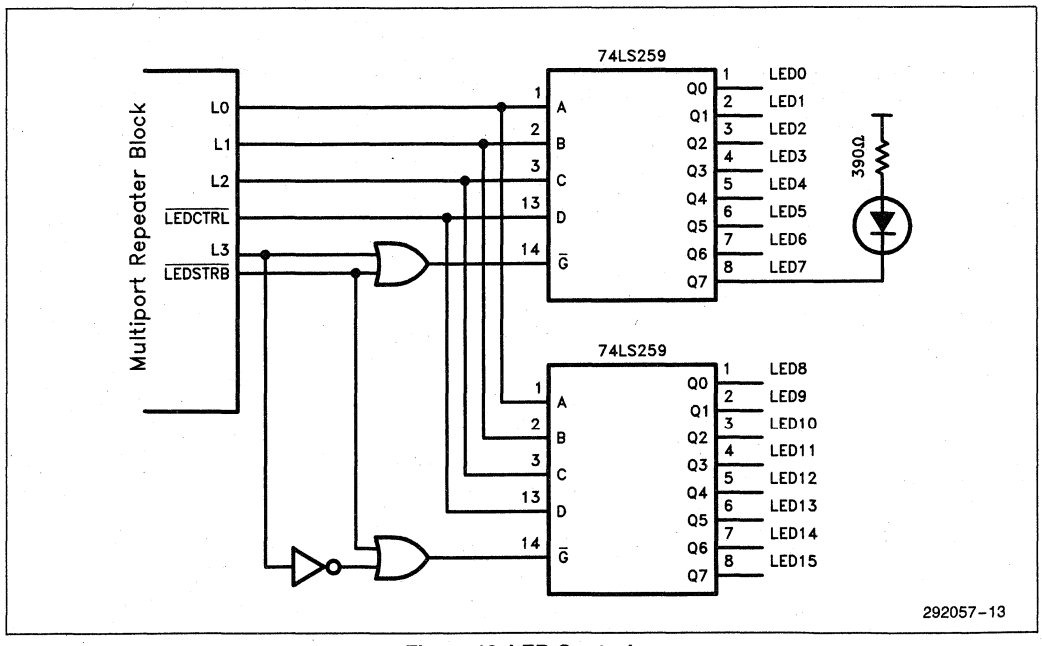


Figure 13. LED Control

Prestandard and standard-compliant networks will coexist at the AUI interface. Network sections based entirely on prestandard components will be able to connect to network sections based entirely on compliant components through coax backbones, or through external MAUs connected to the AUI ports of the repeaters.

The simplest upgrade path will be a DTE designed with the 82521TA Serial Supercomponent. Here, the user will merely have to substitute the standard-compliant Supercomponent, and his design will work. At this time we are planning to include a prestandard compatible mode for the device, which will be a strapping option. Since the 82521TA has defined the pins required for this mode, users can include either mode of operation in their designs, or the ability to select between them.

Upgrading 82504TA designs will be slightly more difficult, since the standard-compliant device will have more functions integrated; e.g., line drivers, line receivers, and interface logic. These functions were not defined by 10BASE-T when the 82504TA was designed, therefore they were intentionally not included. This will require that the system designer change the design for DTE nodes based on the 82504TA, but will allow reductions in the bill of material cost and board space requirements for the design.

Intel intends the 82505TA standard-compliant product to incorporate the Manchester decoder and clock recovery functions; therefore, an 82504TA will not be needed in the repeater. Intel further intends the device to be backward compatible with the previous version, that is, the new controller can be plugged into an old controller socket. The standard-compliant MPR will also include the capability for parallel expansion, allowing repeater design with more than 11 twisted pair ports.

Overall, the upgrade from the current products to standard-compliant products is easy, and incorporates lower cost, higher functionality, or both. The 82521TA SSC was designed to eliminate the effort (and the risk) required for compliance. In both the case of the 82504TA and 82505TA, the upgrade will require minimal redesign, and will maintain or reduce the requirements of material, board space, and power consumption.

5.0 SUMMARY

In this Application Note, a 10 Mb/s Local Area Network has been introduced that uses standard telephone twisted pair wiring and a star configuration for cost savings and flexibility. It is based on the IEEE 802.3 standard for CSMA/CD medium access. It complies with the standard at the MAC and PLS levels, and follows the emerging 10BASE-T standard at the PMA level. This network type is fully software compatible with and can coexist with current Ethernet or Cheapernet networks. The hardware connection is made by including an 802.3 defined AUI port and complying with the repeater standard ANSI/IEEE 802.3c—1988.

Intel has introduced three products for designing network components (DTEs and repeaters). DTE design can be done with either the 82521TA Serial Supercomponent or the 82504TA Transceiver Serial Interface. The Supercomponent contains all the circuitry required between the Ethernet controller and the RJ-45 connector. It also provides a transparent upgrade path to a standard compliant design. Multiport repeaters can be designed using the 82505TA with an 82504TA. It allows for 11 twisted pair ports and 1 AUI port.

Finally, upgrade paths to the upcoming 10BASE-T standard for Twisted Pair Ethernet were presented. This simplest path is for designs which use the supercomponent; however, all designs can be easily upgraded to the standard when it is available.

APPLICATION NOTE

AP-327

July 1989

Two Software Packages for the 82592 Embedded LAN Module

JOSEPH DRAGONY
APPLICATIONS ENGINEER

URI ELZUR
SYSTEM VALIDATION

INTEL CORPORATION

Order Number: 292062-001

1.0 INTRODUCTION

This Application Note is a companion piece to AP-320, *Using the Intel 82592 to Integrate a Low-Cost Ethernet Solution into a PC Motherboard.* While AP-320 deals mostly with hardware issues this Application Note deals almost entirely with software. Two programs are presented. One is written in "C" and the other is written in assembly language. The *NetWare* driver presented in this Application Note is a revised version of the code in section 7 of AP-320.

1.1. Objective

This Application Note was written to serve as a design example to aid the user in developing software for the Intel 82592 LAN Controller. Two programs are provided. The ELM Exerciser Program demonstrates the embedded LAN architecture and provides the user a tool for exercising the 82592 in a system environment. This program is written mainly in the "C" programming language with assembly language used when necessary. The *NetWare* driver provides an example of an interface to a widely used networking package. The *NetWare* driver provides an avenue for evaluation of the ELM concept in a real LAN environment. The *NetWare* driver code provides routines that accomplish all of the common functions required by LAN interfaces. This code should be adaptable to drivers for network software packages other than *NetWare* without too much effort. The *NetWare* driver is written completely in assembly language.

1.2 Acknowledgements

We would like to thank Dror Avni, Gideon Prat, Zeev Sperber and Koby Gottlieb of Intel Israel Design Center for their excellent support during the development of the Exerciser software. We also thank Ben L. Gee of San Jose, California and Drex Dixon of Novell for their advice during the development of the *NetWare* driver software.

2.0 ELM HARDWARE

The ELM is intended to demonstrate the concept of embedded LAN connections. This concept could be implemented either directly on the motherboard of a microcomputer system or as a socket option similar to todays math coprocessor sockets. The ELM illustrates how little board space this concept requires, and also makes it possible to evaluate the performance potential of the nonbuffered architecture. The ELM is not intended as a final solution. Additional hardware features such as a DMA stop register and DMA capable of chaining noncontiguous buffers could simplify the driver software.

The ELM is implemented as a small printed circuit board containing an 82592 Advanced CSMA/CD LAN Controller, two PALs, and two latches. It is connected by a ribbon cable to an analog module, which provides the interface to the media. There are two analog modules available. They are an Ethernet module and an Ethernet/Cheapernet module. Using this approach, other analog modules, for example, StarLAN or twisted Pair Ethernet, could be implemented without modifying the digital module.

The ELM is designed to function in PC AT compatible systems. It has been used in the Intel SYP301 system, Compaq Deskpro 386-16, Compaq Portable 386-20, Compaq Portable 286, and both 6- and 8-MHz IBM PC AT machines. The ELM takes liberties with the refresh cycles of the PC. It does not sense the system's refresh request and can cause refresh cycles to be missed occasionally. In a commercial implementation a timer should be used to limit the amount of time the ELM can control the bus. The ELM hardware and driver software are used daily by one of the authors as his connection to our department LAN and no problems have been caused by the lack of a refresh kickoff timer. The module uses two of the system's 16-bit DMA channels to provide transmit and receive DMA. Channels 6 and 7 are used. The module also uses the Int10 interrupt line. None of these hardware requirements are jumper selectable. The module also requires a small modification to the system motherboard. A connection must be made to the EOP pin of the DMA controller to allow autoinitialization to be controlled by the module for retransmission in case of collision. This can be accomplished by soldering a binding post to the EOP pin of the secondary 8237A DMA controller. In cases where a connection to EOP cannot be made, the software would have to be altered to allow retransmission to be controlled by the CPU.

As well as providing all required address decoding, the two PALs interpret the Tightly Coupled Interface handshake signals from the 82592 and generate control signals to the latches and the DMA controller. These signals accomplish two things. First, at the end of a received frame, the Tightly Coupled Interface generates a handshake. The PALs convert this to a signal that latches the last location of the frame just received. The 82592 transfers length and status information into the memory as the last four words of a received frame. Using this information it is possible to reconstruct a string of frames in memory. This feature of the module allows reception of back-to-back frames. Second, when a collision occurs, the Tightly Coupled Interface generates a handshake, which the PALs use to send an EOP to the system's DMA controller. This allows the ELM to execute a retransmission without intervention by the CPU. This feature serves two purposes. The CPU is free to continue the processing it is involved with, and the node is also guaranteed fair and equal access to the media. When the CPU must actually handle retrans-

mission it is unlikely that the station will be ready to retry access to the link in a timely fashion.

Section 2 Design Documentation for 82592 Embedded LAN Module Novell NetWare* Driver

3.0 OVERVIEW

The Novell *NetWare** Driver for the 82592 Embedded LAN Module (ELM) is the first *NetWare* driver internally generated by MCFG LAN Marketing. The purpose of the Embedded LAN Module project is to demonstrate the feasibility of an embedded Ethernet LAN connection.By providing a driver for a very widely used Network Operating System (the popular *NetWare* from Novell, Inc.) we are attempting to provide an tool for evaluating this concept under real network conditions. This driver is a workstation shell driver. This section of the Application Note is intended to be used in conjunction with the program listing in Appendix C. It is presented as an adjunct to the comments in the source code listing itself. Hopefully the text will shed the needed light where the source code comments fail to illuminate.

The first part of this section contains an overview of the requirements of a *NetWare* driver to allow those unfamiliar with *NetWare* drivers to follow the discussion. A bibliography is provided as an appendix for those who desire more detailed information. The balance of the section is a discussion of each routine the driver software provides. Each routine is first explained from a functional point of view. Then any hardware considerations are discussed. Where it is warranted, alternative approaches to the routine are given.

This document is not meant to be a tutorial on writing Novell *NetWare* driver software. It is a discussion of the generation of a single driver for a particular piece of hardware. This driver is a demonstration tool and is not represented to be a commercial *NetWare* driver. Neither the author nor Intel Corporation accept any responsibility for the use or misuse of this driver or of this documentation. For complete information on *NetWare* driver generation please contact Novell.

Novell's *NetWare* Network Operating System uses an implementation of the Xerox Internetwork Datagram Packet (IDP) protocol called the Internetwork Packet Exchange (IPX) protocol. It provides the developer a set of media independent services, and dictates a set of services that the driver must provide. Information concerning transmit and receive operations are communicated between IPX and the driver by using Event Control Blocks (ECBs). For example, if *NetWare* wants to transmit a packet, a transmit ECB is prepared that contains address information and a list of fragments in memory containing the packet to be transmitted. The driver routine DriverSendPacket is then called. DriverSendPacket processes the ECB and constructs the media specific frame, which allows the information to be transmitted to the target node on the network. When the attempt to transmit the frame has been completed, the driver stuffs a completion code into the proper position in the ECB and passes it back to IPX through a call to the IPX routine IPXHoldEvent. IPX puts the ECB in a queue and later does the processing required to complete the operation.

NetWare requires the driver to provide several routines for its use. Some of these routines may not be required by a driver and can be implemented as a simple return. This driver implements the routines DriverDisconnect and DriverOpenSocket as a return. The remaining routines are implemented and are listed below.

- DriverInitialize configures the LAN adapter hardware and any variables that need to be initialized at start up time such as the node address.
- DriverSendPacket and DriverBroadcastPacket are implemented as a single routine with two labels at the entry point. This routine processes the transmit ECB that is passed to it and make a best effort attempt to send it to the target node. It is not a guaranteed delivery routine.
- DriverISR is the interrupt service routine for the driver and processes all interrupt events.
- DriverPoll checks to see if a transmit is in progress. If there is no active transmit it returns. If a transmit is underway DriverPoll checks to see if it has timed out. If so, the transmission is aborted and its ECB is returned with an error code.
- DriverCancelRequest searches the transmit queue for the specified ECB and removes it from the queue. It then stuffs the completion code and returns.
- DriverCloseSocket unlinks all pending ECBs for the specified socket and returns them to IPX.
- DriverUnhook is used to disinstall the driver if no active file server can be found during initialization. This involves restoring the interrupt vector to its original value and disabling the LAN adapter so it will not affect system operation.
- SetInterruptVector is called by DriverInitialize to insert the interrupt vector for the LAN adapter into the correct location in the system's interrupt vector table after saving any vector that is already there.

4.0 DRIVER SOFTWARE ROUTINES

4.1 DriverInitialize

This is the first routine IPX calls when the driver software is being loaded. This routine is responsible for

*NetWare is a registered trademark of Novell Incorporated

initializing the LAN adapter hardware and any variables or memory structures required by the driver. It also sets the interrupt vector in system RAM after saving any vector already there. When IPX calls this routine it specifies a point in memory for the initialization routine to place the node address.

The first thing this driver does is set the IPX variable "MaxPhysPacketSize" to 1024. This value is used when attaching to a fileserver to negotiate the largest packet size that will be passed between the two stations. This allows transferring packets larger than the 576-byte default packet size between the fileserver and the workstation.

4.1.1 GENERATING A STATION ADDRESS

The next action generates a station address. Since the ELM has no address PROM the driver generates the address by using a combination of hard-coded numbers and the value read from the system's real-time clock. The real-time clock is read using function 2Ch of the DOS interrupt 21h. The first two bytes of the address are 00h and AAh, which are Intel's Ethernet code. The next three bytes are the minutes, seconds, and hundredths of seconds read from the real-time clock. The sixth byte is 7Eh, which was a dysteleological choice on the authors part. This technique gives a high likelihood that several ELMs can be operated in a small network without duplicate addresses occurring. A commercial implementation of the ELM concept should be provided with a hard-coded address in PROM or EPROM on the card. After the address bytes have been moved into the drivers local variable array they are copied to the location indicated by IPX in the DI register.

When the address initialization has been completed, the driver initializes some parameters from the hardware configuration table. This is mainly done as an example, since there is only one possible hardware configuration for this module. However, the code required to step through the tables is provided.

4.1.2 INITIALIZING THE INTERRUPT VECTOR

Initializing the interrupt vector is the next action. The interrupt number is read from the configuration variable config__irq__loc and placed in the AL register. The offset of the interrupt service routine is moved into the BX register then SetInterruptVector is called. SetInterruptVector first generates the mask variables for the 8259A by writing a one into DL and then shifting it left a number of times corresponding to the value passed in AL. The unmask variable is then generated by the negation of the value in DL. SetInterruptVector then saves the vector for the interrupt that the board will use and inserts the vector for DriverISR in its place. The routine then returns control to DriverInitialize.

Upon returning from SetInterruptVector the ELM is enabled by a write to location 303h. The PALs decode this write, and enable DMA and interrupts from the ELM to the system (as well as reads and writes to and from the 82592 registers). A Reset command is then issued to the 82592.

4.1.3 INITIALIZING THE BUFFER VARIABLES

The driver must calculate the effective address of the transmit and receive buffers to use the system DMA. Since the PC architecture uses a static page register for the upper address bits, checks must be made to ensure that the buffers do not cross these hardware imposed boundaries. This is accomplished through a call to set__up__buffers. This routine sets up two buffers in the 10-kB space allocated at load time. One buffer is used for a transmit buffer and as the parameter block for commands that require parameters. This buffer is set up to be at least 1200 bytes long. All remaining space is used for the receive buffer. The receive buffer is implemented as a restartable linear buffer. This approach was taken to allow the use of the IPX routine IPXReceivePacket, which requires the receive packet to be contained in a single, contiguous buffer. IPXReceivePacket does most of the required receive processing itself which makes the driver simpler.

There are four basic conditions that can exist for the buffer space with which the driver must work.

- The buffer space has no hardware boundary. See figure 1.
- The buffer space contains a boundary, and the lower section is too small to use (i.e., less than the 1200 bytes used for the transmit buffer). See figure 2.
- The buffer space contains a boundary, and the upper section is too small to use. See figure 3.
- The buffer space contains a boundary, and both sections are usable. See figure 4.

In the first case the transmit and general purpose buffer will be located in the first 1200 bytes of the buffer space, and the receive buffer will occupy the remainder of the space. In the second case the unusable fragment is discarded by adding the length of the fragment to the original starting address of the buffer. The total buffer area is adjusted by subtracting the length of the fragment from the original length (10-kB) of the original total buffer area. The transmit buffer uses the first 1200 bytes of the buffer space. In the third case the starting address remains the same and the total buffer area is adjusted by subtracting the length of the unusable fragment from the original total buffer area. In these three cases the required addressing variables can now be calculated. The fourth case adds one additional step. Since both fragments are usable the larger fragment must be determined. The receive buffer will be located in the larger fragment. The receive buffer will be at least 5000 bytes and can be as large as 8800 bytes depending on where DOS loads the driver.

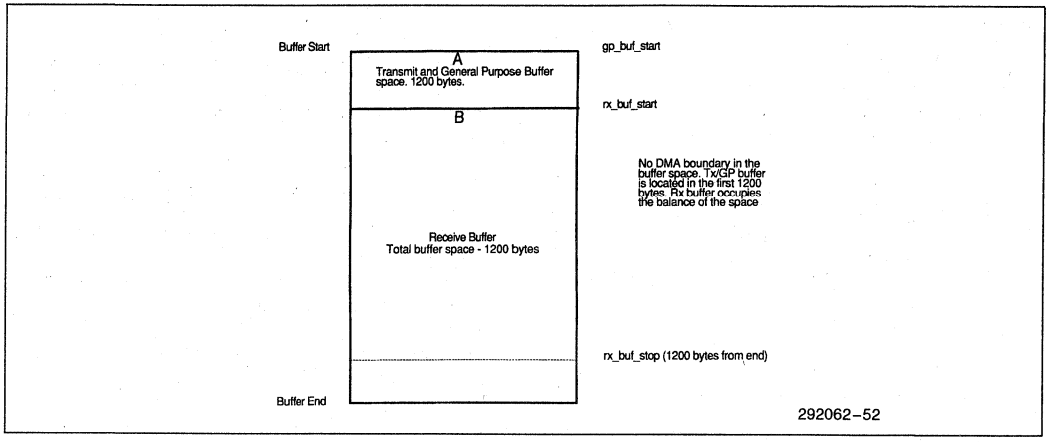


Figure 1. Buffer with No Hardware Boundary

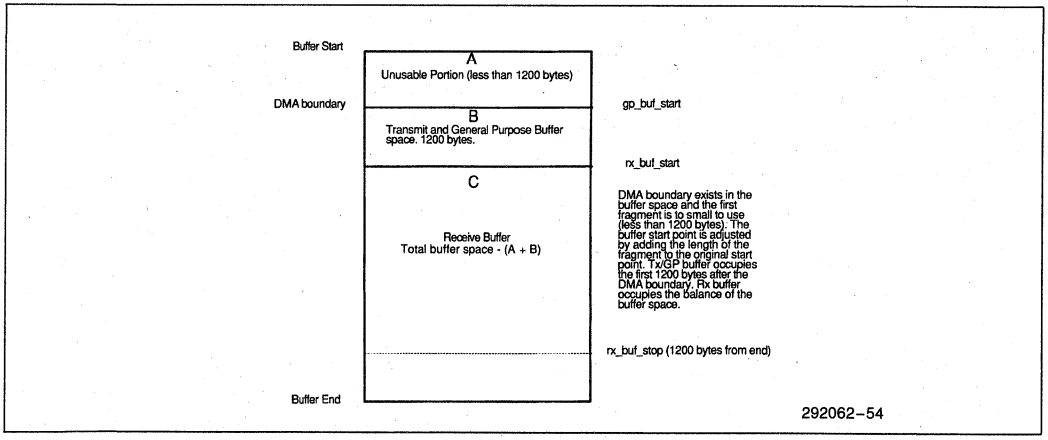


Figure 2. Buffer with Boundary and Unusable Portion at Top

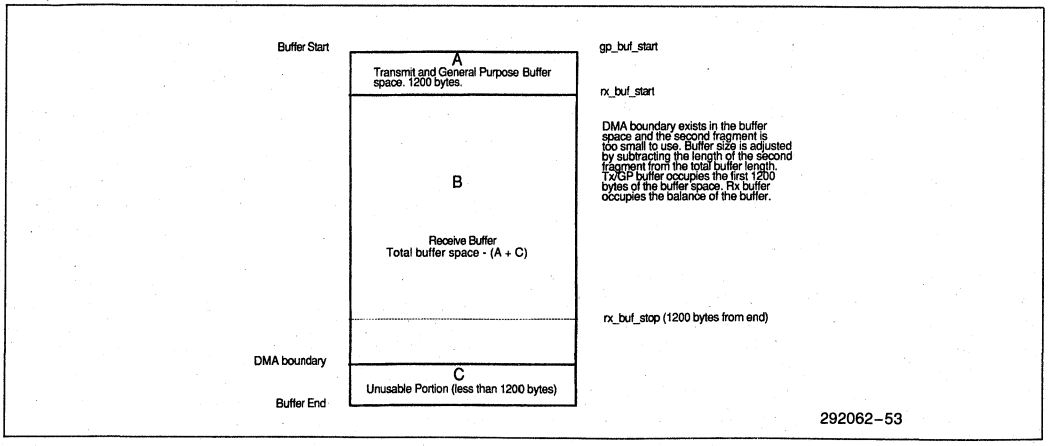


Figure 3. Buffer with Boundary and Unusable Section at Bottom

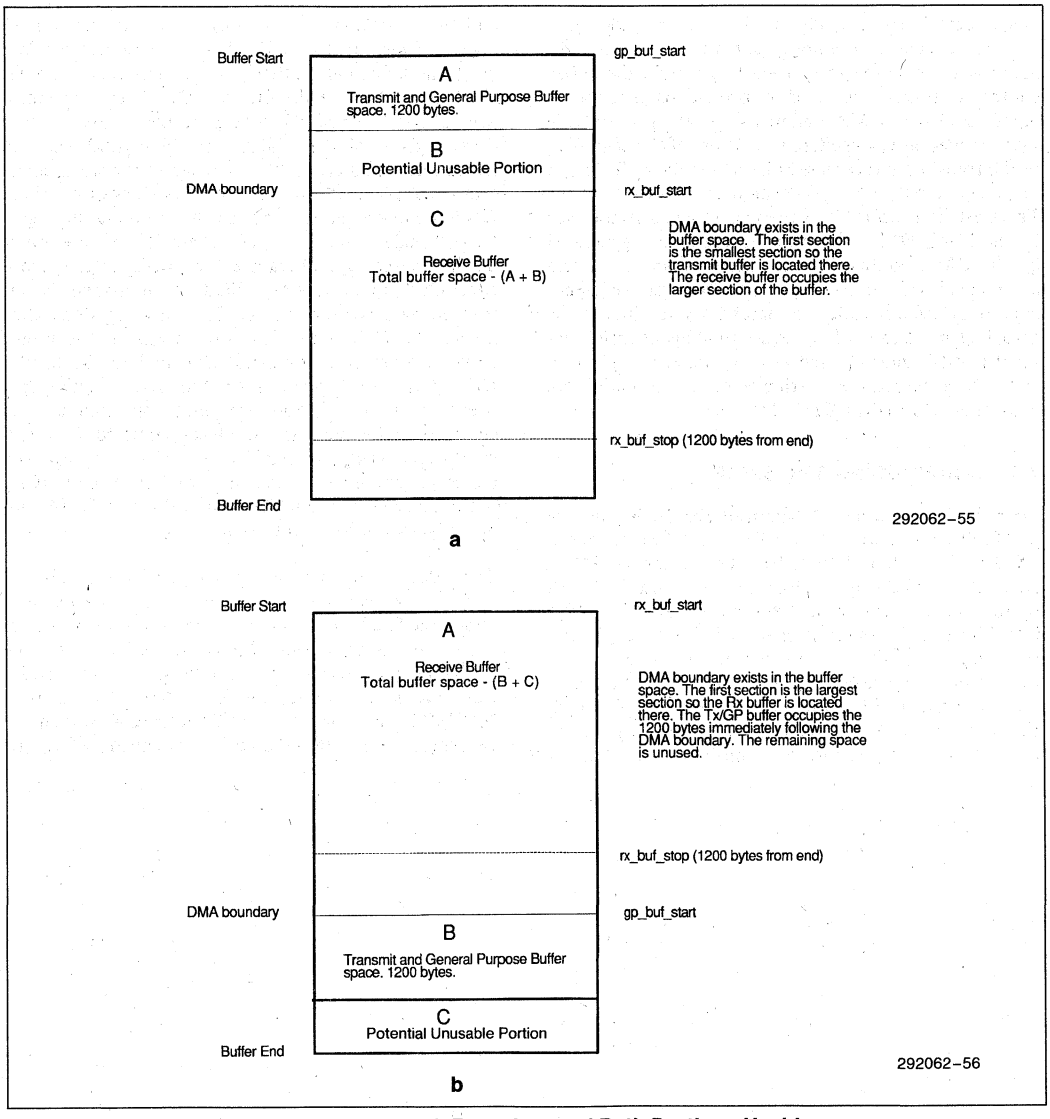


Figure 4. Buffer with Boundary and Both Portions Usable

Once these initial calculations have been made set_up_buffers uses this information to generate the addressing information to be used to program the DMA control channels and their respective page registers. Since the 16-bit DMA channels are set up to provide word moves only, the effective address of the beginning of the transmit and receive buffers must be shifted right one place so only A1-A16 are contained in the variable. The least significant bit of the page register is not used by the 16-bit DMA channels because A16 is generated by the DMA controller. The receive channel requires an artificial segment to be generated because the latches contain an effective address rather than an offset to the actual segment the buffer resides in. This artificial segment is used when the received packet is passed up to IPX. Once the required variables have been initialized, control is returned to DriverInitialize.

4.1.4 CONFIGURING THE 82592

With the DMA variables initialized, the driver can now prepare to configure and initialize the 82592 Advanced CSMA/CD LAN Controller. The transmit DMA channel is used during configuration to allow the 82592 to read parameters from memory. To put the 82592 into 16-bit mode, the first operation to the 82592 after reset must be a Configure command with zero in the byte count of the parameter block. To do this the transmit DMA channel is set up to point to the beginning of the transmit/general purpose buffer area. This is done by first resetting the indexing flip-flop in the 8237A, and then enabling it by writing 10h to the command register. This puts the 8237A into rotating priority, late write, and normal (rather than compressed) timing. Next the address of the first location of the transmit/general purpose buffer is written to the 2-byte base address register of the 8237A (low byte first) and the DMA page register. A "1" is written to the word count register of the DMA controller. This allows two transfers to be made because the 8237A interprets this register as "transfer count - 1." The channel is then set up to do the desired type of transfer by writing to the DMA controller's mode register. Finally, the channel is unmasked by a write to the 8237A mask register. After moving "0's" into the first two words of the buffer space, a Configure command is issued to the 82592. DriverInitialize then enters a polling loop, reading register zero of the 82592 and waiting for the command to complete. After the command has completed, an Interrupt Acknowledge is issued to the 82592 to clear the interrupt generated by the completion of the command. All transfers that the 82592 makes through DMA will be 16-bits wide from this point on.

The DMA channel is set up again as previously described; however, the word count is set to eight. This allows the 82592 to read in its configuration parameters from the transmit/general-purpose buffer area. The configuration parameters are copied into the buffer from the array config_block by the CPU using a MOVSB instruction with a REP prefix. CX contains an 18 decimal when the MOVSB is executed. When the

copy is completed a Configure command is issued and a polling loop is entered to wait for command completion.

The parameters in the configure block set the 82592 to function in the following manner. The serial mode is set to high speed to allow Ethernet operation; both the transmit and receive TCI modes are enabled;and slot time, minimum frame length, preamble length are set to the values required by Ethernet. After the command is completed the generated interrupt is cleared.

4.1.5 SETTING THE STATION'S INDIVIDUAL ADDRESS

The transmit DMA channel is again set up for use, this time with a word count of three for use by the Individual Address Setup command. The node address is copied from its place in memory to the Tx/GP buffer area, and the IASetup command is issued to the 82592. After the command is completed the interrupt is cleared by an Interrupt Acknowledge command.

4.1.6 FINAL INITIALIZATION

The receive DMA channel is now initialized to point to the beginning of the receive buffer. The word count is set so the receive DMA cannot go beyond the end of the assigned receive buffer area.

Next, the interrupt channel is unmasked to allow interrupt driven operation and a Receive Enable command is issued to the 82592. The AX register is set to zero to indicate successful completion of the initialization routine and control is returned to IPX. Should some part of the initialization routine fail, AX would contain a pointer to a $ terminated error message string in memory. On return of control IPX would display the specified message and terminate.

4.2 DriverSendPacket, DriverBroadcastPacket

These two routines are treated as a single routine with two labels at the entry point. The first action taken when these routines are called is to disable interrupts through a CLI command. The routine then determines if any packets are queued for transmission. This is done by checking the segment portion of the double-word variable send_list to see if it is null. If it is, no frames are queued and the packet is put in the first location in the list. Flow then drops through to the start_send routine, which does the actual transmission. (The start_send routine will be detailed later.) If the transmit queue is not empty then DriverSendPacket searches to the end of the queue and adds the packet there. The routine then returns control to IPX. The queued packet will be sent when it is reached in the list. The queue is maintained as a linked list using a dedicated link field in the transmit ECBs. The head is the ECB contained in the send_list variable and the tail is the ECB with a null link field.

The start_send routine is a subfunction of DriverSendPacket. It is not called directly by IPX but it can be called by DriverPoll in response to a transmission timing out when frames are queued for transmission. This routine starts by clearing the interrupt and direction flags through a CLI and CLD instruction respectively. It then retrieves the length of the packet to be transmitted from the transmit ECB packet length field. The packet length is compared to the minimum length required by Ethernet after a byte swap to allow arithmetic operations to be performed on it. If it requires padding the value is stored in the padding variable.

The byte count for the 82592 is then calculated and the construction of the frame in the transmit buffer is begun. Since *NetWare* requires the Ethernet length field be an even number start_send next increments the byte count then performs a bitwise AND operation with FEh. This ensures that the byte count is consistent with the Ethernet length.

The first step in constructing the frame in memory is to move the transmit byte count into the first word of the transmit buffer. The byte count is stored low byte first. The destination address is then copied from the transmit ECB to the buffer by the CPU using MOVSW instructions. It is not necessary to copy the source address to the transmit buffer since the 82592 is configured to do automatic source address insertion. After ensuring that it is an even number, the length is moved into the Ethernet header. Now the fragment list from the transmit ECB must be processed. First the fragment count is moved into the AX register. This value indicates the number of fragments the list contains. By decrementing AX after each fragment is copied to the buffer the completion of the fragment processing can be determined. The address of the first fragment is loaded into DS:SI, and the length of the fragment is loaded into CX. The fragment is then copied into the buffer through a REP MOVSW. If the fragment was an odd length a MOVSB is done to finish the copy. The pointer to the fragment descriptor list is indexed to the next entry. AX is decremented and if it is not zero the operations above are repeated until all the fragments have been copied to the buffer. Once the fragment list is completely processed any required padding is moved into the buffer. The word following the last location in the frame must be a zero since the 82592 in TCI mode checks this location to see if it has a chain of frames to transmit. A zero is interpreted as end of chain, a 04h is interpreted as a new transmit command. This driver does not implement transmit chaining.

The transmit DMA channel is now initialized and transmit_active_flag is set to "1". The DMA address registers and the page register are set to point to the beginning of the transmit buffer. The channel mode is set to move data from memory to the 82592. Four is added to the transmit byte count that was calculated earlier to allow the DMA controller to transfer the two bytes of the byte count field and the transmit chain word that completes a transmit in Tightly Coupled Interface mode. After this addition the transmit byte count is shifted right one bit to convert it to a word count. This value is then moved into the DMA controller's word count registers. Finally, the channel is unmasked.

A Transmit command is now issued to the 82592. IPX provides a time mark called IPXIntervalMarker that represents the PC clock tick. The current value of this variable is read and moved into tx_start_time to be used by the DriverPoll routine to check for transmit timeouts. The TotalTxPacketCount variable is incremented and control is returned to IPX. The 82592 contains a programmable timer that could be used to generate transmit timeouts in an application that does not have such a built-in mechanism. This routine must return with interrupts disabled.

4.3 DriverPoll

DriverPoll is called at intervals by IPX to allow the driver to check for transmit timeouts or other non-interrupt driven events that need to be serviced. The first thing done after disabling interrupts with a CLI is to check if the transmit_active_flag variable is set. If it is not set, a return is performed. If it is set, the tx_start_time variable is subtracted from the current value of IPXIntervalMarker. If the result is less than the value of TxTimeOutTicks, in this case 20, a return is performed. If the transmit has timed out the transmission is aborted and a completion code of TransmitHardwareFailure is moved into the completion code field of the ECB. The ECB is then unlinked from the transmit queue and returned to IPX through a call to IPXHoldEvent.

To accommodate errata No. three of the 82592 A-1 stepping, as stated in revision 1.2 (December, 1988) of the 82592 Errata Sheet, a Switch to Port1 command is issued to the device. This is followed by a Selective Reset in Port1, followed by a switch back to Port0. The receiver is then reenabled by a Receive enable command. The flag that indicated an active transmission is then cleared. The transmit queue is then checked. If the queue is not empty, the ES:SI register pair is set up with the values from the queue, and start_send is called. If the queue is empty control is returned to IPX.

4.4 DriverISR

This routine services all interrupts generated by the ELM. It first calls IPXStartCriticalSection to tell the Asynchronous Event Scheduler (AES) function of IPX that it should not execute until an IPXEndCriticalSection call is issued. This allows interrupts to be reenabled for sources other than IPX's AES which is executed in response to the system clock tick interrupt. DriverISR then saves the machine state by pushing the general purpose registers, the index registers, the base pointer and the ES and DS registers. Next the direction and interrupt flags are cleared with a CLD and CLI instruction, respectively. An EOI is then issued to each of the two system interrupt controllers to clear them. The DS and ES registers are then set to the same value as the CS register because the driver is contained in a single segment. The cause of the interrupt is now determined by reading register "0" in the 82592. A zero is first written to the 82592 to set the internal pointer. The value read from the 82592 is then compared with the values representing a receive, transmit, and retransmit interrupt, then a jump is taken to the proper section of the interrupt service routine. If the value does not match one of the expected values, the variable false_590_int is incremented and a jump to the label int_exit is performed.

4.4.1 RECEIVE CASE

If the value read from the 82592 indicates that a frame has been received, a jump is made to the beginning of the code that services receives. The first action is to read the two latches that contain the address of the last word that was transferred during the receive. This value is moved into the variables rx_buf_tail and rx_buf_ptr. This value is then compared with the value stored in rx_buf_stop by the set_up_buffers routines to determine if most of the receive buffer has been used and a reset is required. If most of the buffer has been used the flag reset_rx_buf is set to indicate that the buffer variables must be initialized before the interrupt service routine is exited. The value read from the latches is then compared with the value in rx_buf_head. This value represents the last location that contains a received frame. If no frames have been received it contains the address of the first location in the receive buffer. If this comparison indicates that no new frame has been received the ten_cent_latch_crash variable is incremented and a jump is made to the label int_exit.

If a frame, or frames, has been received the receive buffer must be processed to allow the received frames to be sent up to IPX in the order in which they were received. This is accomplished by using the count and status information that the 82592 deposits at the end of

each frame when it is in TCI mode. Using the value read from the latches as a base, the routine ProcessFrames indexes back through the chain of received frames. The rx_buf_ptr variable keeps track of the current position in the buffer. The status of the frame is read from the end of the receive buffer and if it is good a jump is done to the label good_rx. If the status is bad, rx_buf_ptr is adjusted to point to the end of the previous frame in the buffer. This value is compared to the value of rx_buf_head, which contains the location last processed by the receive routine or the beginning of the receive buffer if this is the first receive. If the values are equal, all currently received frames have been processed and a jump is made to the label hand_off_packet. At the label good_rx three length checks are made as required by the *NetWare* implementation of Ethernet. First the frame is checked to see that is does not exceed the maximum length of 1102 bytes (1024 data size, 64 *NetWare* bytes, and 14 Ethernet header bytes). Next it is checked to see that it is at least the minimum size of 30 bytes. This 30 byte value is only the IPX packet size, it does not count the Ethernet header or the pad bytes required by Ethernet. The last check ensures that the actual number of bytes received agrees with both the Ethernet and IPX header length fields. If the IPX length is less than the minimum Ethernet frame length the total number of bytes received is expected to be 60. This represents Ethernet's 64 byte minimum frame length less the four CRC bytes, which are not counted as receive bytes. If all these checks pass, the frame is added to the list of received frames by storing its location, length, and source address in an array of structures called rx_list. Each entry consists of 12 bytes. These bytes are the location of the frame in memory, the length of the frame, and the address of the node that sent the packet.

When all received frames have been processed, all good frames are passed up to IPXin the order they were received using calls to IPXReceivePacket. When all entries in rx_list have been processed, the variable rx_buf_head is set to the value read from the latches at the beginning of the interrupt service routine and stored in rx_buf_tail. ProcessFrames then returns to the point from which it was called and execution falls through to int_exit.

4.4.2 TRANSMIT CASE

If the status read from the 82592 indicates that a transmit completion is the cause of the interrupt, a jump is performed to the label sent_packet.The first action is to check that tx_active_flag is set. If it is not set no transmit should have been taking place, so a jump is made to the label int_exit. If tx_active_flag was set, the status is read from the 82592. If the status is bad a jump is made to tx_error, which increments the appropriate counter and moves an error code to the AX register before jumping to the FinishUpTransmit code. If the status is good any retries contained in the status register are added to the RetryTxCount variable, the AX register is XOR'd to indicate a good transmission and execution falls through to FinishUpTransmit. This code inserts the proper completion code in the transmit ECB, unlinks it from the transmit queue, and hands it off to IPX by calling IPXHoldEvent. The transmit queue is then checked to see is any frames are waiting. If send_list is not empty the next frame's ECB address is put into the ES:SI register pair and a call is made to start—send. On return execution jumps to int_exit.

4.4.3 EXITING THE INTERRUPT SERVICE ROUTINE

At int_exit the driver makes a safety check to ensure that the receiver is still enabled. This is done by checking the two bits in status register 3. If the receiver is disabled, a Receive Enable is issued to the 82592. Next, an Interrupt Acknowledge is issued to the 82592 and the interrupt bit is polled to see if any new interrupts have occurred. If a new interrupt has occurred, execution jumps back into the interrupt service routine at the label int_poll_loop. If no new interrupts have occurred, the reset_rx_buf flag is checked to determine if the receive buffer needs to be reinitialized. If reinitialization is required, a final check is made to see if any new frames have been received. If a new frame has been received ProcessFrames is called. On return the Receive DMA channel is masked and the receiver is disabled by issuing a Receive Disable command to the 82592. It is necessary to disable the receiver during the reprogramming of the 8237A because if there is an active request on a channel when it is unmasked the 8237A enters an undefined state which can result in a system crash. The necessary variables are reinitialized as well as the receive DMA controller. The receive DMA channel is then unmasked and the receiver is reenabled by issuing a Receive Enable command to the 82592. The interrupt enable flag in the processor is then cleared through a CLI instruction and IPXEndCriticalSection is called to tell IPX that it is now free to run. A call is made to IPXServiceEvents, and on return the registers are popped to restore the machine state and the interrupt service routine is exited.

This covers the main sections of code that make up this driver. The routines that were not covered in detail are generic in nature and can be understood by a reading of the driver source code included as Appendix C.

Section 3. ELM Exerciser Program

5.0 OVERVIEW

The ELM Exerciser software is specifically written for the Embedded LAN Module Demonstration board but can accommodate other 82592 TCI (Tightly-Coupled-

Interface) implementations, with minimal changes. The ELM Exerciser software supports system and 82592 configuration, command execution and statistics display for the 82592 in the Embedded LAN hardware.

This section of the Application Note includes a description of the Exerciser, a discussion on design considerations for 82592 software drivers and some programming hints.

6.0 INITIALIZATION

System initialization begins with setting up all memory structures. The 8259A PIC IRQ10 is masked, to prevent unsolicited 82592 interrupts before initialization is completed. Control is then transferred to the user. The ELM hardware is enabled by a write to I/O Port 303h. This is done using the "LAN En" command.To activate the 82592, the following sequence should be executed. This sequence can be executed through the "Initialize" command or by executing each command separately. First the 82592 should be reset. This places the device in the default configuration. The default bus width is eight bits. Next a Configure command is issued to the 82592 with "0" in the byte count field. This places the 82592 into 16 bit mode. The 82592 will now use a 16-bit data bus for DMA transfers. For commands and status, only the low byte is used.

The ELM Software Package uses the 82592 in a configuration different from the 82592 default configuration. Whenever a parameter is used that varies from the default, an explanation is given. The 82592 configuration is presented in code example one. The Configure command is issued through channel 0, which is used for memory read I/O write cycles.

'ByteCnt' is a word-wide field containing the number of parameter bytes in the CONFIGURE command (excluding the 'ByteCnt' field). The maximum length is 15 bytes. The 82592 will execute 9 DMA word wide transfers (1 for ByteCnt and 8 for parameters) but will ignore the last (the 16th) byte.

The following 'Params1' fields are different from the default:

- The FIFO limit field is set to 0Fh.This configures the FIFO's as two equal 32 byte banks for receive and transmit. For this configuration, this parameter is internally multiplied by 2 to generate the actual FIFO limit. This parameter configures the transmit side, so transmit FIFO limit equals 2 * 0Fh = 30. The receive FIFO limit is then 32 − 30 = 2. This means that the 82592 will issue a bus request after the first word has been written into it. Note that this configuration is provides the maximum bus latency (Refer to the 82592 User Manual for more detailed explanation). The system bus request mechanism of the transmit and receive FIFO's is tuned using this parameter. For the receive FIFO this determines the number of bytes that may gather in the FIFO, before it requests the bus. The lower the FIFO limit, the earlier the bus is requested and the 82592 will be able to overcome a higher bus latency (the time it takes from request asserted to actual bus transfers). However, if the bus latency is short, fewer bytes will be gathered in the FIFO and before a request is made. This increases the arbitration overhead per transfer.
- The OSC RANGE and SMPLG RATE bits are set to '0' as required in High Speed Mode. (Refer to the 82592 User Manual for more explanations on High Speed Mode).
- The CHAINING bit is set to '0'. Together with RxEOP set to '1' and TxEOP set to '1', this causes the 82592 to signal with the EOP# pin for all the receive frames last byte (BC field) and collided transmit frames. The ELM logic uses EOP# with DRQ lines to determine transmission status and to latch the pointer to the byte count field into the ELM latch.
- BUFFER LENGTH/TCI is set to '80h'. Together with the chaining set to '0', RxEOP set to '1' and TxEOP set to '1', this puts the 82592 in TCI mode. In this mode the EOP# and DRQ pins signal the completion and status of transmit and receive events. This allows retransmission on collision by auto initializing the DMA controller as well as reception of back to back frames without CPU intervention. When the device is not configured to TCI mode, it requires CPU acknowledgement after each received frame. In TCI mode the status is stored in memory, so the 82592 does not need immediate CPU attention. If this parameter is programmed to C0h, the 82592 will not generate an interrupt upon frame reception. The ELM Software packages use this interrupt to invoke the RCV ISR. Note that 'Params3' fields are described in the manual using decimal numbers. They should be translated into a Hexadecimal base for programming, e.g. inter frame spacing of 96 bits is programmed as '60h'.

The 'Params4' Max Retry field is set by default to '0Fh'. In the case of a frame transmission attempt that has experienced 15 retries, the Num__Coll status is set to '0', instead of 16. Namely, the first attempt and 15 retransmissions have collided so the Num__Coll should be 16. Instead the 82592 will set this field to '0'. This is further discussed in the transmit section.

The 'Params7' Monitor Interrupt field is set to '1'. This prevents Monitor interrupts. This bit can remain in its default state, since the Monitor Mode is disabled.

The following 'Params8' fields are different from the default:

- 'Params8' CLK divider is not used.

CODE EXAMPLE 1

```
CONF.CONF_Ptr->ByteCnt = 0x000F; /* 15 BYTES */
CONF.CONF_Ptr->Params1 = 0x804F; /* TCI, Tx FIFO LIMIT = 32 */
CONF.CONF_Ptr->Params2 = 0x0026; /* PREAMBLE LEN =7 */
CONF.CONF_Ptr->Params3 = 0x0060; /* 9.6 uS INTERFRAME, 512 SLOT */
CONF.CONF_Ptr->Params4 = 0x00F2; /* RETRY =15 NO PROMISCUOUS */
CONF.CONF_Ptr->Params5 = 0x4000; /* minimum frame 64 */
CONF.CONF_Ptr->Params6 = 0x00FF; /* NO AUTO RE-XMT ON COLLISION */
CONF.CONF_Ptr->Params7 = 0x873F; /* NO MONITOR INT */
CONF.CONF_Ptr->Params8 = 0xFFF0; /* RX & TX EOP, 32B RX & TX FIFO */
```

- RxEOP and TxEOP are set to '1' to enable the TCI signaling as explained above.
- Status length is set to 6 bytes. Together with the TCI mode configuration,this causes the Frame Counter to be presented in the STATUS 2__0 register. The 7 LSBs of STATUS 2__0 count the number of frames received after the Receive Enable command was issued. Both good and bad frames are counted. The frame counter value is valid when the MSB of Status 2__0 is set. Comparison of the ISR frame counter with the number reported by the frame count contained in Status 2__0 can aid in the ISR software debug.

7.0 TRANSMIT

Transmission is very simple with the 82592. The data is stored in memory. The DMA is initialized to point at the first byte of the data. After the 82592 is given a Transmit command (code 04h, when using channel 0),it will request DMA data transfers, acquire the link as soon as the first byte is stored in the internal transmit FIFO and transmit the data. When transmission is completed, the status field is updated and the INT pin is asserted. However, if a collision occurs, retransmission is performed external to the 82592 by the ELM logic. The ELM hardware uses the EOP# signal to force the 8237A DMA controller to point to the first byte of the frame (DMA is in autoinitialize mode). The 82592 issues a data request to the DMA controller and starts transmission again. This is done without CPU intervention. The IEEE 802.3 time gap of 9.6 uSec for a retransmit attempt (for the first slot) is easily met. Retransmission is attempted until the internal 82592 maximum retry counter expires. In the ELM example, 15 retries will be attempted. Our laboratory experiments indicate that most collisions are resolved within less than 15 retries. In the case of errors the software driver should intervene.

The 82592 is configured for TCI. This causes the 82592 to use its EOP# pin and thus, enables the external ELM hardware to detect transmit collision events. The ELM hardware uses both the EOP# and DRQ lines to force the DMA to autoinitialize for the retransmissions.

The configuration used in the Exerciser software (TCI mode) causes the 82592 to search for a command byte after the last data byte. This byte should always be 0 so it will be interpreted by the 82592 as a NOP command. If the least significant 3 bits in this byte are 100 binary, the 82592 will interpret this byte as a Transmit command and treat the following bytes as the byte count field of a new frame. The 82592 will then attempt to transmit it. XMT chaining is not used in the Exerciser software. The ELM retransmission mechanism uses the 8237A DMA autoinitialize capability. If a chain of frames is stored in the transmit buffer, there will be no way to handle automatic retransmission for all the frames but the first one. There is no way to cause the 8237A DMA controller to jump to the first byte of the n-th frame, required for a retransmission when chaining is used. However, transmit chaining is possible when using the 82560/82561 DMA controller.

In case of a fatal transmission error, the 82592 will signal the event to the CPU through the interrupt and status mechanism. The CPU will issue another Transmit command with the same data (or issue a higher layer activity when not in the ELM Exerciser software environment). The CPU will control the number of times it retransmits the same frame. In case this number exceeds the ELM MAX__RETRY (default = 15. This is a SW variable, not the 82592 internal max retry counter) transmission is stopped and the following message is printed on the screen: "ERROR no. 1 ". The ELM Exerciser software 'MAX__RETRY' default value is 16.

Transmission of an exact number of frames in the range of 1-32000 or an endless number of frames is supported.The background transmit command, found in the EXECUTE menu, communicates with the transmit interrupt service routine (ISR) The XMT__LOOP flag variable is reset by the background routine immediately before transmission and set by the transmit ISR. This handshake allows transmission of 1 frame at a time. The background transmit command polls the XMT__LOOP flag until it is found to be set. The display is updated periodically. To stop the transmission loop hit any key on the keyboard.

8.0 INTERRUPT SERVICE ROUTINE

When an execution command has been completed, or a frame has been received, the 82592 will assert its INT pin. This is true if the BUFFER LENGTH/TCI parameter of the Configure command was 80h. If programmed to C0h, the 82592 will not generate an interrupt upon frame reception. The ELM Exerciser software uses this interrupt to invoke the receive ISR). The 82592 INT pin drives IRQ10 of the system bus. IRQ10 is connected to the slave PIC on the motherboard. The slave PIC output drives one of the master PIC inputs. If the 82592 INT pin is asserted, the slave PIC generates (if not masked) an Int signal to the Master PIC which transfers (if not masked) the signal to the CPU. Both PICs have experienced an INT event. This requires that an EOI be issued to both PICs before exiting the ISR.

The 82592 will present the event that caused interrupt generation in its status register number 0, along with bit 7 set to indicate an unacknowledged interrupt event. The content of the status registers will not be altered until the CPU acknowledges the interrupt. When configured to TCI mode, the 82592 will store one more event in an internal queue. This allows the reception of an endless number of frames, while transmission is performed in parallel and its status stored in this special internal queue. This optional pending event will be presented in the STATUS register after each CPU interrupt acknowledge sequence. The 82592 will use one receive event to report all the frames received from the first INT assertion until its acknowledgement. The 82592 will not wait for the CPU to acknowledge the interrupt; nor will the 82592 generate new interrupts by toggling its INT pin. New 82592 interrupts for all events will be generated after the CPU acknowledges the previous INT request.The CPU can do so by issuing a command to the 82592 with the ACK bit (bit 7), in the 82592 command byte, set.

At program initialization, the assembly language procedure "ini__int" is executed. It saves the local environment pointers (private stack and segment registers) and replaces the current 72h INT Vector in the interrupt table, with a pointer to the int__hnd routine. The int__hnd routine is the routine invoked upon any 82592 interrupt. The original 72h interrupt vector, is saved in a place known to be empty in the DOS interrupt vector table (62h). The ELM Exerciser software will restore the original vector before exiting to DOS at the end of the execution.

CODE EXAMPLE 2

```
;************************************************************
; INI_INT : SET CONDITIONS TO WORK WITH 8259A COMMUNICATION INT
;       IRQ10 IS CONNECTED TO 82592 INT PIN
;       INT VEC 72h IS JUMP TARGET FOR IRQ10MASKING IRQ10 IS DONE
;       AT THE CALLING LEVEL
;************************************************************

_ini_int proc far

        push bp
        mov bp,sp
        push es
        push bx
        mov bx, _mstck2
        mov cs:lss, bx          ;save local stack segment
        mov bx, _mstckl
        mov cs:lsp, bx          ;save local stack pointer
        mov cs:lds, ds          ;save local data segment
        mov cs:les, es          ;save local extra segment
        mov al,vec              ;hex communication interrupt no.
        call sys35              ;es:bx hold returned address
        mov al,62h
        call sys25              ;save for later retrieve in vec 62
        mov al,vec              ;hex int no. of irq10 (LAN)
        mov bx,offset int_hnd   ;address of interrupt handler routine
        push cs
        pop es                  ;es-segment of interrupt handler
        call sys25              ;install via DOS system call
        pop bx
        pop es
        mov sp,bp
        pop bp
        ret
_ini_int endp
```

The assembly language procedure "int.hnd" is invoked when the 82592 interrupts the CPU using IRQ10. First it checks whether this is the first entry of "int.hnd" or if the interrupt service routine has been reentered while the previous interrupt was still being processed. The section on the interrupt service routine discusses interrupt nesting. In the ELM Exerciser Software, 82592 interrupt nesting is prevented. Therefore, the above test should always return a "no nesting" answer. Then the host process environment is stored in memory and the local environment is restored (see the section "Interfacing with DOS"). Next, the machine state is saved by pushing the general purpose registers (AX, BX, CX and DX) and the index registers (SI and DI). The execution until now is considered critical, in the sense that it should not be interrupted. The processor hardware prevents any new interrupts of the same level or lower in priority from interrupting the execution.

The routine "irq10__mask" masks interrupt request 10 of the 8259A PIC. This is the interrupt generated by the 82592. This prevents the 82592 from causing nested interrupts. Now an STI instruction can be issued. This will enable processing of other interrupts by the processor in a timely fashion. An optional approach is discussed later in the section on interrupt nesting.

CODE EXAMPLE 3

```
;*************************************************************
;  INT_HND: INTERRUPT HANDLER FOR IRQ10 (VEC 72)
;  DEALS WITH THE LOWER LEVEL OF THE TREATMENT, AND CALLS
;  THE C HANDLER

;*************************************************************

int_hnd proc

   inc cs:counter     ;saves ss,sp,ds,es only for the first entry
   cmp cs:counter, 1
   jne inter
first_entl:
   mov cs:hsp,sp      ;save registers of host process in memory
   mov cs:hss,ss
   mov cs:hes,es
   mov cs:hds,ds
   mov sp,cs:lsp      ;pop local registers from memory
   mov ss,cs:lss
   mov ds,cs:lds
   mov es,cs:les
inter:
   push bp
   mov bp,sp
   push ax
   push bx
   push cx
   push dx
   push si
   push di
   call _irq10_mask
   sti
   push ax
   mov ax,20h         ;end of interrupt to 8259A Slave
   out 0A0h,al
   mov ax,20h         ;end of interrupt to 8259A Master
   out 20h,al
   pop ax
ll:
   call _c_int
```

CODE EXAMPLE 3 (Continued)

```
exit:
  call _parazit_rcv
  cli                ;enter a critical section must disable int only
                     ;if eoi is before this section
  call _irq10_unmask
  pop di
  pop si
  pop dx
  pop cx
  pop bx
  pop ax
  mov sp,bp
  pop bp
  dec cs:counter     ;nesting check
  cmp cs:counter,0
  jnz inexit
last:
  mov sp,cs:hsp      ;pop host registers from memory
  mov ss,cs:hss
  mov ds,cs:hds
  mov es,cs:hes

inexit:
  sti
ext_int:
  iret
int_hnd endp
```

Before program control is transferred to the "C" routine "c_int()" the two 8259A PICs are acknowledged. The "c_int()" routine reads the 82592 status from the 6 byte status register. To make sure Status byte 0 is the first byte read and that the following bytes are read in order, a RLS_PTR (0Fh) command, with 0 as a pointer value (points to Status_0), is issued. If no command is issued to the 592 during the Status read sequence, each successive read to the 82592 will increment the internal status pointer and guarantee properly ordered status bytes. Now the 82592 Interrupt can be acknowledged. This allows the next event to be presented in the status register, and the INT pin to be asserted. Since the PIC is masked all further interrupts from the 82592 will be ignored until the ISR has been exited. The 'INT_PROC_Stat' variable holds the interrupt event presented in STATUS_0. For receive and transmit events, a lengthy processing is required. It is described below.

CODE EXAMPLE 4

```
far c_int()
{
register a; /* for a faster detection of the INT event */

      write_592(0x0F);          /* RLS_PTR to 0 command */
      get_592_status();         /* IN operation from the 82592 */
      outp(ADDR_592,0x80);      /* acknowledge to 82592 */
      a = STAT_REG_Stat[0];     /* Status byte 0 holds the event */
      if (a & 0x80)             /* INT bit active */
        {
        a &= 0x0F;              /* clear irelevant bits */
        switch (a)
        {
        case 8:
              INT_PROC_Stat = "RCV"; /* last INT event for screen display */
              int_rcv();        /* call the RCV ISR */
              break;

              case 4:
                INT_PROC_Stat = "XMT";
                int_xmt(); /* TRANSMIT event */
                break;

              case 12:
                INT_PROC_Stat = "ReX"; /* Re-TRANSMIT event */
                int_xmt();
                break;

              case 1:
                INT_STAT_RDY = TRUE; /* Individual Address */
                INT_PROC_Stat = "IA "; /* executed */
                break;

              case 2:
                INT_STAT_RDY = TRUE; /* Configure executed */
                INT_PROC_Stat = "CNF";
                break;

              case 5:
                INT_STAT_RDY = TRUE; /* TDR executed */
                INT_PROC_Stat = "TDR";
                break;

              case 6:
                INT_STAT_RDY = TRUE; /* Dump Executed */
                INT_PROC_Stat = "DMP";
                break;
```

CODE EXAMPLE 4 (Continued)

```
case 7:
  INT_RDY = TRUE; /* Diagnose Passed */
  INT_PROC_Stat = "DgP";
  break;

case 10:
  INT_STAT_RDY = TRUE; /* RCV Aborted */
  INT_PROC_Stat = "RxD";
  break;

case 13:

  INT_STAT_RDY = TRUE; /* Execution aborted */
  INT_PROC_Stat = "ExA";
  break;

case 15:
  INT_STAT_RDY = TRUE; /* Diagnose Failed */
  INT_PROC_Stat = DgF";
  break;

default:
  INT_STAT_RDY = TRUE;
  INT_PROC_Stat = " ? ";
  break;

  } /* end switch */
                } /* end if */
  } /* exit interrupt handler */
```

Before leaving the "int__hnd" interrupt service routine, the 82592 interrupt is unmasked in the 8259A PIC. The machine state is then restored by popping the registers pushed on the stack at the entry of the interrupt handler. The nesting control 'counter' variable is decremented and finally the host process environment is restored.

8.1 RCV Interrupt Service Routine: "int__rcv( )"

The RCV interrupt service routine handles the reception of one or more frames. All the frames received are stored in memory in successive addresses by the 8237A DMA controller. After the 82592 has completed receiving a frame it interrupts the CPU. The 82592 will continue to receive frames as long as the DMA continues to service its requests to store data in memory.

The "int__rcv( )" routine implements a cyclic RCV buffer handling. The buffer size is configurable. The ELM Exerciser software uses 16 KB, a smaller size can be used if interrupt latency is small enough. The maximum receive buffer size for this architecture is limited to 128 kB due to the physical page register implementation of the PC-AT DMA subsystem. In the ELM it is further limited to 64 kB due to the fact that A0 through A15 are latched in the TCI latches. If A1 through A16 are latched then the full 128 kB that the DMA can access could be used as a receive buffer. To allow maximum buffer size, the ELM Exerciser software locates its receive buffer at the beginning of a physical address segment, i.e. address of type xxOOOOh, (which of course is not a must). This allows a simplification in address calculation because the lowest 16 bits can be directly used for address calculations. There is no need to add a displacement from the beginning of the physical address segment.

The receive ISR starts with the frame received last (pointed by the latch) and processes received frames, backwards. The last received frame byte count field is used to find the byte count field of the next to last frame. After the pointer to the beginning of the frame has been reproduced, the received frames can be processed. This goes on until the first frame has been found. The frames can then be processed in the order in which they were received.

Throughout the RCV ISR session, the length of the current frame is being subtracted from the "Cur-Latch-abs" variable (pointer to the current frame Byte Count field), to obtain the previous frame's byte count field. When the 'Cur-Latch-abs' variable points to the byte preceding the first byte received in this session of the ISR, the receive ISR has completed processing all the frames received in this session.

As seen in figure 5, the 82592 appends the frame's status and byte count fields, after the last data byte. In the case of odd frame length (data field in bytes), the 592 will leave one byte empty so the status and byte count are stored on a word boundary when the 82592 is set to 16 bit mode.

8.2 Execution Algorithm

READING THE LATCH

The first action taken in the receive ISR is to read the 16-bit latch. It holds the low 16 bit physical address of the byte count field of frame number N, the last completely received frame (latch_content = 470h in the example of figure 1). Note that more than 1 frame can

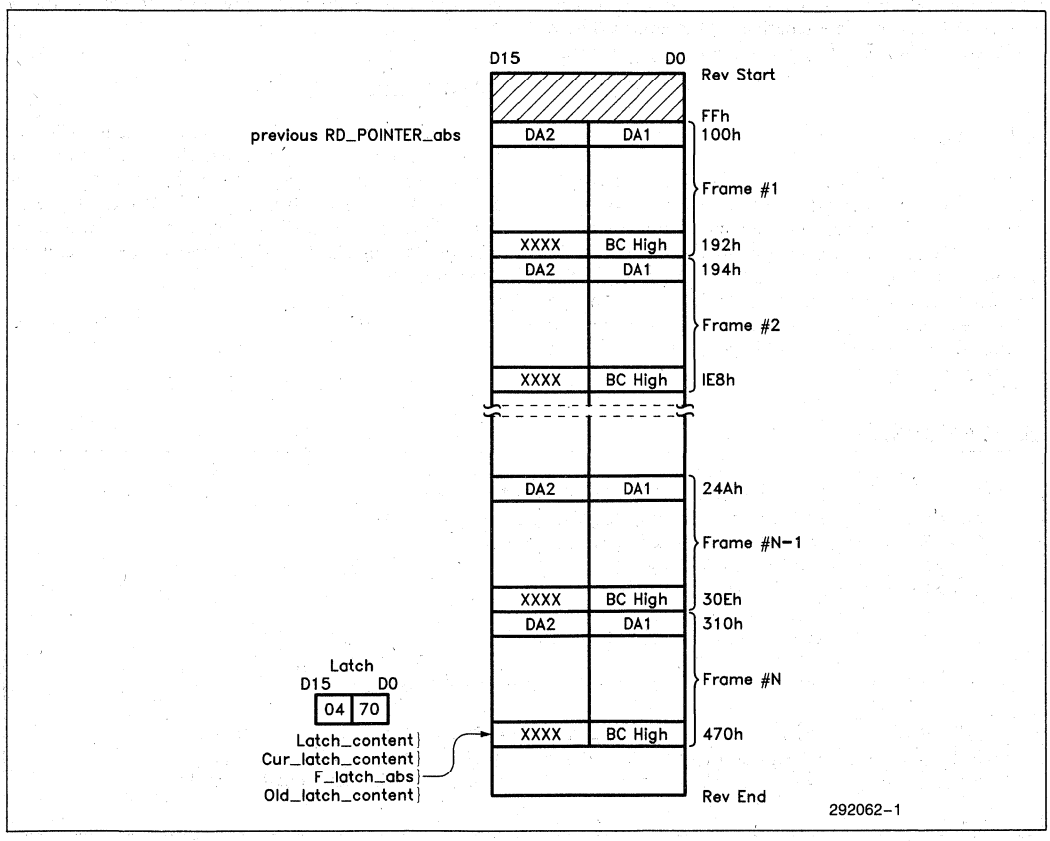


Figure 5. Memory Organization of RCV Frames

be handled by "int_rcv()" in one session. It is not necessarily the very same frame, designated 1, that its reception completion has generated the INT signal.In this program, the latch is read twice. The latch is read in two 8 bit accesses since 16-bit I/O is not supported so the latch content can be altered in between. As the DMA can work in parallel with INT processing, it can capture the bus during the receive ISR latch read operation. If a frame was completed exactly at this time, a new byte count field pointer is loaded into the latch. The second receive ISR read operation will then get half of the new BC field pointer). If the read operations yield the same result, the data is valid. In case of mismatch, the read operations are repeated. Latch read action is very short compared with frame reception, so there will be no need for more than two iterations. This can also be accomplished by doing a word read at the low byte of the two latches. Since the ELM does not return MEMCS16 the processor will execute two back to back byte reads at consecutive I/O locations. These two reads are locked thus pointer integrity is guaranteed.

FALSE ALARM

A "false alarm" is detected by the fact that the latch content is equal to its value in the previous service sequence. If the latch was not updated no frame was received since the last receive ISR invocation. If exactly the buffer size bytes were stored in memory from the last time a receive was serviced, this test may indicate wrong results. However, the buffer size should be big enough to accommodate the longest CPU latency in servicing interrupts.

BYTE COUNT AND STATUS

The frame length is read from the byte count field (two words). The general status maintained by the ELM Exerciser software is updated from the status field attached to the end of the frame before the byte count field. Then the frame may be copied to the user's application receive area. By now one frame has been received and processed (FRTT_CNT = 1).Note that the byte count field includes the number of the destination address field bytes, source address field bytes, information field (type field included) and two status bytes. The Byte Count field itself (four bytes) plus two bytes of the status field are not included in the calculation. In 16-bit mode, the 82592 extends the status and byte count fields to words instead of bytes This is treated in more detail in the 82592 User Manual. The previous frame's byte count field has now been located (byte count + 6) bytes previous to the current one. See figure 5.

DATA CHECK

This utility allows checking the received data. The basic assumption is, of course, that the data is known in advance so we have a reference to compare with. In the ELM case, the data transmitted is sequential (word wide), in case the user did not change the default transmission data blocks.This is illustrated in the following code example.

CODE EXAMPLE 5

```
/*********************************************
* checks the data words to be sequential data *
* 0,1,2, if data length is longer than 4. *
* length = 4 data = 0000 0000 *
* length = 5 data = 0000 0001 00 *
* length = 6 data = 0000 0001 0000 *
* length = 7 data = 0000 0001 0002 00 *

*********************************************/

/* RD_POINTER_seg.Ptr is pointing to the first byte of the frame
Cur_Latch_abs is pointing to the byte count of the previous frame
Prev_latch_abs is pointing to the byte count of the current frame
*/
```

CODE EXAMPLE 5 (Continued)

```
find_boundary(f_byte)
unsigned int f_byte; /* offset of first byte in frame */
{

/* Don't compare last 5 words, 2* BC, 2* STATUS + last word is zero */
  if_byte < Prev_latch_abs)
    return((Prev_latch_abs - f_byte - 10)/2);
  else if(f_byte > Prev_latch_abs)
    {
    if((Prev_latch_abs - Sixteen_1) > 10)
        {
        wrap_required = TRUE;
        return((RD_END_abs - f_byte)/2);
        }
    else
        return(((RD_END_abs - f_byte)/2) - (10 - (Prev_latch_abs -
Sixteen_1)));
}
else
        }
        RX_CHECK = TRUE; RX_CHECK_DT = 0xa5a5; RX_CHECK_BNUM = 0xa5a5;
        }
}

wrap()
{
wrap_around = TRUE;
bound_dist == find_boundary(Sixteen_1);
local.temp.xoffset = Sixteen_1;
}

check_data(f_byte)
unsigned int f_byte; /* offset of first byte in frame */
{
int i,j;
int far * tmp;
unsigned int tmp1, delta, dtlen;

/* reset flags */

  wrap_around = wrap_required = FALSE;

/* first find buffer boundary cross point from first DA word if any */
  bound_dist = find_boundary(f_byte);

/* second find DA and compare to IA */
  local.tmp = mk_pointer(Prev_latch_abs, -(Cur_Byte_Count + 4));

  if (*(local.tmp++) !=IASU.IASU_Ptr -- IaAdd1)
    {
    RX_CHECK = TRUE; RX_CHECK_DT = *(local.tmp); RX_CHECK_BNUM = 0xaa1;
return;
    }

  if (bound_dist- == 0)
    wrap();
```

CODE EXAMPLE 5 (Continued)

```
  if (*(local.tmp++) != IASU.IASU_Ptr -> IaAdd2)
    {
    RX_CHECK = TRUE; RX_CHECK_DT = *(local.tmp); RX_CHECK_BNUM = 0xaa2;
return;
    }

  if (bound_dist- == 0)
    wrap();
  if (*(local.tmp++) !=IASU.IASU_Ptr -> IaAdd3)
    {
    RX_CHECK = TRUE; RX_CHECK_DT = *(local.tmp); RX_CHECK_BNUM = 0xaa3;
return;
    }

/* second, disregard SA */

  if (bound_dist- == 0)
    wrap();
  local.tmp ==;
  if (bound_dist- == 0)
    wrap();
  local.tmp ++;
  if (bound_dist- == 0)
    wrap();
  local.tmp ==;

/* third check if frame includes data != 0 using LEN field */

  if (bound_dist- == 0)
    wrap();
  if ((dtlen = *(local.tmp++)) == 0)
    {
    RX_CHECK = TRUE; RX_CHECK_DT = *(local.tmp); RX_CHECK_BNUM = 0xaa6;
return;
    }

/* if frame includes data != 0 compare all bytes till boundary cross */

  i = 0;

  if (bound_dist > 0)
    {
    for (i=0; i<bound_dist; i++)
        {
        if(*(local.tmp++) = i)
          {
        RX_CHECK = TRUE; RX_CHECK_DT = *(local.tmp); RX_CHECK_BNUM = i;
return;
          }
        }
    }
```

CODE EXAMPLE 5 (Continued)

```
/* if frame includes data != 0 compare all bytes after boundary cross */
  if (wrap_required && !wrap_around)
    {
    bound_dist = find_boundary(Sixteen_l);
    local.temp.xoffset == Sixteen_l;
    for (j=0; j<bound_dist; j++)
        {
        if(*(local.tmp++) != (j+i))
          {
        RX_CHECK = TRUE; RX_CHECK_DT = *(local.tmp); RX_CHECK_BNUM = i+j;
        return;
          }
        }
    }
/* compare LEN field */
  if ((dtlen % 2) == 1)
    dtlen++;
  if (dtlen != Cur_Byte_Count-16)
    {
    RX_CHECK = TRUE; RX_CHECK_DT = dtlen; RX_CHECK_BNUM =0xdd7;
    return;
    }
/*compare status bytes */

}
/*****************************************************************************/
```

main loop

The main loop of the receive ISR determines if there is "more__to__read()", i.e., whether the ISR had serviced all the frames received this time or more frames are stored in memory and were not processed yet. The algorithm of "more__to__read()", may be found in code example 8.

Each received frame is processed following the algorithm depicted above for the first frame traced by the "int__rcv( )" routine.

NOTE Software Variable Naming Convention

The structure "VARIABLE.Ptr" holds a valid "C" pointer to a specific memory structure. It is arranged in a way that allows access to its two 16-bit entities, the offset and the segment (or base). These are "VARIABLE.Addr.xoffset" and "VARIABLE.Addr.xsegment". Another suffix is used for variables holding the 16 low bits of a physical address. These are denoted by "VARIABLE—abs".

CODE EXAMPLE 6

```
/*********************************************************************
The RECEIVE INT service routine
RCV_AREA.START_Ptr is always at the beginning of a Physical segment *
RD_POINTER_seg is a temp variable
*********************************************************************/

int_rcv()
{
int i,k;
unsigned int cbc_h;
int trace_cnt;
int far * tmp_ptr;
unsigned int tmp, tmp_l;
```

CODE EXAMPLE 6 (Continued)

```
int false_int;

if ((i=latch_read()) != 0)
 {
 return;                        /* in case of fatal error */
 }                              /* Compile with DEB_PRT switch,
                                          to monitor this event */

Old_latch_content = latch_content; /* for parazit_rcv() */

Cur_Latch_abs = latch_content;     /* = 470h */

F_Latch_abs = latch_content;                    /* = 470h */
RD_POINTER_seg.Ptr = mk_pointer(F_Latch_abs, 2); /* 472h */
if (RD_POINTER_abs == RD_POINTER_seg.Addr.xoffset)
 { /* 100h == 472 */
 false_int++;               /* Compile with DEB_PRT switch,
                                          to monitor this event */
 return;
 }

cbc_h = * (mk_pointer(Cur_Latch_abs,0));        /* 470h */
Cur_Byte_Count = * (mk_pointer(Cur_Latch_abs, -2)); /* 46Eh */
Cur_Byte_Count = (cbc_h << 8) + (Cur_Byte_Count & 0x00FF);
if ((Cur_Byte_Count % 2) == 1) Cur_Byte_Count++; /* odd data bytes but all
fields are word aligned */

tmp_ptr = mk_pointer(Cur_Latch_abs, -6); /* status 46Ah */

if (*(tmp_ptr) & 0x00EF)
 {
 RX_ERR = TRUE;
 if (*(tmp_ptr) & 0x0080)
      SCB.SCB_Ptr -> SRT_FRM += 1;
 if (*(tmp_ptr) & 0x0040)
      SCB.SCB_Ptr -> NO_EOF += 1;
 if (*(tmp_ptr) & 0x0020)
      SCB.SCB_Ptr -> TOO_LNG += 1;
 if (*(tmp_ptr) & 0x0008)
      SCB.SCB_Ptr -> NO_SFD += 1;
 if (*(tmp_ptr) & 0x0004)
      SCB.SCB_Ptr -> NAD_MCH += 1;
 if (*(tmp_ptr) & 0x0002)
      SCB.SCB_Ptr -> IA_MCH += 1;
 if (*(tmp_ptr) & 0x0001)
      SCB.SCB_Ptr -> RCV_CLD += 1;
 }
tmp_ptr = mk_pointer(Cur_Latch_abs, -4);
if(*(tmp_ptr) & 0x0020)
 {
 SCB.SCB_Ptr -> RCV_OK += 1;
 UPDATE_RXCNT = TRUE;
 }
```

CODE EXAMPLE 6 (Continued)

```
if(*(tmp_ptr) & 0x001D)
 {
 RX_ERR = TRUE;
 if (*(tmp_ptr) & 0x0010)
      SCB.SCB_Ptr -> LEN_ERR += 1;
 if (*(tmp_ptr) & 0x0008)
      SCB.SCB_Ptr -> CRCErrs += 1;
 if (*(tmp_ptr) & 0x0004)
      SCB.SCB_Ptr -> AlinErrs += 1;
 if (*(tmp_ptr) & 0x0001)
      SCB.SCB_Ptr -> OvernErrs+= 1;
 }

FRTT_CNT = 1;       /* Frames Received This Time count.Compile with DEB_PRT
switch, to monitor number of frames received each int_rcv invocation */
Prev_latch_abs = Cur_Latch_abs; /* for Data Check */
RD_POINTER_seg.Ptr = mk_pointer(Cur_Latch_abs, -(Cur_Byte_Count + 6));
Cur_Latch_abs = RD_POINTER_seg.Addr.xoffset; /* previous frame BC pointer
308h */
RD_POINTER_seg.Ptr = mk_pointer(Cur_Latch_abs, 2); /* frame N 1st byte 310h
*/
tmp = RD_POINTER_seg.Addr.xoffset;

/******* Data Check ********
      RD_POINTER_seg.Ptr points to the first byte of the frame
      Cur_Latch_abs points to the byte count of the previous frame */

if (CheckEnable == ON)
 check_data(tmp);
while ((k = more_to_read(tmp)) != 0)         /* MAIN LOOP */
 {
 cbc_h = * (mk_pointer(Cur_Latch_abs,0));
 Cur_Byte_Count = * (mk_pointer(Cur_Latch_abs, -2));
 Cur_Byte_Count = (cbc_h << 8) + (Cur_Byte_Count & 0x00FF);
 if ((Cur_Byte_Count % 2) == 1)
       Cur_Byte_Count++;

 tmp_ptr = mk_pointer(Cur_Latch_abs, -6);
 if (*(tmp_ptr) & 0x00EF)
      {
      RX_ERR = TRUE;
      if(*(tmp_ptr) & 0x0080)
       SCB.SCB_Ptr -> SRT_FRM += 1;
      if(*(tmp_ptr) & 0x0040)
       SCB.SCB_Ptr -> NO_EOF += 1;
      if(*(tmp_ptr) & 0x0020)
       SCB.SCB_Ptr -> TOO_LNG += 1;
      if(*(tmp_ptr) & 0x0008)
       SCB.SCB_Ptr -> NO_SFD += 1;
      if(*(tmp_ptr) & 0x0004)
       SCB.SCB_Ptr -> NAD_MCH += 1;
      if(*(tmp_ptr) & 0x0002)
       SCB.SCB_Ptr -> IA_MCH += 1;
      if(*(tmp_ptr) & 0x0001)
       SCB.SCB_Ptr -> RCV_CLD += 1;
      }
 tmp_ptr = mk_pointer(Cur_Latch_abs, -4);
```

CODE EXAMPLE 6 (Continued)

```
if(*(tmp_ptr) & 0x0020)
     {
     SCB.SCB_Ptr -> RCV_OK += 1;
     UPDATE_RXCNT = TRUE;
     }
if(*(tmp_ptr) & 0x001D)
     {
     RX_ERR = TRUE;
     if (*(tmp_ptr) & 0x0010)
      SCB.SCB_Ptr -> LEN_ERR += 1;
     if(*(tmp_ptr) & 0x0008)
      SCB.SCB_Ptr -> CRCErrs += 1;
     if (*(tmp_ptr) & 0x0004)
      SCB.SCB_Ptr -> AlinErrs += 1;
     if (*(tmp_ptr) & 0x0001)
      SCB.SCB_Ptr -> OvernErrs+= 1;
     }

Prev_latch_abs = Cur_Latch_abs;     /* for Data Check */
RD_POINTER_seg.Ptr = mk_pointer(Cur_Latch_abs, -(Cur_Byte_Count + 6));
Cur_Latch_abs = RD_POINTER_seg.Addr.xoffset;
RD_POINTER_seg.Ptr = mk_pointer(Cur_Latch_abs, 2);
tmp = RD_POINTER_seg.Addr.xoffset;

/******* Data Check ********

     RD_POINTER_seg.Ptr points to the first byte of the frame
     Cur_Latch_abs points to the byte count of the previous frame */
if(CheckEnable == ON)
     check_data(tmp);
FRTT_CNT++;
if (FRTT_CNT >1250)
     break;              /* GUARD BAND */
};   /* MAIN LOOP END */

RD_POINTER_seg.Ptr = mk_pointer(F_Latch_abs, 2);
RD_POINTER_abs = RD_POINTER_seg.Addr.xoffset; /* next int_rcv session 1st
byte 472h */
RxCount += FRTT_CNT;
copy_data();     /* from RCV buffer to user's application */
INT_STAT_RDY = TRUE;     /* for screen status update */
return;       /* end of int_rcv() */
}
```

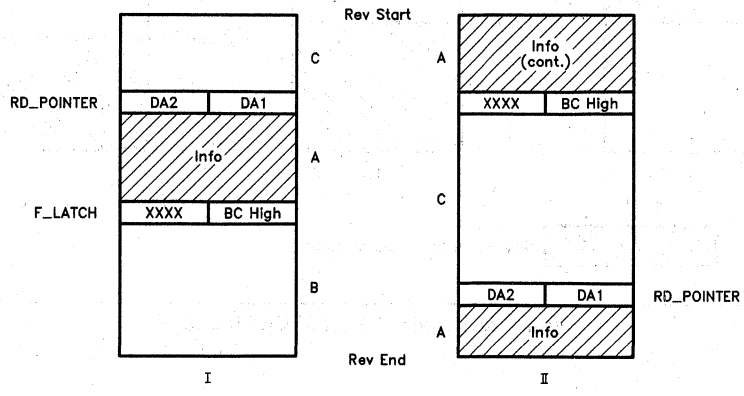
address calculation

In code example 3, the routine "mk__pointer(address, delta)" is used. This routine handles a 16-bit physical address to pointer conversion (see Code Example 7) and the buffer internal address calculation (which are not straightforward).

As noted before, the RCV buffer is a cyclic buffer. When calculating an address starting from one address and adding/subtracting some 'delta' value, the buffer boundaries can be crossed. Overlapping the buffer boundaries is further illustrated in Figure number 6. When subtracting the byte count of frame N, which starts in (1) and ends in (2), a direct decrement will not give the right results. If we calculate backwards and fall out of the receive area, (address region within which the buffer for all frames resides) "mk-pointer()" adds the length of the receive area to get the right address. If we go in the other direction, "mk__pointer" subtracts the receive area length. Due to receive area alignment into a physical address segment, the case in which the address we get is lower than RCV__AREA__START is impossible.

CODE EXAMPLE 7

```
/*************************************
* mk_pointer transforms a 16 bit  *
* abs to a pointer within the RCV_AREA *

*************************************/

int far *
mk_pointer(abs,delta)
unsigned int abs;
int delta;
{
unsigned int The_length;

The_length = (unsigned) DECRCVBLEN; /* buffer length */
RD_POINTER_seg.Addr.xoffset = abs + delta;
if(RD_POINTER_seg.Addr.xoffset > RD_END_abs)
  {
/* check if going backwards */
if (delta < 0)
  RD_POINTER_seg.Addr.xoffset += The_length;
/* check if going forward */
if (delta > 0)
  RD_POINTER_seg.Addr.xoffset -= The_length;
  }

/* if result is before RCV buffer starts in the segment. */

/* impossible result for aligned RCV_AREA */

return(RD_POINTER_seg.Ptr);

}
```



292062-2

The frames begin at RD__POINTER and end at F__LATCH.

Figure 6. Overlapped Buffer Boundary

Another address calculation is performed to determine whether there is "more_to_read()" (see code example 8) i.e., whether all the frames received this time were processed and whether the next frame to be read is valid. Cases of invalidity are detected by this routine. This error is denoted "SW_OVERRUN", for software overrun errors. In case the interrupt handling was too slow, the DMA could have over written received frames which were not processed yet. In this case, a valid byte count field can be replaced by arbitrary data. This will mislead the backwards address calculation. It may also cause a detectable address error, where the previous frame byte count field is located in an area known to be outside the last valid receive area. The following variables are used in this calculation:

Cur-Latch-abs, labeled CUR LATCH in figure 7, is used to hold the address of the current frame's byte count in memory, while the ISR processes them. At the beginning, it holds the last frame's byte count (frame n), then it will hold frame n-1's byte count address and so on.

F-latch-abs, labeled NEW LATCH in figure 7, holds the address of the byte count of the last received frame (frame N) in this "int_rcv()" invocation. It is used to detect a frame that is wrapped-around the end of the receive buffer.

RD-POINTER-abs, labeled OLD LATCH in figure 7, always points to the first byte of the next frame to be processed in the next "int_rcv()" invocation. It is updated at the end of each service cycle, to hold F-latch-abs + 2. (Note: 'RD-POINTER-abs' is not equivalent to 'RD-POINTER-seg.Ptr' which serves as a template for various pointer calculations.)

The following scenarios are possible for such intermediate results.

Legal intermediate results should fall in the "A" area if the results are in "A", namely between RD-POINTER (start of this session RCV-AREA) and F-LATCH (First Latch, end of this session RCV-AREA). Note that F-LATCH may be greater or smaller than RD-POINTER. See Figure 7A.

If however, the result falls in "B" or "C" areas it is not legal. This may happen if the buffer was too small and the 82592 had overrun it in reception. This would occur if the whole buffer was filled and then more frames were received, before being processed by the ISR. See figure 7B.

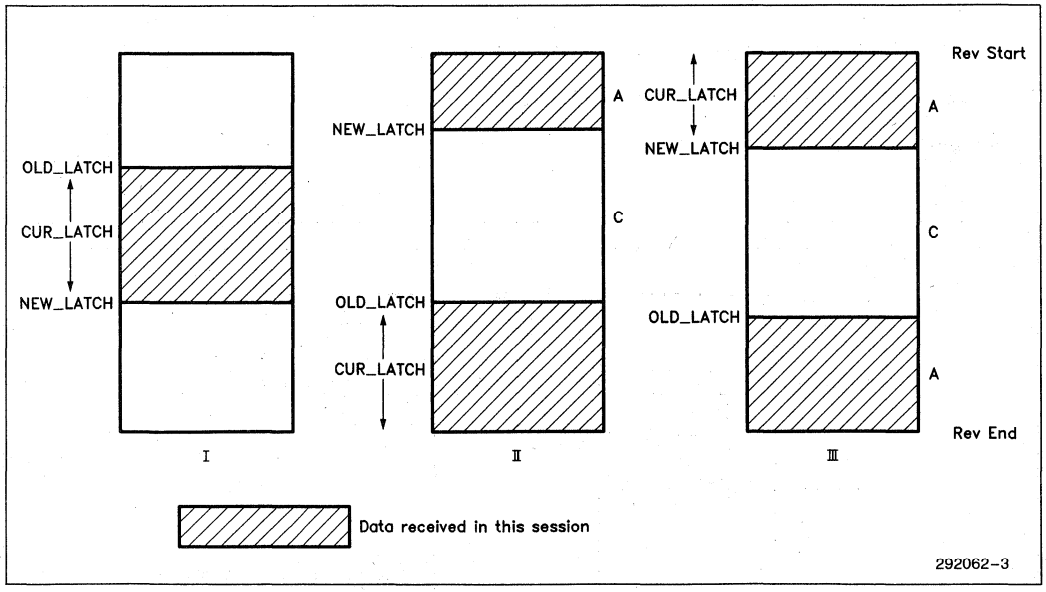


Figure 7a. Legal "CUR_LATCH" Values

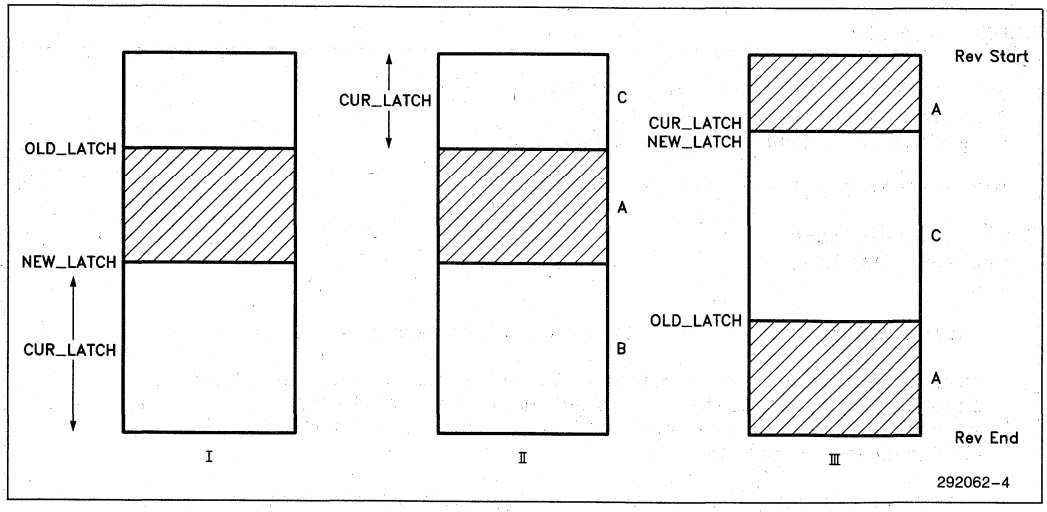


Figure 7b. More Frames TO READ

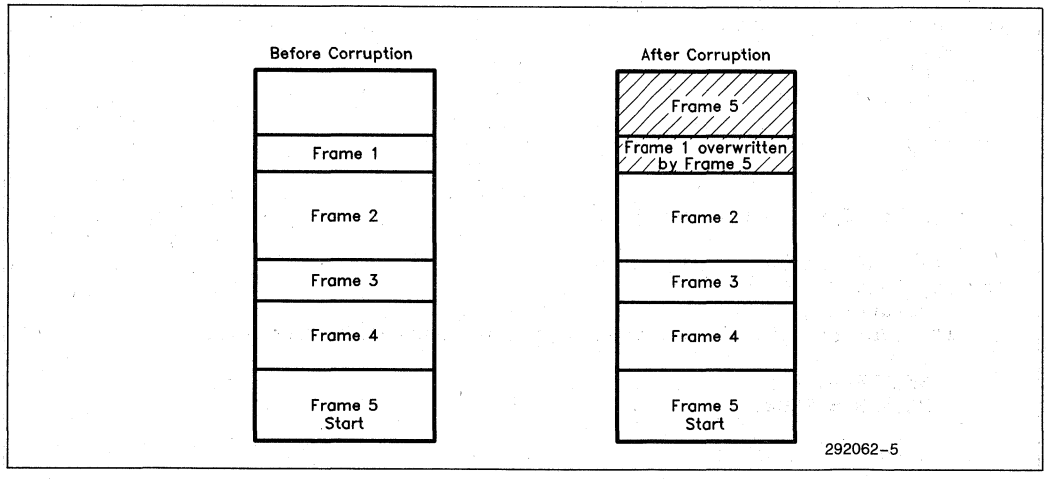


Figure 8. Receive Buffer Corrupted by Overrun

Frame #5 erases the byte count information stored by frame #1. When we calculate our way backwards we will not find the byte count field of frame #1 and read instead a data word of frame #5. This data may be interpreted wrong and cause an avalanche in the frame reconstruction process. This event can be detected by the above test.

This can be completely avoided if a stop register is implemented in hardware. This register would hold RD-POINTER (the beginning of the current RCV-AREA) and prevent write access to areas previously written by 592/DMA and not serviced yet. In case the maximum delay of the system is determined and known, one can allocate a big enough buffer to accommodate that delay in the ISR. CRC should be stored in memory and checked by the S/W to assure data integrity.

CODE EXAMPLE 8

```
/***********************************
* checks whether more frames to *
* be read as indicated by the latch *

***********************************/

more_to_read(tmp)
unsigned int tmp;
{

  unsigned int cur_content, f_content, rd_content;

  cur_content = tmp; /* Cur_Latch_abs [flow chart: cur latch] */
  f_content = F_Latch_abs; /* [flow chart: new latch] */
  rd_content = RD_POINTER_abs; /* [flow chart: old latch] */
  if(f_content > rd_content)
    {
    if((cur_content > rd_content) && (cur_content <= f_content))
      return(1);
    if ((cur_content < rd_content)
(cur_content > f_content))
      {
      SW_OVR++;
      RX_ERR = TRUE;
      }
return(0);
    }

  if (f_content < rd_content)
    {
    if ((cur_content > rd_content)
(cur_content <= f_content))
      return(1);
    if ((cur_content < rd_content) && (cur_content > f_content))
      {
      SW_OVR++;
      RX_ERR = TRUE;
      }
    return(0);
  }
}
```

CODE EXAMPLE 9

```
far parasite_rcv()
{
int i;
unsigned int j;

if((i=latch_read()) != 0)
  {
  LATCH_ERROR++;
  return;
  }
if((RxCount != 0) && (latch_content != Old_latch_content))
  {
  PARASITE++;
  int_rcv();
  }
}
```

Frames may be received during execution of the receive ISR. These frames are handled before the receive ISR is exited. This saves the interrupt latency involved and the context switching overhead. Therefore, it will increase the overall performance of the code. For all interrupt events, the "parasite_rcv" routine is invoked, before exiting the "int_hnd" routine (see code examples 9 and 3). The routine "parasite_rcv" checks whether a frame has been received using the address latch for indication. It compares the current 'latch_content' with the latest value known to this ISR invocation (stored in 'Old-latch-content'). Obviously, a different value will indicate that a new frame has been received from the time this ISR has been invoked until it is about to be exited.

In the case of an odd frame length (data field in bytes), the 592 will leave one byte empty so the status and byte count are stored on a word boundary (when the 82592 is set to 16 bit mode).

8.3 Transmit Interrupt Service Routine: "int_xmt( )"

The transmit ISR is simpler than the receive ISR. It mainly deals with status update and software generated retransmissions. First, collision status is checked. Num_Coll is the number of times this frame has experienced a collision during a transmission attempt. Num_Coll is equal to, or greater than 0, and smaller or equal to the configuration parameter Max Retry. FRM_COLL is an indication generated by the ELM Exerciser software based on the status reported by the 82592. All the other status reports reflect the 82592 status report, with no further processing. FRM_COLL contains the number of frames that have experienced at least one collision. COLL indicates that the last transmission attempt has experienced a collision; but, transmission was stopped due to other fatal events). MAX_COLL indicates that the 82592 has attempted to transmit this frame "Max Retry" times plus 1. All these attempts have experienced collisions.

The events of transmit deference, heartbeat and frame too long are merely registered and transmission is considered successful. A frame too long error may indicate a hardware error that caused the 82592 to load an incorrect value into its byte count counter; e.g., 700h was loaded into the 82592 byte count counter. However, 700h is greater than the maximum allowed length of an Ethernet frame. This could have happened because of a hardware or software malfunction.

The events of underrun, lost CRS, lost CTS, late collision or Max—Coll indicate a fatal error with which the 82592 cannot cope. The decision taken here is to retransmit in these cases. However, this decision can be left for a higher software layer, where such a layer exists.A status report mismatch is fixed for the MAX_COLL status. There is a special case when the 82592 is configured for 15 retries. After 15 retries it increments the internal counter to hold (15 + 1)MOD16 and hence Num_Coll holds zero. In this case 16 is added to Num_Coll. A frame transmission that has been completed successfully still may have suffered from collisions. Hence, this field should be checked even in TX_OK cases.

Next, the statistics update flag is set ('INT_STAT_RDY').

The last section of this code handles the counter load for XMT LOOP cases.

CODE EXAMPLE 10

```
/*****************************
* XMT interrupt service *

/*****************************

int_xmt()
{
TxCount++;        /* update XMT frame counter */
                  /* for all events, do */
SCB.SCB_Ptr->Num_COLL += (STATUS.STATUS_Ptr-> Status_1_0 & 0x000F);
if((STATUS.STATUS_Ptr -> Status_1_0 & 0x000F)
(STATUS.STATUS_Ptr -> Status_1_0 & 0x0020) )
 SCB.SCB_Ptr->FRM_COLL++;
if(STATUS.STATUS_Ptr -> Status_1_1 & 0x0080)
SCB.SCB_Ptr->COLL += 1;
/* status report, only */
if(STATUS.STATUS_Ptr-> Status_1_0 & 0x00D0)
 {
 if(STATUS.STATUS_Ptr->Status_1_0 & 0x0080)
      SCB.SCB_Ptr->TX_DEF += 1;
 if(STATUS.STATUS_Ptr->Status_1_0 & 0x0040)
      SCB.SCB_Ptr->HRT_BEAT += 1;
 if(STATUS.STATUS_Ptr->Status_1_0 & 0x0010)
      SCB.SCB_Ptr->FRTL += 1;
 TX_ERR = TRUE;       /* for status display purposes */
}

/* Fatal errors, ELM Software Package initiates another transmission of

the frame */
 if( (STATUS.STATUS_Ptr->Status_1_1 & 0x000F)
(STATUS.STATUS_Ptr->Status_1_0 & 0x0020))
 {
 if(STATUS.STATUS_Ptr->Status_1_1 & 0x0001)
      SCB.SCB_Ptr->UndernErrs+= 1;
 if(STATUS.STATUS_Ptr->Status_1_1 & 0x0004)
      SCB.SCB_Ptr->LOST_CRS+= 1;
 if(STATUS.STATUS_Ptr->Status_1_1 & 0x0002)
      SCB.SCB_Ptr->LOST_CTS+= 1;
 if(STATUS.STATUS_Ptr->Status_1_1 & 0x0008)
      SCB.SCB_Ptr->LTCOL += 1;
 if(STATUS.STATUS_Ptr-> Status_1_0 & 0x0020)
      {
      SCB.SCB_Ptr->MAX_COLL += 1;
      if(!(STATUS.STATUS_Ptr->Status_1_0 & 0x000F))
       SCB.SCB_Ptr->Num_COLL += 0x10;
      }

  FLAG_RE_XMT  =  TRUE;        /* signals  Re_XMT  is  required */
 INT_STAT_RDY = TRUE;
TX_ERR = TRUE;     /* for status display purposes */
 return;
 }
/* xmt ok */
 if (STATUS.STATUS_Ptr->Status_1_1 & 0x0020)
 {
 SCB.SCB_Ptr->TX_OK += 1;
 UPDATE_TXCNT = TRUE;
 RETRY_CNT = 0;
 }
```

CODE EXAMPLE 10 (Continued)

```
/********** statistics **************/

INT_STAT_RDY = TRUE;
/********* if xmt loop ***************/ TermCount-;
/* reload running counter */
if((XMT_FOREVER == TRUE) && (TermCount == 0))
 TermCount = 0x7FFF;
if (TermCount > 0)
 XMT_LOOP = TRUE; /* set condition for next frame transmission, if
transmission in loop is requested */

return;
}
```

9.0 SOFTWARE DESIGN HINTS

9.1 Segment Boundaries

The PC AT DMA subsystem uses an 8-bit page register which allows 24 bit addressing. The page register divides the 16 MB memory space into 128KB physical segments in the 16-bit channels. This is because the address 0 through 15 generated by the 8237A drive address 1 through 16 of the system. in the 16-bit channels the page register is used to generated address 17 through 23. If a buffer is allocated so that it lies in two physical segments, a special logic should take care of segment boundary crossing and update the page register. To prevent this complicated logic, memory buffer allocation should prevent physical segment boundary crossing.

9.2 Set Pointer

The internal status registers of the 82592 are read in sequence. One of the ISR operations is reading the status. One should make sure that the first byte of status is read first. The background utilities can be interrupted while reading the status. In that case, the internal 82592 status register pointer is not set to 0. However, the background software will now get the status pointer set to 0, while it expected it to be different. Hence, during status read in the background, the interrupts should be disabled.

9.3 Interfacing with DOS

The 82592 can interrupt the program at any instant. Since the ELM Exerciser software is run under the DOS, the 82592 may interrupt a DOS system call. DOS saves a small stack for its own use. It does not support other uses for this stack. Calling routines or passing parameters by using the DOS stack may cause a stack overflow error and consequently a system collapse. In order to prevent this from happening, a private stack was constructed. Its size is dependent upon the routine calls within the ISR and upon the stack required for passing parameters to or from called routines.

9.4 Screen Operations

It is not advisable to use DOS screen operations from within the ISR.This may cause the system to collapse, due to DOS not being reentrant. Within the ISR, one can use a software flag to indicate that data for screen update is available. In the main program outside the ISR, DOS services or direct BIOS calls can be made.

9.5 Nested interrupts

The first check made when interrupt processing starts is whether this is a nested interrupt. A memory variable "counter" is incremented every time execution of int__hnd starts and decremented just before exiting. If "counter" is greater than 1, then this is not the first entry, it is a nested interrupt. In the code example, the processing continues at the label "inter." This demonstrates that, since we are using the local environment, execution has to continue using the current stack pointer and it should not be set to initial value. In this case, local environment restoration is skipped.

Another method to prevent interrupt nesting is by not issuing the "STI" command in the "int__hnd" routine. This blocks the CPU interrupt input and prevents external interrupt sources from preempting the "int__hnd" execution. The drawback of this approach is that it can significantly enlarge the interrupt latency of other devices. These devices may not be designed to cope with long interrupt latencies.

APPENDIX A
LIST OF USEFUL DOCUMENTS

Documents used in the development of the *NetWare* driver.

1) *Advanced NetWare V2.1 Internetwork Packet Exchange Protocol (IPX) with Asynchronous Event Scheduler (AES)* Revision 1.00. Copyright Novell, Inc.

2) *NetWare V2.1 Driver Specification for Network Interface Cards* Copyright Novell, Inc.

3) *Advanced NetWare Theory of Operations* Version 2.1 Copyright Novell, Inc.

Other Useful Documents

4) *Internet Transport Protocols* (Xerox Corporation; Xerox System Integration Standard; Stamford, Connecticut; December,1981; XSIS-028112)

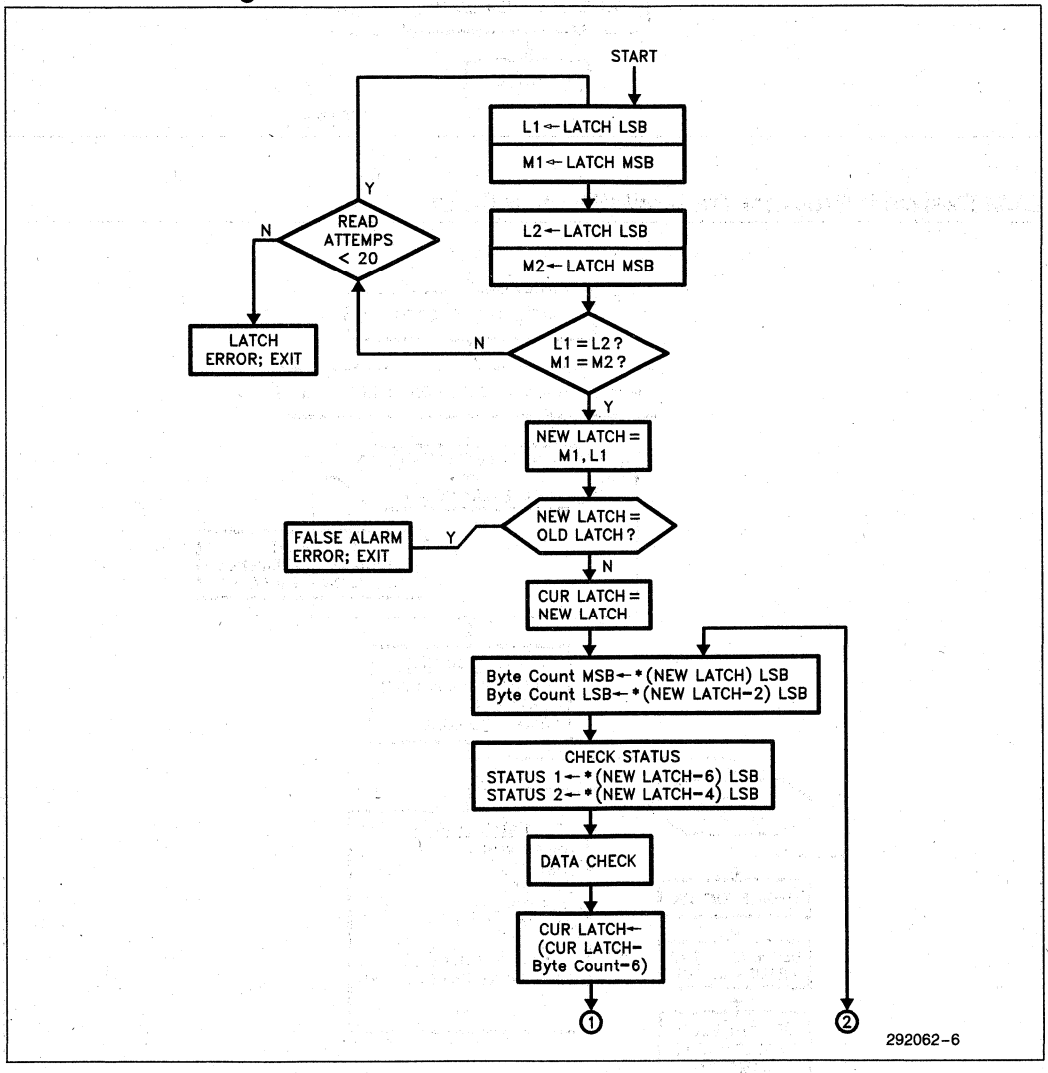
5) *Local Area Network (LAN) Component User's Manual* 1988 Edition Copyright Intel Corporation

6) *AP-320 Using the Intel 82592 to Integrate a Low Cost Ethernet Solution into a PC Motherboard* Copyright Intel Corporation, 1988

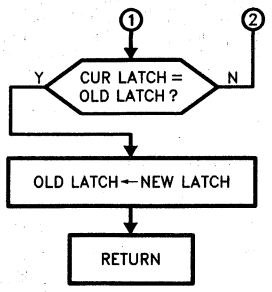
7) *82590-82592 Advanced LAN Controller A-1 Step Errata version 1.2*, December, 1988

APPENDIX B
ELM EXERCISER FLOWCHARTS

ELM Exerciser Program RCV ISC Flow Chart

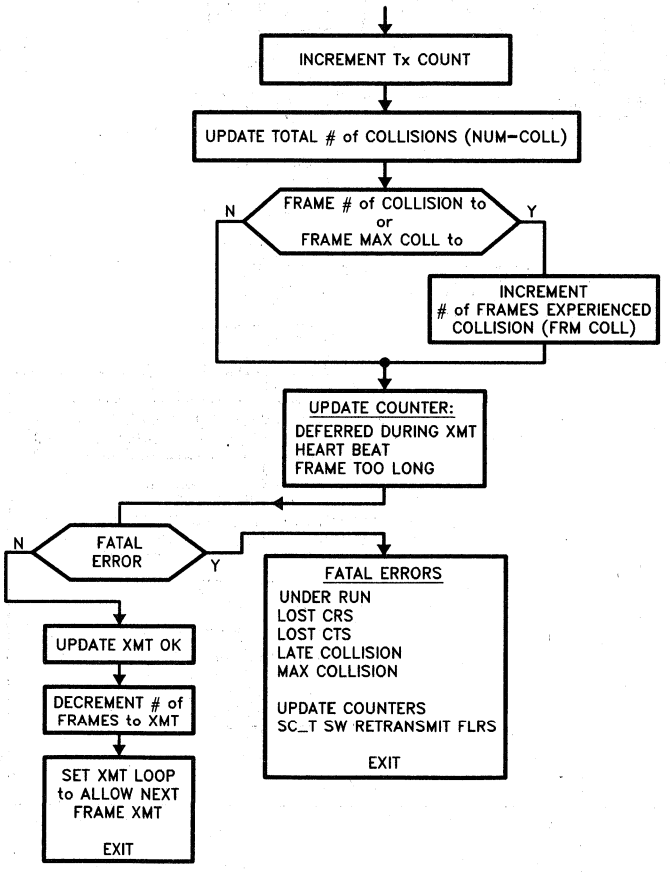


ELM Exciser Program RCV ISC Flow Chart (Continued)



292062-7

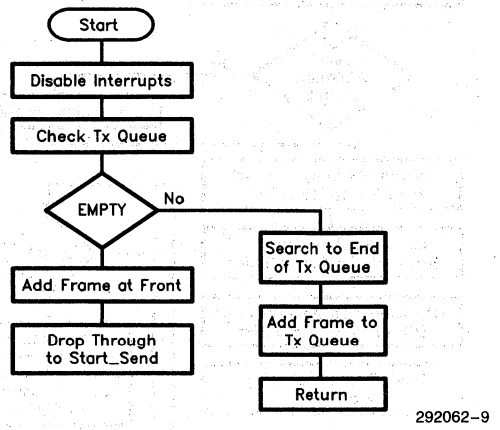
ELM Exciser Program Transmit ISR Flow Chart



292062-8

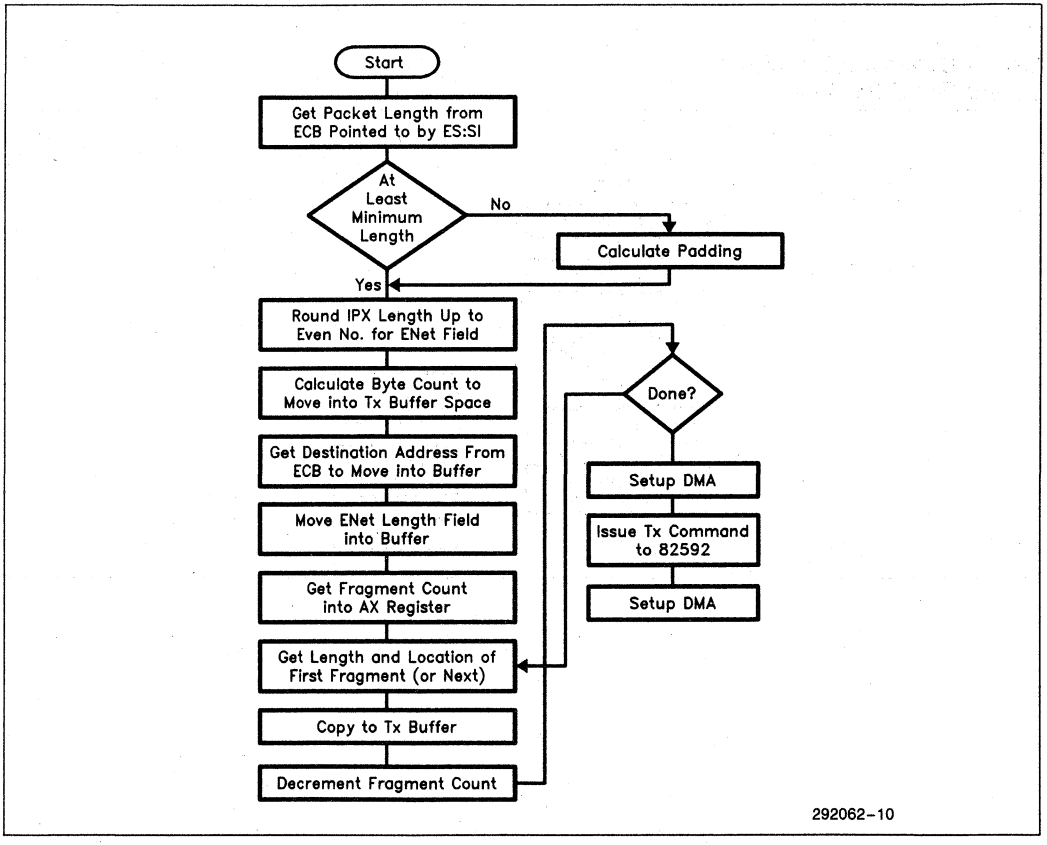
NetWare Driver Flowcharts

Driver Broadcast Packet
Driver Send Packet

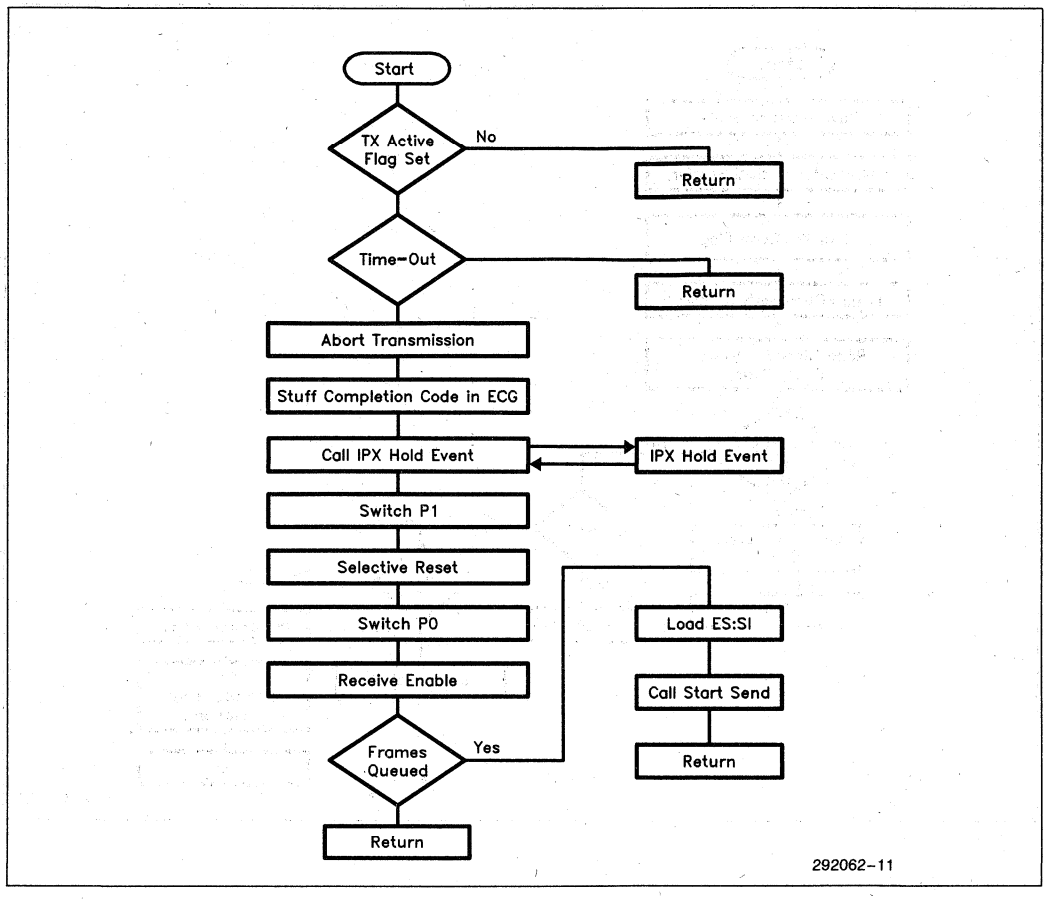


292062-9

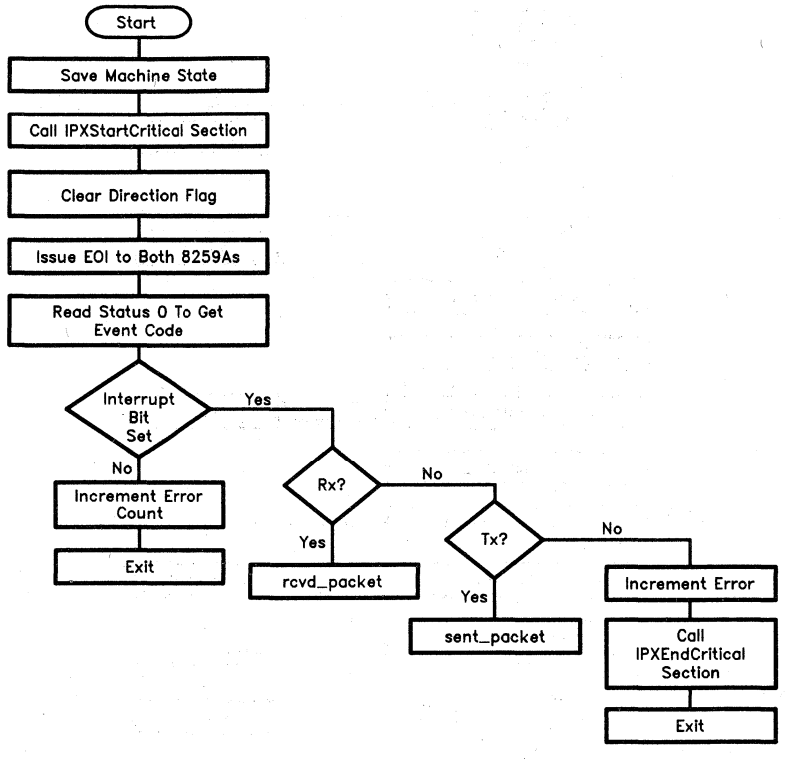
Start Send


Start
Get Packet Length from ECB Pointed to by ES:SI
At Least Minimum Length
No
Calculate Padding
Yes
Round IPX Length Up to Even No. for ENet Field
Calculate Byte Count to Move into Tx Buffer Space
Get Destination Address From ECB to Move into Buffer
Move ENet Length Field into Buffer
Get Fragment Count into AX Register
Get Length and Location of First Fragment (or Next)
Copy to Tx Buffer
Decrement Fragment Count
Done?
Setup DMA
Issue Tx Command to 82592
Setup DMA
292062-10

Driver Poll

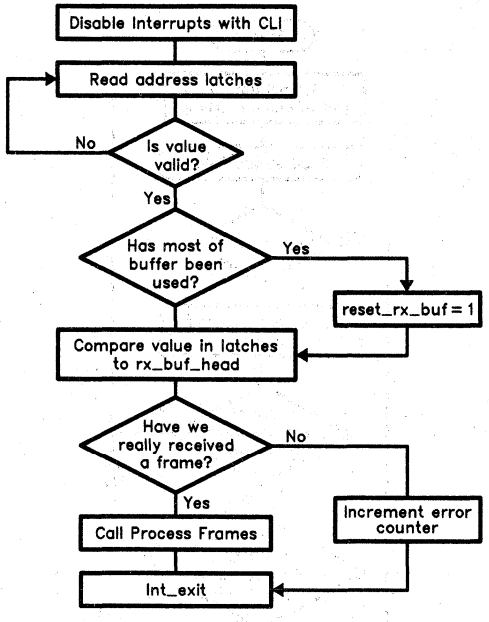

Start
TX Active Flag Set
No
Return
Time-Out
Return
Abort Transmission
Stuff Completion Code in ECG
Call IPX Hold Event
IPX Hold Event
Switch P1
Selective Reset
Switch P0
Receive Enable
Frames Queued
Yes
Load ES:SI
Call Start Send
Return
Return
292062-11

Driver ISR

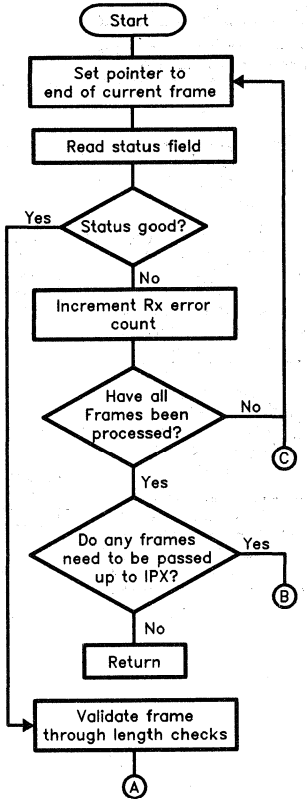


292062-12

rcvd__packet

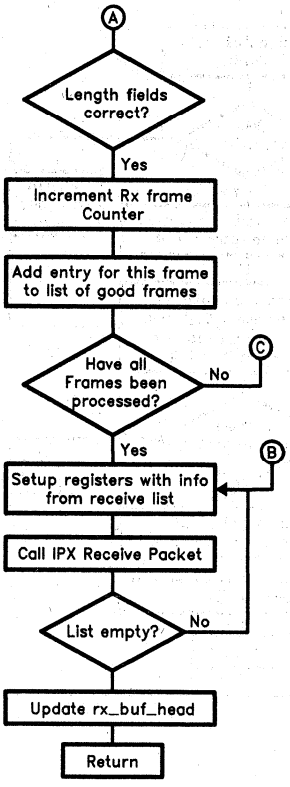

Disable Interrupts with CLI
Read address latches
No
Is value valid?
Yes
Has most of buffer been used?
Yes
reset_rx_buf = 1
Compare value in latches to rx_buf_head
Have we really received a frame?
No
Yes
Increment error counter
Call Process Frames
Int_exit
292062-13

Process Frames



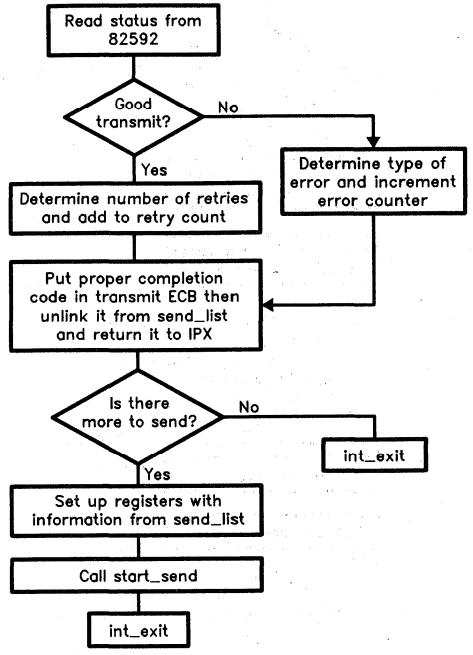
292062-14

Process Frames (Continued)



292062-15

Sent_Packet



292062-16

APPENDIX C
NETWARE DRIVER SOURCE CODE LISTING

NetWare Driver Source Code Listing

```
$mod186

;************************************************************************
;
;     !!!!!! FOR EVALUATION PURPOSES ONLY !!!!!!
;
;     NetWare(R) Driver for the LAN-On-Motherboard Module
;
;     This shell driver is written for use in SYP301 systems.
;
;     Joe Dragony  DFG Technical Marketing
;
;     REVISION 3.11
;
;     Last revision: Date 04-12-89   Time 16:30
;
;
;
;************************************************************************

%*define(slow) local label (
    jmp    short %label
%label:
)

%*define(wait) local label (
    mov  cx, 03Fh
%label:
    nop
    loop  %label
)

%*define(fastcopy) local label (
    shr cx, 1
    rep movsw
    jnc %label
    movsb
%label:
)

%*define(inc32 m) (
    add word ptr %m[0], 1
    adc word ptr %m[2], 0
)
```

292062-17

NetWare Driver Source Code Listing (Continued)

```
    name    LANOnMotherboardModule

CGroup      group    Code, mombo_init

    assume    cs: CGroup, ds: CGroup

Code    segment word public 'CODE'

    public    DriverSendPacket
    public    DriverBroadcastPacket
    public    DriverOpenSocket
    public    DriverCloseSocket
    public    DriverPoll
    public    DriverCancelRequest
    public    DriverDisconnect
    public    SDriverConfiguration

    public    LANOptionName

    extrn     IPXGetECB: NEAR
    extrn     IPXReturnECB: NEAR
    extrn     IPXReceivePacket: NEAR
    extrn     IPXReceivePacketEnabled: NEAR
    extrn     IPXHoldEvent: NEAR
    extrn     IPXServiceEvents: NEAR
    extrn     IPXIntervalMarker: word
    extrn     MaxPhysPacketSize: word
    extrn     ReadWriteCycles: byte
    extrn     IPXStartCriticalSection: NEAR
    extrn     IPXEndCriticalSection: NEAR


;;;;;;;;;;;;;;;;;;;;
;    Equates
;;;;;;;;;;;;;;;;;;;;

TRUE            equ  1
FALSE               equ  0
CR              equ  0Dh
LF              equ  0Ah
BAD             equ  0FFh
BPORT               equ  0
IRQLOC              equ  19
DMA0LOC             equ  23
DMA6LOC             equ  25
TransmitHardwareFailure      equ  0FFh
PacketUnDeliverable    equ  0FEh
PacketOverflow         equ  0FDh
ECBProcessing          equ  0FAh
TxTimeOutTicks         equ  20
```

292062-18

NetWare Driver Source Code Listing (Continued)

```
;  Latch definitions
TenCentLo    equ  301h
TenCentHi    equ  302h

;  Enables for 10cent
EnLAN     equ  303h
DisLAN    equ  304h

;  8259  definitions

InterruptControlPort        equ    020h
InterruptMaskPort           equ    0A1h  ;for secondary 8259A
ExtraInterruptControlPort   equ    0A0h
EOI                         equ    020h

;  8237 definitions

DMAcmdstat     equ  0D0h
DMAreq         equ  0D2h
DMAsnglmsk     equ  0D4h
DMAmode        equ  0D6h
DMAff          equ  0D8h
DMAtmpclr      equ  0DAh
DMAclrmsk      equ  0DCh
DMAallmsk      equ  0DEh
DMA6page       equ  089h
DMA6addr       equ  0C8h
DMA6wdcount    equ  0CAh
DMA7page       equ  08Ah
DMA7addr       equ  0CCh
DMA7wdcount    equ  0CEh
DMAtx6         equ  01Ah     ;demand mode, autoinit, read transfer
DMAtx7         equ  01Bh     ;demand mode, autoinit, read transfer
DMArx6         equ  006h     ;demand mode, no autoinit, write transfer
DMArx7         equ  007h     ;demand mode, no autoinit, write transfer
DMA6msk        equ  006h
DMA6unmsk      equ  002h
DMA7msk        equ  007h
DMA7unmsk      equ  003h
DMAena        equ  010h
```

292062-19

NetWare Driver Source Code Listing (Continued)

```
; 82592 Commands

C_NOP      equ  00h
C_SWP1     equ  10h
C_SELRST   equ  0Fh
C_SWP0     equ  01h
C_IASET    equ  01h
C_CONFIG   equ  02h
C_MCSET    equ  03h
C_TX       equ  04h
C_TDR      equ  05h
C_DUMP     equ  16h
C_DIAG     equ  07h
C_RXENB    equ  18h
C_ALTBUF   equ  09h
C_RXDISB   equ  1Ah
C_STPRX    equ  1Bh
C_RETX     equ  0Ch
C_ABORT    equ  0Dh
C_RST      equ  0Eh
C_RLSPTR   equ  0Fh
C_FIXPTR   equ  1Fh
C_INTACK   equ  80h



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Data Structures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
even
hardware_structure    struc
    io_addr1         dw     ?
    io_range1        dw     ?
    io_addr2         dw     ?
    decode_range2    dw     ?
    mem_addr1        dw     ?
    mem_range1       dw     ?
    mem_addr2        dw     ?
    mem_range2       dw     ?
    int_used1        db     ?
    int_line1        db     ?
    int_used2        db     ?
    int_line2        db     ?
    dma_used1        db     ?
    dma_chan1        db     ?
    dma_used2        db     ?
    dma_chan2        db     ?
hardware_structure    ends
```

292062-20

NetWare Driver Source Code Listing (Continued)

```
ecb_structure     struc
    link                        dd   0
    esr_address                 dd   0
    in_use                      db   0
    completion_code             db   0
    socket_number               dw   0
    ipx_workspace               db   4   dup (0)
    transmitting               db   0
    driver_workspace            db   11  dup (0)
    immediate_address           db   6   dup (0)
    fragment_count              dw   1
    fragment_descriptor_list    db   6   dup (?)
ecb_structure     ends

fragment_descriptor    struc
    fragment_address  dd   ?
    fragment_length   dw   ?
fragment_descriptor    ends

rx_buf_structure    struc
    rx_dest_addr         db   6 dup (?)
    rx_source_addr       db   6 dup (?)
    rx_physical_length   dw   ?
    rx_checksum          dw   ?
    rx_length            dw   ?
    rx_tran_control      db   ?
    rx_hdr_type          db   ?
    rx_dest_net          db   4 dup (?)
    rx_dest_node         db   6 dup (?)
    rx_dest_socket       dw   ?
    rx_source_net        db   4 dup (?)
    rx_source_node       db   6 dup (?)
    rx_source_socket     dw   ?
rx_buf_structure    ends

tci_status    struc
    status0    db    ?
    dead1      db    ?
    status1    db    ?
    dead2      db    ?
    bc_lo      db    ?
    dead3      db    ?
    bc_hi      db    ?
tci_status    ends
```

292062-21

NetWare Driver Source Code Listing (Continued)

```
ipx_header_structure    struc
    checksum             dw   ?
    packet_length        dw   ?
    transport_control    db   ?
    packet_type          db   ?
    destination_network  db   4 dup (?)
    destination_node     db   6 dup (?)
    destination_socket   dw   ?
    source_network       db   4 dup (?)
    source_node          db   6 dup (?)
    source_socket        dw   ?
ipx_header_structure    ends


;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    even

tx_start_time   dw  0
adapter_io      dw  ?
config          dw  ?
send_list       dd  0          ;points to list of ECBs to be sent
buffer_segment  dw  ?
rx_ecb          dd  ?
tx_ecb          dd  ?

config_block   db  0Fh,00h,48h,80h,26h,00h,60h,00h,0F2h,00h,00h,40h,0F7h,00h,3Fh,87h,0F0h,0FFh

temp_flag          db    0
int_mask_register  dw    ?
old_irq_vector     dd    ?
int_vector_addr    dw    ?
int_mask           db    ?
int_unmask         db    ?
command_reg        dw    300h        ;82592 port 0 address
read_in_length     dw    ?
config_dma0_loc    db    ?
config_dma1_loc    db    ?
config_irq_loc     db    ?
config_bport       dw    ?
tx_active_flag     db    0
frame_status       db    0
status10           db    0
status11           db    0
status20           db    0
status21           db    0
```

292062-22

NetWare Driver Source Code Listing (Continued)

```
even

gp_buf              dw   5000 dup (0)    ;twice the required size
gp_length           dw   1388h
gp_buf_offset       dw   cgroup:gp_buf
gp_offset_adjust    dw   0
gp_buf_start        dw   0       ;A1-A16 of General Purpose Buffer EA
gp_buf_page         dw   0       ;A17-A23 of General Purpose Buffer EA
tx_byte_cnt         dw   0       ;IPX packet length plus header length
rx_buf_start        dw   0       ;A1-A16 of General Purpose Buffer EA
rx_buf_page         dw   0       ;A17-A23 of General Purpose Buffer EA
rx_buf_head         dw   0       ;current rx head, buffer has been flushed to here
rx_buf_tail         dw   0       ;value read from 10 cent latches
rx_buf_ptr          dw   0       ;used during rx list generation
rx_buf_stop         dw   0       ;point to reset the DMA controller
rx_buf_length       dw   0
rx_buf_segment      dw   0       ;calculated at init for use by IPXReceivePacket
curr_rx_length      dw   0
rx_list             dw   180 dup (0)
num_of_frames       dw   0
reset_rx_buf        dw   0
padding             dw   0

;
;    Define Hardware Configuration
;
ConfigurationID         db    'NetWareDriverLAN WS  '

SDriverConfiguration    LABEL      byte

reserved1               db    4 dup (0)
node_addr               db    6 dup (0)
reserved2               db    0     ;non-zero means is a real driver.
node_addr_type          db    0     ;address is determined at initialization
max_data_size           dw    1024 ;largest read data request will handle
lan_desc_offset         dw    LANOptionName
lan_hardware_id         db    0AAh     ;Bogus Type Code
transport_time          dw    1     ;transport time
reserved_3              db    11 dup (0)
major_version           db    01h ;Bogus version number
minor_version           db    00h
flag_bits               db    0
selected_configuration  db    0     ;board configuration (interrupts, IO addresses, etc.)
number_of_configs       db    01
config_pointers         dw    configuration0
```

292062-23

NetWare Driver Source Code Listing (Continued)

```
LANOptionName       db    'Intel LAN-On-Motherboard Module',0,'$'

configuration0      dw    300h, 16, 0, 0          ;IO ports and ranges
        db      0
        dw      0 , 0
        db      0
        dw      0 , 0         ;memory decode
        db      0FFh, 10, 0, 0    ;interrupt level 10
        db      0FFh, 6, 0FFh, 7    ;DMA channels 6 and 7
        db      0,0
        db      'IRQ 10, IO Addr = 300h, DMA 6 and 7, For Evaluation Only', 0

;*******************************************************
;
;    Error Counters
;
;*******************************************************
    Public DriverDiagnosticTable,DriverDiagnosticText

DriverDiagnosticTable  LABEL  byte

DriverDebugCount          dw    DriverDebugEnd-DriverDiagnosticTable
DriverVersion             db    01,00
StatisticsVersion         db    01,00
TotalTxPacketCount        dw    0,0
TotalRxPacketCount        dw    0,0
NoECBAvailableCount       dw    0
PacketTxTooBigCount       dw    -1        ;not used
PacketTxTooSmallCount     dw    -1        ;not used
PacketRxOverflowCount     dw    0
PacketRxTooBigCount       dw    0
PacketRxTooSmallCount     dw    0
PacketTxMiscErrorCount    dw    -1        ;not used
PacketRxMiscErrorCount    dw    -1        ;not used
RetryTxCount              dw    0
ChecksumErrorCount        dw    -1        ;not used
HardwareRxMismatchCount   dw    0
NumberOfCustomVariables   dw    (DriverDiagnosticText-DriverDebugEnd1)/2

DriverDebugEnd1  LABEL    byte
```

292062-24

NetWare Driver Source Code Listing (Continued)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Driver Specific Error counts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


rx_errors              dw  0
underruns              dw  0
no_cts                 dw  0
no_crs                 dw  0
rx_aborts              dw  0
no_590_int             dw  0
false_590_int          dw  0
lost_rx                dw  0
stop_tx                dw  0
ten_cent_latch_crash   dw  0
rx_disb_failure        dw  0
tx_abort_failure       dw  0
rx_buff_ovflw          dw  0
tx_timeout             dw  0

DriverDiagnosticText        LABEL    byte


    db      'RxErrorCount',0
    db      'UnderrunCount',0
    db      'LostCTSCount',0
    db      'LostCRSCount',0
    db      'RxAbortCount',0
    db      'No590InterruptCount',0
    db      'False590InterruptCount',0
    db      'LostOurReceiverCount',0
    db      'QuitTransmittingCount',0
    db      'TencentLatchCrashCount',0
    db      'RxDisableFailureCount',0
    db      'TxWontAbort',0
    db      'ReceiveBufferOverflow',0
    db      'TxTimeoutErrorCount',0

    db      0,0


DriverDebugEnd    LABEL  word
```

292062-25

NetWare Driver Source Code Listing (Continued)

```
;*****************************************************************
;
;    Interrupt Procedure
;
;*****************************************************************

even

RxErrorTypeCheck:


BufferOverflow:
    inc   rx_buff_ovflw
    jmp   int_exit


not_590_int:
    inc  no_590_int
    jmp  int_exit

DriverISR        PROC    far
public      DriverISR

    call IPXStartCriticalSection    ;tell AES we're busy
    pusha
    push ds                         ;save machine state
    push es
    cld
    in   al, InterruptMaskPort      ;read current interrupt mask
    or   al, int_mask               ;mask our channel
%slow
    out  InterruptMaskPort, al            ;write it to the 8259A
    mov  al, EOI
    out  InterruptControlPort, al     ;issue EOI's to the 8259A's
    out  ExtraInterruptControlPort, al
    sti                 ;enable interrupts to be friendly
    mov  ax, cs
    mov  ds, ax         ;DS points to C/DGroup
    mov  es, ax         ;ES also
    mov  dx, command_reg
    mov  al, 0
    out  dx, al         ;set status reg to point to reg 0
%slow
    in   al, dx         ;read status from 82592
    test al, 80h        ;check if the INT bit is set
    jz   not_590_int
```

292062-26

NetWare Driver Source Code Listing (Continued)

```
  int_poll_loop:
    and  al, NOT 20h   ;ignore the EXEC bit
    mov  ah, al        ;save the status in AH
    cmp  ah, 0D8h      ;did I receive a frame?
    jz   rcvd_packet
    cmp  ah, 84h        ;did I finish a transmit?
    jz   sent_packet_jmp
    cmp  ah, 8Ch        ;did I finish a retransmit?
    jz   sent_packet_jmp
    inc  false_590_int  ;unwanted interrupt
    jmp  int_exit

sent_packet_jmp:
    jmp  sent_packet
bad_rcv:
    inc  rx_errors
    jmp  RxErrorTypeCheck

int_exit_jmp:
    jmp  int_exit

;When the address bytes are being read it is possible that another frame
;could come in and cause a coherency problem with the ten-cent latches.
;I am dealing with this possibility by reading TenCentHi twice and making
;sure the values match. If they don't the read is redone.

rcvd_packet:
    cli
    mov  dx, TenCentHi ;read high address byte of last frame received
    in   al, dx
    mov  ah, al        ;save it in ah
    mov  dx, TenCentLo ;read low address byte of last frame received
    in   al, dx
    mov  rx_buf_tail, ax   ;this is the last location containing rx data
;Read TenCentHi again to make sure it hasn't changed.......
    mov  dx, TenCentHi       ;read high address byte again
    in   al, dx
    cmp  al, ah            ;do values match?
    jz   addr_ok           ;if so, proceed
    jmp  rcvd_packet       ;else, read the latches again
addr_ok:
    mov  ax, rx_buf_tail    ;this is a valid address
    mov  rx_buf_ptr, ax     ;this is the last location containing rx data
    cmp  rx_buf_stop, ax    ;is most of the buffer already used?
    ja   BufferOK          ;if not, proceed
    mov  reset_rx_buf, 1    ;else, set flag for exit routine
BufferOK:
    cmp  ax, rx_buf_head    ;have we really received a frame?
    ja   process_new_frames   ;if so, process it
    inc  ten_cent_latch_crash ;else, increment error count and exit
    jmp  int_exit
```

292062-27

NetWare Driver Source Code Listing (Continued)

```
process_new_frames:
    call ProcessFrames

int_exit:
    push cs
    pop  ds
    cmp  tx_active_flag, 0
    jnz  finish_exit

;    verify that our receiver is still going.

    mov  dx, command_reg
    mov  al, 60h        ;point to status byte 3
    out  dx, al
%slow
    in   al, dx         ;read status byte 3
    test al, 60h        ;check to see if receiver is enabled
    jnz  finish_exit    ;if so, proceed
    jmp  LostOurReceiver    ;else, take corrective action

int_pending:
    jmp  int_poll_loop


finish_exit:
    cli
    mov  dx, command_reg
    mov  al, C_INTACK
    out  dx, al          ;issue interrupt acknowledge to the 590
%slow
    xor  al, al         ;clear al
    out  dx, al         ;set status reg to point to reg 0
%slow
    in   al, dx         ;read status 0
    test al, 80h        ;is INT bit set?
    jnz  int_pending    ;if so, service pending interrupt
    cmp  reset_rx_buf, 1    ;do we need to reinitialize receive DMA channel?
    jnz  no_rx_buf_reset
    mov  al, dma7msk        ;mask receive DMA channel
    out  DMAsnglmsk, al
    mov  dx, TenCentHi      ;read high address byte of last frame received
    in   al, dx
    mov  ah, al             ;save it in ah
    mov  dx, TenCentLo      ;read low address byte of last frame received
    in   al, dx
    cmp  ax, rx_buf_head    ;do we have a new frame?
    jna  no_new_frames
    mov  rx_buf_tail, ax    ;this is the last location containing rx data
    mov  rx_buf_ptr, ax     ;set pointer for use during buffer processing
    call ProcessFrames
```

292062-28

NetWare Driver Source Code Listing (Continued)

```
no_new_frames:
    mov  dx, command_reg
    mov  al, C_RXDISB      ;issue rx disable to kill any active requests
    out  dx, al
    mov  al, C_SWP1
    out  dx, al
    mov  al, C_SELRST
    out  dx, al
%slow
    mov  al, C_SWP0
    out  dx, al
    out  DMAff, al         ;data is don't care
    mov  ax, rx_buf_start   ;set dma up to point to the beginning of rx buf
    mov  rx_buf_head, ax
    shl  rx_buf_head, 1
    out  DMA7addr, al
    mov  al, ah
%slow
    out  DMA7addr, al
    mov  al, DMArx7
%slow                      ;set proper mode for receive
    out  DMAmode, al
    mov  ax, rx_buf_length  ;set up  rx buf
%slow
    out  DMA7wdcount, al
    mov  al, ah
%slow
    out  DMA7wdcount, al
    mov  dx, DMAsnglmsk
    mov  al, DMA7unmsk
%slow
    out  dx, al
    mov  dx, command_reg
    mov  al, C_RXENB       ;make sure receiver is enabled
    out  dx, al
    mov  reset_rx_buf, 0    ;clear the flag

no_rx_buf_reset:
    cli
    call IPXEndCriticalSection
    in   al, InterruptMaskPort
    and  al, int_unmask
%slow
    out  InterruptMaskPort, al
    pop  es
    pop  ds
    popa
    sti
    iret
```

292062-29

NetWare Driver Source Code Listing (Continued)

```
LostOurReceiver:
    inc  lost_rx
    mov  al, C_RXENB
    mov  dx, command_reg
    out  dx, al
    jmp  finish_exit

too_big:
    inc  PacketRxOverflowCount
    jmp  int_exit

sent_packet:
    cli
    cmp  tx_active_flag, 0
    jz   false_tx_int     ;shouldn't have been transmitting
    in   al, dx
    mov  status10, al
%slow
    in   al, dx
    mov  status11, al
    test status11, 20h
    jz   tx_error
    mov  al, status10       ;extract the total number of retries from
    and  ax, 0Fh           ;the status register and add to retry count
    add  RetryTxCount, ax
    xor  ax, ax           ;status = 0, good transmit

FinishUpTransmit:
    les  si, send_list
    cmp  es: [si].transmitting, TRUE    ;if the transmitting flag is not set
    jnz  ecb_cancelled                  ;then an ECB has been cancelled and
    mov  es: [si].completion_code, al   ;this is a fresh one
    mov  ax, es: word ptr [si].link
    mov  word ptr send_list, ax
    mov  ax, es: word ptr [si].link + 2
    mov  word ptr send_list + 2, ax
    mov  es: [si].in_use, 0         ;finish the transmit

    call IPXHoldEvent
ecb_cancelled:
    push cs
    pop  ds
    mov  cx, word ptr send_list + 2
    mov  tx_active_flag, cl
    jcxz int_exit_jmp1
    mov  es, cx                   ;segment of next SCB in list
    mov  si, word ptr send_list   ;offset of next SCB in list
    call start_send
    jmp  finish_exit

int_exit_jmp1:
    jmp  int_exit
```

292062-30

NetWare Driver Source Code Listing (Continued)

```
false_tx_int:
    jmp  int_exit

tx_error:
    test status10, 20h          ;Max collisions??
    jnz  QuitTransmitting
    test status11, 01h          ;Tx underrun??
    jz   lost_cts
    inc  underruns
lost_cts:
    test status11, 02h          ;did we lose clear to send??
    jz   lost_crs
    inc  no_cts
lost_crs:
    test status11, 04h          ;did we lose carrier sense??
    jz   hmmm
    inc  no_crs
hmmm:
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit

QuitTransmitting:
    mov  al, status10
    and  ax, 0Fh
    add  RetryTxCount, ax
    inc  stop_tx
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit

DriverISR    endp




;
;    ProcessFrames:  a routine to process received frames and hand them
;                    off to IPX
;
;    Assumes: rx_buf_tail and rx_buf_ptr have been set up with the value
;             read from the ten cent latches.
;
;    Returns: nothing
;
```

292062-31

NetWare Driver Source Code Listing (Continued)

```
ProcessFrames  proc  near

do_next_frame:
    sti
    mov  bx, rx_buf_ptr      ;end of current frame to process
    sub  bx, 6               ;set bx up to point to beginning of the status
    mov  es, rx_buf_segment  ;this is necessary because latches hold EA not
                             ;offset relative to CGROUP
    mov  al, es:[bx].status1
    test al,20h              ;test for good receive
    jnz  good_rx
    inc  rx_errors
    mov  cl, es:[bx].bc_lo
    mov  ch, es:[bx].bc_hi   ;cx has actual number of bytes read
    dec  cx                  ; toss byte count & status
    and  cl, 0feh            ; round up
    sub  bx, cx              ;bx points to first location of frame
    cmp  rx_buf_head, bx
    je   hand_off_packet_jmp ;this was the first frame in the sequence
    cmp  rx_buf_head, bx
    ja   check_rx_queue      ;this frame is a fragment in the beginning of
    mov  rx_buf_ptr, bx      ;the receive buffer
    sub  rx_buf_ptr, 2
to_do_next_frame:
    jmp  do_next_frame
hand_off_packet_jmp:
    jmp  hand_off_packet

check_rx_queue:
    cmp  num_of_frames, 0     ;have any frames been processed?
    jne  hand_off_packet_jmp  ;if yes, give them to IPX
    jmp  process_exit         ;if not, go back to ISR

good_rx:
    mov  cl, es:[bx].bc_lo
    mov  ch, es:[bx].bc_hi  ;cx has actual number of bytes read
    mov  curr_rx_length, cx
    dec  cx                  ;toss byte count & status
    and  cl, 0feh            ;round up
    sub  bx, cx             ;bx points to first location of frame
    cmp  rx_buf_head, bx
    ja   check_rx_queue
    mov  rx_buf_ptr, bx
    sub  rx_buf_ptr, 2      ;rx_buf_ptr = last location of n-1 frame
    sub  cx, 14              ;sub length of 803.2 header
    cmp  cx, 1024 + 64
    jbe  not_too_big
    inc  PacketRxTooBigCount
    jmp  do_next_frame
not_too_big:
    cmp  cx, 30
    jae  not_too_small
    inc  PacketRxTooSmallCount
    jmp  do_next_frame
```

292062-32

NetWare Driver Source Code Listing (Continued)

```
not_too_small:
    mov  ax, es:[bx].rx_length    ; get IPX length
    xchg al, ah
    inc  ax
    and  al, 0feh
    xchg al, ah
    cmp  ax, es:[bx].rx_physical_length     ; same as 802.3 length ?
    jne  to_do_next_frame
    xchg al, ah
    cmp  ax, 60 - 14          ;at least min length minus header
    ja   len_ok               ;yes, continue
    mov  ax, 60 - 14          ;no, round up
len_ok:
    cmp  ax, cx               ;match physical length
    jz   not_inconsistent     ;yes, continue
    inc  HardwareRxMismatchCount
    jmp  do_next_frame
not_inconsistent:
    %inc32  TotalRxPacketCount      ; Double Word Increment
    mov  ax, 12
    mul  num_of_frames
    mov  di, ax
    mov  rx_list [di], bx     ;first location of ethernet frame
    add  rx_list [di], 14     ;first location of ipx packet
    mov  ax, rx_buf_segment
    mov  rx_list [di + 2], ax
    mov  ax, word ptr es:[bx].rx_length
    xchg al,ah
    mov  rx_list [di + 4], ax
    mov  ax, word ptr es:[bx].rx_source_addr + 0
    mov  word ptr rx_list [di + 6], ax
    mov  ax, word ptr es:[bx].rx_source_addr + 2
    mov  word ptr rx_list [di + 8], ax
    mov  ax, word ptr es:[bx].rx_source_addr + 4
    mov  word ptr rx_list [di + 10], ax
    add  num_of_frames, 1
    cmp  num_of_frames, 50    ;prevent list overflow
    je   hand_off_packet
    cmp  rx_buf_head, bx
    je   hand_off_packet
    jmp  do_next_frame
```

292062-33

NetWare Driver Source Code Listing (Continued)

```
hand_off_packet:
    mov  si, rx_list[di]      ;offset in receive buffer space
    mov  es, rx_list[di + 2]  ;receive buffer bogus segment
    mov  cx, rx_list[di + 4]  ;IPX packet length
    lea  bx, rx_list[di + 6]  ;pointer to immediate address
    cli
    push ds
    call IPXReceivePacketEnabled  ;since packet is contiguous let IPX do
    pop  ds                       ;the work
    sub  num_of_frames, 1         ;decrement count
    jz   adjust_rx_head           ;if all frames are processed adjust head
    sub  di, 12                        ;otherwise index to next list entry
    jmp  hand_off_packet      ;and loop to process next frame
adjust_rx_head:
    mov  ax, rx_buf_tail      ;location of last location used in receive
    add  ax, 2                ;index to next word location
    mov  rx_buf_head, ax      ;set rx_buf_head to new value for next receive
process_exit:
    ret                       ;interrupt

ProcessFrames  endp




;
;   Driver Send Packet
;   Driver Broadcast Packet
;
;   Assumes
;     ES:SI points to a fully prepared Event Control Block
;     DS = CS
;     Interrupts are DISABLED but may be reenabled temporarily if necessary
;
;     don't need to save any registers
;
DriverBroadcastPacket:
DriverSendPacket    PROC    NEAR
    cli                           ; disable the interrupts
    mov    es: [si].transmitting, FALSE ;make sure the flag is initially clear
    mov  cx, word ptr send_list + 2   ;it will be used later to prevent a
    jcxz AddToFrontOfList     ;cancelled ECB from being given to IPX twice
;search to the end of the list, and add there.
    mov  di, word ptr send_list

AddToListLoop:
    mov  ds, cx
    mov  cx, ds: word ptr [di].link + 2
    jcxz AddListEndFound
    mov  di, ds: word ptr [di].link
    jmp  AddToListLoop
```

292062-34

NetWare Driver Source Code Listing (Continued)

```
AddListEndFound:
    mov  es: word ptr [si].link, cx        ;move null pointer to newest SCB's
    mov  es: word ptr [si].link + 2, cx    ;link field
    mov  ds: word ptr [di].link, si
    mov  ds: word ptr [di].link + 2, es
    mov  ax, cs
    mov  ds, ax    ;set ds back to entry condition
    ret

AddToFrontOfList:
    mov  es:word ptr[si].link, cx
    mov  es:word ptr[si].link + 2, cx
    mov  word ptr send_list, si
    mov  word ptr send_list + 2, es
;drop through to Start Send

DriverSendPacket    endp

;
;    Start Send
;
;    assumes:
;     ES: SI    points to the ECB to be sent.
;     interrupts are disabled
;

start_send    PROC    NEAR
public      start_send
    cli            ; disable the interrupts
    cld
    mov    es: [si].transmitting, TRUE
;save SCB address in variable tx_ecb to liberate registers
    mov  word ptr tx_ecb, si
    mov  word ptr tx_ecb + 2, es
    push ds        ;save ds for future use
;get IPX packet length out of the first fragment (IPX header)
    lds  bx, es: dword ptr [si].fragment_descriptor_list
    mov  ax, ds: [bx].packet_length
    pop  ds           ;restore ds to CGROUP
    push ax           ;save length for later use in 590 length field
    xchg al, ah       ;byte swap for 592 length field calculation
    add  ax, 18       ;add in the overhead bytes DA,SA,CRC,length

    mov  padding, 0
    cmp  ax, 64
    ja   long_enough
    mov  padding, 64      ;minimum length frame
    sub  padding, ax      ;pad length
    mov  ax, 64
```

292062-35

NetWare Driver Source Code Listing (Continued)

```
long_enough:
    sub  ax, 10       ;SA and CRC are done automatically
    inc  ax
    and  al, 0FEh      ;frame must be even
    mov  tx_byte_cnt, ax
    mov  di,gp_buf_offset
    mov  bx, cs
    mov  es, bx
;move the byte count into the transmit buffer
    stosw
;move the destination address from the tx ECB to the tx buffer
    mov  bx, si
    lea  si, [bx].immediate_address
    mov  ds,word ptr tx_ecb + 2
    movsw
    movsw
    movsw
    mov  ax,cs        ; get back to the code (Dgroup) section
    mov  ds,ax

;now the 590 length field
    pop  ax
    xchg ah, al
    inc  ax
    and  al, 0FEh        ;make sure E-Net length field is even
    xchg ah, al
    stosw
    lds  si, tx_ecb
    mov  ax, ds: [si].fragment_count
    lea  bx, [si].fragment_descriptor_list
move_frag_loop:
    push ds              ; save the segment
    mov  cx, ds: [bx].fragment_length
    lds  si, ds: [bx].fragment_address
    %fastcopy
    pop  ds              ; get the segment back
    add  bx, 6
    dec  ax
    jnz  move_frag_loop
;start transmitting
    mov  cx, cs
    mov  ds, cx
;add any required padding
    mov  cx, 4          ;make sure frame ends with a NOP
    add  cx, padding
    shr  cx, 1
    rep  stosw
    mov  tx_active_flag, 1
    xor  ax, ax
    out  DMAff, al          ;data is don't care, AX has been zeroed
    mov  ax, gp_buf_start
%slow
    out  DMA6addr, al
```

292062-36

NetWare Driver Source Code Listing (Continued)

```
    mov  al, ah
%slow
    out  DMA6addr, al
    mov  ax, gp_buf_page
%slow
    out  DMA6page, al       ;DMA page value
%slow
    mov  al, DMAtx6          ;setup channel 1 for tx mode
    out  DMAmode, al
    mov  ax, tx_byte_cnt
    add  ax, 4          ;add two for byte count, two for tx chain fetch
    shr  ax, 1          ;convert to word value and account for odd
    adc  ax, 0          ;byte DMA transfer
    out  DMA6wdcount, al
%slow
    mov  al, ah
    out  DMA6wdcount, al
%slow
    mov  al, DMA6unmsk
    out  DMAsnglmsk, al
    mov  dx, command_reg
    mov  al, C_TX
    out  dx, al
    mov  ax, IPXIntervalMarker        ;get a fix on the time that transmission
    mov  tx_start_time, ax    ;started and save it for later use
    %inc32  TotalTxPacketCount        ;increment counter
    ret

start_send    endp


DriverOpenSocket:
DriverDisconnect:
    ret
```

292062-37

NetWare Driver Source Code Listing (Continued)

```
;***************************************************************************
;
;    Driverpoll
;
;    Poll the driver to see if there is anything to do
;
;    Is there a transmit timeout? If so, abort transmission and return
;    ECB with bad completion code. Check to see if frames are queued.
;    If they are set up ES:SI and call DriverSendPacket.
;
;***************************************************************************

DriverPoll    PROC    NEAR
    cmp  tx_active_flag, 0
    jz   NotWaitingOnTx
    mov  dx, IPXIntervalMarker
    sub  dx, tx_start_time
    cmp  dx, TxTimeOutTicks
    jb   NotTimedOutYet

; This transmit is taking too long so let's terminate it now
;
; Issue an abort to the 82592
    mov  dx, command_reg
    mov  al, C_ABORT         ;abort transmit
    out  dx, al
    inc  tx_timeout
    les  si, tx_ecb
    mov  es: [si].completion_code, PacketUnDeliverable   ;stuff completion code of a failed tx
    mov  ax, es: word ptr [si].link
    mov  word ptr send_list, ax
    mov  ax, es: word ptr [si].link + 2
    mov  word ptr send_list + 2, ax

; Finish the transmit

    mov  es: [si].in_use, 0
    call IPXHoldEvent
```

292062-38

NetWare Driver Source Code Listing (Continued)

```
;make sure that execution unit didn't lock up because of abort errata

    mov  dx, command_reg
    mov  al, C_SWP1
    out  dx, al
%wait
    mov  al, C_SELRST
    out  dx, al
%wait
    mov  al, C_SWP0
    out  dx, al
%wait
    mov  al, C_RXENB
    out  dx, al
    mov  tx_active_flag, 0

;See if any frames are queued

    mov  cx, word ptr send_list + 2
    jcxz queue_empty
    mov  es, cx
    mov  si, word ptr send_list
    call start_send

queue_empty:
NotWaitingOnTx:
NotTimedOutYet:
    ret

DriverPoll    endp
```

292062-39

NetWare Driver Source Code Listing (Continued)

```
;
;   Driver Cancel Request
;
;   Assumes on entry:
;     ES:SI is pointer to ECB we want to cancel
;     DS is setup
;     Interrupts are DISABLED
;
;   Assumes any registers may be destroyed.
;
;   Returns completion code in AL:
;     00    Buffer was located and canceled.
;     FF    Buffer was not found to be in use by the driver
;

DriverCancelRequest     PROC    NEAR

;first, see if it is the one we are currently sending.
    mov     dx, es
    cmp     word ptr send_list, si
    jnz     NotFirstOne
    cmp     word ptr send_list + 2, dx
    jnz     NotFirstOne
;we need to cancel the first entry.  first, unlink it
;from the send list.
    mov     ax, es: word ptr [si].link
    mov     word ptr send_list, ax
    mov     cx, es: word ptr [si].link + 2
    mov     word ptr send_list + 2, cx
    mov     es: [si].completion_code, 0FCh
    mov     es: [si].in_use, 0
    xor     ax, ax
    ret
;we need to search down the send list

NotFirstOne:
    mov     cx, word ptr send_list + 2
    mov     di, word ptr send_list

ScanTheSendListLoop:
    jcxz    NotFound
;move to the next link
    mov     es, cx
    mov     bx, di
    mov     cx, es: word ptr [bx].link + 2
    mov     di, es: word ptr [bx].link
;next node is pointed to by CX:DI
;previous node is pointed to by ES:BX
;see if we found it
    cmp     di, si
    jnz     ScanTheSendListLoop
    cmp     cx, dx
    jnz     ScanTheSendListLoop
```

292062-40

NetWare Driver Source Code Listing (Continued)

```
;we found it. now unlink it.
    push    ds
    mov     ds, cx
    mov     ax, ds: word ptr [si].link
    mov     es: word ptr [bx].link, ax
    mov     ax, ds: word ptr [si].link + 2
    mov     es: word ptr [bx].link + 2, ax
    mov     ds: [si].completion_code, 0FCh
    mov     ds: [si].in_use, 0
    pop     ds
    xor     ax, ax
    ret

NotFound:
    mov     al, 0FFh
    ret

DriverCancelRequest     endp


;
;   Driver Close Socket
;
;   Assumes on entry:
;     DX has socket number
;     DS is setup
;     Interrupts are DISABLED
;
;   Assumes any registers may be destroyed.
;

DriverCloseSocket  PROC NEAR
    mov     cx, word ptr send_list + 2
    jcxz    DriverCloseExit
    les     si, send_list

DriverCloseLoop:
    cmp     es: [si].socket_number, dx
    jnz     DriverToNext
    push    dx
    call    DriverCancelRequest
    pop     dx
    jmp     DriverCloseSocket

DriverToNext:
    mov     cx, es: word ptr [si].link + 2
    jcxz    DriverCloseExit
    les     si, es: [si].link
    jmp     DriverCloseLoop
```

292062-41

NetWare Driver Source Code Listing (Continued)

```
DriverCloseExit:
    ret

DriverCloseSocket  endp

Code    ends



mombo_init    segment 'CODE'

    public    DriverInitialize, DriverUnHook
no_card_message            db  CR,LF,'No adapter installed in PC$'
config_failure_message     db  CR,LF,'Configuration Failure$'
iaset_failure_message      db  CR,LF,'IA Setup Failure$'
ConfigDataUnderrunMess     db  CR,LF,'Configuration underrun$'

;
;   Driver Initialize
;
;   assumes:
;     DS, ES are set to CGroup (== CS)
;     DI points to where to stuff node address
;     Interrupts are ENABLED
;     The Real Time Ticks variable is being set, and the
;     entire AES system is initialized.
;
;   returns:
;     If initialization is done OK:
;          AX has a 0
;     If board malfunction:
;          AX gets offset (in CGroup) of '$'-terminated error string
;

DriverInitialize PROC    NEAR
    mov    MaxPhysPacketSize, 1024
    cli
    cld
    mov  ax, cs
    mov  ds, ax
    mov  es, ax
;get DOS time and use for address.
    mov    ah,02Ch
    int    21h
    mov    bx, OFFSET CGroup: node_addr
    mov    byte ptr cgroup:[bx], 00h
    mov    byte ptr cgroup:[bx+1], 0AAh
    mov    byte ptr cgroup:[bx+2], ch
    mov    byte ptr cgroup:[bx+3], dl
    mov    byte ptr cgroup:[bx+4], dh
    mov    byte ptr cgroup:[bx+5], 7Eh
    mov    si, bx
```

292062-42

NetWare Driver Source Code Listing (Continued)

```
    movsw           ;stuff address at point IPX indicated
    movsw
    movsw
    sti

;initialize the configuration table
    mov    al,selected_configuration
    cbw
    shl    ax,1             ; multiply by two
    add    ax,OFFSET CGROUP:config_pointers    ;ax contains the offset value
    mov    bx,ax                    ;of the default configuration
    mov    bx,[bx]                    ;list
    mov    Config,bx
    mov    al,[bx+DMA0LOC]
    mov    config_dma0_loc,al
    mov    al,[bx+DMA6LOC]
    mov    config_dma1_loc,al
    mov    al,[bx+IRQLOC]
    mov    config_irq_loc,al
    mov    ax,[bx+BPORT]
    mov    command_reg, 300h

SetTheInterruptVector:
;
;    SET UP THE INTERRUPT VECTORS
;
    push   di
    mov    al, config_irq_loc
    mov    bx, OFFSET CGroup: DriverISR
    call   SetInterruptVector
    pop    di
    mov    dx, EnLAN
    out    dx, al          ;enable LAN on MB module
%slow
    mov    dx, command_reg
    mov    al, C_RST
    out    dx, al      ; reset the 82592 controller

;generate 20 bit address for DMA controller from configure block location
;this is necessary to accomodate the page register used in the PC DMA

    call   set_up_buffers

;set up DMA channel for configure command
    xor    ax, ax
    out    DMAff, al           ;data is don't care
%slow
    mov    al, DMAena
    out    DMAcmdstat, al
    mov    ax, gp_buf_start
%slow
    out    DMA6addr, al
    mov    al, ah
```

292062-43

NetWare Driver Source Code Listing (Continued)

```
%slow
    out     DMA6addr, al
    mov     ax, gp_buf_page
%slow
    out     DMA6page, al       ;DMA page value
    mov     ax, 1
%slow
    out     DMA6wdcount, al     ;make two transfers
    mov     al, ah
%slow
    out     DMA6wdcount, al
    mov     al, DMAtx6          ;setup channel 6 for tx mode
%slow
    out     DMAmode, al
    mov     al, DMA6unmsk
%slow
    out     DMAsnglmsk, al
    xor     ax, ax
    mov     di, gp_buf_offset  ;mov zeroes into the byte count field of the
    stosw               ;buffer to put the 82592 into 16 bit mode
    stosw
%slow
    mov     dx, command_reg
    mov     al, C_CONFIG        ;configure the 82592 for 16 bit mode
    out     dx, al            ;issue configure command
%slow

wide_mode_wait_loop:
    xor     al, al
%slow
    out     dx, al      ;point to register 0
%slow
    in        al, dx      ;read register 0
    and     al,0DFh     ;disregard exec bit
    cmp     al, 82h     ; is configure finished?
    jz        do_config
    loop    wide_mode_wait_loop
    mov     ax, OFFSET CGroup: no_card_message
    jmp     init_exit

do_config:
    mov     al, C_INTACK
    out     dx, al           ;clear interrupt
    xor     ax, ax
%slow
    out     DMAff, al           ;data is don't care
    mov     ax, gp_buf_start
%slow
    out     DMA6addr, al
    mov     al, ah
%slow
    out     DMA6addr, al
    mov     ax, gp_buf_page
```

292062-44

NetWare Driver Source Code Listing (Continued)

```
%slow
    out     DMA6page, al        ;DMA page value
%slow
    mov     al, DMAtx6          ;setup channel 1 for tx mode
    out     DMAmode, al
%slow
    mov     ax, 8
    out     DMA6wdcount, al
%slow
    mov     al, ah
    out     DMA6wdcount, al
%slow
    mov     al, DMA6unmsk
    out     DMAsnglmsk, al
    mov     ax, ds
    mov     es, ax
    mov     si, offset cgroup:config_block
    mov     di, gp_buf_offset
    mov     cx, 18
rep movsb
    mov     dx, command_reg
    mov     al, C_CONFIG        ; configure the 82592
    out     dx, al
%slow
    xor     cx, cx

config_wait_loop:
%slow
    xor     al, al
%slow
    out     dx, al      ;point to register 0
%slow
    in        al, dx     ;read register 0
    and     al, 0DFh    ;discard extraneous bits
    cmp     al, 82h     ; is configure finished?
    jz        config_done
    loop    config_wait_loop
    mov     ax, OFFSET CGroup: config_failure_message
    jmp     init_exit

config_done:
;clear interrupt caused by configuration
    mov     al, C_INTACK
    out     dx, al

;do an IA_setup
    mov     di, gp_buf_offset
    mov     al, 06h     ;address byte count
    stosb
    mov     al, 00h
    stosb
    mov     si, OFFSET CGROUP:node_addr
    mov     cx, SIZE node_addr
rep movsb
```

292062-45

NetWare Driver Source Code Listing (Continued)

```
    out     DMAff, al           ;data is don't care
%slow
    mov     ax, gp_buf_start
    out     DMA6addr, al
    mov     al, ah
%slow
    out     DMA6addr, al
    mov     ax, gp_buf_page
%slow
    out     DMA6page, al        ;DMA page value
%slow
    mov     al, DMAtx6          ;setup channel 1 for tx mode
    out     DMAmode, al
%slow
    mov     ax, 3
    out     DMA6wdcount, al
%slow
    mov     al, ah
    out     DMA6wdcount, al
%slow
    mov     al, DMA6unmsk
    out     DMAsnglmsk, al

    mov     dx, command_reg
    mov     al, C_IASET             ;set up the 82592 individual address
    out     dx, al
    xor     cx, cx    ;cx is used by the loop instruction below. this
      ;causes the loop to be executed 64k times max
ia_wait_loop:
    xor     al, al
    out     dx, al
%slow
    in        al, dx
    and     al, 0DFh    ;discard extraneous bits
    cmp     al, 81h     ; is command finished?
    jz        ia_done
    loop    ia_wait_loop
    mov     ax, OFFSET CGroup: iaset_failure_message
    jmp     init_exit

ia_done:
    mov     al, C_INTACK
    out     dx, al          ;clear interrupt from iaset
;initialize the receive DMA channel
    xor     al, al
    out     DMAff, al
    mov     ax, rx_buf_start  ;set dma up to point to the beginning of rx buf
%slow
    out     DMA7addr, al
    mov   al, ah
%slow
    out     DMA7addr, al
    mov     ax, rx_buf_page   ;set rx page register
```

292062-46

NetWare Driver Source Code Listing (Continued)

```
%slow
    out     DMA7page, al
    mov     al, DMArx7
%slow
    out     DMAmode, al
    mov     ax, rx_buf_length    ;set wordcount to proper value
%slow
    out     DMA7wdcount, al
    mov    al, ah
%slow
    out     DMA7wdcount, al
    mov     al, dma7unmsk        ;unmask receive DMA channel
%slow
    out     DMAsnglmsk, al

;unmask our interrupt channel
    in       al, InterruptMaskPort
    and    al, int_unmask
%slow
    out     InterruptMaskPort, al

;enable the receiver
    mov     dx, command_reg     ;enable receives
    mov     al, C_RXENB
    out     dx, al
    xor     ax, ax
    mov     cx, 1

init_exit:
    ret


DriverInitialize    endp
```

292062-47

NetWare Driver Source Code Listing (Continued)

```
;    Set up Buffers:
;    This routine generates the page and offset addresses for the 16 bit
;    DMA. It checks for a page crossing and uses the smaller half of the
;    buffer area for Tx and general purpose if a crossing is detected. If
;    no crossing is detected the general purpose/transmit buffer is placed
;    at the beginning of the buffer area. This routine also generates a
;    segment address for the receive buffer which allows the value read
;    from the "10 cent" latches to be used as read for the offset passed
;    to IPXReceivePacket. This saves some arithmetic steps when tracing
;    back through the rx buffer chain.
;

set_up_buffers      proc      near

    mov  ax, offset cgroup: gp_buf
    mov  gp_buf_offset, ax
    mov  bx, cs
    mov  dx, cs
    shr  ax, 1
    mov  cx, 3
    shl  bx, cl
    rol  dx, cl         ;get upper 3 bits for page register
    and  dx, 0007h      ;clear all but the lowest 3 bits
    add  ax, bx         ;ax contains EA of first location in buffer
    adc  dx, 0          ;if addition caused a carry add it to page
    mov  cx, 0FFFFh      ;of buffer to page break
    sub  cx, ax         ;cx contains the number of bytes to page break
    cmp  cx, 01388h
    jb     intel_hop
    jmp  copacetic      ;it's cool, whole buffer space is in one page
intel_hop:
    cmp  cx, 0258h
    ja   low_ok         ;low fragment is a usable size, check upper fragment
    add  ax, cx         ;move pointer past the page break to discard fragment
    sub  gp_length, cx ;adjust length variable to reflect shorter length
    mov  gp_offset_adjust, cx
    shl  gp_offset_adjust, 1  ;convert to byte format
    mov  cx, gp_offset_adjust
    add  gp_buf_offset, cx    ;adjust gp_buf starting point to reflect change
    jmp  copacetic            ;both buffers will be in the same page, rx buf shortenec

low_ok:
    cmp  cx, 1130h
    jb   high_ok
    mov  gp_length, cx ;adjust length variable, discard upper buffer fragment
    jmp  copacetic     ;both buffers will be in the same page, rx buf shortened

high_ok:               ;now since both fragments are usable we have to find the
    cmp  cx, 09C4h     ;actual page break. the large half will be the receive
    ja   rx_first      ;buffer and the small half will be the gp-tx buffer.
    mov  gp_buf_page, dx
    shl  gp_buf_page, 1
    mov  gp_buf_start, ax
```

292062-48

NetWare Driver Source Code Listing (Continued)

```
    mov  rx_buf_start, 0000h
    mov  rx_buf_head, 0000h
    add  dx, 1                ;next page
    mov  rx_buf_page, dx
    shl  rx_buf_page, 1
    shl  ax, 1
    adc  dx, 0
    mov  bx, cx     ;save number of bytes to page break
    mov  cx, 12
    shl  dx, cl
    mov  rx_buf_segment,dx
    sub  gp_length, bx
    mov  cx, gp_length
    mov  rx_buf_length, cx
    sub  cx, 258h
    shl  cx, 1
    add  cx, ax
    mov  rx_buf_stop, cx
    jmp  buffers_set

rx_first:
    mov  rx_buf_page, dx
    shl  rx_buf_page, 1
    mov  rx_buf_start, ax
    mov  rx_buf_head, ax
    shl  rx_buf_head, 1
    mov  rx_buf_length, cx
    mov  rx_buf_stop, 0FB9Eh  ;1200 bytes from end of buffer
    mov  gp_buf_start, 0000h
    add  dx, 1                ;next page
    mov  gp_buf_page, dx
    shl  gp_buf_page, 1
    add  cx, 1
    shl  cx, 1
    mov  gp_offset_adjust, cx
    add  gp_buf_offset, cx
    sub  dx, 1
    shl  dx, 1
    shl  ax, 1
    adc  dx, 0
    mov  cx, 12
    shl  dx, cl
    mov  rx_buf_segment,dx
    jmp  buffers_set

copacetic:
    mov  gp_buf_start, ax  ;A1-A16 of gp buffer, gp buffer is first
    add  ax, 258h          ;1200 bytes for gp buffer at front of buffer space
    mov  rx_buf_start, ax  ;rx buffer starts 1200 bytes in
    mov  rx_buf_head, ax
    shl  rx_buf_head, 1
    sub  gp_length, 258h
    mov  cx, gp_length
    mov  rx_buf_length, cx
```

292062-49

NetWare Driver Source Code Listing (Continued)

```
    shl  dx, 1              ;convert segment to byte address
    mov  rx_buf_page, dx
    mov  gp_buf_page, dx
    shl  ax, 1              ;convert offset to byte address
    adc  dx, 0              ;adjust segment for shift
    mov  cx, 12
    shl  dx, cl
    mov  rx_buf_segment, dx   ;load variable for transfers to IPX
    mov  cx, rx_buf_length
    sub  cx, 258h          ;setup marker for low rx buffer space, >600 words
    shl  cx, 1
    add  ax, cx
    mov  rx_buf_stop, ax

buffers_set:
    ret

set_up_buffers      endp


;
;    SetInterruptVector
;
;    Set the interrupt vector to the interrupt procedure's address
;    save the old vector for the unhook procedure
;
;    assumes: bx has the ISR offset
;      al has the IRQ level
;      interrupts are disabled
;

SetInterruptVector    PROC    NEAR
;mask on the appropriate interrupt mask
    push   ax
    xchg   ax, cx
    and    cx, 07h
    mov    dl, 1
    shl    dl, cl          ;get the appropriate bit location
    mov    int_mask, dl       ;set the interrupt bit variable
    not    dl
    mov    int_unmask, dl    ;set the interrupt mask variable
    mov    ax, InterruptMaskPort
    mov    int_mask_register, ax
    in    al, InterruptMaskPort
    or    al, int_mask
%slow
    out    InterruptMaskPort, al
    pop    ax
    cld
    cbw
    xor    cx, cx
    mov    es, cx
    add    al, 68h     ;adding 8 converts int number to int type, i.e.,
             ;int 4 = type 12, int 5 = type 13 etc.
```

292062-50

NetWare Driver Source Code Listing (Continued)

```
    shl     ax, 1
    shl     ax, 1       ;two shifts = mul by 4 to create offset of vector
    xchg    ax, di
    mov     int_vector_addr, di     ;save this address for unhook
    mov     ax, es: [di]        ;save old interrupt vector
    mov     word ptr old_irq_vector, ax
    mov     ax, es: [di] + 2
    mov     word ptr old_irq_vector + 2, ax
    xchg    ax, bx      ;bx has the ISR offset
    stosw
    mov     ax, cs
    stosw
    ret

SetInterruptVector    endp




;
;    Driver Unhook
;
;    Assumes
;     DS = CS = IPX segment
;     Interrupts are DISABLED
;
;    Assumes any registers but DS, SS, SP may be destroyed
;
;    This procedure restores the original interrupt vector
;
;    This procedure will never be called if DriverInitialize
;    did not complete successfully.
;


DriverUnhook    PROC    NEAR
    in      al, InterruptMaskPort
    or      al, int_mask
    %slow
    out     InterruptMaskPort, al
    xor     ax, ax
    mov     es, ax    ;es is set to vector table segment
    mov     bx, word ptr int_vector_addr
    mov     ax, word ptr old_irq_vector
    mov     es: [bx], ax        ;restore old interrupt offset
    mov     ax, word ptr old_irq_vector + 2
    mov     es: [bx + 2], ax    ;restore old interrupt segment
    ret

DriverUnhook    endp


mombo_init    ends
end
```

292062-51

APPLICATION NOTE

AP-331

August 1989

Using the Intel 82592 to Implement a Nonbuffered Master Adapter for ISA Systems

JOSEPH DRAGONY
APPLICATIONS ENGINEER

Order Number: 292066-001

1.0 INTRODUCTION

The modern office has become increasingly computerized due to the availability of reasonably priced, yet very powerful, microcomputers. One of the rapidly growing uses of these powerful computers is desktop publishing. This technology allows text and graphics output to be generated that rivals the quality of work that could only be produced by very expensive phototypesetting equipment a few years ago. Another major application is Computer Aided Design (CAD). One thing that both of these applications have in common is that the output devices they require are still relatively expensive. Networking has enabled sharing these expensive peripherals, such as sophisticated laser printers, plotters, and FAX equipment, that would not be practical if attached to a single user machine. Since these peripherals are seldom in constant use by a single user, sharing them throughout an office over a LAN allows much better utilization of each unit. Through print spooling, the sharing of the equipment is transparent to the user except for the short walk to the print station to retrieve any spooled jobs. The cost reduction aspects of networking are beginning to be reflected in the network hardware itself. Media cost has been reduced, first from Ethernet to Cheapernet. Now the move to Twisted Pair Ethernet (TPE) lowers medium costs even further. The increased market for LAN adapters is also driving cost reduction in the adapter market. The 82592 Nonbuffered Master (NBM) is a simple, cost effective integrated LAN adapter for Industry Standard Architecture (ISA) workstations which addresses this need for cost reduction, coupled with high performance.

The NBM592 takes advantage of the increased bandwidth capabilities of the bus and memory subsystems in current ISA computers, commonly known as "AT" type computers. It is based on the Intel 82592 Advanced CSMA/CD LAN Controller. The NBM592 has its own DMA, which consists of an 82C37A DMA controller and support logic implemented in PALs and TTL. Part of this logic implements the master handshake which allows the NBM592 to take control of the host bus. This DMA is used to transfer data from the network directly into the host memory subsystem. The NBM contains no local buffer memory. This allows the cost of local buffer memory to be trimmed from the cost of the adapter. The low cost and very high performance of this adapter architecture make it uniquely suited to todays market.

Because the NBM592 is derived from the Embedded LAN Module (ELM) the reader might find the following Application Notes helpful. AP-320 *Using the Intel 82592 to Integrate a Low-Cost Ethernet Solution into a PC Motherboard,* and AP-327 *Two Software Packages for the 82592 Embedded LAN Module.* These publications are available from the Intel Literature Department.

1.1 Objective

The objective of this Application Note is to present the NBM592 architecture using the 82592. The implementation that will be described here uses readily available off-the-shelf devices. This low level of integration is presented as a starting point. Gate array or other ASIC technology could be used to reduce the parts count of this architecture while lowering cost and possibly increasing performance. The software aspects of this solution will also be discussed. A *NetWare** shell driver is the vehicle for illustrating the programming of the NBM.

1.2 Acknowledgements

I acknowledge and thank Dan Gavish of the Intel Israel System Validation group, David Bar-On of Moran Systems, Haifa, Israel, and Yosi Mazor of Intel MCFG LAN Marketing for their efforts in the definition, development, and debugging of the hardware. I also thank Ben L. Gee of San Jose, California for his work in modifying the Embedded LAN Module driver to run on the NBM592 hardware.

2.0 HARDWARE OVERVIEW

The NBM592 is an extension of the ELM architecture. The NBM592 differs from the ELM in the fact that it contains its own DMA resources. The NBM592 also contains logic to implement the ISA bus master handshake, which allows the NBM592 to operate as a master adapter on the ISA bus. This allows the adapter to transfer data from the network directly into host memory at higher speeds than the system DMA channels are capable of.

The NBM592 was specifically designed to work in 6- and 8-MHz IBM PC AT machines. Although the NBM592 has been tested successfully in a variety of other machines, a thorough worst-case timing analysis would be required to ensure proper functioning in clone machines using integrated chipsets. To implement the master handshake logic in a gate array or other ASIC, this analysis would need to be done for all of the current motherboard chip sets to ensure clone compatibility.

Figure 1 contains a block diagram of the NBM592 circuitry. The circuitry in the shaded area marked DMA logic and the TCI address latches from the area marked CSMA/CD logic would be good candidates for integration into gate array or other dense ASIC logic. The cost reduction benefits would depend on the level of integration.

*NetWare is a registered trademark of Novell Incorporated.

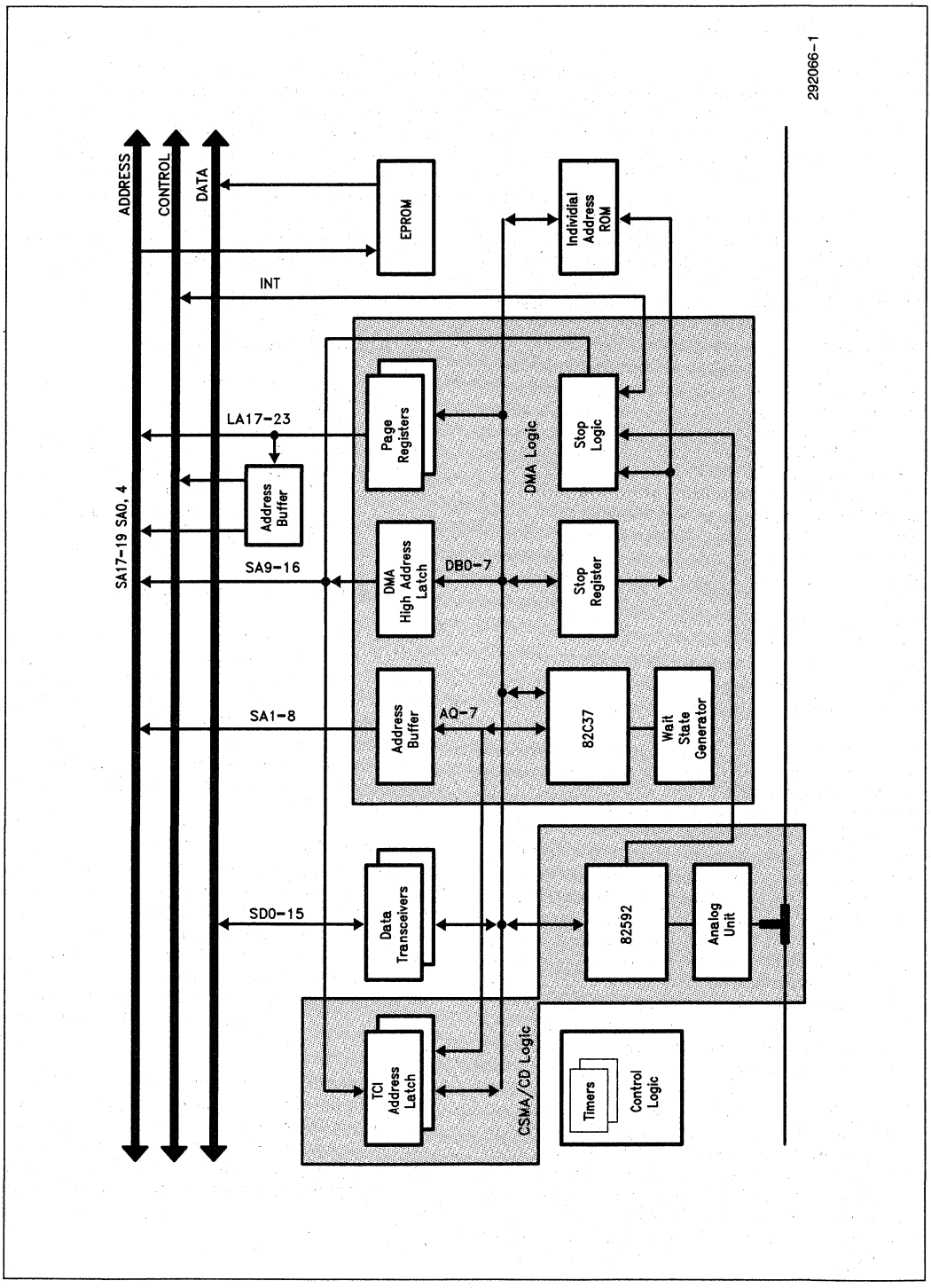


Figure 1. Nonbuffered Adapter Block Diagram

2.1 DMA Functional Block

The DMA functional block is comprised of an 8-MHz 82C37A DMA controller, page registers for the upper addresses in DMA cycles, a receive ring buffer overflow prevention circuit (stop register), a watchdog timer that limits NBM592 DMA bursts to less than 15 us, and a wait state generator for DMA cycles. Also contained in this block are two latches that store the address of the last memory location containing receive data.

2.2 CSMA/CD Functional Block

The CSMA/CD functional block is implemented by the 82592 Advanced CSMA/CD LAN Controller. This device supports all industry standard CSMA/CD LANs, such as IEEE 10BASE5, 10BASE2, 10BROAD36, 10BASE-T, and 1BASE5. The 82592 also supports proprietary CSMA LANs from 1 to 20 Mb/S such as the IBM PC Network. The 82592 also implements the CSMA/DCR protocol that provides deterministic collision resolution on CSMA LANs. This feature can be used when the worst case time for accessing the medium must be known.

The 82592 also implements a Tightly Coupled Interface (TCI) to industry standard DMA controllers that allows back-to-back frame reception and retransmission on collision to be done without CPU intervention. When the 82592 is configured to TCI mode it generates four additional DMA requests after the last byte of the frame has been transferred to memory. The first two of these transfers are used to move the status for the current reception into memory. The second two transfers write the number of bytes transferred into memory. By using this byte count value it is possible to reconstruct the chain of packets in memory so they can be handed off to the higher layers of the software. This will be discussed more fully in the software section of the Application Note.

2.3 Analog Interface

The analog interface for the NBM592 consists of a separate daughterboard that attaches to the NBM592 through an SBX connector. By using this approach it is possible to support IEEE 10BASE5, 10BASE2, 10Base-T, 1BASE5, and other proprietary network standards by simply removing one daughterboard from the digital assembly and installing a different analog interface. There are currently three analog interface modules, an Ethernet module, a Cheapernet module, and a Twisted Pair Ethernet (TPE) module, which is based on the Intel 82521 Serial Supercomponent.

2.4 System Bus Interface

The system interface for the NBM592 is I/O mapped. It uses 16 bytes of read/write I/O space. The 82592 command and status registers, 82C37A registers, page registers, and stop register are all accessible in this 16-byte address space. The IA ROM contents can also be read in this window.

3.0 DMA OPERATION

3.1 Better System Bus Utilization

The NBM592 operates as a DMA master on the I/O channel of the host computer. This means that all address and control signals are generated by the NBM592 while it is actively transferring data. The NBM592 DMA block is based on the 8-MHz 82C37A DMA controller. By providing its own DMA the NBM592 is able to transfer data between the network and memory at a higher rate than the system DMA channels would allow. Two of the four available channels of the on board 82C37A are used by the NBM592. In the default configuration Channel 0 is used as the transmit channel and Channel 1 the receive channel. Channels 2 and 3 are not used. The transmit and receive channels may be exchanged by using jumpers. The 82C37A provides address lines A16 through A1. Address lines SA16-SA9 on the ISA bus are latched from the multiplexed address/data bus of the 82C37A by ADSTB. Address lines SA8-SA1 are driven by the A7–A0 outputs of the 82C37A through a transceiver. A0 is pulled low during DMA transfers because all transfers are word aligned. The upper address bits are provided by the page registers, which are programmed during initialization. IC10 is the page register used for the Transmit channel, and IC11 is the page register used for the Receive channel. This architecture allows DMA transfers across a 128-kB memory space for both transmit and receive.

3.2 System Bus Arbitration

When the 82592 needs to perform DMA cycles it asserts its request to the on-board 82C37A. The 82C37A then asserts its HRQ pin. This pin is connected to the DRQ6 line in the I/O channel. When $\overline{\text{DACK6}}$ is returned the NBM592 drives the $\overline{\text{MASTER}}$ line in the I/O channel low, waits one clock and then drives the address bus. One clock later the NBM592 drives the control lines. The NBM592 may then perform DMA cycles for up to 15 μs. This time limitation exists to ensure that the system can access the bus to perform refresh cycles.

3.3 Transmit DMA Channel

If a collision occurs during transmit, the NBM592 must be able to reinitialize the DMA controller to point back to the beginning of the transmit buffer. This reinitialization must be done without CPU intervention to be ready to retransmit the frame within the 9.6 μs Interframe Spacing (IFS) time. The NBM592 does this by performing the TCI handshake with the 82592 to determine when a collision has occurred. The $\overline{EOP}$ pin on the 82C37A is then activated by the TCI logic. Since the 82C37A has been programmed to autoinitialize mode it resets its address to the beginning of the transmit buffer. After the IFS time and the random backoff time, if any, the 82592 will begin to make DMA requests and the frame will be retransmitted.

3.4 Receive DMA Channel

The receive DMA channel in the NBM592 uses a ring buffer. This is done by programming channel 1 of the 82C37A to autoinitialize mode. When the DMA channel reaches the end of the receive buffer space, it autoinitializes to the beginning of the receive buffer space and continues reception there. This approach ensures that the maximum possible buffering capacity is always available to the adapter. The integrity of the receive buffer is protected by a stop register, which is discussed in detail in section 3.4.1. A pair of latches is used to store the last address in memory that contains receive data. These latches are triggered by the TCI handshake at the completion of a receive operation. At that instant the latches are clocked and the address on the A16–A1 lines are latched. When the NBM592 receives a packet it appends four words of information. The upper bytes of these four words are not used. The lower bytes contain the status of the reception and the byte count of the frame. The byte count, along with the value from the TCI latches, is used to recover the received frame chain from the receive buffer. This process is discussed in section 7.3.1.

3.4.1 Stop Register

The Stop register (IC9) holds the stop address for the receive ring buffer. This implementation uses a 256-byte resolution. A finer resolution would require additional components. The CPU loads it with a new value as each receive buffer is processed. The value in the stop register is compared by IC8 to the corresponding address lines during DMA receive cycles (DMA_MW). When the contents of the latch and the address bus contain the same value, the $\overline{OVERFLOW}$ signal is activated. The $\overline{OVERFLOW}$ signal is latched by PAL 4, and the interrupt line is asserted. The OVERFLOW bit can be read by the CPU by an I/O read at offset 0Eh from the base address of the adapter. The bit appears on the D0 data line. When $\overline{OVERFLOW}$ is active, the Receive channel of the DMA is disabled until the stop register is reloaded by the CPU. This prevents corruption of the receive buffer structure during extremely heavy network traffic conditions. The stop register can be tested on power up by reading the overflow bit. The lower five bits of the stop register are used to select the IA PROM address. The $\overline{OVERFLOW}$ line is pulled up to V_{CC} to allow removing the overflow comparator and register IC8 and IC9 for a lower cost version of the board. If the stop register circuitry is removed it would be advisable to use the linear restartable buffering approach that was used in the ELM driver. In this approach, as frames are received, the driver software checks to see how much of the receive buffer remains available. When most of the buffer has been consumed the software reinitializes the DMA controller to point back to the beginning of the buffer space, and reception can resume.

3.5 Wait State Generator

The DMA circuitry also contains an optional wait state generator. Zero to three DMA wait states can be selected by a jumper that controls the wait state generator (IC23). Timing calculations show that one wait state is needed for a 6 MHz AT, and that no wait states are needed for an 8-MHz AT. The basic DMA transfer cycle is 3 clocks, two clocks for the command ($\overline{RD}$ or $\overline{WR}$) and one clock for address setup time. Wait states extend the command. The address setup time can be extended to two clocks by programming the 82C37A to normal write cycle. In this case a wait state should be added.

4.0 HARDWARE DESIGN CONSIDERATIONS

There are several circuits that are designed in specific ways, or use specific signals, to handle special cases of 82C37A/82592 interfacing. The reasons for the approach chosen in each case are individually discussed below.

4.1 Transmit EOP

When programmed for late write the 82C37A can receive DRQ after the end of S3. This can cause an extra DMA cycle. This can be solved by generating a special write during DMA cycles (for the 82592) This write starts when the $\overline{MEMR}$ signal is activated and ends when the 82C37A-generated $\overline{IO\_WR}$ signal ends.

The worst case timing for the sampling of the $\overline{BAD\_TX}$ signal shows a problem. The solution is to connect $\overline{592WR}$ to the flip-flop clock input, instead of $\overline{DMA\_MW}$. $\overline{BADTX}$ is sampled by the rising edge of $\overline{DMA\_MW}$. $\overline{BADTX}$ is $\overline{EOP}$ * DRQ delayed by one pal. $\overline{DMA\_MW}$ is not delayed by the PALs; thus timing requirements are met and $\overline{DISDACK}$ is generated properly.

4.2 Separate Receive and Transmit Page Registers

An address setup time of 120 ns is required before asserting the command (Read or Write). This is done by adding a 1-clock delay to the first command in each burst, and asserting an additional wait state to this first command. The signal EN_CMD is generated by PAL3. EN_CMD is activated two clocks after DACK0 or DACK1, and remains active until the end of the cycle. A wait state is added to this first command by the qualification of the RD_OR_WR signal by EN_CMD. The EN_CMD controls the enable line of the command bus buffer. This buffer drives SA0 and BHE, which are part of the address and must be active before command. These lines are driven by the PALs.

4.3 Extra Wait State

The 82C37A IOW signal can be active after the end of the 82C37A S4. This will cause an extra wait state to be inserted. This can be eliminated by using the 82C37A MEMW signal for the wait state generator instead of the 82C37A IOW. This line will always go inactive before the end of S4. This prevents the insertion of extra wait states.

4.4 TCI-Direction

When the TCI latch is read by the CPU the data buffer direction line that is driven from the RD line is in the wrong direction. This is because the DMA controller clear mask register can be accessed by writing to the same address. To prevent this the RD line has been disabled.

When addressing the DMA clear mask register the 37_CS is deactivated. This prevents access to this register. The local RD signal is driven by IORD during slave cycles.

4.5 Bus Contention

Since the 82C37A specification does not guarantee that the mid address (strobed by ADSTB) will float before the command is active, contention can occur on these lines. The solution is to delay the command (Read or Write) until after each ADSTB cycle. This is done by generating EN_CMD, which disables the command for one clock and adds one wait state in addition to those added by the wait state generator. This is implemented in PAL3.

The high data buffer is enabled by EN_ADDR, which is active only during master cycles. This prevents the NBM592 from enabling the high data buffer (SD15–SD8) during slave accesses to odd I/O addresses. If this was not done, contention with low data multiplexed into the high data by the motherboard would occur.

4.6 DRAM Precharge Time

There is a problem with the worst case timing of the 82C37A when more than one transfer cycle is executed. The problem is that the worst case time between two commands can be lower than the precharge time required by the DRAMs. If extreme values are taken for two delay parameters, (maximum value for inactive time and minimum value for active time) the DRAM precharge time will be violated. We assume that for the same signal the difference between those two parameters does not exceed 30 ns. This satisfies the precharge time for the DRAM chips. The required precharge time is 100 ns. For the address setup time, the assumption is that command is activated after the address is stable (i.e., the address setup time is greater than zero). The address path to the memory chips consists of the delay through the 74LS245 transceiver on the NBM plus the delay of the 74F158 in the host system. The total delay is 19 ns. The command path to RAS consists of one 74LS244 on the NBM plus a 74F10 and a 74F00 in the host system. This path totals 9 ns. The required setup time specified for the DRAM chips is 0 ns. Therefore, a 10 ns setup time from the 82C37A will satisfy the required setup time. The same analysis holds for read and write cycles.

The 82C37A worst case timing does not guarantee that the Read command signal will stay active after the Write command is deactivated. For proper board operation the Read must stay active after the deactivation of the Write signal.

5.0 DATA PATH

The data path includes the 82592, the interface to the analog circuit, the 16-bit address latch for the TCI address (IC2 and 4), the 16-bit data transceiver for buffering data (IC3 and 5), and the IA PROM which contains the station address (IC6). 82592 connections and signal names are the same as in the ELM. The low address latch latches its data directly from the 82C37A lines in order to minimize the loading on the bus. The high address latch latches its data from the system's SA lines.

The Station IA is read from the PROM, which is enabled by PROM_CS. To read the IA PROM, the CPU first preloads the stop register with the address of the byte to be read. The CPU then reads from I/O address 30Ah. In the current design the 82592, the DMA, and all the other circuitry is clocked by the same 8-MHz clock. In future versions the 82592 can be clocked by a 16-MHz clock. In this case, the 8-MHz clock to the rest of the board will be generated by dividing the 16-MHz clock by 2, in the unused flip-flop of IC15A. This requires jumper changes. There is an option of driving the 82592 clock from a 16 MHz clock. The clock can be divided by 2 to produce the local 8-MHz Clk to the 82C37. The local oscillator could be eliminated by using a 10-MHz 82C37A and using a

Table 1. NBM592 I/O Map

Address	DMA Register	Write	Read
300h	0	Base and Current Address No. 0	Current Address No. 0
301h	8	Command Register	Status Register
302h	1	Base and Current Word Count No. 0	Current Word Count No. 0
303h	9	Request Register	
304h	2	Base and Current Address No. 1	Current Word Count No. 1
305h	A	Single Mask	
306h	3	Base and Current Word Count No. 1	
307h	B	Mode Register	
308h	4	592 Port 0	592 Port 0
309h	C	Byte Pointer FF	
30Ah	5	Page Register 0 (Tx)	IA PROM
30Bh	D	Master Clear	Temporary Register
30Ch	6	Page Register 1 (Rx)	Low TCI Byte
30Dh	E		High TCI Byte
30Eh	7	Stop Register	Overflow Flag
30Fh	F	Write Mask Register	

buffered version of the TxC signal generated by the 82C501AD on the analog module to clock all NBM592 circuitry.

Provisions have been made for a boot EPROM (IC25). This optional device is accessed during the system boot process. The BIOS searches for a remote boot ROM, and if one is found the ROM initialization code is executed. IC24 serves as its address decoder. EPROM size and memory allocation are jumper selectable (see APPENDIX B for details). If a remote boot is not needed, both IC24 and IC25 can be omitted.

6.0 PC AT I/O CHANNEL INTERFACE

The board was designed to occupy no more than 16 I/O addresses; to meet this restriction, during slave mode access to the 82C37A SA0 is routed to A3 and during DMA cycles A3 is routed to SA4.

The NBM592 uses a 16-bit DATA path. All signals in the data path are buffered. SA lines are decoded directly by the PAL, and driven by the DMA through buffers. LA lines are driven by the latches. $\overline{\text{MEMRD}}$ and $\overline{\text{MEMWR}}$ lines are driven by the DMA. $\overline{\text{SMEMRD}}$ and $\overline{\text{SMEMWR}}$ are buffered by the system. The $\overline{\text{IORD}}$ and $\overline{\text{IOWR}}$ signals are inputs and are buffered by the PALs. AEN is an input to the decode PAL.

The NBM592 can used IRQ 10, 11, 12, 14, or 15. IRQ10 is the default interrupt request, driven by the 82592 interrupt signal OR'd with the overrun latch. The interrupt line is jumper selectable. Jumper locations to select the various lines are given in the jumper tables.

The master handshake requires the use of one host DMA channel. In the NBM592 host DMA channels 5, 6, or 7 can be used. The channel is used in cascade mode to allow the NBM592 to master the host bus. The default connection is channel 6 with channels 5 or 7 available through jumper selection (see APPENDIX B for details). The $\overline{\text{MASTER}}$ signal is activated by the PALs when the board DMA is active.

6.1 Refresh Watchdog Timer

There are two watchdogs on the NBM592. The watchdogs are driven from the local 8-MHz clock. Watchdog No.1 is used to ensure that the refresh mechanism will be able to gain control of the bus when it needs to.

Refresh cycles occur approximately every 15 μs. When a refresh request occurs the DMA must release the bus within 15 μs. This is done by using a time constant of 12 μs in the watchdog. When a refresh request is sensed the watchdog starts to run. The watchdog timer will expire after approx. 12 μs. This corresponds to W5 * W6 at 8-MHZ, the 3 extra μs will be spent transferring bus control between the two DMAs. After the bus is relinquished, the request is regenerated one clock after $\overline{\text{DACK6}}$ is inactivated. Analysis and lab inspection show that while working with no wait-states, 82592 bursts do not exceed 12 μs. Therefore, this circuitry may be removed from future versions of the board.

6.2 Floppy Disk Watchdog Timer

Watchdog No.2 is an optional floppy disk watchdog (SPARE-1). The purpose of this watchdog is to avoid the possibility of bus starvation to the floppy disk during DMA bursts by the NBM592. DRQ2 is used to sense activity of the floppy drives. The watchdog drops the 82592 request with a delay after a floppy DMA request is encountered. This watchdog is disabled by a jumper, as it is redundant. This is an optional feature and is not used in the present implementation.

The + 12-V line in the ISA bus provides power to the analog module.

The Reset line from the bus is used to reset the NBM592 circuitry during system initialization.

7.0 SOFTWARE

The software discussion in this Application Note is based on a driver intended to be used with Novell *NetWare* V2.1. The driver is based on the driver that appears in AP-327, *Two Software Packages for the 82592 Embedded LAN Module.* There are two major differences between the driver in AP-327 and the driver in this Application Note. First, this driver uses a ring buffer approach, as opposed to the linear restartable buffer used in the ELM. Secondly, this driver uses macros for conditional blocks to allow the code to be written in a manner resembling a high-level language. This makes the code more readable for those with limited assembly language experience.

While this driver is written to run with a specific networking package, it contains all the functions that would normally be required by any networking package. Once a good understanding of the code is gained it should be possible to modify most of the procedures to operate under another networking package. The main differences will be the format of the communicating structures between the driver and the lowest layer of the networking software. The procedures that will be discussed in detail in this Application Note are DriverInitialize, DriverSendPacket, DriverISR, and DriverPoll. These four procedures are the backbone of the driver and represent the most important code for understanding the functionality of the NBM592. Procedures called from within the primary procedures will also be covered.

The source code for the procedures discussed below is included as APPENDIX E.

7.1 Initialization

In our software example, initialization is carried out by the procedure DriverInitialize. This procedure is called by the networking software when it is loaded. This procedure initializes the hardware and any software variables that must be initialized at run time. The transmit and receive buffer variables are initialized through a call to SetUpBuffers. DriverInitialize also calls the procedure SetInterruptVector to initialize the proper entry in the system interrupt vector table, after first saving the vector that is already there.

7.1.1 DriverInitialize

The first function that DriverInitialize performs is to set the variable MaxPhysPacketSize to 1024. This value is used to negotiate the maximum size of the frames that will be transferred between the fileserver and the workstation.

Next, the base I/O address is read from the configuration table and this value is added to the offset value for each register in the NBM592 I/O space. This includes the 82C37A registers, the stop register, the TCI latches, and the IA PROM address.

The CPU now reads the Master DMA channel number from the configuration table and calculates the required variables. This is the host DMA channel that will be used to implement the master handshake between the NBM592 and the host.

The next operation is to read the station address from the address PROM. This is done by first writing the address of the byte to be read to the Stop register and then reading from the IA PROM port. The value written into the stop register is used to drive the address inputs of the IA PROM. The code to read the PROM is implemented as a loop, with the value written to the latch staring at zero and incrementing through five to read the six bytes of station address. These six bytes of address are stored in the array node_addr and also are written into a location in IPX's space. The location IPX wants the address written to is passed in the DI register when DriverInitialize is called.

After the station address has been read and stored, DriverInitialize loads the AL register with the number of the interrupt line that the NBM592 will use, loads the BX register with the offset of the procedure DriverISR,

and calls the procedure SetInterruptVector. Details of this routine are provided in section 7.1.2. After SetInterruptVector returns, a call is made to the procedure SetUpBuffers. SetUpBuffers initializes all the buffer management variables. Details of this procedure appear in section 7.1.3.

After SetUpBuffers returns, DriverInitialize is ready to configure the DMA channels that the NBM592 will use. One host DMA channel and two of the on-board DMA channels will be configured. The host DMA channel is configured to cascade mode. This allows the onboard DMA to use this channel for arbitration in the ISA bus. The onboard DMA controller is configured for extended write, active low DREQ, and rotating priority. The transmit DMA channel in the onboard controller is programmed next. The channel is configured to autoinitialize mode to allow retransmission on collision without CPU intervention. This channel will be used to transfer the configuration and address parameters to the 82592.

The 82592 operates in the 8-bit-bus mode after reset. It is put into the 16-bit-bus mode by giving it a Configure command with zero in the byte count field. This is the first command that the driver issues to the 82592. The transmit channel is set up to point to the beginning of the transmit buffer area. The word count is set to 1 because the 82C37A interprets this register as transfers-to-be-made − 1. A Configure command is now given to the 82592. DriverInitialize now enters a polling loop to determine when the command has been completed. The software can tell when the command is complete by reading the 82592 Status0 register and testing to see if the interrupt bit is set. This loop will be repeated a maximum of 65,536 times. If the command has not completed by that time, a pointer to an error message is moved into the AX register and control is returned to IPX. At that point the error message will be displayed and the loading of the driver will be aborted.

After the first Configure command has completed, another Configure must be done to actually load the desired parameters into the 82592. The transmit channel is set up to point to the beginning of the transmit buffer space and the word count is set to eight. This will allow the nine required transfers to be made. The byte count and configuration parameters are copied into the transmit buffer area and a Configure command is issued to the 82592. Once again a polling loop is entered to wait for command completion.

To set the station address the transmit channel is set up to point to the beginning of the transmit buffer and the word count is programmed to 3. The byte count and station address are copied into the transmit buffer and an IA Setup command is issued to the 82592. The 82592 is again polled for command completion.

Now that the 82592 is initialized, the receive DMA channel can be set up. This channel is also programmed to autoinitialize mode and the word count is set to the size of the receive buffer − 1. This will cause the DMA to wrap around to the beginning of the receive buffer when it reaches the end. This results in a ring buffer. The receive stop register is programmed with a value near the end of the buffer. The receiver is enabled by issuing a Receive Enable command to the 82592. The AX register is zeroed to indicate that the initialization completed successfully and control is returned to IPX. The hardware is now ready for operation.

7.1.2 SetInterruptVector

The CPU reads the value of the interrupt line to be used from the configuration table. It puts this value in the AL register. The offset of the Interrupt Service Routine (ISR) is placed in the BX register and SetInterruptVector is called. This procedure calculates the mask and unmask variables for the interrupt channel that will be used for the driver. This channel is then masked to prevent any unwanted interrupts. The CPU now calculates the address in the interrupt vector table where the vector will be stored. After saving the vector that is already at the location to be used SetInterruptVector installs the interrupt vector for the NBM592. The procedure ends in a return that passes control back to Driver initialize.

The next initialization task is to set up the transmit and receive buffer space to accommodate the architecture of the NBM592 DMA subsystem. This is done by a call to SetUpBuffers.

7.1.3 SetUpBuffers

The NBM592 DMA architecture is essentially the same as the ISA DMA subsystem. It is made up of an 82C37A supplying A16–A1 and a page register supplying A17–A23. A0 is pulled low when DMA transfers are being made because all transfers are done on word boundaries. Because of the fact that no carry can be generated from A16 to A17, the buffers must be located such that no 128-kB boundary exists in them. If the address of the 82C37A is allowed to roll over from FFFFh to 0000h the page register will remain unchanged. This will cause memory locations at the very bottom of the 128-kB page to be overwritten. SetUpBuffers prevents the occurrence of this problem by checking to see if a boundary exists in the buffer area and then allocating the buffer space to the transmit and receive buffers accordingly. It is strongly recommended that commercial implementations of the NBM concept use counters instead of latches for the upper address bits. This would eliminate the problems associated with the page register implementation and would simplify buffer setup and processing.

The SetUpBuffers procedure in this Application Note is an improved version of the procedure presented in AP-327. Although the general approach is the same, several changes have been made to accommodate the ring buffer implementation.

7.2 DriverSendPacket

Transmission on the network is accomplished by the procedure DriverSendPacket. When the IPX wants to send a packet to the fileserver or another station, it prepares a Transmit ECB and calls DriverSendPacket. The address of the ECB is passed in the ES:SI register pair. The procedure checks to see if frames are already queued for transmission. If frames are queued, DriverSendPacket adds the new ECB to the end of the queue and returns control to IPX. If no frames are queued for transmission, execution falls through to the procedure StartSend.

7.2.1 StartSend

The procedure StartSend is responsible for actually building the frame in the transmit buffer, setting up the DMA controller, and issuing the Transmit command to the 82592. This routine also calculates any padding needed to bring the frame up to minimum Ethernet length.

The first action that StartSend takes is to set the transmitting flag in the driver workspace area of the ECB. This flag is used to ensure that only valid transmit ECBs are returned to IPX by the transmit ISR. If a transmit request is cancelled and then the interrupt for the cancelled transmit occurs, the code could erroneously return a packet that had never been transmitted. Having this flag available prevents this.

If the IPX packet plus the Ethernet overhead bytes do not add up to a frame size of 64 bytes StartSend calculates the number of padding bytes required and stores this value in memory for later use. After the padding calculations have been done StartSend begins to build the transmit frame in memory. The frame begins with the 82592 byte count. This includes the IPX packet, the Ethernet header and CRC bytes, and the chaining byte at the end of the frame. In this application the chaining byte will always be zero, since chaining is not supported.

The transmit ECB contains a fragment list which describes the length and location of each fragment in memory that makes up the frame to be sent. This list is processed by StartSend with the fragments being copied in order into the transmit buffer. After the copy is complete any required pad bytes are moved into the end of the buffer.

After the transmit frame has been built in memory the DMA controller and page register are programmed with the address of the beginning of the transmit buffer. The word count for the frame is written to the DMA controller and then it is unmasked. Writing a Transmit command to the 82592 causes it to begin making DMA requests and transmission begins. The starting time of the transmission is saved in memory and StartSend returns control to the calling code.

7.3 DriverISR

DriverISR is the interrupt service procedure. It calls the procedures RcvdPacket or SentPacket after it has determined the source of the interrupt.

The first task in the Interrupt Service Routine (ISR) is to save the machine state. This is done by pushing all the registers on the stack as soon as the ISR is entered. Once the machine state is saved the program is free to use all of the processor's registers for its own purposes. The segment registers are then set so they all point to the same segment since the driver is implemented as a .COM program. In this memory model code and data share the same segment.

The ISR code next issues an End Of Interrupt (EOI) to the two 8259A Programmable Interrupt Controllers (PIC). This allows the PICs to accept interrupts from other sources. Since the PICs are configured in the edge triggered mode they can be cleared before the 82592 interrupt has been cleared. In a system that uses level triggered interrupts, the interrupt from the 82592 would have to be cleared first. If this architecture were migrated to the PS/2™ it would require the 82592 to be acknowledged first because the PS/2 systems use level triggered interrupts.

DriverISR now checks to see what event caused the interrupt. This is accomplished by comparing the status read from the Status0 register of the 82592 to the event codes for receive, transmit and retransmit. The value read from Status0 is AND'd with 0DFh prior to the comparison to mask the state of the Exec bit. This simplifies the comparison step. If the event code is not receive, transmit, or retransmit, the driver increments an error counter called false_590_int and proceeds to the exit code. If the event code is receive, the procedure RcvdPacket is called. If the event code is transmit or retransmit, the procedure SentPacket is called. Upon return the driver proceeds through the exit code.

7.3.1 RcvdPacket

The driver's first action upon entering the RcvdPacket procedure is to read the TCI address latches. These two 8-bit latches contain A1–A16 of the ending address of

the last frame received. This address is used as the starting point for the buffer reconstruction process. It is saved in a variable called rx__buff__tail. The last four words of the receive buffer contain status and length information for the packet. By subtracting the length of the current buffer from the current address read from the TCI latches, the end of the previous frame can be found. By repeating this process the complete chain of unprocessed frames can be reconstructed. The status bytes are used to determine whether the frame should be processed or discarded. The procedure Normalize Pointer is used to account for the possibility that the packet is wrapped around in the receive ring buffer while the status and length bytes are being read.

Each packet contains two length fields. One is contained in the IPX header and the other is contained in the Ethernet header. The length of the packet is validated by doing several length checks. The length of the Ethernet header is subtracted from the total bytes received prior to doing the length checks. The first length check determines if the packet exceeds the 1088 byte maximum length for this driver (1024 data bytes and 64 *NetWare* bytes). The next length check determines if the frame is shorter than the minimum dictated by IPX (30 bytes plus padding). The final check makes sure that the IPX length and the actual number of bytes received agree. If any of these length checks fail, the appropriate error counter is incremented and the frame is discarded. If all the length checks pass, the packet is added to a list of good received frames by putting a pointer to the first byte of the frame into the array rx__ list and incrementing the variable num__of__frames. Since the length of rx__list is limited to 30 entries a check is made to see if this is the last entry in rx__list. This cycle is repeated until all frames have been processed, or all entries in rx__list have been used. When one of these two events occur the driver enters a small loop of code that takes care of handing the received packets off to IPX.

The handoff loop is controlled by the variable num__ of__frames. After each frame is handed off to IPX num__of__frames is decremented. When num__of__ frames reaches zero there are no more frames to hand off and the loop terminates. The list is processed by reading the offset of the first byte of the frame from rx__list. This offset is used to read the socket number from the IPX header of the frame. The socket number is used as a parameter for a call to the IPX routine IPXGetECB. If there is an ECB available for that socket IPX passes a pointer back to the driver. If an ECB is available the loop calls the procedure DeliverPacket, which does the processing necessary to transfer a packet from the driver to IPX. If no ECB is available, the next frame is processed.

After all frames have been processed the stop register is checked to see if a receive overflow occurred. If an overflow occurred the variable rx__buff__overflow is incremented. The stop register is then updated by writing the value of rx__buf__tail − 256 into it. The value of receive buffer head is then updated by writing the value of rx__buf__tail + 2 into it. The variable rx__buf__ head now points to the first byte of the next receive buffer. Execution now returns to DriverISR.

7.3.2 SentPacket

The first action taken in the SentPacket procedure is to test the software flag tx__active__flag. If this flag is not set then a transmit had not been initiated and the transmit interrupt is erroneous. In this case, control simply returns to DriverISR. If tx__active__flag is set the driver reads the status of the transmission from the 82592. The driver tests the status to see if the transmission completed successfully. If an error occurred the status is tested to determine the type of failure, and the corresponding error counter is incremented. If the transmit completed successfully, this code is skipped. Next the driver extracts the total number of collisions the frame experienced and adds this value to the variable RetryTxCount. The driver writes a completion code for the transmission into the ECB, unlinks it from the transmit queue, and returns it to IPX through a call to IPXHoldEvent. The driver now checks the transmit queue to see if any packets are awaiting transmission. If there is a packet in the queue, the driver loads the ES:SI register pair with the address of the ECB and calls StartSend. If no packets are queued, control is returned to DriverISR.

7.3.3 Exiting DriverISR

After control is returned to DriverISR the driver checks to make sure that the receiver is enabled. If it is not, then a Receive Enable command is issued to the 82592. DriverISR then issues an Interrupt Acknowledge to the 82592. This clears the interrupt that caused entry into the DriverISR code and allows any new interrupt that may have occurred during processing to move into the 82592 Status0 register. DriverISR reads this register to determine if a new interrupt occurred. If a new interrupt is detected then execution loops back to the beginning of the ISR and the new interrupt is processed. If no new interrupt is detected a call is made to IPXServiceEvents to tell IPX it has events to process, the machine state is restored to its condition when DriverISR was entered, and an IRET instruction returns control to the code that was executing when the interrupt occurred.

7.4 DriverPoll

The procedure DriverPoll is called by IPX to allow the driver to check for timed-out transmits and any other non-interrupt-driven events that need to be handled. DriverPoll first checks to see if tx__active__flag is set. If it is not, then control is returned to IPX. If tx__active__flag is set, then the driver checks to see if the current transmission has timed out. This is done by reading IPXIntervalMarker, subtracting tx__start__ time from the value read, and comparing the result with TxTimeOutTicks. If the result of the subtraction is greater than TxTimeOutTicks, the transmission is aborted by issuing an Abort command to the 82592. DriverPoll then writes a bad completion code into the ECB for the packet, unlinks the ECB from the transmit queue, and returns the ECB to IPX through a call to IPXHold Event. The 82592 is then given a Selective Reset command to put it in a known state. All configuration parameters are maintained when a Selective Reset is done but the Receive, Execution, and FIFO machines are all put in a known state. A Receive Enable command is given to the 82592 to reenable the receiver. DriverPoll then checks the transmit queue to see if any packets are queued. If a packet is awaiting transmission, the ES:SI register is loaded with the address of the ECB and StartSend is called. If no packets are waiting control is simply returned to IPX.

APPENDIX A
SPECIAL CONSIDERATIONS FOR A-1 STEPPING ANOMALIES

There are a few anomalies in the operation of the A-1 stepping of the 82592 that require workarounds in the NBM592. They are discussed below.

Receive EOP

When bad frames are received, the 82592 signals this to the hardware by not dropping the DREQ signal during EOP. This can cause the 82C37A to issue another DMA cycle. This extra cycle can corrupt the receive buffer chain. The NBM592 contains an Extra DMA Read Elimination circuit to prevent this extra read cycle. The DREQ signal is disabled during the time that the EOP is active. The DREQ1# signal to the 82C37A is qualified by the $\overline{EOP}$ signal to eliminate another cycle. The $\overline{EOP}$ goes active after the activation of the RD# pin, this deactivates DREQ1#, thus the DMA will not issue another transfer. The $\overline{EOP}$ signal to the DMA controller is blocked during receive cycles. This is done because the receive channel is reinitialized when the upper limit of the receive buffer is reached, not at the end of each receive.

NOTE:
The NBM592 does not discard bad received frames.

Transmit EOP

The NBM592 also contains an Extra DMA Write Elimination circuit for the Transmit channel. In some case, when $\overline{EOP}$ and DREQ are driven active, the 82C37A can execute an extra write cycle. This redundant cycle can be wrongly interpreted by the 82592. The NBM592 contains a circuit that eliminates this extra cycle, if it occurs. To disable the extra write, a signal DISDACK# is generated. This signal, when active, disables the DACK to the 82592, causing the 82592 to ignore the write cycle. This signal is activated when at the rising edge of the 82592 WR# signal (at DMA cycles), a bad Transmit event is identified. A BAD_TX# signal is generated when the DRQ signal is active during the occurrence of the 82592 EOP#. The DISDACK# signal is deactivated when either the next write arrives, or when the current DMA burst ends (this is identified by the DACK signal going inactive). This function is implemented in IC15B and PAL2.

In many cases the extra write cycle does not occur. The DISDACK signal, in this case, would cause the elimination of the next write cycle. This write cycle would be the first cycle of the next frame, which would cause undefined data to be sent instead of the next frame. This problem was remedied by canceling the DISDACK signal at the end of the current burst. This is done by connecting the preset signal of the DISDACK flip-flop to the DMA controller's DACK0 signal. Thus if the extra write occurs, it is eliminated.

Transmit Error EOP

When a collision occurs after all the transmit data has been transferred to the FIFO, the 82592 does not issue the bad transmit EOP. This will cause the DMA controller to continue with the retransmit cycle from the current address, instead of autoinitializing to the beginning of the transmit buffer. One solution to this problem is to program the 82C37A count register to the actual transmit count. This will cause the 82C37A to autoinitialize. This solution can cause a problem with good transfers. The 82592, when transmitting in TCI mode, can transmit a chain of frames. After the end of a good transmit the 82592 issues another DMA cycle to read the next memory location in the transmit buffer. If the first three bits of that location are binary 100, then the 82592 attempts to transmit another frame. With the proposed solution to errata No.1, the 82592 will read the chained command from the byte-count field of the Transmit buffer. To prevent an extra frame transmission, the NBM592 uses the 82592 $\overline{EOP}$ signal, which is active during the chain command read, to force the D0 line high. This is done by generating a KIL_DATA# signal which disables the bidirectional data buffers. A pull-up resistor on the local D0 forces this line to '1', thus ensuring that the value read by the 82592 will not be binary 100.

APPENDIX B
JUMPERS

Jumpers are provided to allow selection of interrupt line used, DMA wait states, and the host DMA channel used for the master handshake. The tables below show how to set the jumpers in each jumper block as required for your configuration.

```
INTERRUPT JP1
IRQ10 (default)    pins 1-6
IRQ12              pins 3-8
IRQ13              pins 4-9
IRQ14              pins 5-10

HOST DMA CHANNEL JP2
DRQ5               pins 1-7 and 2-8
DRQ6 (default)     pins 3-9 and 4-10
DRQ7               pins 5-11 and 6-12

EPROM SIZE JP3
8 kB               pin 1-7 not connected
                   pin 2-8 not connected
16 kB              pin 1-7 connected
                   pin 2-8 not connected
32 kB              pin 1-7 connected
                   pin 2-8 connected
```

EPROM ADDRESS JP4

	Jumpers						
	A19	A18	A17	A16	A15	A14	A13
Address	1-8	2-9	3-10	4-11	5-12	6-13	7-14
C8000	NC	NC	C	C	NC	C*	C**
CA000	NC	NC	C	C	NC	C*	NC**
CC000	NC	NC	C	C	NC	NC*	C**
CE000	NC	NC	C	C	NC	NC*	NC**
D0000	NC	NC	C	NC	C	C*	C**
D2000	NC	NC	C	NC	C	C*	NC**
D4000	NC	NC	C	NC	C	NC*	C**
D6000	NC	NC	C	NC	C	NC*	NC**
D8000	NC	NC	C	NC	NC	C*	C**
DA000	NC	NC	C	NC	NC	C*	NC**
DC000	NC	NC	C	NC	NC	NC*	C**
DE000	NC	NC	C	NC	NC	NC*	NC**
E0000	NC	NC	NC	C	C	C*	C**

* When using a 32 kB EPROM this jumper is not used for address selection. Pin 13 should be connected to JP6.
** When using a 16 kB or 32 kB EPROM this jumper is not used for address selection. Pin 14 should be connected to JP5.

```
I/O ADDRESS SELECT JP10
300h-30Fh          pins 1-2 connected
310h-31Fh          pins 1-2 not connected

WAIT STATES JP11
0 WS                      pins 1-5
1 WS (use for 6 MHz AT)   pins 2-6
2 WS                      pins 3-7
3 WS                      pins 4-8
```

APPENDIX C
PAL EQUATIONS

PAL 1

```
PAL 1
module PAL1 flag '-R3'
title 'NBM592 - PAL1 X023'
IC18 device 'P20L10' ;
SA0                 pin 1;  "in
SA1                 pin 2;  "in
SA2                 pin 3;  "in
SA3                 pin 4;  "in
IORD_BAR            pin 5;  "in
IOWR_BAR            pin 6;  "in
LIMIT_LATCH_BAR     pin 7;  "in
BOARD_CS_BAR        pin 8;  "in
EN_ADDR_BAR         pin 9;  "in
EPROM_CS_BAR        pin 10; "in
NC1                 pin 11; "not used
NC2                 pin 13; "not used
LD_RX_BAR           pin 14; "out
LD_TX_BAR           pin 15; "out
LD_LIMIT_BAR        pin 16; "out
OE1_BAR             pin 17; "out
OE2_BAR             pin 18; "out
i592CS0_BAR         pin 19; "out
i37CS_BAR           pin 20; "out
PROM_CS_BAR         pin 21; "out
D0                  pin 22; "out
RD_BAR              pin 23; "out
H,L,X=1,0,.X.;
```

292066-2

Equations

```
!LD_TX_BAR = !BOARD_CS_BAR & SA3 & !SA2 & SA1 & !SA0 & !IOWR_BAR; "0AH
!LD_RX_BAR = !BOARD_CS_BAR & SA3 &  SA2 & !SA1 & !SA0 & !IOWR_BAR; "0CH
!LD_LIMIT_BAR = !BOARD_CS_BAR & SA3 & SA2  &  SA1 & !SA0 & !IOWR_BAR; "0EH
!OE1_BAR = !BOARD_CS_BAR & SA3 & SA2 & !SA1 & !SA0 & !IORD_BAR; "0CH
!OE2_BAR = !BOARD_CS_BAR & SA3 & SA2 & !SA1 &  SA0 & !IORD_BAR; "0DH
!i592CS0_BAR = !BOARD_CS_BAR & SA3 & !SA2 & !SA1 & !SA0 & EPROM_CS_BAR; "08H
!i37CS_BAR = !BOARD_CS_BAR & EPROM_CS_BAR &( (!SA3) # (SA0 & SA1) # (!SA2 & SA0) ) ;
enable D0 = !BOARD_CS_BAR & SA3 & SA2 & SA1 & !SA0 & !IORD_BAR; "0EH
!D0 = LIMIT_LATCH_BAR;
!PROM_CS_BAR = !BOARD_CS_BAR & SA3 & !SA2 & SA1 & !SA0 & !IORD_BAR; "0AH
enable RD_BAR = EN_ADDR_BAR;
!RD_BAR = !IORD_BAR # !EPROM_CS_BAR;
end PAL1
```

292066-3

PAL 2

```
module PAL2 flag '-R3'
title 'NBi592 -pal2  REV X024'
IC19 device 'P20L10' ;
RD_BAR                  pin 1;  "in
NC1                     pin 2;  "in (spare)
i592EOP_BAR             pin 3;  "in
i592DREQ0               pin 4;  "in
i592DREQ1               pin 5;  "in
DAK0_BAR                pin 6;  "in
DAK1_BAR                pin 7;  "in
RESET                   pin 8;  "in
WATCHDOG_BAR            pin 9;  "in
DISDACK_BAR             pin 10; "in
i592DRQ0DD              pin 11; "in
LIMIT_LATCH_BAR         pin 13; "in
LTCW                    pin 14; "out
DREQ0_BAR               pin 15; "out
DREQ1_BAR               pin 16; "out
i592DACK_BAR            pin 17; "out
MSEOP_BAR               pin 18; "out
KILL_DATA_BAR           pin 19; "out
DAK0                    pin 20; "out
WD_TX_BAR               pin 21; "out (for internal use)
WD_RX_BAR               pin 22; "out (for internal use)
BADTX_BAR               pin 23; "out
```

292066-4

Equations

```
DAK0 = !DAK0_BAR;

LTCW = !RD_BAR & !i592EOP_BAR & !DAK1_BAR;

!WD_TX_BAR = i592DREQ0 & !DAK0_BAR & !WATCHDOG_BAR & !RESET # !WD_TX_BAR
& !WATCHDOG_BAR & !RESET;"Arm when DACK active"

!WD_RX_BAR = i592DREQ1 & !DAK1_BAR & !WATCHDOG_BAR & !RESET # !WD_RX_BAR
& !WATCHDOG_BAR & !RESET;"Arm when DACK active"

!DREQ1_BAR = i592DREQ1 & DAK1_BAR & WD_RX_BAR & LIMIT_LATCH_BAR #
i592DREQ1 & i592EOP_BAR & WD_RX_BAR & LIMIT_LATCH_BAR;

!DREQ0_BAR = i592DREQ0 & i592DRQ0DD & !RESET & WD_TX_BAR # !DREQ0_BAR &
DAK0_BAR & !RESET & WD_TX_BAR; " KEEP TIL DACK

!i592DACK_BAR = !DAK1_BAR# !DAK0_BAR & DISDACK_BAR;

enable MSEOP_BAR = !i592EOP_BAR & !DAK0_BAR;

MSEOP_BAR = 0; "EOP TX UNIT

!BADTX_BAR = !i592EOP_BAR & i592DREQ0; "DREQ active at EOP

!KILL_DATA_BAR = !DAK0_BAR & !i592EOP_BAR;

end PAL2
```

292066-5

PAL 3

```
module PAL3 flag '-R3'
title 'NBi592 - PAL3  REV X024'
IC20 device 'P20L10' ;
DMA_MW_BAR                pin 1; "in
CTS_BAR                   pin 2; "in
IOWR_BAR                  pin 3; "in
DACK6_BAR                 pin 4; "in
DASTB                     pin 5; "in
DDASTB                    pin 6; "in
FLOPPY                    pin 7; "in
RESET                     pin 8; "in
W5                        pin 9; "in
W6                        pin 10; "in
DHALDA_BAR                pin 11; "in
NC2                       pin 13; "in (spare)
MASTER_BAR                pin 14; "out
WR_BAR                    pin 15; "out
EN_CMD_BAR                pin 16; "out
NC3                       pin 17; "I/O (spare)
LPBK_BAR                  pin 18; "out
WATCHDOG_BAR              pin 19; "out
QRD_OR_WR                 pin 20; "out
SA0                       pin 21; "I/O
EN_ADDR_BAR               pin 22; "out
A3                        pin 23; "out
H,L,X=1,0,.X.;
```

292066-6

Equations

```
enable SA0 = !EN_ADDR_BAR;
SA0 = 0;
enable WR_BAR = EN_ADDR_BAR;
WR_BAR = IOWR_BAR;
enable MASTER_BAR = !DACK6_BAR;
MASTER_BAR = 0;
!WATCHDOG_BAR = FLOPPY & !RESET
                                        "ARM by FLOPPY WATCHDOG
#  W5 & W6 & !RESET
                                        "ARM by 15 µs WATCHDOG
# !WATCHDOG_BAR & !DACK6_BAR & !RESET; "DROP after release
QRD_OR_WR = (!DMA_MW_BAR # !WR_BAR) & !EN_CMD_BAR;
!EN_ADDR_BAR = !DACK6_BAR & !DHALDA_BAR;
!EN_CMD_BAR = !DASTB & !DDASTB & !DACK6_BAR;
enable A3 = EN_ADDR_BAR;
A3 = SA0;
LPBK_BAR = !CTS_BAR;
end PAL3
```

292066-7

PAL 4

```
module PAL4
title 'MBN592 - PAL4 REV X023'
IC21 device 'P20L10' ;
AEN                    pin 1;   "in
SA9                    pin 2;   "in
SA8                    pin 3;   "in
SA7                    pin 4;   "in
SA6                    pin 5;   "in
SA5                    pin 6;   "in
SA4                    pin 7;   "in
RESET                  pin 8;   "in
RANGE                  pin 9;   "in
EPROM_CS_BAR           pin 10;  "in
OVERFLOW_BAR           pin 11;  "in
i592INT                pin 13;  "in
EN_DATA_BAR            pin 14;  "out
IRQ10                  pin 15;  "out
BOARD_CS_BAR           pin 16;  "out
EN_CMD_BAR             pin 17;  "in
KILL_DATA_BAR          pin 18;  "in
HLDA37                 pin 19;  "out
LD_LIMIT_BAR           pin 20;  "in
EN_ADDR_BAR            pin 21;  "in
LIMIT_LATCH_BAR        pin 22;  "out
BHE_BAR                pin 23;
H,L,X=1,0,.X.;
QADD=[X,X,SA9,SA8,SA7,SA6,SA5,SA4];
```

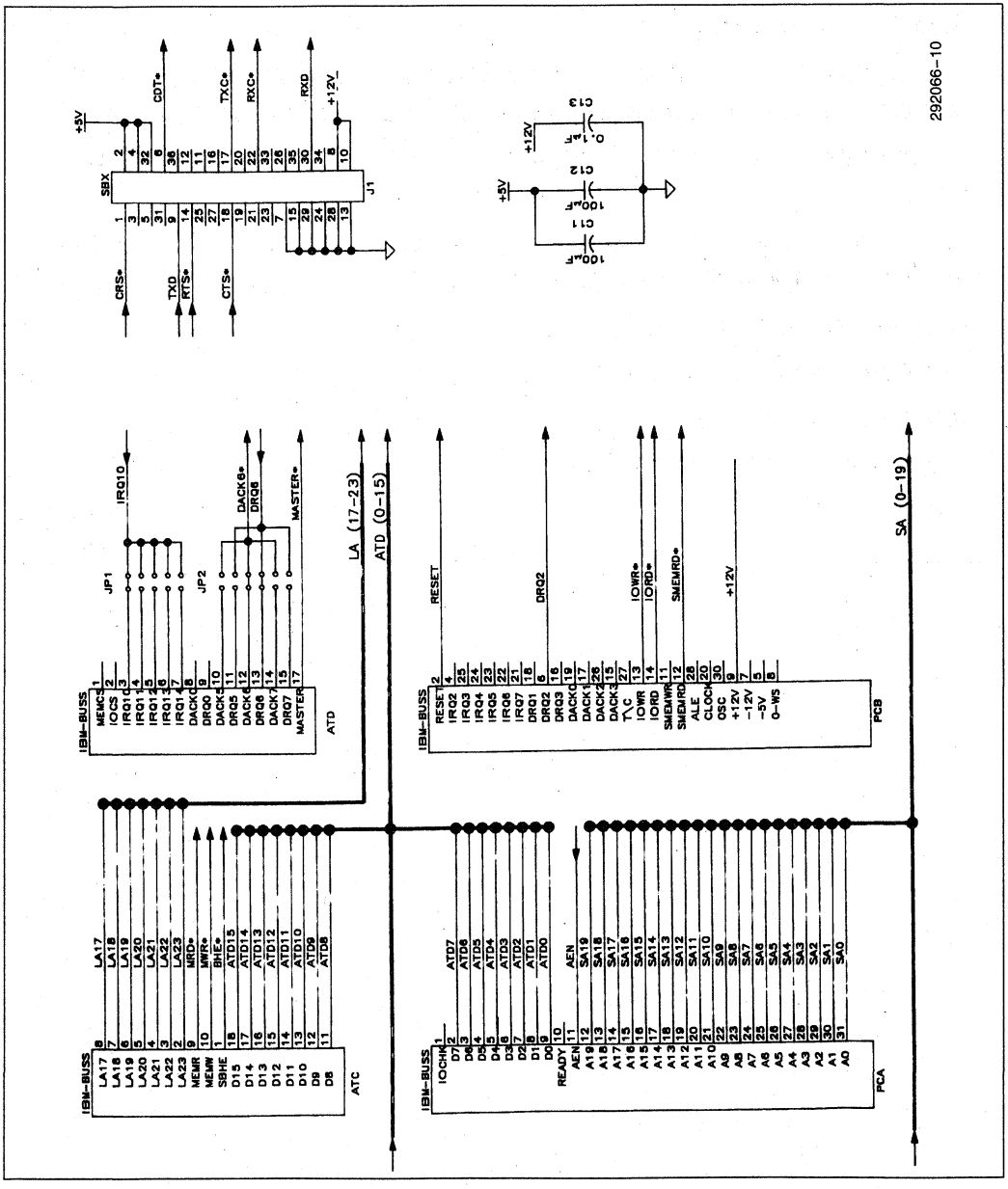
292066-8

Equations

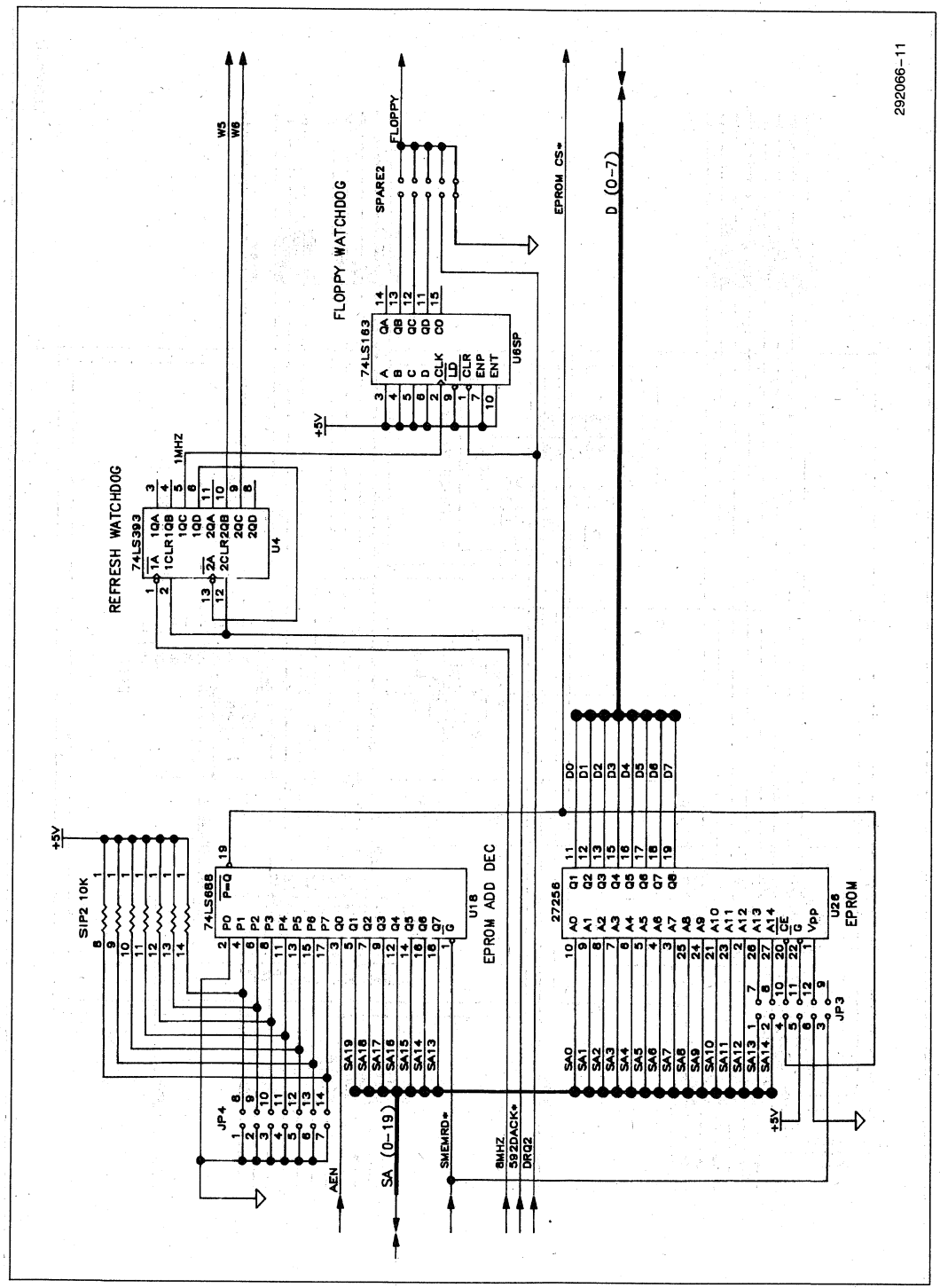
```
enable EN_CMD_BAR = 0;

enable LD_LIMIT_BAR = 0;

enable EN_ADDR_BAR  = 0;

enable KILL_DATA_BAR= 0;

!BOARD_CS_BAR = (!AEN & SA9 & SA8 & !SA7 & !SA6 & !SA5 & EN_ADDR_BAR & !SA4 &
!RANGE) # ( !AEN & SA9 & SA8 & !SA7 & !SA6 & !SA5 & EN_ADDR_BAR & SA4 & RANGE );

!LIMIT_LATCH_BAR = ( !OVERFLOW_BAR # !LIMIT_LATCH_BAR & LD_LIMIT_BAR ) &
!RESET;

!EN_DATA_BAR = !EN_ADDR_BAR & !EN_CMD_BAR & KILL_DATA_BAR #
!BOARD_CS_BAR # !EPROM_CS_BAR;

HLDA37 = !EN_ADDR_BAR;

enable BHE_BAR = !EN_ADDR_BAR;

BHE_BAR = 0;

IRQ10 = i592INT # !LIMIT_LATCH_BAR;

end PAL4
```

292066-9

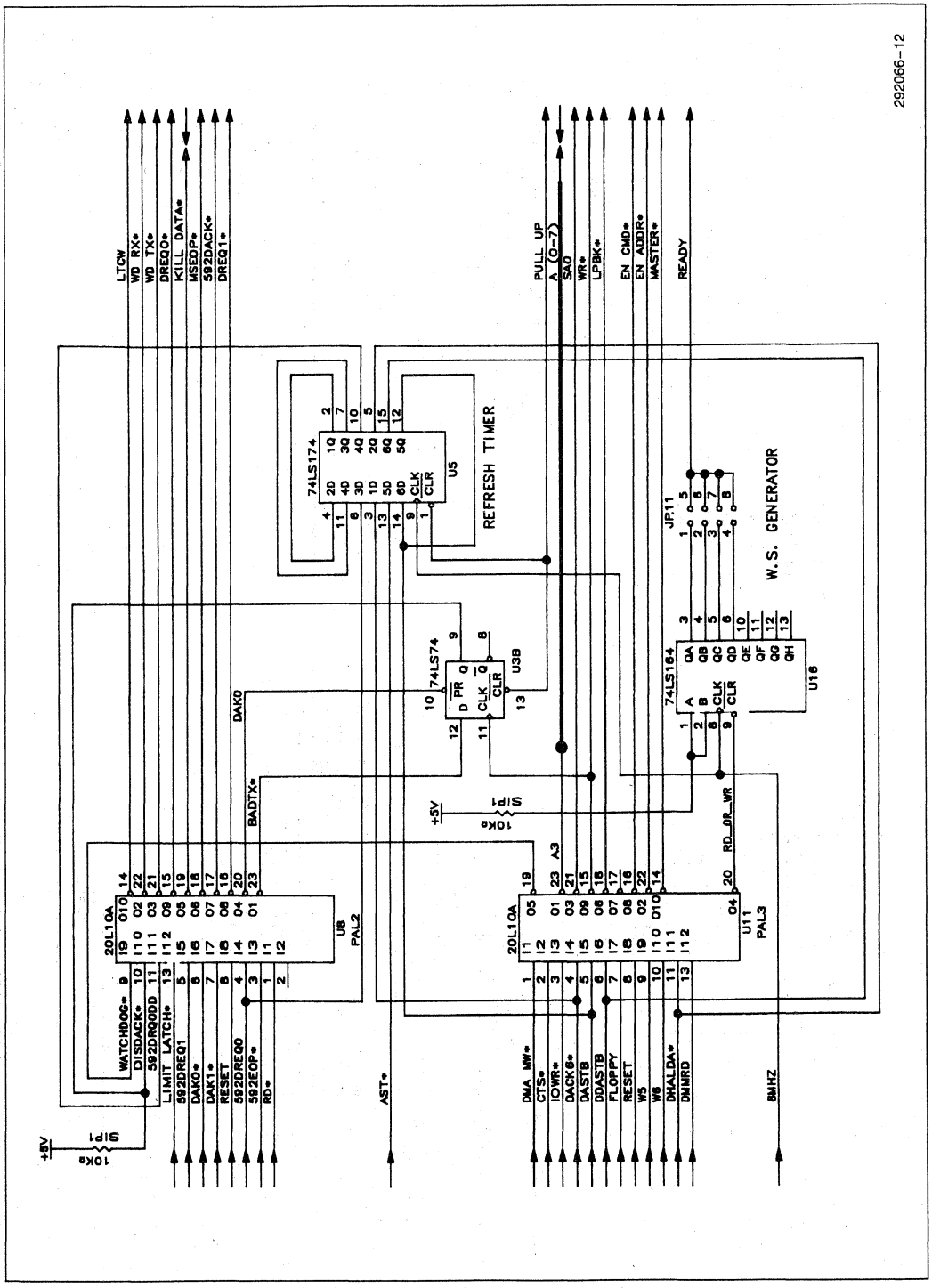
APPENDIX D
SCHEMATICS AND PARTS LIST



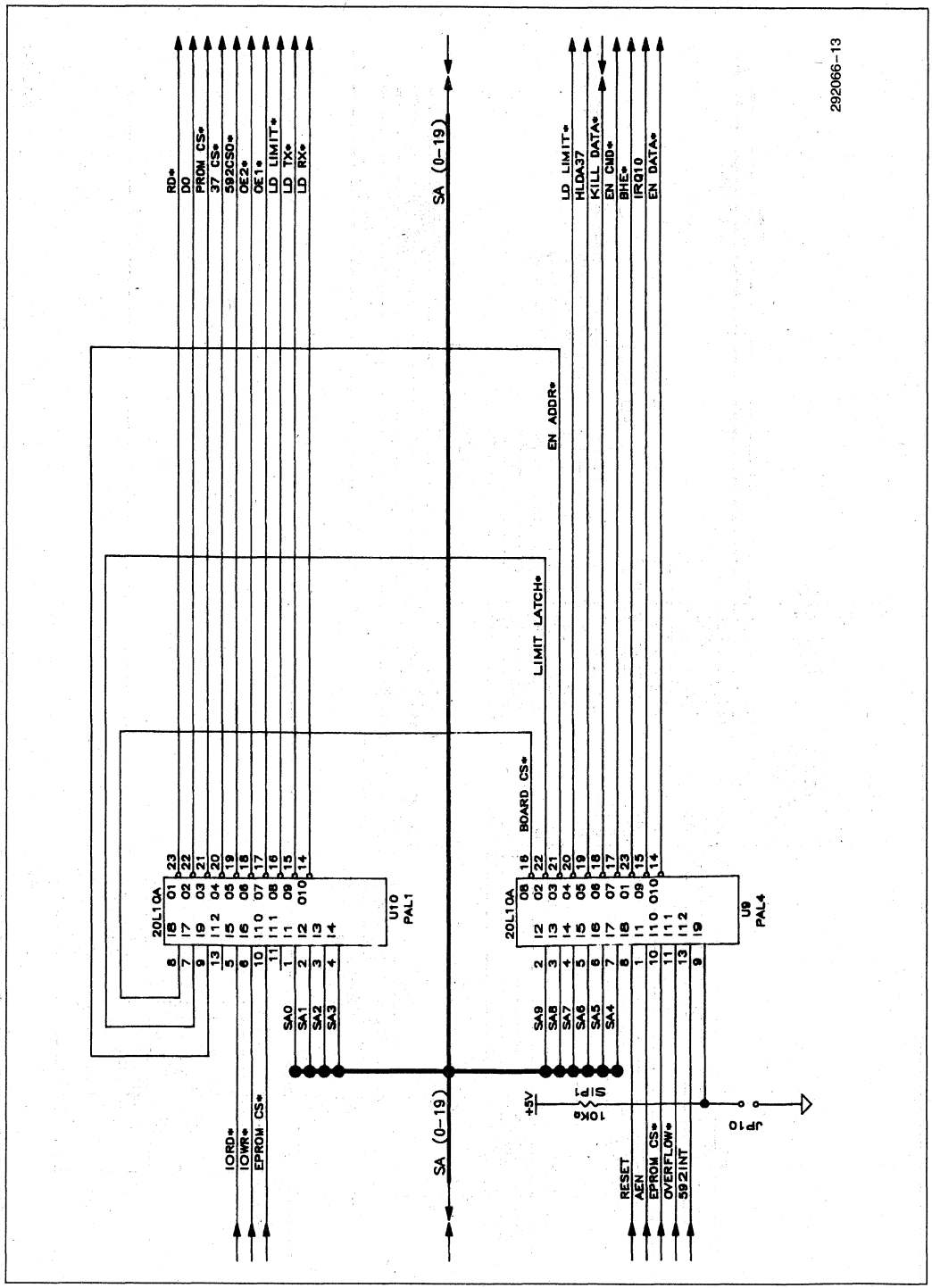
Nonbuffered Master Adapter Connectors



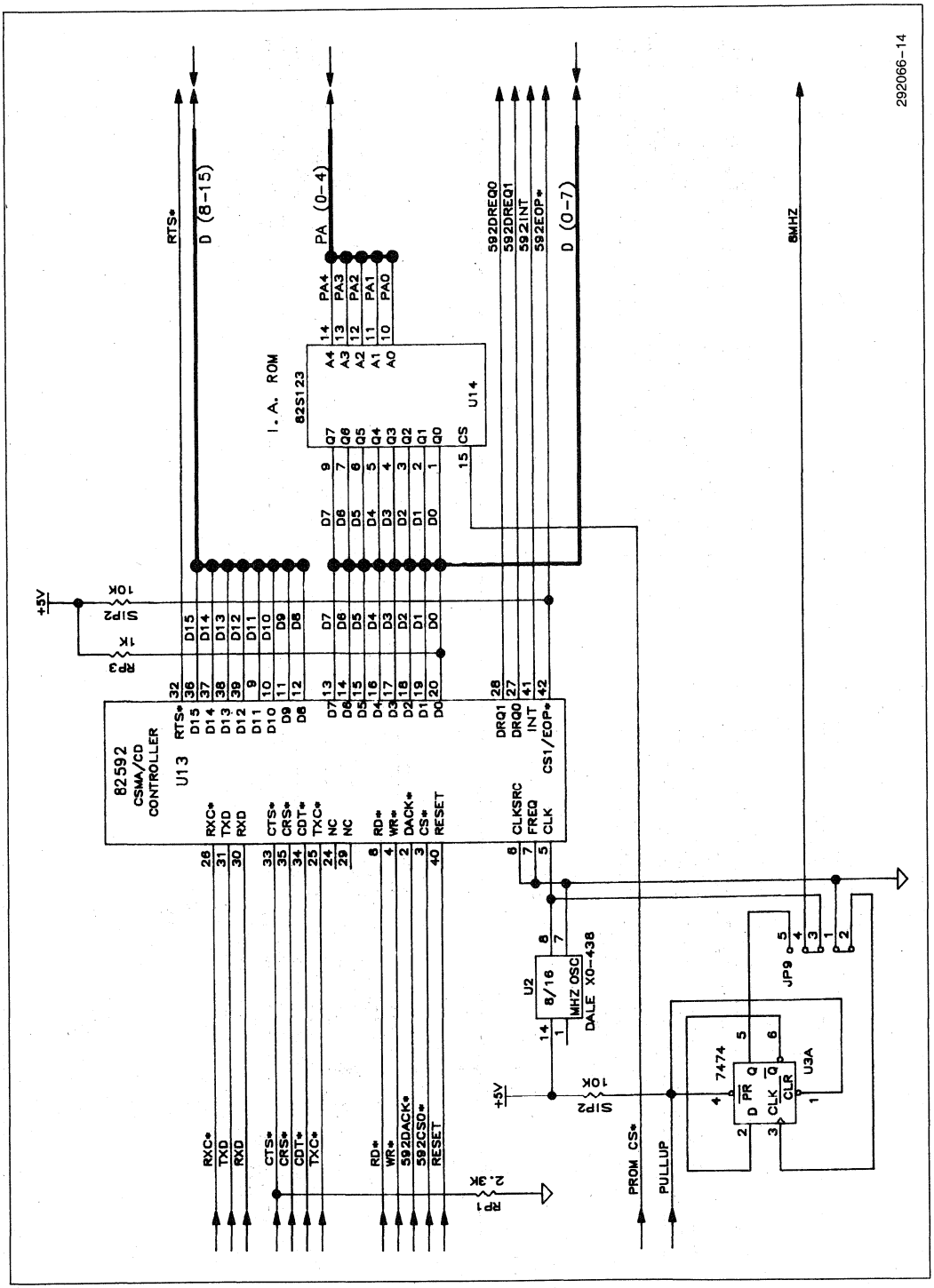
Nonbuffered Master Adapter EPROM, Watchdog Timers



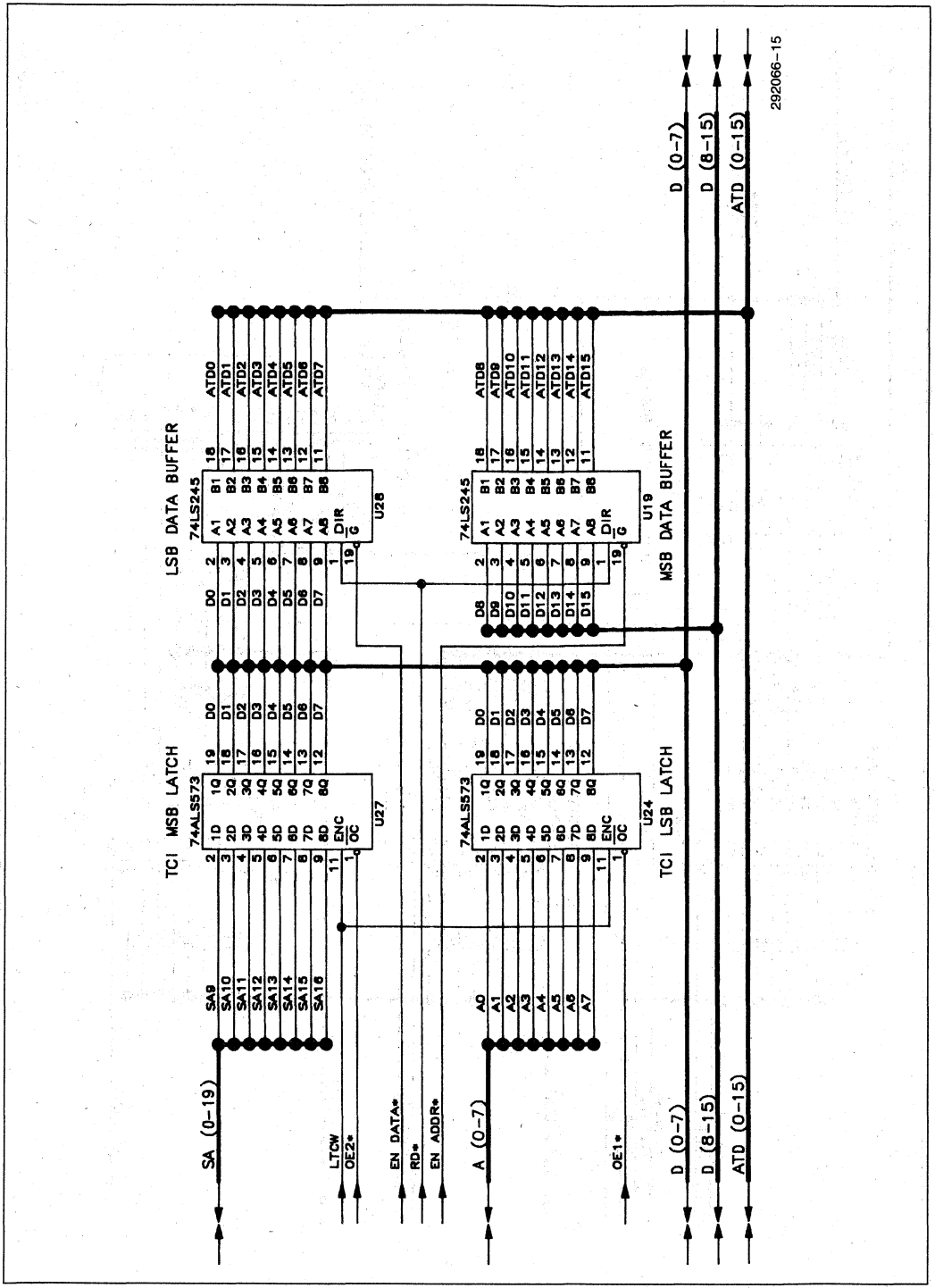
Nonbuffered Master Adapter Control-PAL2, PAL3



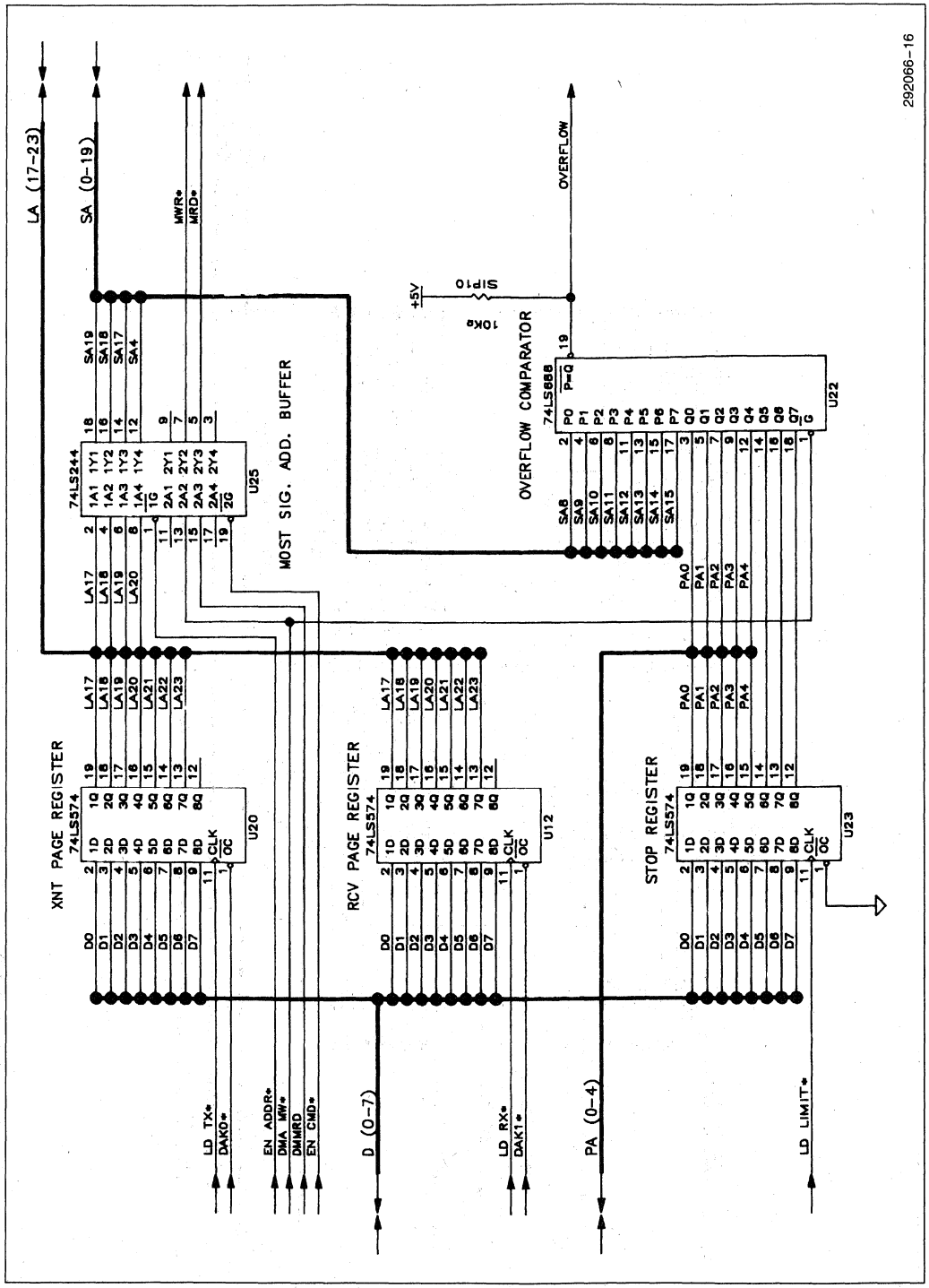
Nonbuffered Master Adapter Control—PAL1, PAL4



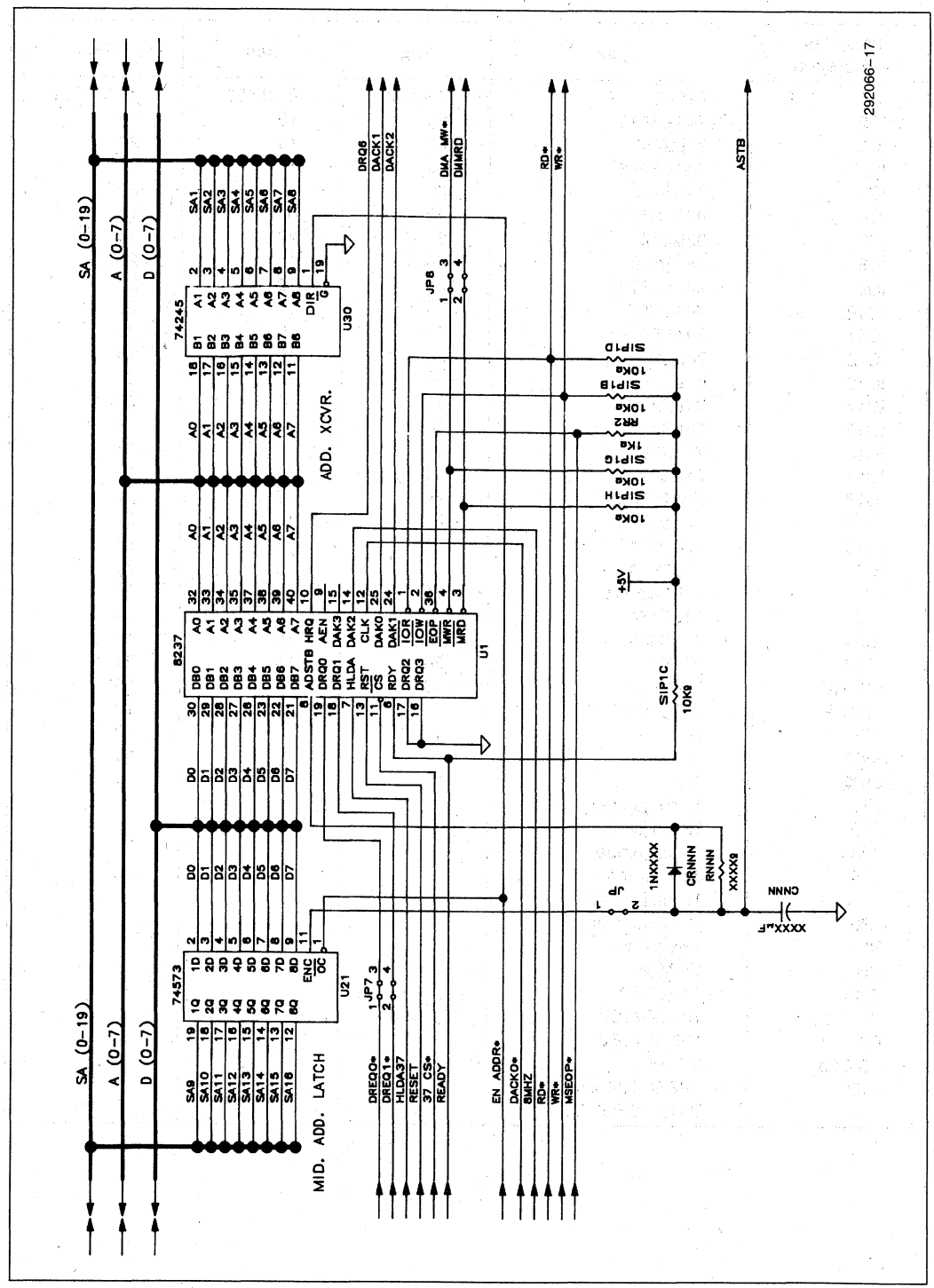
Nonbuffered Master Adapter 82592, I.A.ROM, Clk Generator



Nonbuffered Master Adapter I/O Bus Data Buffers, TCI Address Latch



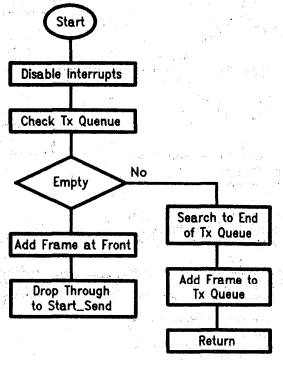
Nonbuffered Master Adapter Page Registers, Stop Register & Comparator



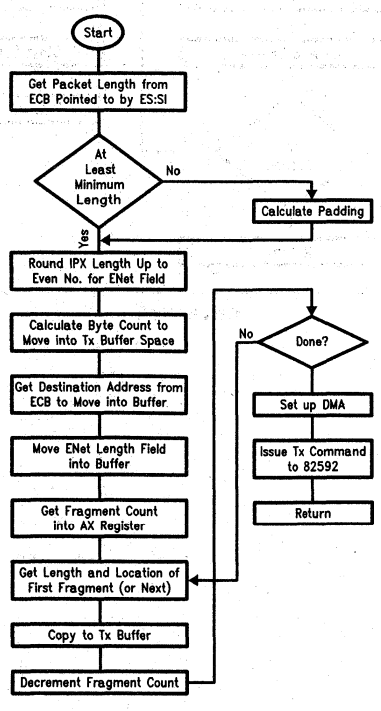
Nonbuffered Master Adapter DMA Controller

Parts List Reference	Type	V_{CC}	Gnd	Pins
IC1	82592	1,44,43	21,22,23	44 (PLCC)
IC2	74ALS573	20	10	20
IC3	74LS245	20	10	20
IC4	74ALS573	20	10	20
IC5	74LS245	20	10	20
IC6	82S123	16	8	16
IC7	82C37A	31	20	40
IC8	74LS688	20	10	20
IC9	74ALS574	20	10	20
IC10	74ALS574	20	10	20
IC11	74ALS574	20	10	20
IC12	74ALS573	20	10	20
IC13	74LS245	20	10	20
IC14	74LS244	20	10	20
IC15	74LS74	14	7	14
IC16	74LS393	14	7	14
IC17	74AS174	16	8	16
IC18	PAL20L10	24	12	24
IC19	PAL20L10	24	12	24
IC20	PAL20L10	24	12	24
IC21	PAL20L10	24	12	24
IC22	OSC 8 MHz	14	7	14
IC23	74LS164	14	7	14
IC24	74LS688	20	10	20
IC23	27256	28	14	28
SPARE1	74LS163	16	8	16
SPARE2	DIP SWITCH			10
SIP1	10K	1		10
SIP2	10K	1		10
SIP3	10K	1		10
JP1	10 pin jumper			
JP2	12 pin jumper			
JP3	10 pin jumper			
JP4	3 pin jumper			
JP5	16 pin jumper			
JP6	12 pin jumper			
RR1	13K ¼ W			
RR2	1K ¼ W			
RR3	1K ¼ W			
C1	100 μF/16V			
C2	100 μF/16V			
C3-C20	0.1 μF			
P1B	IBM CONN. B03,B29 B01,B10,B31 62			

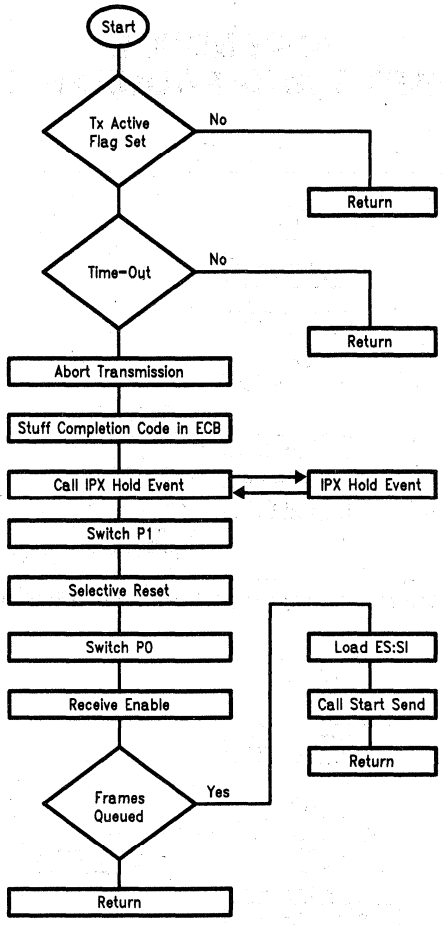
APPENDIX E
FLOWCHARTS AND PROGRAM LISTINGS



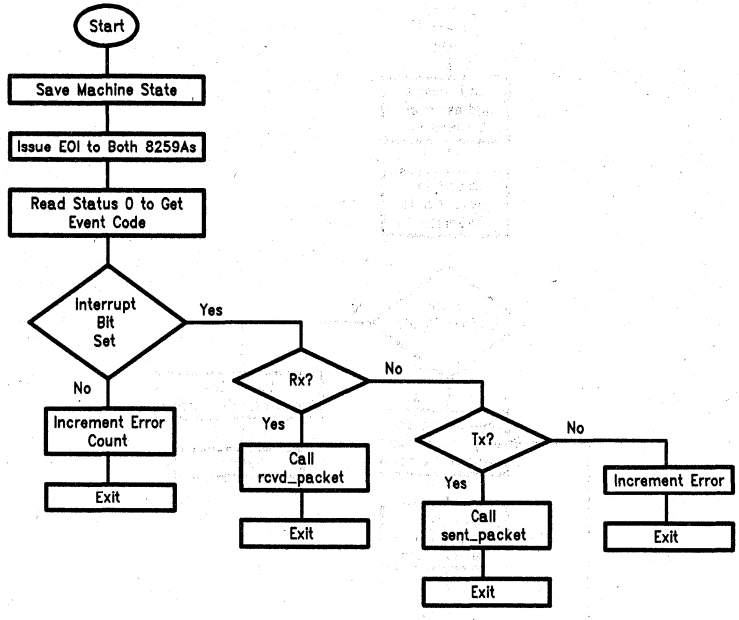
Driver Broadcast Packet-Driver Send Packet



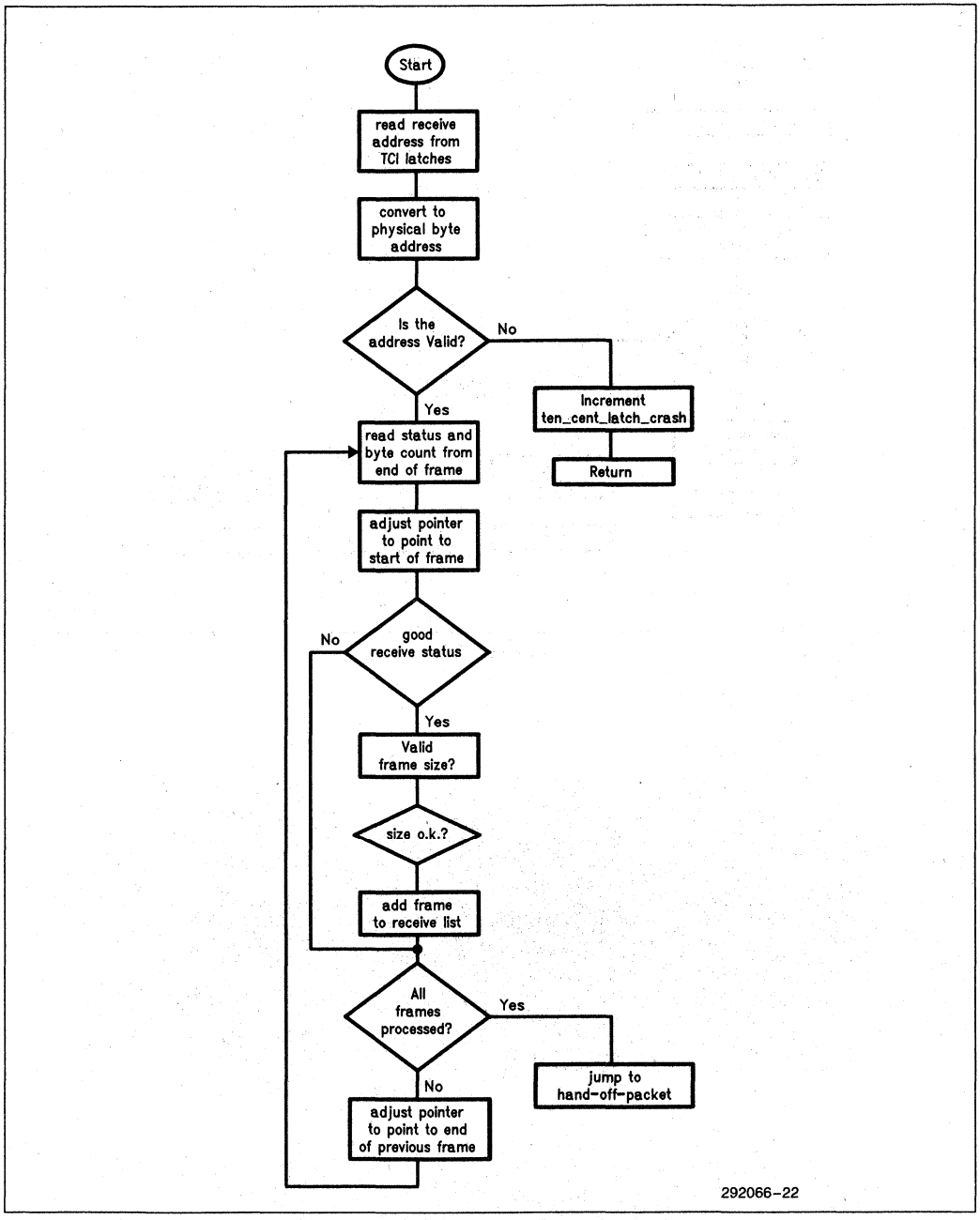
Start Send


Start
Tx Active Flag Set
No
Return
Time-Out
No
Return
Abort Transmission
Stuff Completion Code in ECB
Call IPX Hold Event
IPX Hold Event
Switch P1
Selective Reset
Switch P0
Load ES:SI
Receive Enable
Call Start Send
Return
Frames Queued
Yes
Return
292066-20

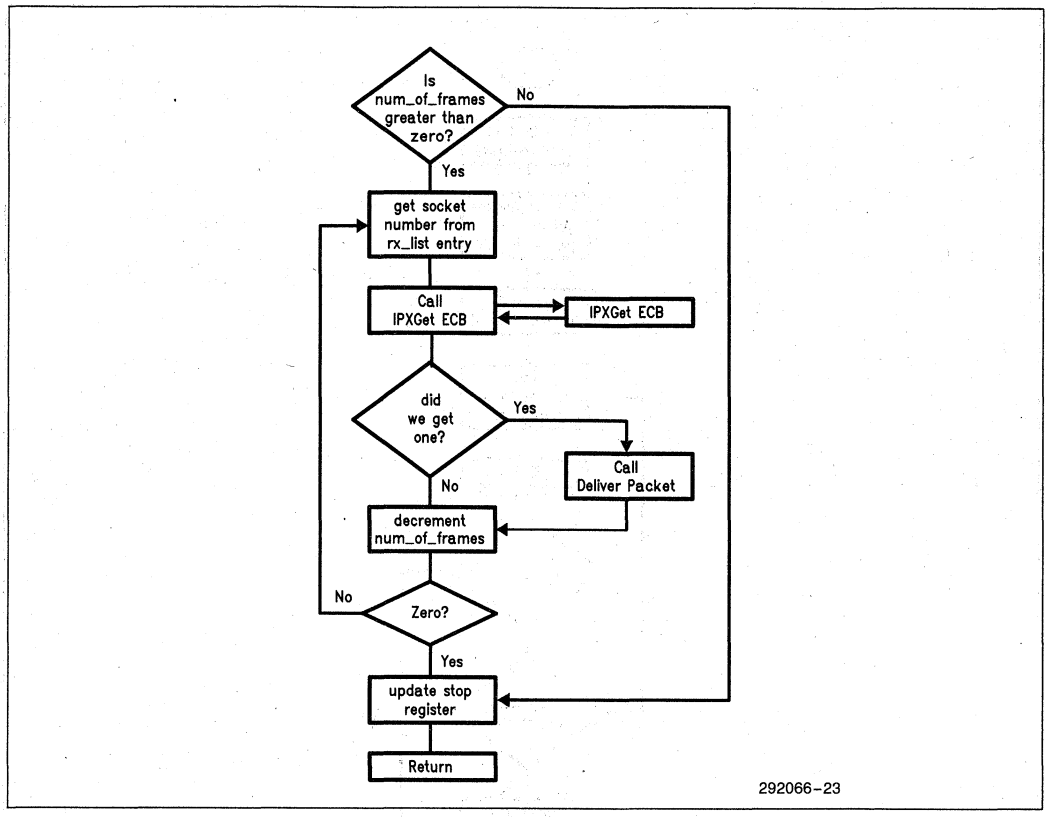
Driver Poll


Start
Save Machine State
Issue EOI to Both 8259As
Read Status 0 to Get Event Code
Interrupt Bit Set
Yes
No
Increment Error Count
Exit
Rx?
Yes
No
Call rcvd_packet
Exit
Tx?
Yes
No
Call sent_packet
Exit
Increment Error
Exit
292066-21

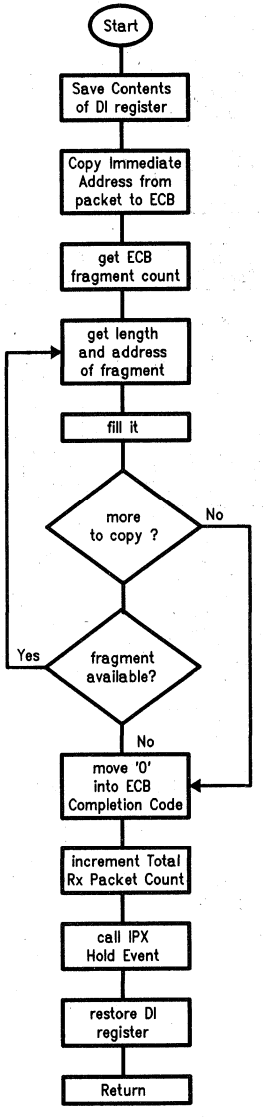
Driver ISR


Start
read receive address from TCI latches
convert to physical byte address
Is the address Valid?
No
Increment ten_cent_latch_crash
Return
Yes
read status and byte count from end of frame
adjust pointer to point to start of frame
good receive status
No
Yes
Valid frame size?
size o.k.?
add frame to receive list
All frames processed?
Yes
jump to hand-off-packet
No
adjust pointer to point to end of previous frame
292066-22

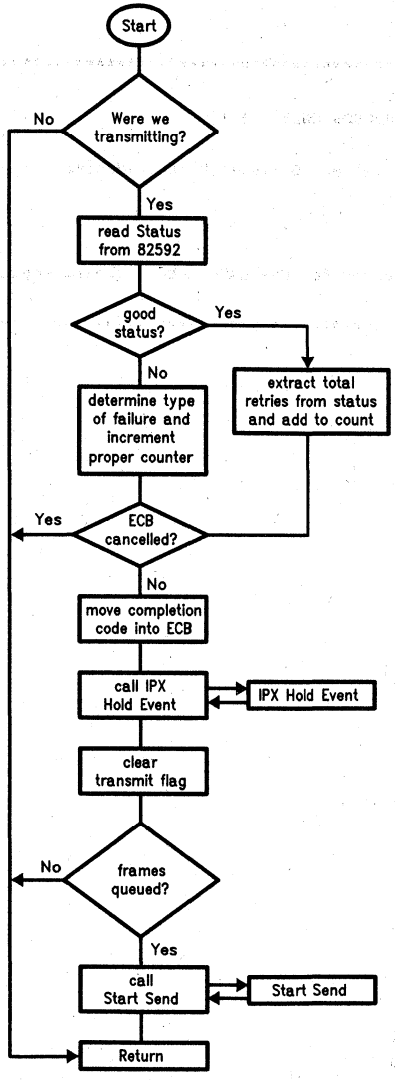
Rcvd Packet



Hand_Off_Packet



Deliver Packet


Start
Were we transmitting?
No
Yes
read Status from 82592
good status?
Yes
No
extract total retries from status and add to count
determine type of failure and increment proper counter
ECB cancelled?
Yes
No
move completion code into ECB
call IPX Hold Event
IPX Hold Event
clear transmit flag
frames queued?
No
Yes
call Start Send
Start Send
Return
292066-25

Sent Packet

```
$mod186
$nogen
;*****************************************************************
;
;  !!!!!! FOR EVALUATION PURPOSES ONLY !!!!!!
;
;  NetWare Driver for the Intel Non Buffered Master adapter.
;
;  Written by Ben L Gee.
;
;  Based on Joe Dragony's driver for the LAN-On-Motherboard Module.
;
;*****************************************************************

      name Shell_Module

false   equ 0
true  equ 1

$include(relid.inc)
$include(smacro.inc)

%set(V2_1,1)
%set(V2_0,0)

%*define(slow) local label (
      jmp short %label
%label:
)

%*define(fastcopy) local label (
      shr cx, 1
      rep movsw
      jnc %label
      movsb
%label:
)

%*define(inc32 m) (
      add word ptr %m[0], 1
      adc word ptr %m[2], 0
)
```

292066-26

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       Data Structures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ECBStructure struc
      Link                    dd 0
      ESRAddress              dd 0
      InUseFlag               db 0
      CompletionCode          db 0
      SocketNumber            dw 0
      IPXWorkspace            db 4 dup (0)
      Transmitting            db 0
      DriverWorkspace  db 11 dup (0)
      ImmediateAddress db 6 dup (0)
      FragmentCount           dw 1
      FragmentDescriptorList  db 6 dup (?)
ECBStructure ends

FragmentDescriptor struc
      FragmentAddress  dd ?
      FragmentLength          dw ?
FragmentDescriptor ends

rx_buf_structure struc
      rx_dest_addr            db 6 dup (?)
      rx_source_addr          db 6 dup (?)
      rx_physical_length      dw ?
      rx_checksum             dw ?
      rx_length               dw ?
      rx_tran_control  db ?
      rx_hdr_type             db ?
      rx_dest_net             db 4 dup (?)
      rx_dest_node            db 6 dup (?)
      rx_dest_socket          dw ?
      rx_source_net           db 4 dup (?)
      rx_source_node          db 6 dup (?)
      rx_source_socket dw ?
rx_buf_structure ends

tci_status struc
      status0  db ?
               db ?
      status1  db ?
               db ?
      bc_lo           db ?
               db ?
      bc_hi           db ?
               db ?
tci_status ends
```

292066-27

```
ipx_header_structure struc
      checksum          dw ?
      packet_length           dw ?
      transport_control       db ?
      packet_type             db ?
      destination_network     db 4 dup (?)
      destination_node db 6 dup (?)
      destination_socket      dw ?
      source_network          db 4 dup (?)
      source_node             db 6 dup (?)
      source_socket           dw ?
ipx_header_structure ends


CGroup  group Code, mombo_init

      assume cs: CGroup, ds: CGroup

Code  segment word public 'CODE'

      public DriverSendPacket
      public DriverBroadcastPacket
      public DriverOpenSocket
      public DriverCloseSocket
      public DriverPoll
      public DriverCancelRequest
      public DriverDisconnect
      public SDriverConfiguration
      public DriverISR

      public LANOptionName

      extrn IPXGetECB: NEAR
      extrn IPXReturnECB: NEAR
      extrn IPXReceivePacket: NEAR
      extrn IPXReceivePacketEnabled: NEAR
      extrn IPXHoldEvent: NEAR
      extrn IPXServiceEvents: NEAR
      extrn IPXIntervalMarker: word
      extrn MaxPhysPacketSize: word
      extrn ReadWriteCycles: byte
      extrn IPXStartCriticalSection: NEAR
      extrn IPXEndCriticalSection: NEAR
```

292066-28

```
;
;       Define Hardware Configuration
;
ConfigurationID db 'NetWareDriverLAN WS  '

SDriverConfiguration LABEL byte

                db 4 dup (0)
node_addr               db 6 dup (0)
                db 0
node_addr_type          db 0    ; address is determined at initialization
max_data_size           dw 1024 ; largest read data request will handle (512, 1024, 2048, 4096)
lan_desc_offset         dw LANOptionName
lan_hardware_id         db %LanType
transport_time          dw 1    ; transport time
reserved_3              db 11 dup (0)
major_version           db %MajorVersion
minor_version           db %MinorVersion
flag_bits               db 0
selected_configuration db 0    ; board configuration (interrupts, IO addresses, etc.)
number_of_configs       db  10
config_pointers         dw CFG0, CFG1, CFG2, CFG3, CFG4
                dw CFG5, CFG6, CFG7, CFG8, CFG9

LANOptionName           db 'Intel '
                db 'Non Buffered Master'
                db ' (For Evaluation Only)'
                db '  V%MajorVersion.%MinorVersion'
                db ' (%VersionDate)'
                db 0, '$'


;
;         Hardware Setting table structure
;

HardwareStructure struc
        H_IOBase        dw ?
        H_IOLength      dw ?
        H_Aux1          dd ?
%if (%V2_1) then (
                        db ?
)fi
        H_RAMSegment    dw ?
        H_RAMSize       dw ?            ; unsigned
%if (%V2_1) then (
                        db ?
)fi
        H_Aux2          dd ?
        H_IRQUsedFlag   db ?
        H_IRQ           db ?
        H_Aux3          dw ?
        H_DMA0UsedFlag db ?
        H_DMA0          db ?
```

292066-29

```
        H_DMA1UsedFlag db ?
        H_DMA1         db ?
%if (%V2_1) then (
        H_Flag1        db ?
        H_Flag2        db ?
)fi
        H_Description  db ?
HardwareStructure ends


%*define(CFG(p1,p2,p3,p4,m1,m2,m3,m4,i1,i2,i3,i4,d1,d2,d3,d4,f1,f2,msg)) ( label byte
      dw %p1, %p2, %p3, %p4
      %if (%V2_1) then ( db 0 ) fi
      dw %m1
      %if (%V2_0) then ( dw %m2 * 16 ) fi
      %if (%V2_1) then ( dw %m2 ) fi
      %if (%V2_1) then ( db 0 ) fi
      dw %m3
      %if (%V2_0) then ( dw %m4 * 16 ) fi
      %if (%V2_1) then ( dw %m4 ) fi
      db %i1, %i2, %i3, %i4, %d1, %d2, %d3, %d4
      %if (%V2_1) then ( db %f1, %f2 ) fi
      %if (%p1 ne 0) then (
       db 'I/O Base = %p1'
       %if (%p3 ne 0) then (
              db ' and %p4'
       ) fi
       %if ((%m2 ne 0) or (%i1 ne 0) or (%d1 ne 0)) then (
              db ', '
       ) fi
      ) fi
      %if (%m2 ne 0) then (
       db 'RAM Base = %m1'
       %if (%m4 ne 0) then (
              db ' and %m3'
       ) fi
       %if ((%i1 ne 0) or (%d1 ne 0)) then (
              db ', '
       ) fi
      ) fi
      %if (%i1 ne 0) then (
       db 'INT = %i2'
       %if (%i3 ne 0) then (
              db ' and %i4'
       ) fi
       %if (%d1 ne 0) then (
              db ', '
       ) fi
      ) fi
```

292066-30

```
      %if (%d1 ne 0) then (
       db 'DMA = %d2'
       %if (%d3 ne 0) then (
              db ' and %d4'
       ) fi
      ) fi
      db %msg, 0
)

;        I/O          Memory  Int       DMA     flags
CFG0  %CFG(300h,16,0,0,0,0,0,0,-1,10,0,0,-1,6,0,0,0,0,'')
CFG1  %CFG(310h,16,0,0,0,0,0,0,-1,11,0,0,-1,7,0,0,0,0,'')
CFG2  %CFG(300h,16,0,0,0,0,0,0,-1,12,0,0,-1,5,0,0,0,0,'')
CFG3  %CFG(310h,16,0,0,0,0,0,0,-1,14,0,0,-1,6,0,0,0,0,'')
CFG4  %CFG(300h,16,0,0,0,0,0,0,-1,15,0,0,-1,7,0,0,0,0,'')
CFG5  %CFG(310h,16,0,0,0,0,0,0,-1,10,0,0,-1,5,0,0,0,0,'')
CFG6  %CFG(300h,16,0,0,0,0,0,0,-1,11,0,0,-1,6,0,0,0,0,'')
CFG7  %CFG(310h,16,0,0,0,0,0,0,-1,12,0,0,-1,7,0,0,0,0,'')
CFG8  %CFG(300h,16,0,0,0,0,0,0,-1,14,0,0,-1,5,0,0,0,0,'')
CFG9  %CFG(310h,16,0,0,0,0,0,0,-1,15,0,0,-1,6,0,0,0,0,'')


      even
;*********************************************************
;
;     Error Counters
;
;*********************************************************
      Public DriverDiagnosticTable, DriverDiagnosticText

DriverDiagnosticTable LABEL byte

DriverDebugCount       dw DriverDebugEnd-DriverDiagnosticTable
DriverVersion          db %MajorVersion, %MinorVersion
StatisticsVersion      db 01, 00
TotalTxPacketCount     dw 0, 0
TotalRxPacketCount     dw 0, 0
NoECBAvailableCount    dw 0
PacketTxTooBigCount    dw -1          ; not used
PacketTxTooSmallCount  dw -1          ; not used
PacketRxOverflowCount  dw 0
PacketRxTooBigCount    dw 0
PacketRxTooSmallCount  dw 0
PacketTxMiscErrorCount dw -1          ; not used
PacketRxMiscErrorCount dw 0
RetryTxCount           dw 0
ChecksumErrorCount     dw -1          ; not used
HardwareRxMismatchCount dw 0
NumberOfCustomVariables dw (DriverDiagnosticText-DriverDebugEnd1)/2

DriverDebugEnd1 LABEL byte
```

292066-31

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       Driver Specific Error counts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


false_590_int           dw 0
no_crs                  dw 0
no_cts                  dw 0
lost_rx                 dw 0
MaxCollisions           dw 0
no_590_int              dw 0
rx_buff_ovflw           dw 0
ten_cent_latch_crash    dw 0
tx_timeout              dw 0
underruns               dw 0

DriverDiagnosticText LABEL byte

      db 'False590InterruptCount', 0
      db 'LostCRSCount', 0
      db 'LostCTSCount', 0
      db 'LostOurReceiverCount', 0
      db 'MaxCollisionsCount', 0
      db 'No590InterruptCount', 0
      db 'ReceiveBufferOverflow',0
      db 'TencentLatchCrashCount', 0
      db 'TxTimeoutErrorCount', 0
      db 'UnderrunCount', 0

      db 0, 0

DriverDebugEnd LABEL word

even
;;;;;;;;;;;;;;;;;;;;
;       Equates
;;;;;;;;;;;;;;;;;;;;

CR      equ 0Dh
LF      equ 0Ah

TransmitHardwareFailure equ 0FFh
PacketUnDeliverable     equ 0FEh
PacketOverflow          equ 0FDh
ECBProcessing           equ 0FAh
TxTimeOutTicks          equ 10

;  Adapter I/O ports

Addr592                 dw 08h          ; + IOBase (I/O)
IA_PROM_port            dw 0ah          ; + IOBase (I)
AddrLatchLow            dw 0ch          ; + IOBase (I)
AddrLatchHigh           dw 0dh          ; + IOBase (I)
LimitRegister           dw 0eh          ; + IOBase (I/O)
```

292066-32

```
;  8259  definitions

InterruptControlPort          equ 020h
InterruptMaskPort             equ 021h
ExtraInterruptControlPort     equ 0A0h
ExtraInterruptMaskPort        equ 0A1h
EOI                   equ 020h

;  8237 definitions

; Command Register
RotatingPriority      equ 010h
ExtendedWrite         equ 020h
ActiveLowDREQ         equ 040h

; Mode Register
WriteTransfer         equ 00000100b
ReadTransfer          equ 00001000b
AutoInitialization    equ 00010000b
DemandMode            equ 00000000b
CascadeMode           equ 11000000b

              even
DMAcmdstat            dw 001h         ; + IOBase (I/O)
DMAsnglmsk            dw 005h         ; + IOBase (O)
DMAmode               dw 007h         ; + IOBase (O)
DMAff                 dw 009h         ; + IOBase (O)

XmtDMApage            dw 00ah         ; + IOBase (O)
XmtDMAaddr            dw 000h         ; + IOBase (I/O)
XmtDMAwdcount         dw 002h         ; + IOBase (I/O)

RcvDMApage            dw 00ch         ; + IOBase (O)
RcvDMAaddr            dw 004h         ; + IOBase (I/O)
RcvDMAwdcount         dw 006h         ; + IOBase (I/O)

MasterDMAcmdstat      equ 0D0h
MasterDMAsnglmsk      equ 0D4h
MasterDMAmode         equ 0D6h
MasterDMAff           equ 0D8h

MasterDMApage         dw ?
MasterDMAaddr         dw 0c0h
MasterDMAwdcount      dw 0c2h

XmtDMAtx              db DemandMode + AutoInitialization + ReadTransfer
XmtDMAmsk             db 4
XmtDMAunmsk           db 0

RcvDMArx              db DemandMode + AutoInitialization + WriteTransfer + 1
RcvDMAmsk             db 4 + 1
RcvDMAunmsk           db 0 + 1
```

292066-33

```
MasterDMAmodevalue      db CascadeMode
MasterDMAmsk            db 4
MasterDMAunmsk          db 0

; 82592 Commands

C_NOP           equ 00h
C_SWP1          equ 10h
C_SELRST        equ 0Fh
C_SWP0          equ 01h
C_IASET         equ 01h
C_CONFIG        equ 02h
C_MCSET         equ 03h
C_TX    equ 04h
C_TDR           equ 05h
C_DUMP          equ 16h
C_DIAG          equ 07h
C_RXENB         equ 18h
C_ALTBUF        equ 09h
C_RXDISB        equ 1Ah
C_STPRX         equ 1Bh
C_RETX          equ 0Ch
C_ABORT         equ 0Dh
C_RST           equ 0Eh
C_RLSPTR        equ 0Fh
C_FIXPTR        equ 1Fh
C_INTACK        equ 80h

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      even

gp_buf_size     equ 600         ; in words
max_rx_buf_size equ 2200        ; in words
gp_buf          dw gp_buf_size + max_rx_buf_size dup (?)
gp_buf_pointer dd ?
gp_buf_start    dw ?    ; A1-A16 of General Purpose Buffer EA
gp_buf_page     dw ?    ; A17-A23 of General Purpose Buffer EA
tx_byte_cnt     dw ?    ; IPX packet length plus header length
rx_buf_start    dw ?    ; A1-A16 of Receive Buffer EA
rx_buf_page     dw ?    ; A17-A23 of Receive Buffer EA
rx_buf_head     dw ?    ; current rx head, buffer has been flushed to here
rx_buf_tail     dw ?    ; value read from 10 cent latches
rx_buf_length   dw ?    ; word size of rx_buf
rx_buf_segment dw ?     ; calculated at init for use by IPXReceivePacket
rx_buf_first    dw ?    ; offset from rx_buf_segment of start of rx_buf
rx_buf_limit    dw ?    ; offset from rx_buf_segment of limit of rx_buf
rx_buf_size     dw ?    ; byte size of rx_buf
Logical2Physical dw ?  ; add this to convert from rx_buf_segment to rx_buf_page
rx_list         dw 30 dup (?)
num_of_frames   dw ?
```

292066-34

```
padding         dw ?
SendList        dd 0    ; points to list of ECBs to be sent
tx_start_time   dw 0
tx_active_flag db 0

;*************************************************************************
;
;  Interrupt Procedure
;
;*************************************************************************

even
DriverISR PROC far

      pusha
      push ds
      push es
      mov ax, cs
      mov ds, ax                    ; DS points to C/DGroup
      mov es, ax                    ; ES also

      mov al, EOI
      out InterruptControlPort, al
      out ExtraInterruptControlPort, al

      mov dx, Addr592
      mov al, 0
      out dx, al                    ; set status reg to point to reg 0
      %slow
      in al, dx
      test al, 80h
      %ifz
          inc no_590_int
      %else
          %do
       and al, NOT 20h          ; ignore the EXEC bit
       mov ah, al               ; save the status in AH
       cmp ah, 0D8h             ; did I receive a frame?
       %ife
           call RcvdPacket
       %else
           cmp ah, 84h          ; did I finish a transmit?
           %ife
              call SentPacket
           %else
              cmp ah, 8ch    ; did I finish a retransmit?
              %ife
                  call SentPacket
              %else
                  inc false_590_int  ; unwanted interrupt
              %endif
           %endif
       %endif
```

292066-35

```
        push cs
        pop ds
        cmp tx_active_flag, false
        %ife
            ; verify that our receiver is still going.
            mov dx, Addr592
            mov al, 60h         ; point to status byte 3
            out dx, al
            %slow
            in al, dx
            test al, 60h
            %ifz
                inc lost_rx
                mov al, C_RXENB
                mov dx, Addr592
                out dx, al
            %endif
        %endif

        mov dx, Addr592
        mov al, C_INTACK
        out dx, al              ; issue interrupt acknowledge to the 590

        %slow
        xor al, al
        out dx, al              ; set status reg to point to reg 0
        %slow
        in al, dx
        test al, 80h
          %whilenz
      %endif

      call IPXServiceEvents

      pop es
      pop ds
      popa
      iret

DriverISR endp

even
RcvdPacket proc near

      ; When the address bytes are being read it is possible that
      ; another frame could come in and cause a coherency problem
      ; with the ten-cent latches.  I am dealing with this
      ; possibility by reading AddrLatchHigh twice and making
      ; sure the values match.  If they don't the read is redone.
      cli
      mov dx, AddrLatchHigh   ; read high address byte of last frame received
      in al, dx
```

292066-36

```
%do
    mov bh, al              ; save it in bh
    mov dx, AddrLatchLow    ; read low address byte of last frame received
    in al, dx
    mov bl, al
    ; Read AddrLatchHigh again to make sure it hasn't changed.......
    mov dx, AddrLatchHigh   ; read high address byte again
    in al, dx
    cmp al, bh
%whilene

shl bx, 1           ; convert to byte address
sub bx, Logical2Physical ; bx - magic = physical - (physical-logical)
mov rx_buf_tail, bx     ; logical

mov si, bx              ; this is the last location containing rx data
call NormalizePointer   ; normalize si
cmp bx, si              ; was it already a valid pointer ?
%ifne                   ; if not, big trouble...
    inc ten_cent_latch_crash
%else1
    %do
  mov es, rx_buf_segment
  mov ch, es: [si]      ; get bc_hi
  sub si, 2
  call NormalizePointer
  mov cl, es: [si]      ; get bc_lo
  sub si, 2
  call NormalizePointer
  mov ah, es: [si]      ; get status1
  sub si, 2
  call NormalizePointer
  mov al, es: [si]      ; get status0

  ; cx has actual number of bytes read
  dec cx                ; toss byte count & status
  and cl, 0feh          ; round up
  sub si, cx            ; si points to first location of frame
  call NormalizePointer
  mov bx, si            ; save in bx

  test ah, 20h          ; test for good receive
  %ifz                  ; bad receive
      inc PacketRxMiscErrorCount
      jmp short SkipThisFrame
  %endif                ; good receive

  sub cx, 14            ; sub length of 802.3 header
  cmp cx, 1024 + 64
  %ifa
      inc PacketRxTooBigCount
      jmp short SkipThisFrame
  %endif
```

292066-37

```
cmp cx, 30
%ifb
    inc PacketRxTooSmallCount
    jmp short SkipThisFrame
%endif

lea si, [bx].rx_length
call NormalizePointer
mov ax, es: [si]        ; get IPX length
xchg al, ah
inc ax
and al, 0feh
xchg al, ah
lea si, [bx].rx_physical_length
call NormalizePointer
cmp ax, es: [si]        ; same as 802.3 length ?
%ifne
    inc HardwareRxMismatchCount
    jmp short SkipThisFrame
%endif
xchg al, ah
cmp ax, 60 - 14         ; at least min length minus header
%ifbe
    mov ax, 60 - 14     ; no, round up
%endif
cmp ax, cx              ; match physical length
%ifne
    inc HardwareRxMismatchCount
%else
    mov di, num_of_frames
    add di, di
    mov rx_list[di], bx ; first location of ethernet frame
    inc num_of_frames
    cmp num_of_frames, length rx_list
    je hand_off_packet
%endif
  SkipThisFrame:
mov si, bx
cmp rx_buf_head, si     ; first frame of sequence ?
je hand_off_packet      ; yes, go process list
sub si, 2
call NormalizePointer
  %forever                      ; no, continue processing frames
hand_off_packet:
%endif

cli
mov di, num_of_frames
add di, di
%ifnz
    %do
 sub di, 2
 mov si, rx_list[di]
 lea si, [si].rx_dest_socket
```

292066-38

```
        call NormalizePointer
        mov es, rx_buf_segment
        mov ax, es: [si]
        call IPXGetECB
        %ifnz
            call DeliverPacket
        %endif
        dec num_of_frames
          %whilenz
      %endif

      ; update the limit register
      mov dx, LimitRegister
      in al, dx
      test al, 1
      %ifnz
          inc rx_buff_ovflw          ; just for the record
      %endif
      mov si, rx_buf_tail
      sub si, 256
      call NormalizePointer
      mov ax, si                     ; move new limit value to ax
      add ax, Logical2Physical       ; convert to physical address
      mov al, ah                     ; only need bits A15..A8
      out dx, al                     ; store it in the limit register

      mov si, rx_buf_tail
      add si, 2
      call NormalizePointer
      mov rx_buf_head, si            ; set rx_buf_head to new value for next receive
      ret

RcvdPacket endp

even
DeliverPacket proc near

      push di
      mov di, rx_list[di]
      mov bp, si                     ; copy ecb offset to bp
      xchg si, di                    ; es:di = ecb
      mov ds, rx_buf_segment         ; ds:si = packet
      assume ds: nothing
      add si, 6                      ; skip destination address
      lea di, [di].ImmediateAddress
      call NormalizePointer
      movsw
      call NormalizePointer
      movsw
      call NormalizePointer
      movsw
      add si, 4                      ; skip etype and checksum
      call NormalizePointer
      mov dx, ds: [si]        ; get length from IPX header
```

292066-39

```
xchg dh, dl
sub si, 2                       ; point to checksum
call NormalizePointer
mov di, bp

; disburse the packet

; ds:si = packet data source
; es:bp = ECB
; ax     = fragment count
; dx     = amount of data in source
; bx     = pointer to the FragmentDescriptorList
; cx     = size of this fragment

mov cx, es: [bp].FragmentCount
lea bx, [bp].FragmentDescriptorList
%do
    push es
    push cx
    mov cx, es: [bx].FragmentLength
    les di, es: [bx].FragmentAddress

    mov ax, rx_buf_limit
    sub ax, si
    cmp ax, cx
    %ifb
  xchg cx, ax         ; cx = amount to copy
  sub ax, cx          ; ax = amount not copied

  cmp dx, cx
  %ifb
      mov cx, dx
  %endif
  sub dx, cx

  %fastcopy
  mov si, rx_buf_first
  mov cx, ax
    %endif

    cmp dx, cx
    %ifb
  mov cx, dx
    %endif
    sub dx, cx

    %fastcopy
    pop cx
    pop es
    add bx, 6
%loop
```

292066-40

```
        ; deliver the ECB
        mov si, bp
        mov es: [si].CompletionCode, 0
        push cs
        pop ds
        assume ds: CGroup
        %inc32 TotalRxPacketCount
        call IPXHoldEvent
        pop di
        ret

DeliverPacket endp


even
;       input:
;         si = pointer into rx_buf
;
;       output:
;         si = valid pointer into rx_buf
;         no other registers modified
;
NormalizePointer proc near

        cmp si, cs: rx_buf_first
        %ifae
            cmp si, cs: rx_buf_limit
            %ifb
          ret
            %endif
            sub si, cs: rx_buf_size
            ret
        %endif
        add si, cs: rx_buf_size
        ret

NormalizePointer endp

even
SentPacket proc near

        cli
        cmp tx_active_flag, true
        %ife
            in al, dx
            mov ah, al
            %slow
            in al, dx
            xchg ah, al
            test ah, 20h
            %ifz
```

292066-41

```
        test al, 20h            ; Max collisions??
        %ifnz
            inc MaxCollisions
        %endif
        test ah, 01h            ; Tx underrun??
        %ifnz
            inc underruns
        %endif
        test ah, 02h            ; did we lose clear to send??
        %ifnz
            inc no_cts
        %endif
        test ah, 04h            ; did we lose carrier sense??
        %ifnz
            inc no_crs
        %endif
        mov al, TransmitHardwareFailure
          %else
        ; extract the total number of retries from the status
        and ax, 0Fh             ; register and add to retry count
        add RetryTxCount, ax
        xor ax, ax              ; status = 0, good transmit
          %endif

          mov cx, word ptr SendList[2]
          %ifcxnz
        mov es, cx                      ; segment of next SCB in list
        mov si, word ptr SendList[0]  ; offset of next SCB in list
        cmp es: [si].Transmitting, true ; if not canceled
        %ife
            mov es: [si].CompletionCode, al
            mov ax, es: word ptr [si].Link[0]
            mov word ptr SendList[0], ax
            mov ax, es: word ptr [si].Link[2]
            mov word ptr SendList[2], ax
            ; finish the transmit
            mov es: [si].InUseFlag, 0
            call IPXHoldEvent
        %endif
          %endif

          mov tx_active_flag, false
          mov cx, word ptr SendList[2]
          %ifcxnz
        mov es, cx                      ; segment of next SCB in list
        mov si, word ptr SendList[0]  ; offset of next SCB in list
        call StartSend
          %endif
      %endif
      ret

SentPacket endp

;
```

292066-42

```
;  Driver Send Packet
;  Driver Broadcast Packet
;
;  Assumes
;     es: SI points to a fully prepared Event Control Block
;     DS = CS
;     Interrupts are DISABLED but may be reenabled temporarily if necessary
;
;     don't need to save any registers
;
even
DriverBroadcastPacket:
DriverSendPacket PROC NEAR

      mov es: [si].Transmitting, false

      mov cx, word ptr SendList[2]
      %ifcxnz
          ; search to the end of the list, and add there.
          mov di, word ptr SendList[0]
          %do
        mov ds, cx
        mov cx, ds: word ptr [di].Link[2]
        jcxz AddListEndFound
        mov di, ds: word ptr [di].Link[0]
          %forever

      AddListEndFound:
          mov es: word ptr [si].Link[0], cx  ; move null pointer to newest SCB's
          mov es: word ptr [si].Link[2], cx  ; link field
          mov ds: word ptr [di].Link[0], si
          mov ds: word ptr [di].Link[2], es
          mov ax, cs
          mov ds, ax                 ; set ds back to entry condition
          ret
      %endif

      mov es: word ptr[si].Link[0], cx
      mov es: word ptr[si].Link[2], cx
      mov word ptr SendList[0], si
      mov word ptr SendList[2], es
      ; drop through to Start Send

DriverSendPacket endp

;
;   Start Send
;
;   assumes:
;     es: SI   points to the ECB to be sent.
;     interrupts are disabled
;
```

292066-43

```
even
StartSend PROC NEAR

      cld
      mov es: [si].Transmitting, true
      push ds          ; save ds for future use
      ; get IPX packet length out of the first fragment (IPX header)
      lds bx, es: dword ptr [si].FragmentDescriptorList
      mov ax, ds: [bx].packet_length
      pop ds                    ; restore ds to CGROUP
      push ax          ; save length for later use in 590 length field
      xchg al, ah               ; byte swap for 592 length field calculation
      add ax, 18                ; add in the overhead bytes DA, SA, CRC, length

      mov padding, 0
      cmp ax, 64
      %ifb
          mov padding, 64       ; minimum length frame
          sub padding, ax       ; pad length
          mov ax, 64
      %endif
      sub ax, 10                ; SA and CRC are done automatically
      inc ax
      and al, 0FEh              ; frame must be even
      mov tx_byte_cnt, ax
      les di, gp_buf_pointer
      ; move the byte count into the transmit buffer
      stosw
      ; move the destination address from the tx ECB to the tx buffer
      mov bx, si
      lea si, [bx].ImmediateAddress
      mov ds, word ptr SendList[2]
      movsw
      movsw
      movsw
      mov ax, cs                ; get back to the code (Dgroup) section
      mov ds, ax

      ; now the 590 length field
      pop ax
      xchg ah, al
      inc ax
      and al, 0FEh              ; make sure E-Net length field is even
      xchg ah, al
      stosw
      lds si, SendList
      mov ax, ds: [si].FragmentCount
      lea bx, [si].FragmentDescriptorList
      %do
          push ds               ; save the segment
          mov cx, ds: [bx].FragmentLength
          lds si, ds: [bx].FragmentAddress
          %fastcopy
          pop ds                ; get the segment back
```

292066-44

```
        add bx, 6
        dec ax
    %whilenz
    ; start transmitting
    mov cx, cs
    mov ds, cx
    ; add any required padding
    mov cx, 4               ; make sure frame ends with a NOP
    add cx, padding
    shr cx, 1
    rep stosw
    mov tx_active_flag, true

    mov dx, DMAff
    out dx, al              ; data is don't care

    mov dx, XmtDMAaddr
    mov al, byte ptr gp_buf_start[0]
    out dx, al
    mov al, byte ptr gp_buf_start[1]
    out dx, al

    mov ax, gp_buf_page
    mov dx, XmtDMApage      ; DMA page value
    out dx, al

    mov al, XmtDMAtx ; setup channel 1 for tx mode
    mov dx, DMAmode
    out dx, al

    mov ax, tx_byte_cnt
    inc ax                  ; convert to word value and account for odd
    shr ax, 1               ; byte DMA transfer
    mov dx, XmtDMAwdcount
    out dx, al
    %slow
    mov al, ah
    out dx, al

    mov al, XmtDMAunmsk
    mov dx, DMAsnglmsk
    out dx, al

    mov dx, Addr592
    mov al, C_TX
    out dx, al

    mov ax, IPXIntervalMarker
    mov tx_start_time, ax
    %inc32   TotalTxPacketCount

    ret

StartSend endp
```

292066-45

```
DriverOpenSocket:
DriverDisconnect:
      ret

;***************************************************************************
;
;     Driverpoll
;
;     Poll the driver to see if there is anything to do
;
;     Is there a transmit timeout? If so, abort transmission and return
;     ECB with bad completion code. Check to see if frames are queued.
;     If they are set up es: SI and call DriverSendPacket.
;
;***************************************************************************

even
DriverPoll PROC NEAR

      cmp tx_active_flag, true
      %ife
          mov dx, IPXIntervalMarker
          sub dx, tx_start_time
          cmp dx, TxTimeOutTicks
          %ifa
        ; This transmit is taking too long so let's terminate it now
        mov tx_active_flag, false

        ; Issue an abort to the 82592
        mov dx, Addr592
        mov al, C_ABORT        ; abort transmit
        out dx, al

        inc tx_timeout
        mov cx, word ptr SendList[2]
        %ifcxnz
            mov es, cx                    ; segment of next SCB in list
            mov si, word ptr SendList[0]    ; offset of next SCB in list
            cmp es: [si].Transmitting, true ; if not canceled
            %ife
                mov es: [si].CompletionCode, TransmitHardwareFailure ; stuff completion code of
                mov ax, es: word ptr [si].Link[0]                    ; a failed tx
                mov word ptr SendList[0], ax
                mov ax, es: word ptr [si].Link[2]
                mov word ptr SendList[2], ax

                ; Finish the transmit
                mov es: [si].InUseFlag, 0
                call IPXHoldEvent
            %endif
        %endif
```

292066-46

```
        ; make sure that execution unit didn't lock up because of abort errata
        mov dx, Addr592
        mov al, C_SWP1
        out dx, al
        mov al, C_SELRST
        %slow
        out dx, al
        mov al, C_SWP0
        %slow
        out dx, al
        mov al, C_RXENB
        %slow
        out dx, al

        ; See if any frames are queued
        mov cx, word ptr SendList[2]
        %ifcxnz
            mov es, cx                  ; segment of next SCB in list
            mov si, word ptr SendList[0] ; offset of next SCB in list
            call StartSend
        %endif
          %endif
      %endif
      ret

DriverPoll endp

;
;   Driver Cancel Request
;
;   Assumes on entry:
;     es: SI is pointer to ECB we want to cancel
;     DS is setup
;     Interrupts are DISABLED
;
;   Assumes any registers may be destroyed.
;
;   Returns completion code in AL:
;     00   Buffer was located and canceled.
;     FF   Buffer was not found to be in use by the driver
;

even
DriverCancelRequest proc near

      ; first, see if it is the one we are currently sending.
      mov dx, es
      cmp word ptr SendList[0], si
      %ife
          cmp word ptr SendList[2], dx
          %ife
       ; we need to cancel the first entry.  first, unlink it
       ; from the send list.
       mov ax, es: word ptr [si].Link[0]
```

292066-47

```
        mov word ptr SendList[0], ax
        mov cx, es: word ptr [si].Link[2]
        mov word ptr SendList[2], cx
        mov es: [si].CompletionCode, 0FCh
        mov es: [si].InUseFlag, 0
        xor ax, ax
        ret
          %endif
      %endif

      ; we need to search down the send list
      mov cx, word ptr SendList[2]
      mov di, word ptr SendList[0]

      %do
          %do
        jcxz NotFound
        ; move to the next link
        mov es, cx
        mov bx, di
        mov cx, es: word ptr [bx].Link[2]
        mov di, es: word ptr [bx].Link[0]
        ; next node is pointed to by CX:DI
        ; previous node is pointed to by es: BX
        ; see if we found it
        cmp di, si
          %whilenz
          cmp cx, dx
      %whilenz

      ; we found it.  now unlink it.
      push ds
      mov ds, cx
      mov ax, ds: word ptr [si].Link[0]
      mov es: word ptr [bx].Link[0], ax
      mov ax, ds: word ptr [si].Link[2]
      mov es: word ptr [bx].Link[2], ax
      mov ds: [si].CompletionCode, 0FCh
      mov ds: [si].InUseFlag, 0
      pop ds
      xor ax, ax
      ret

NotFound:
      mov al, 0FFh
      ret

DriverCancelRequest endp
```

292066-48

```
;
;   Driver Close Socket
;
;   Assumes on entry:
;     DX has socket number
;     DS is setup
;     Interrupts are DISABLED
;
;   Assumes any registers may be destroyed.
;

even
DriverCloseSocket proc near

      mov cx, word ptr SendList[2]
      jcxz DriverCloseExit
      les si, SendList
      %do
          cmp es: [si].SocketNumber, dx
          %ife
       push dx
       call DriverCancelRequest
       pop dx
       jmp DriverCloseSocket
          %endif
          mov cx, es: word ptr [si].Link[2]
          jcxz DriverCloseExit
          les si, es: [si].Link[0]
      %forever
DriverCloseExit:
      ret

DriverCloseSocket endp


Code  ends



mombo_init segment 'CODE'

      public DriverInitialize, DriverUnHook
no_card_message        db  CR, LF, 'No adapter installed in PC$'
config_failure_message db  CR, LF, 'Configuration Failure$'
iaset_failure_message  db  CR, LF, 'IA Setup Failure$'

;                0    1    2    3    4    5    6    7
DMAPageRegisters       db 87h, 83h, 81h, 82h, 8fh, 8bh, 89h, 8ah

config_block   dw 15          ; 0..1:  byte count
       db 48h          ; 2:  High-Speed Mode, Fifo Limit = 8
       db 80h          ; 3:  TCI mode
       db 00100110b    ; 4:
```

292066-49

```
        db 00h          ; 5:
        db 96           ; 6:  Interframe Spacing
        db 00h          ; 7:
        db 0F2h         ; 8:
        db 00000000b    ; 9:
        db 00h          ; 10:
        db 64           ; 11:  Minimum Frame Size
        db 11110111b    ; 12:  Auto Restransmit
        db 00h          ; 13:
        db 3Fh          ; 14:
        db 87h          ; 15:
        db 0D0h         ; 16:
        db 0FFh         ; 17:

InterruptBit            db ?
InterruptMask           db ?
                even
OldIRQVector            dd ?
InterruptMaskRegister   dw ?
InterruptVectorAddress  dd ?

;
;   Driver Initialize
;
;   assumes:
;     DS, ES are set to CGroup (== CS)
;     DI points to where to stuff node address
;     Interrupts are ENABLED
;     The Real Time Ticks variable is being set, and the
;     entire AES system is initialized.
;
;   returns:
;     If initialization is done OK:
;       AX has a 0
;     If board malfunction:
;       AX gets offset (in CGroup) of '$'-terminated error string
;

DriverInitialize PROC NEAR

      mov MaxPhysPacketSize, 1024
      cld
      cli

      ; initialize the configuration table
      mov al, selected_configuration
      cbw
      shl ax, 1               ; multiply by two
      mov bx, ax
      mov bx, config_pointers[bx]
      mov ax, [bx].H_IOBase
      add Addr592, ax
      add AddrLatchLow, ax
      add AddrLatchHigh, ax
```

292066-50

```
add LimitRegister, ax
add DMAcmdstat, ax
add DMAsnglmsk, ax
add DMAmode, ax
add DMAff, ax
add XmtDMAaddr, ax
add XmtDMAwdcount, ax
add XmtDMAPage, ax
add RcvDMAaddr, ax
add RcvDMAwdcount, ax
add RcvDMAPage, ax
add IA_PROM_port, ax

; setup the dma registers

mov al, [bx].H_DMA0
cbw
mov si, ax

mov cl, DMAPageRegisters[si] ; get the page register address
xor ch, ch
mov MasterDMAPage, cx       ; save it

and al, 03h
add MasterDMAmsk, al
add MasterDMAunmsk, al
add MasterDMAmodevalue, al
add ax, ax
add ax, ax
add MasterDMAaddr, ax
add MasterDMAwdcount, ax

; load the node address
lea si, node_addr           ; targets are es:si and es:di
xor ax, ax                  ; ah = prom address
mov cx, size node_addr
%do
    mov al, ah
    mov dx, LimitRegister   ; set prom address
    out dx, al
    mov dx, IA_PROM_port
    in al, dx               ; read prom value
    stosb                   ; store it at es:di
    xchg si, di
    stosb                   ; and at es:si
    inc ah                  ; increment prom address
%loop

;
; SET UP THE INTERRUPT VECTORS
;
mov al, [bx].H_IRQ
mov bx, OFFSET CGroup: DriverISR
call SetInterruptVector
```

292066-51

```
mov dx, Addr592
mov al, C_RST
out dx, al                      ; reset the 82592 controller

; generate 20 bit address for DMA controller from
; configure block location this is necessary to
; accomodate the page register used in the PC DMA

call SetUpBuffers

; configure the master channel for cascade mode

mov al, MasterDMAmsk
mov dx, MasterDMAsnglmsk
out dx, al                      ; disable the channel

mov al, MasterDMAmodevalue
mov dx, MasterDMAmode           ; get the mode register address
out dx, al                      ; set the mode

mov al, MasterDMAunmsk
mov dx, MasterDMAsnglmsk
out dx, al                      ; enable the channel

; set up DMA channel for configure command

mov al, XmtDMAmsk
mov dx, DMAsnglmsk
out dx, al                      ; disable the channel

mov al, RcvDMAmsk
mov dx, DMAsnglmsk
out dx, al                      ; disable the channel

mov dx, DMAFF
out dx, al                      ; data is don't care

mov al, ActiveLowDREQ + ExtendedWrite + RotatingPriority
mov dx, DMAcmdstat
out dx, al

mov dx, XmtDMAaddr
mov al, byte ptr gp_buf_start[0]
out dx, al
mov al, byte ptr gp_buf_start[1]
out dx, al

mov ax, gp_buf_page
mov dx, XmtDMApage              ; DMA page value
out dx, al

mov ax, 1
mov dx, XmtDMAwdcount           ; make two transfers
out dx, al
```

292066-52

```
mov al, ah
%slow
out dx, al

mov al, XmtDMAtx        ; setup tx mode
mov dx, DMAmode
out dx, al

mov al, XmtDMAunmsk
mov dx, DMAsnglmsk
out dx, al

les di, gp_buf_pointer          ; mov zeroes into the byte count field
stosw                           ;  of the buffer to put the 82592 into
stosw                           ;  16 bit mode
mov dx, Addr592
mov al, C_CONFIG        ; configure the 82592 for 16 bit mode
out dx, al                      ; issue configure command
%slow

xor cx, cx
%do
    xor al, al
    out dx, al                  ; point to register 0
    %slow
    in   al, dx                 ; read register 0
    and al, 0DFh        ; disregard exec bit
    cmp al, 82h         ; is configure finished?
%loopne

%ifne
    mov ax, OFFSET CGroup: no_card_message
    ret
%endif

mov al, C_INTACK
out dx, al                      ; clear interrupt

mov dx, DMAff
out dx, al                      ; data is don't care

mov dx, XmtDMAaddr
mov al, byte ptr gp_buf_start[0]
out dx, al
mov al, byte ptr gp_buf_start[1]
out dx, al

mov ax, gp_buf_page
mov dx, XmtDMApage              ; DMA page value
out dx, al

mov al, XmtDMAtx        ; setup channel 1 for tx mode
mov dx, DMAmode
out dx, al
```

292066-53

```
mov ax, 8
mov dx, XmtDMAwdcount
out dx, al
%slow
mov al, ah
out dx, al

mov al, XmtDMAunmsk
mov dx, DMAsnglmsk
out dx, al

mov si, offset cgroup: config_block
les di, gp_buf_pointer
mov cx, 18
rep movsb
mov dx, Addr592
mov al, C_CONFIG        ; configure the 82592
out dx, al

xor cx, cx
%do
    xor al, al
    %slow
    out dx, al                  ; point to register 0
    %slow
    in al, dx                   ; read register 0
    and al, 0DFh        ; discard extraneous bits
    cmp al, 82h         ; is configure finished?
%loopne

%ifnz
    mov ax, OFFSET CGroup: config_failure_message
    ret
%endif

; clear interrupt caused by configuration
mov al, C_INTACK
out dx, al

; do an IA_setup
les di, gp_buf_pointer
mov al, 06h                     ; address byte count
stosb
mov al, 00h
stosb
mov si, OFFSET CGROUP: node_addr
mov cx, SIZE node_addr
rep movsb

mov dx, DMAff
out dx, al                      ; data is don't care
```

292066-54

```
mov dx, XmtDMAaddr
mov al, byte ptr gp_buf_start[0]
out dx, al
mov al, byte ptr gp_buf_start[1]
out dx, al

mov ax, gp_buf_page
mov dx, XmtDMApage              ; DMA page value
out dx, al

mov al, XmtDMAtx        ; setup channel 1 for tx mode
mov dx, DMAmode
out dx, al

mov ax, 3
mov dx, XmtDMAwdcount
out dx, al
%slow
mov al, ah
out dx, al

mov al, XmtDMAunmsk
mov dx, DMAsnglmsk
out dx, al

mov dx, Addr592
mov al, C_IASET         ; set up the 82592 individual address
out dx, al

xor cx, cx
%do
    xor al, al
    out dx, al
    %slow
    in al, dx
    and al, 0DFh        ; discard extraneous bits
    cmp al, 81h         ; is command finished?
%loopne

%ifne
    mov ax, OFFSET CGroup: iaset_failure_message
    ret
%endif

mov al, C_INTACK
out dx, al                      ; clear interrupt from iaset

;initialize the receive DMA channel

mov dx, DMAff
out dx, al

mov dx, RcvDMAaddr
mov al, byte ptr rx_buf_start[0] ; set dma up to point to the
```

292066-55

```
        out dx, al                      ;  beginning of rx_buf
        mov al, byte ptr rx_buf_start[1]
        out dx, al

        mov ax, rx_buf_page             ; set rx page register
        mov dx, RcvDMApage
        out dx, ax

        mov al, RcvDMArx
        mov dx, DMAmode
        out dx, al

        mov dx, RcvDMAwdcount
        mov ax, rx_buf_length
        dec ax
        out dx, al
        mov al, ah
        %slow
        out dx, al

        ; initialize the limit register
        mov ax, rx_buf_limit
        sub ax, 2
        mov bx, rx_buf_segment
        shl bx, 4
        add ax, bx                      ; compute physical address
        mov al, ah
        mov dx, LimitRegister
        out dx, al

        mov al, RcvDMAunmsk             ; unmask receive DMA channel
        mov dx, DMAsnglmsk
        out dx, al

        ; enable the receiver
        mov dx, Addr592         ; enable receives
        mov al, C_RXENB
        out dx, al
        sti
        xor ax, ax
        mov cx, 1
        ret

DriverInitialize endp
```

292066-56

```
;*****************************************************************************
;
;       SetInterruptVector
;
;       Set the interrupt vector to the interrupt procedure's address.
;       Save the old vector for the unhook procedure.
;
;       assumes:  cs:bx is the ISR routine
;         al has the IRQ level 10..15
;         interrupts are disabled
;
;*****************************************************************************

SetInterruptVector proc near

        ; mask on the appropriate interrupt mask
        push ax
        xchg ax, cx
        mov dl, 1

        sub cl, 8
        shl dl, cl
        mov InterruptBit, dl
        not dl
        mov InterruptMask, dl

        in al, ExtraInterruptMaskPort
        and al, dl
        %slow
        out ExtraInterruptMaskPort, al

        mov InterruptMaskRegister, ExtraInterruptMaskPort

        ; also mask on level 2 of first controller
        in al, InterruptMaskPort
        and al, not 4
        %slow
        out InterruptMaskPort, al

        pop ax

        cld
        cbw

        xor cx, cx
        mov es, cx

        add al, 70h - 8
        shl ax, 1
        shl ax, 1
        mov di, ax

        mov word ptr InterruptVectorAddress[0], di
        mov word ptr InterruptVectorAddress[2], es
```

292066-57

```
        mov ax, es: [di][0]
        mov word ptr OldIRQVector[0], ax
        mov ax, es: [di][2]
        mov word ptr OldIRQVector[2], ax

        mov ax, bx
        stosw
        mov ax, cs
        stosw

        ret

SetInterruptVector endp



;  Set up Buffers:
;  This routine generates the page and offset addresses for the 16 bit
;  DMA. It checks for a page crossing and uses the smaller half of the
;  buffer area for Tx and general purpose if a crossing is detected. If
;  no crossing is detected the general purpose/transmit buffer is placed
;  at the beginning of the buffer area. This routine also generates a
;  segment address for the receive buffer which allows the value read
;  from the "10 cent" latches to be used as read for the offset passed
;  to IPXReceivePacket. This saves some arithmetic steps when tracing
;  back through the rx buffer chain.
;

gp_length               dw gp_buf_size + max_rx_buf_size
gp_offset_adjust        dw 0

SetUpBuffers proc near

        mov ax, offset cgroup: gp_buf
        mov bx, cs
        mov dx, cs
        shr ax, 1
        mov cx, 3
        shl bx, cl
        rol dx, cl              ; get upper 3 bits for page register
        and dx, 0007h           ; clear all but the lowest 3 bits
        add ax, bx              ; ax contains A16..A1 of first location in buffer
        adc dx, 0               ; if addition caused a carry add it to page
        xor cx, cx              ; of buffer to page break
        sub cx, ax              ; cx contains the number of words to page break

        cmp cx, gp_buf_size + max_rx_buf_size
        %ifae
            jmp copacetic       ; it's cool, whole buffer space is in one page
        %endif
```

292066-58

```
        cmp cx, gp_buf_size
        %ifbe
            add ax, cx              ; move pointer past the page break to discard fragment
            sub gp_length, cx       ; adjust length variable to reflect shorter length
            jmp copacetic           ; both buffers will be in the same page, rx buf shortened
        %endif

        cmp cx, max_rx_buf_size
        %ifae
            mov gp_length, cx   ; adjust length variable, discard upper buffer fragment
            jmp copacetic       ; both buffers will be in the same page, rx buf shortened
        %endif

        ; now since both fragments are usable we have to find the
        ; actual page break. the large half will be the receive
        ; buffer and the small half will be the gp-tx buffer.
        cmp cx, (gp_buf_size + max_rx_buf_size) / 2
        %ifbe
            ; transmit buffer first
            mov gp_buf_page, dx
            mov gp_buf_start, ax
            mov rx_buf_start, 0000h
            inc dx                      ; next page
            mov rx_buf_page, dx
            mov ax, gp_length
            sub ax, cx
            mov rx_buf_length, ax
        %else
            ; receive buffer first
            mov rx_buf_page, dx
            mov rx_buf_start, ax
            mov rx_buf_length, cx
            mov gp_buf_start, 0000h
            inc dx                      ; next page
            mov gp_buf_page, dx
        %endif
        jmp SetUpBuffers_exit

copacetic:
        mov gp_buf_start, ax    ; A1-A16 of gp buffer, gp buffer is first
        add ax, gp_buf_size     ; allocate gp_buf at front of buffer space
        mov rx_buf_start, ax    ; rx buffer starts 1200 bytes in
        mov cx, gp_length
        sub cx, gp_buf_size
        mov rx_buf_length, cx
        mov rx_buf_page, dx
        mov gp_buf_page, dx

SetUpBuffers_exit:
        mov ax, gp_buf_start
        mov dx, gp_buf_page
        shr dx, 1
        rcr ax, 1
        shr dx, 1
```

292066-59

```
        rcr ax, 1
        shr dx, 1
        rcr ax, 1               ; ax = a19..a4 of gp_buf
        mov dx, cs
        sub ax, dx
        shl ax, 4
        mov bx, gp_buf_start
        shl bx, 1
        and bx, 0fh
        or ax, bx               ; compute offset within cgroup
        mov word ptr gp_buf_pointer[0], ax
        mov word ptr gp_buf_pointer[2], cs

        mov ax, rx_buf_length
        shl ax, 1
        mov rx_buf_size, ax

        mov ax, rx_buf_start            ; get the physical word
        mov dx, rx_buf_page             ;   address of rx_buf
        shl ax, 1
        rcl dx, 1                       ; convert to byte address
        push ax
        xor al, al                      ; save bits A19..A8
        mov cx, 12
        %do
            shl ax, 1
            rcl dx, 1                   ; compute the closest segment
        %loop                           ;  boundry to rx_buf
        pop ax
        mov ah, 80h                     ; increment offset by 8000h bytes
        sub dx, 800h                    ; decrement segment by 800h paragraphs
        mov rx_buf_segment, dx
        mov rx_buf_first, ax
        mov rx_buf_head, ax
        add ax, rx_buf_size
        mov rx_buf_limit, ax

        mov ax, rx_buf_start
        shl ax, 1
        sub ax, rx_buf_first
        mov Logical2Physical, ax        ; logical to physical mapper
        ret

SetUpBuffers endp
```

292066-60

```
;
;   Driver Unhook
;
;   Assumes
;     DS = CS = IPX segment
;     Interrupts are DISABLED
;
;     Assumes any registers but DS, SS, SP may be destroyed
;
;     This procedure restores the original interrupt vector
;
;     This procedure will never be called if DriverInitialize
;     did not complete successfully.
;


DriverUnhook PROC NEAR

      mov dx, InterruptMaskRegister
      in al, dx
      or al, InterruptBit
      %slow
      out dx, al
      les bx, InterruptVectorAddress
      mov ax, word ptr OldIRQVector[0]
      mov es: [bx], ax ; restore old interrupt offset
      mov ax, word ptr OldIRQVector[2]
      mov es: [bx][2], ax    ; restore old interrupt segment
      ret

DriverUnhook endp

mombo_init ends
end
```

292066-61

```
;*************************************************************************
;
;
;  SMacro.inc:  A set of macros that allows assembly code to be
;               written in a structured fashion resembling a high
;               level language.
;
;  Written by Ben L Gee. San Jose, Ca.  (408)578-1123
;
;  This code may be used freely as long as the authors name appears
;  in the listing.
;
;
;*************************************************************************

%set(lev,0)
%set(number,0)

%*define(ifa) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jna l%num
)

%*define(ifae) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jnae l%num
)

%*define(ifb) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jnb l%num
)
```

292066-62

```
%*define(ifbe) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jnbe l%num
)

%*define(ifc) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jnc l%num
)

%*define(ifcxnz) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jcxz l%num
)

%*define(ifnc) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jc l%num
)
```

292066-63

```
%*define(ife) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jne l%num
)

%*define(ifne) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      je l%num
)

%*define(ifz) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jnz l%num
)

%*define(ifnz) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       jz l%num
)
```

292066-64

```
%*define(else) (
%if (%lev eq 1) then (%set(t, %level01H)) fi
%if (%lev eq 2) then (%set(t, %level02H)) fi
%if (%lev eq 3) then (%set(t, %level03H)) fi
%if (%lev eq 4) then (%set(t, %level04H)) fi
%if (%lev eq 5) then (%set(t, %level05H)) fi
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jmp short l%num
l%t:
)

%*define(else1) (
%if (%lev eq 1) then (%set(t, %level01H)) fi
%if (%lev eq 2) then (%set(t, %level02H)) fi
%if (%lev eq 3) then (%set(t, %level03H)) fi
%if (%lev eq 4) then (%set(t, %level04H)) fi
%if (%lev eq 5) then (%set(t, %level05H)) fi
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jmp l%num
l%t:
)

%*define(endif) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
l%num:
%set(lev, %lev-1)
)
```

292066-65

```
%*define(do) (
%set(lev, %lev+1)
%set(number, %number+1)
%set(level%lev, %number)
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
l%num:
)

%*define(forever) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jmp l%num
%set(lev, %lev-1)
)

%*define(whilea) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      ja l%num
%set(lev, %lev-1)
)

%*define(whileae) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jae l%num
%set(lev, %lev-1)
)

%*define(whileb) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jb l%num
%set(lev, %lev-1)
)
```

292066-66

```
%*define(whilebe) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jbe l%num
%set(lev, %lev-1)
)

%*define(whilec) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jc l%num
%set(lev, %lev-1)
)

%*define(whilecxz) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jcxz l%num
%set(lev, %lev-1)
)

%*define(whilenc) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jnc l%num
%set(lev, %lev-1)
)

%*define(whilee) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      je l%num
%set(lev, %lev-1)
)
```

292066-67

```
%*define(whilene) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jne l%num
%set(lev, %lev-1)
)

%*define(whilez) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
      jz l%num
%set(lev, %lev-1)
)

%*define(whilenz) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       jnz l%num
%set(lev, %lev-1)
)

%*define(loop) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       loop l%num
%set(lev, %lev-1)
)

%*define(loope) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       loope l%num
%set(lev, %lev-1)
)
```

292066-68

```
%*define(loopz) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       loopz l%num
%set(lev, %lev-1)
)

%*define(loopne) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       loopne l%num
%set(lev, %lev-1)
)

%*define(loopnz) (
%if (%lev eq 1) then (%set(num, %level01H)) fi
%if (%lev eq 2) then (%set(num, %level02H)) fi
%if (%lev eq 3) then (%set(num, %level03H)) fi
%if (%lev eq 4) then (%set(num, %level04H)) fi
%if (%lev eq 5) then (%set(num, %level05H)) fi
       loopnz l%num
%set(lev, %lev-1)
)
```

292066-69

```
;*************************************************************
;
; Relid.inc  Include file containing revision information
;        for the NBM592 driver software.
;
; Written by Ben L. Gee  San Jose, California
;
;*************************************************************

%*define(MajorVersion)(1)
%*define(MinorVersion)(00)
%*define(VersionDate)(890129)
%*define(LanType)(171)                ; not yet assigned

; 890124 use extended write dma mode
; 890129 correct ECB cancel bug
```

292066-70

intel®

APPLICATION NOTE

AP-326

July 1989

PS592E-16 Buffered Adapter LAN Solution for the Micro Channel Architecture

DARYOOSH KHALILOLAHI
TECHNICAL MARKETING ENGINEER

Order Number: 292060-001

1.0 INTRODUCTION

As the performance of personal computers increases, their role in the office environment expands. This expansion, coupled with the rapid increase in the number of personal computers, makes interconnection an indespensible option. Sharing expensive peripherals (such as high quality printers) reduces the cost. Sharing a single data base improves data control and security. Having electronic mail capabilities improves communication. Proliferation of personal computers as the workstations of choice provides yet another new application for networking. Clusters of workstations connected in Local Area Networks (LANs) can improve productivity by leveraging other station's (in the same or other clusters) computing and storage capabilities. In such an environment the network throughput of workstation nodes is increasingly important.

The best choices for Local Area Networks are those that provide reliability, low cost, ease of expansion, and the backing of major VLSI manufacturers. In recent years IEEE 802.3 10BASE5 (Ethernet), 10BASE2 (Cheapernet), and Twisted Pair Ethernet (TPE) 10BASE-T have emerged as popular choices.

The PS592E is a 16-bit nonintelligent, 32-KByte, buffered slave adapter. It interfaces IBM Micro Channel (Personal System 2 models 50, 60, 70 and 80) computers to an Ethernet or Cheapernet based network.The 82592 LAN Controller and 82561 DMA Controller are used to receive and transmit frames between the network and local memory. The board can perform default cycle (zero wait-state, 200 ns) memory data transfers on the Micro Channel. The board comes with two interchangeable network serial interface modules for Ethernet and Cheapernet applications. A TPE network module will be available in the near future.

A menu driven exerciser software and a NetWare driver are provided with the demo board.

2.0 OBJECTIVE

This application note describes how the Intel 82561 and 82592 are used to build a high-performance, cost-effective LAN adapter that implements the traditional buffered architecture. The last chapter describes an easy migration to a 32-bit adapter design.

2.1 Acknowledgements

I ackowledge and thank Yosi Mazor and Joe Dragony, of Intel's (Folsom, Calif.) Data Communications Focus Group, and Adi Golbert of Intel's (Israel) architecture definition group for their work in developing the hardware and the software and their contribution to this application note.

2.2 Terminology

In the PAL equations and schematics a "_" at the end of a signal name indicates that the signal is active low, "#" stands for logical OR, "&" stands for logical AND, and "!" stands for logical inversion. In the schematics any signal name starting with the letter "L" indicates that the signal is latched on the board or that all the signals used in generating this signal are latched.

3.0 ORGANIZATION

Chapter 4 provides an overview of the 82561 and 82592 functionality. The reader needs a basic knowledge of these components to better understand the following chapters. Chapter 5 provides a functional description of the PS592E. In this chapter, the design is divided into three architectural subsections (host interface, memory subsystem, and network interface). PAL equations and schematics are broken down according to the architectural division. Chapter 6 is the software chapter; samples from the Novell NetWare driver are given. Chapter 7 provides the performance benchmarks for the board. Chapter 8 shows how the design can be modified (including new PAL equations) to upgrade it to a 32-bit adapter. The appendix gives a brief description for most of PS592E internal signals.

4.0 COMPONENT OVERVIEW

4.1 82592 LAN Controller

The CHMOS 82592 is CSMA/CD controller with a 16-bit data path. It can be configured to support a wide variety of industry standard networks, including Ethernet, Cheapernet, TPE, PCNet, and StarLan. The 82592 consists of three subsystems: parallel, serial, and FIFO. The parallel subsystem provides an 8- or 16-bit interface to the external bus. The 82592 supports memory transfers (at up to 16 MB/s), accepts commands from the processor that controls the bus, and provides status to it. The 82592 can support simultaneous transmission and reception including autoretransmit, transmit frame chaining, and back-to-back frame reception. The serial subsystem consists of a highly flexible CSMA/CD unit, a data encoder/decoder, collision detect and carrier sense logic, and a clock generator. In high- integration mode it supports NRZI, Manchester, or Differential Manchester encoding and decoding at bit rates up to 4 Mb/s. In high-speed mode the 82592 is capable of 20-Mb/s Manchester or NRZI encoding. The FIFO subsystem consists of a transmit FIFO, a receive FIFO, and control logic (with programmable threshold). A total of 64 bytes of FIFO can be divided between receive and transmit. This can be done in any of four possible combinations (16/48, 32/32, 48/16, 16/16 byte resolution).

4.2 82561 Host Interface and Memory controller

The CHMOS 82561 is a high-performance DMA controller designed to work in a tightly coupled fashion with the 82592 in a PC AT or PS/2 adapter application.

Two independent DMA channels support transfers of up to 10 MB/s to/from the local SRAM/LAN Controller. Up to 32 KB of ring buffer memory can be I/O or memory mapped into the address space. Host accesses to the local memory can be made with zero wait states. These accesses can be 16- or 32-bit wide. The 82561, without CPU intervention, supports all of the 82592 tightly coupled functions. It can also reclaim bad receive buffers.

The 82592/82561 is an ideal choice for 16 or 32-bit buffered adapters. The combination provides ease of design, high performance, low component count, low power requirements, and competitive cost.

NOTE:
The 82560 and 82561 have similiar functionality. The only exception is that the double-host bus mode of the 82561 supports 32-bit-wide local memory. The 82592/82560 combination is equally suitable for a 16-bit-wide buffered adapter design. The 82561 is used in the PS592E design to demonstrate a 32-kB (8k X 32) buffered memory implementation.

5.0 IMPLEMENTATION

The board is divided into three sections (Figure 1), the host interface, the memory subsystem, and the network subsystem. Both the 82592 and 82561 operate on the 10-Mhz clock generated by the serial side. In the following sections of this chapter a component (designated by its U No. on the board and the schematic) is defined as part of a subsystem if one or more of its output pins are in that subsystem. The host CPU generates a request to the 82561 to access any port (including SRAM) on the board. SRAM accesses are 16-bit wide. All other transfers are 8-bit wide. The local memory is accessed either directly (nonpipeline mode) or through the data latches (pipeline mode). The data transfers between the local memory and the 82592 are 16 bits wide and are controlled by the 82561. During DMA transfers low and high banks of memory are accessed alternately.

5.1 Host interface

This subsystem consists of the POS ID register (U2), POS configuration register (U3), the command register (U1), the status register (U4), the address decoder (U14, U25, U24, U29, U23, and U5), the address latches (U9 and U12), the data latches/transceivers (U32, U22, U26, and U16), and their control (U31 and U21), the request generator (USP, U24, U30, U23, U21, U5, and U19), and the controller (U7) and its support logic (U20, U23, U30, and U13).

5.1.1 ADDRESS DECODING

After power-up the host reads the POS Read Identification register of the board, and if it is what the host expects, the host will configure the board. The ID resides in location 16 and 17 of the on-board 32-byte PROM. The first six Bytes of the PROM hold the board's network address. The PAL equation for the PROM chip select is given in section 5.2. For a complete list of the ports accessed by the 82561 GCS__ output see the table in section 5.1.4.

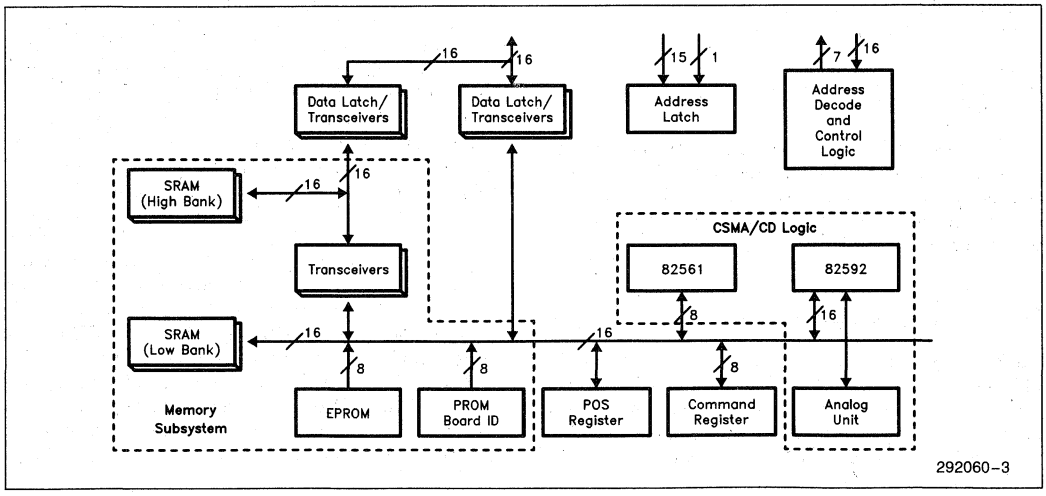


Figure 1. PS592E/16 Block Diagram

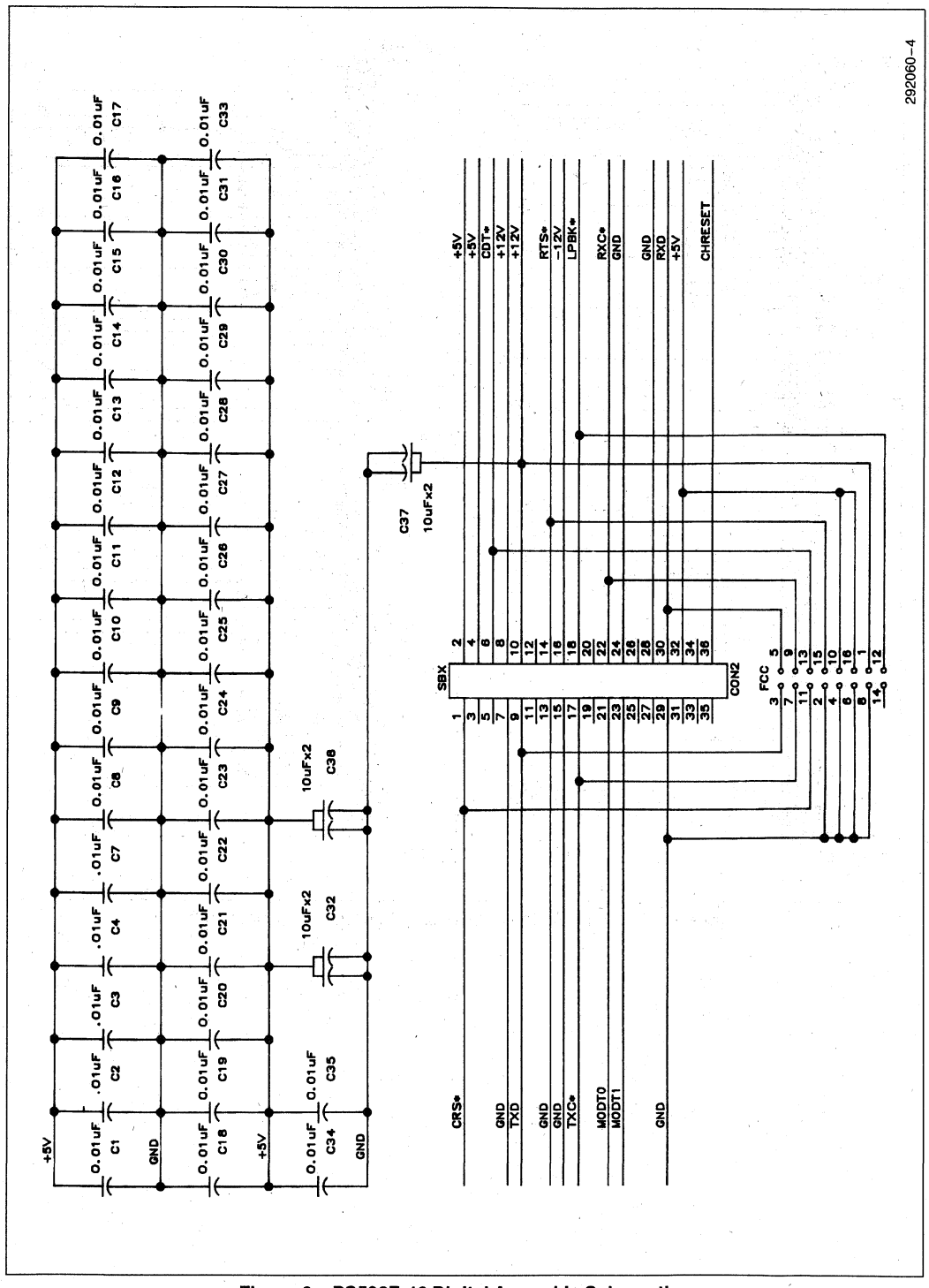


Figure 2a. PS592E-16 Digital Assembly Schematics

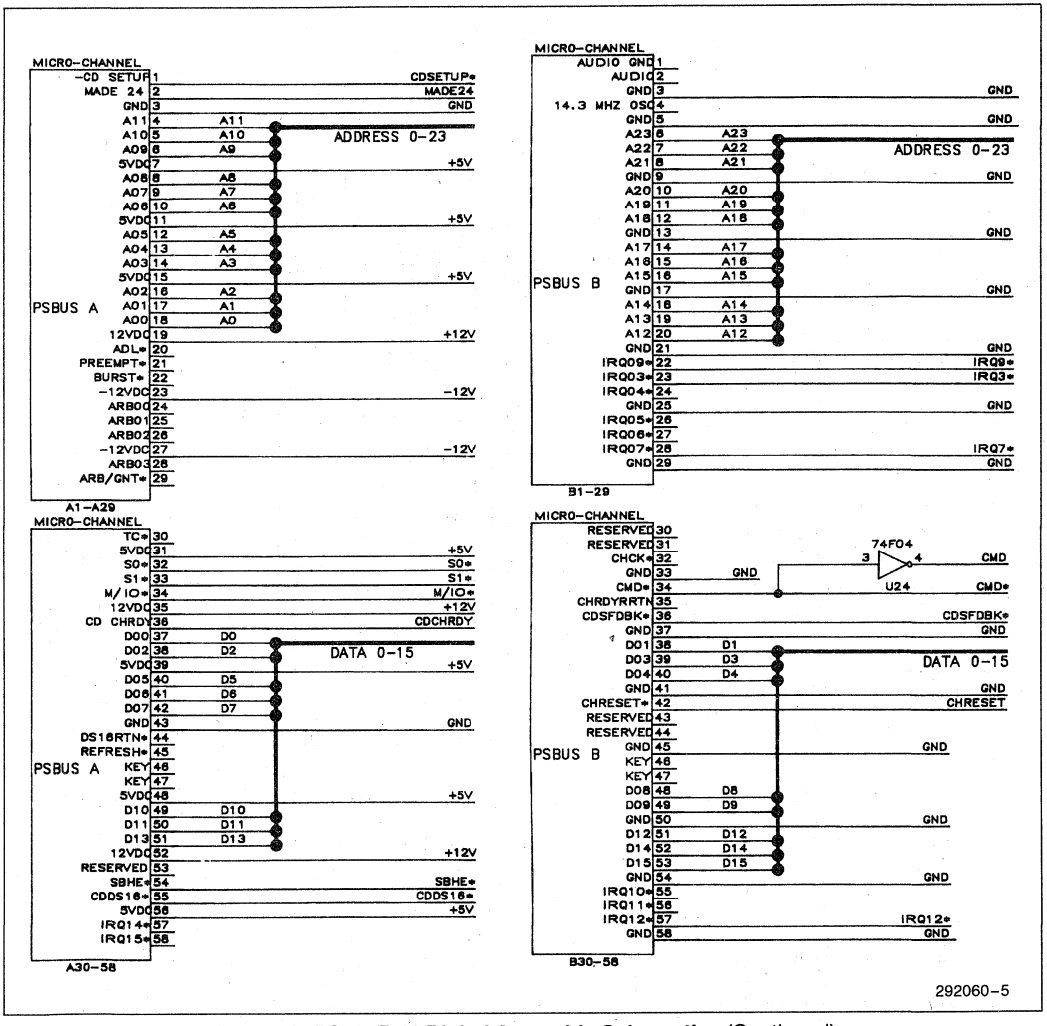


Figure 2b. PS592E-16 Digital Assembly Schematics (Continued)

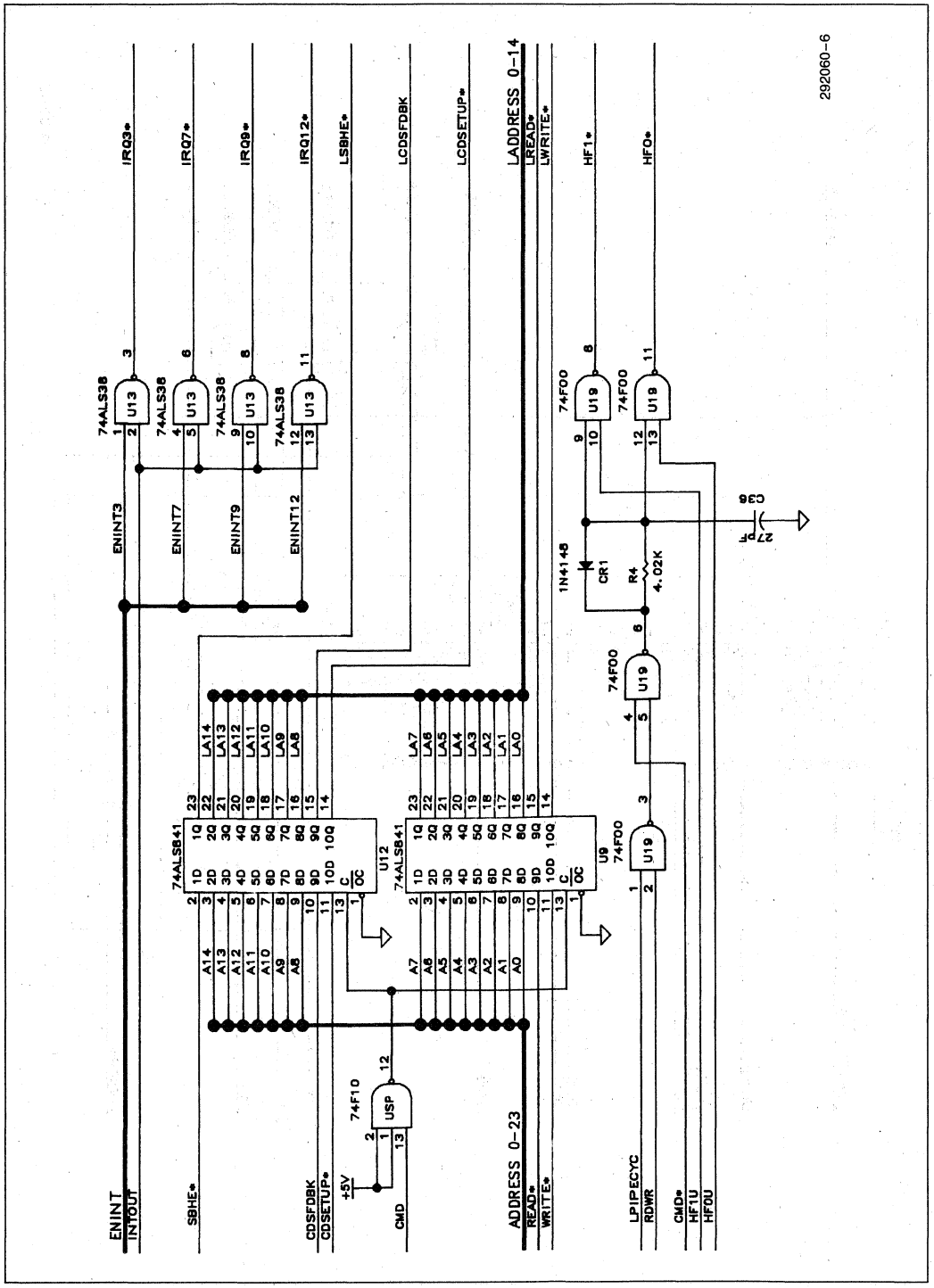


Figure 2c. PS592E-16 Digital Assembly Schematics (Continued)

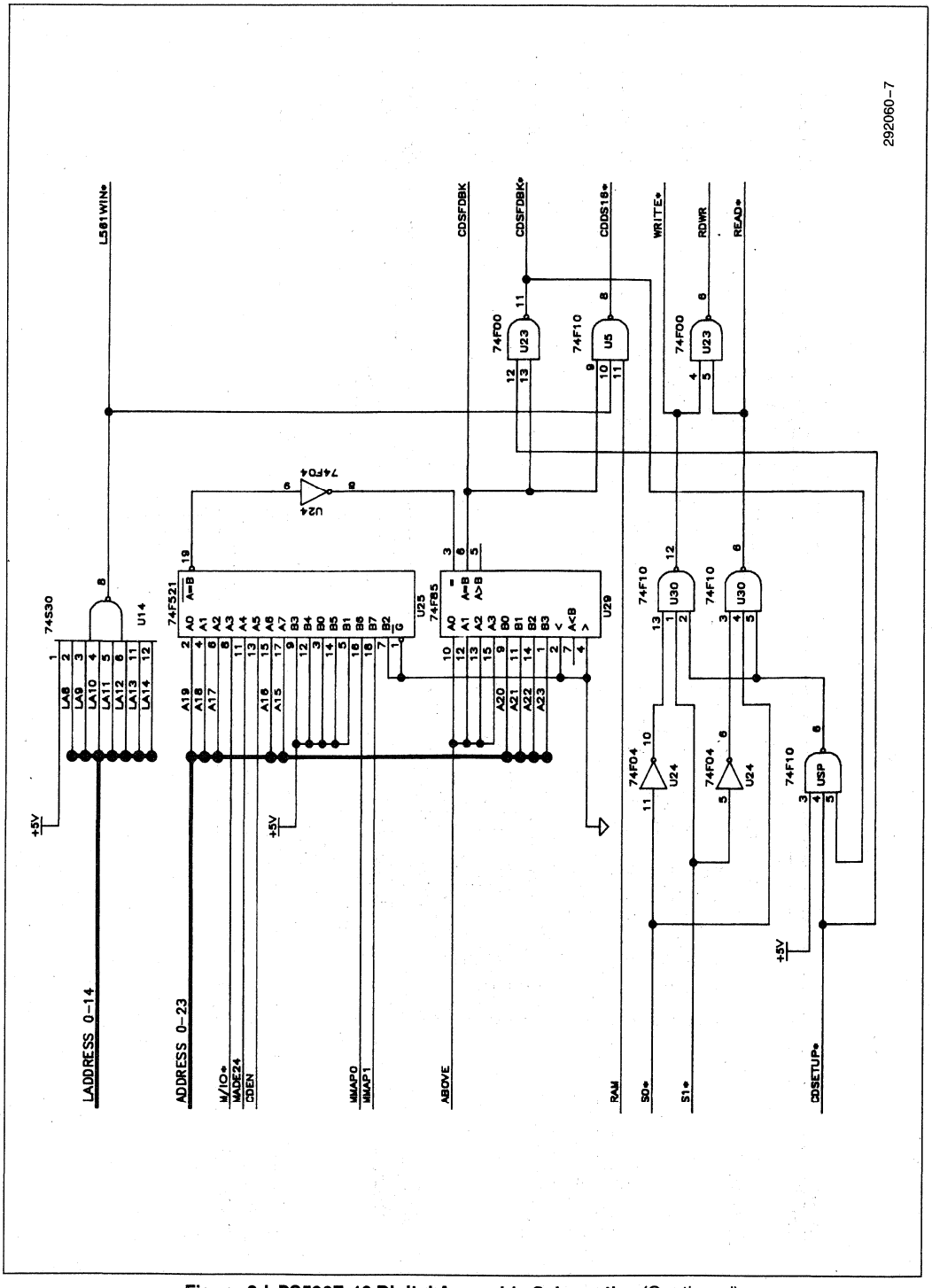


Figure 2d. PS592E-16 Digital Assembly Schematics (Continued)

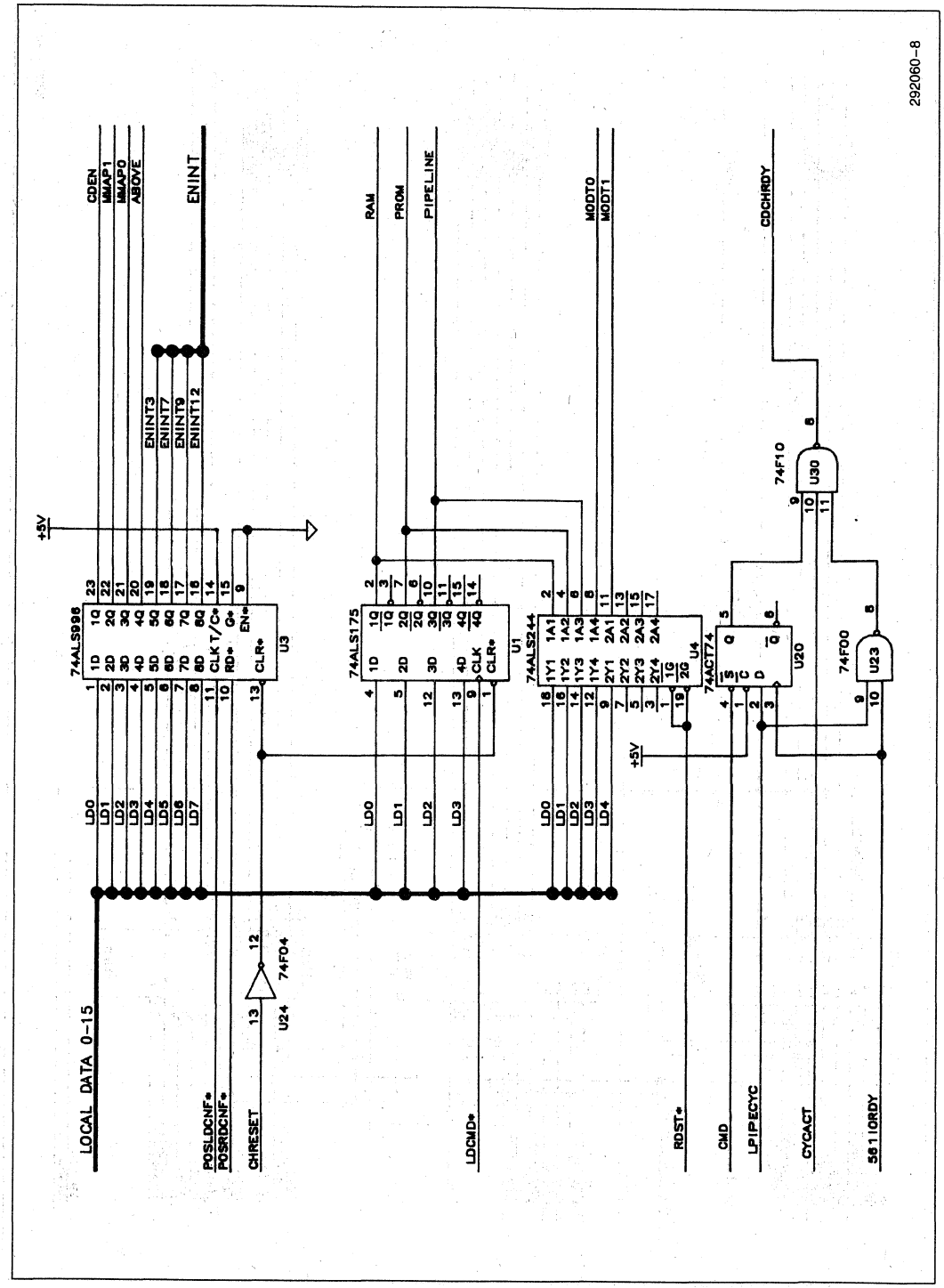


Figure 2e. PS592E-16 Digital Assembly Schematics (Continued)

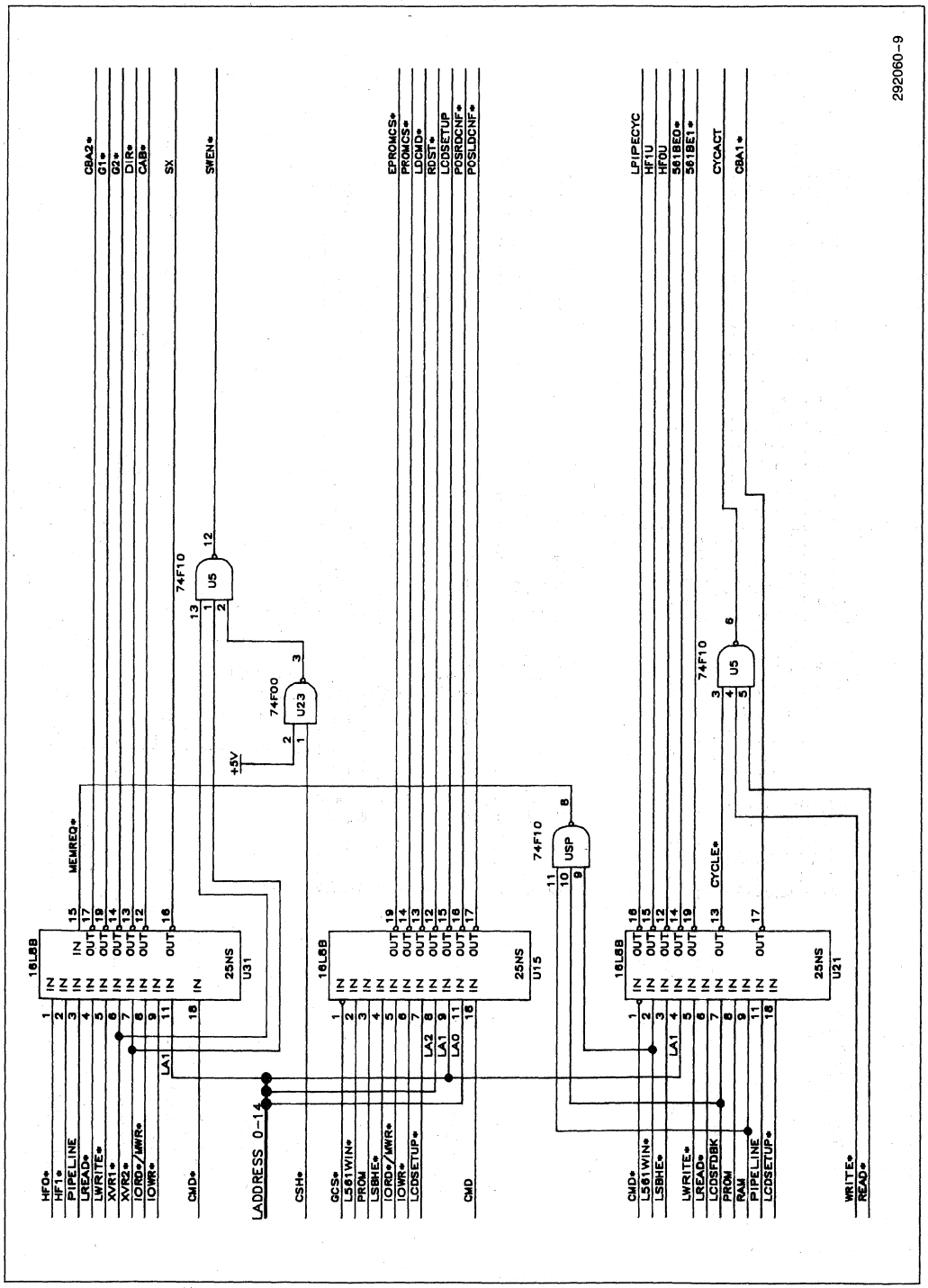


Figure 2f. PS592E-16 Digital Assembly Schematics (Continued)

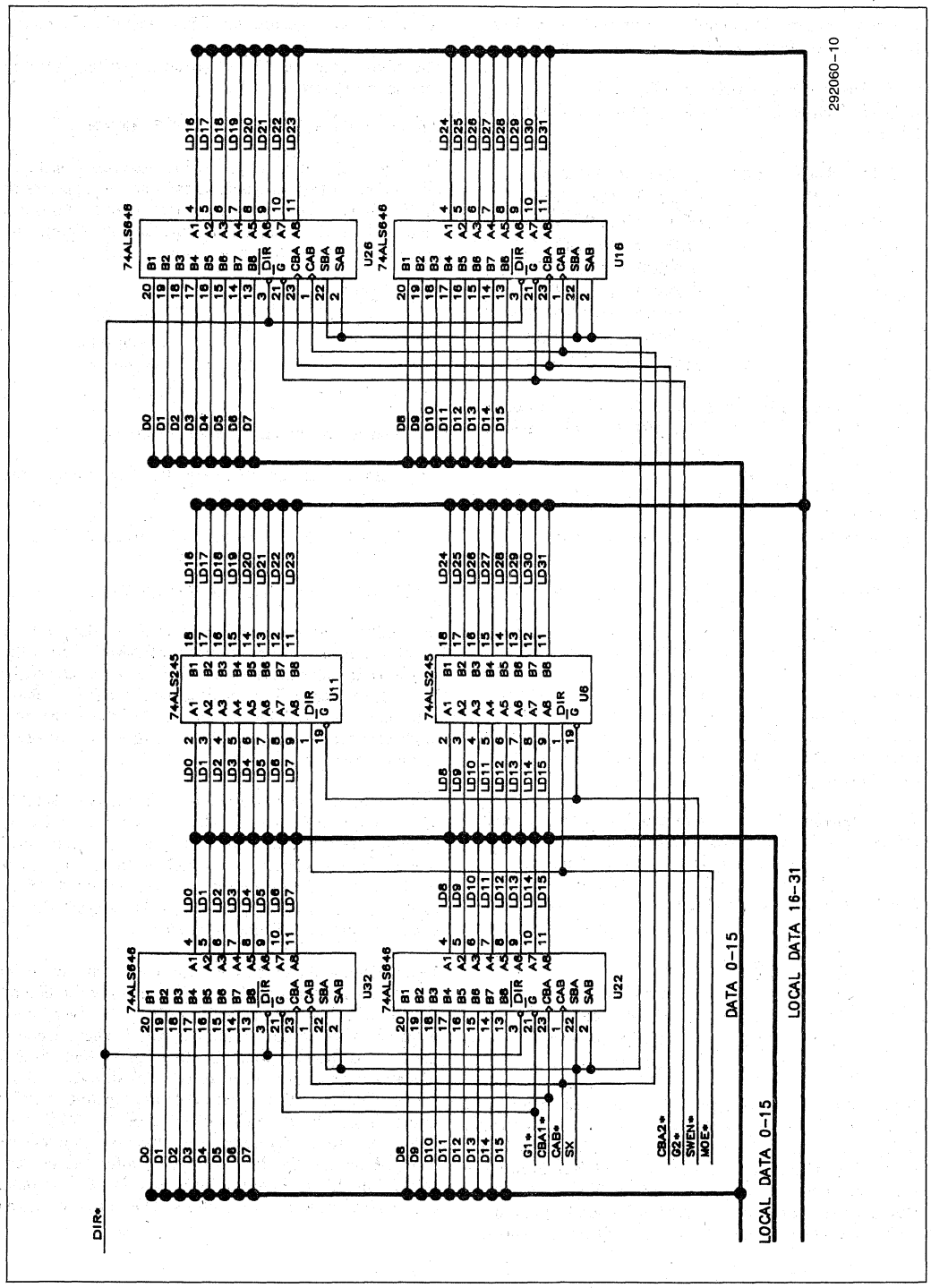


Figure 2g. PS592E-16 Digital Assembly Schematics (Continued)

The board has one POS configuration register (address 102 h). The contents of this register are shown below.

Bit 0: Enable/Disable the adapter
Bit 1–3 : Mapping window (four below and four above 1 Megabyte)

Bit 3	Bit 2	Bit 1	Memory window
0	0	0	0C0000 h to 0C7FFF h
0	0	1	0C8000 h to 0CFFFF h
0	1	0	0D0000 h to 0DFFFF h
0	1	1	0D8000 h to 0DFFFF h
1	0	0	FC0000 h to FC7FFF h
1	0	1	FC8000 h to FCFFFF h
1	1	0	FD0000 h to FD7FFF h
1	1	1	FD8000 h to FDFFFF h

Bit 4–7: Selective Interrupt level. Bit 4 when set selects IRQ3__; Bit 5 selects IRQ7__; Bit 6 selects IRQ9__; Bit 7 selects IRQ12__.

The following PAL equations are for reading from and writing to the POS configuration register.

```
POSRDCNF__ = ! (!GCS__&!LA0&LA1&!LA2&!IORD__
                &!LCDSETUP__);

POSLDCNF__ = ! (!GCS__&!LA0&LA1&!LA2&!IOWR__
                &!LCDSETUP__);
```

The upper 256 bytes of the address space (Base + 7F00 to Base + 7FFF) are used to acccess the command register and the 82561 register ports. All accesses in this range must be byte accesses. An active SBHE__ (odd byte) is interpereted as a request for accessing the command register and an inactive SBHE__ (even byte) for 8256$\frac{1}{8}$2592 accesses.

The remainder of the 32-kB memory window is shared by the local SRAM, PROM, and an optional 8-or 16-kB EPROM. The sharing is accomplished by a paging scheme implemented in a 3-bit command register. Bit 0 and 1 of the command register are decoded as shown in the following table.

Bit 1	Bit 0	
0	0	EPROM access
0	1	RAM access
1	0	PROM access
1	1	reserved

Bit 2 of the command register determines the mode for accessing the buffer memory. When this bit is 0 the host accesses to the local SRAM are made with 3 or 4 wait states. These are referred to as nondefault cycles in the Micro Channel documents. When this bit is 1, the host accesses to the local SRAM are made without any wait states. This is referred to as default cycles. This is achieved by configuring the 82561 to its pipeline mode.

The following is the PAL equation for writing into the command register.

```
LDCMD__ = !(!GCS__&!L561WIN__&!LSBHE__&!IOWR__);
```

The status register is five bits wide. The three least significant bits of this read only register are the contents of the command register. The two most significant bits are generated by the network module and determine the type of the network module installed.

Bit 4	Bit 3	
0	0	Ethernet
0	1	Cheapernet
1	0	Reserved
1	1	TPE

The reserved combination is for future use.

The following is the PAL equation for reading the status register.

```
RDST__ = !(!GCS__&!L561WIN__&!LSBHE__&!IORD__);
```

5.1.2 DATA TRANSCEIVERS/LATCHES

Bit 3 of the command register determines the mode in which the host accesses the local SRAM. When 0 the access is in nonpipeline mode (wait states asserted). When 1, the access is in the pipeline mode (Microchannel default cycles). This bit, and bit 0 of the 82561 host mode register should be set to the same value before a memory access is attempted.

In the nonpipeline mode the data transceivers/latches act as simple transceivers. The cycles are extended by pulling CHRDY low until the transfer (to/from) local memory is completed. A1, which is the second least significant bit of host address, determines which 16-bit bank of memory is being accessed. All non-SRAM accesses are in the nonpipeline mode.

In the pipeline mode the data transciever/latches act as data latches. In this mode a read ahead / write behind operation is performed by the 82561 after every odd-word access requested by the host CPU. These accesses are to sequential locations in the local SRAM. In this mode no wait states are asserted (default cycle on the Micro Channel). The direction of the pipeline transfer is determined by the value of bit 1 of the 82561 host mode register, therefore the direction cannot be changed on the fly. In read cycles, after the current transfer the 82561 updates the buffer with the contents of the next local memory address. This is referred to as

"read ahead" (in anticipation of the next host read request). In write cycles, the data is copied from the data latch to the local memory after the host has finished writing to the data latch. This is referred to as "write behind". Pipeline transfers are made after the host requests accessing an odd word, they are double-word wide (both memory chip selects are activated).

The control signals to the transcievers/latches are generated by the following PAL equations.

```
G1__  = ! ((!MEMREQ__ & PIPELINE & !LREAD__ &
          !LA1 & !CMD__)                          #
          (!XVR1__ & !CMD__)                      #
          (!IORD__ & PIPELINE & !XVR2__));

G2__  = ! ((!HF0__ & HF1__ & PIPELINE & !LREAD__ &
          LA1 & !CMD__)                           #
          (!HF0__ & HF1__ & !PIPELINE & !XVR2__ &
          !CMD__)                                 #
          (!IORD__ & PIPELINE & !XVR2__));

CAB__ = ! (!IOWR__ & !XVR2__);

CBA1__ = ! (!CMD__ & PIPELINE & !LWRITE__ & !LA1 &
          RAM & L561WIN__ & LCDSFDBK);

CBA2__ = ! (!HF0__ & HF1__ & !CMD__ & PIPELINE &
          !LWRITE__ & LA1);

DIR__ = ! ((PIPELINE & !XVR2__ & !IORD__)         #
          (!(!HF0__ & HF1__) # !PIPELINE) & !LWRITE__
          );

SX    = (!MEMREQ__ & PIPELINE # !IORD__ &
          PIPELINE & !XVR2__);
```

5.1.3 ADDRESS LATCHES

The host address and status are latched using the falling edge of CMD__. The latches become transparent when CMD__ goes inactive.

5.1.4 REQUEST GENERATOR

Active S0__/S1__ and CDSFDBK__ or CDSETUP__ initiate a Host request. "CYCACT" indicates an active request. The following table shows the complete list of Host requests to the 82561.

The following PAL equations realize the above table. HF0U and HF1U are the unqualified HF line. They are qualified using the host's status and command lines before they become 82561 input requests.

```
HF1U = ((LRDWR & LCDSFDBK) & (!L561WIN__ #
       !RAM)) #
       (!LCDSETUP__ & LRDWR);

HF0U = ((LRDWR & LCDSFDBK) & ((!L561WIN__ &
       !LSBHE__) # (L561WIN__
       & RAM & !PIPELINE) # (L561WIN__ & RAM &
       LA1)
       # (L561WIN__ & !RAM))) # (!LCDSETUP__ &
       LRDWR);
```

NOTE

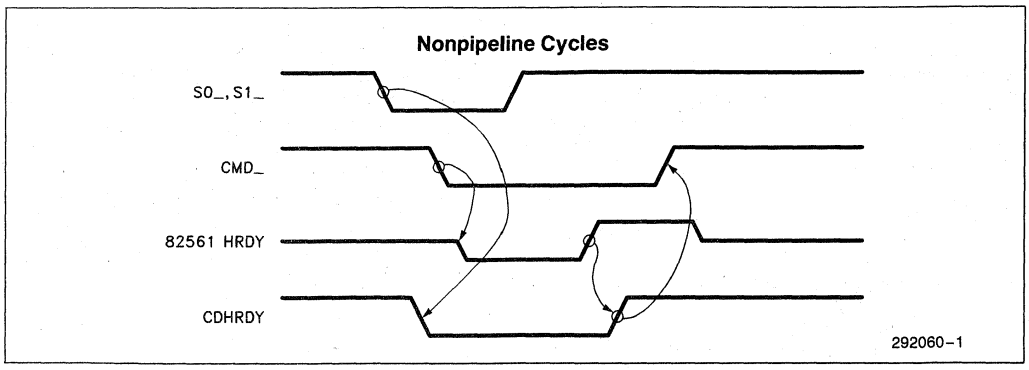
1. The request is qualified earlier (with status line decode) in the case of pipeline cycles. In the pipeline mode the 82561 provides half-clock glitch protection (on its HF inputs). In nonpipeline cycles the request is qualified later (with CMD__), when the address decode is free of any glitch.
2. CR1, R4, and C36 are added to delay the low-to-high transition of the signal. The 82561 spec requires 100-ns inactive time on its HF inputs. The CMD inactive time can be as short as 80 ns. During back-to-back nonpipeline write cycles this can cause the 82561 to miss the deassertion of the first request. The added circuitry guarantees that the HF inputs of the 82561 will be inactive for at least 100 ns.

5.1.5 MEMORY CONTROLLER AND ITS SUPPORT LOGIC

To access any port on the board the host generates a request to the 82561. S0__ and S1__ are decoded to determine if the request is a read or a write. The interrupt from the 82561 activates one of the four interrupt lines on the Micro Channel, depending on the POS configuration. In the nonpipeline mode CDCHRDY is pulled low immediately after the cycle starts. In the pipeline mode the CDCHRDY is high when the cycle starts.

CYCACT	LCDSETUP__	L561WIN__	RAM	SBHE__	HF1__	HF0__	
0	1	X	X	X	1	1	Idle
1	1	0	X	0	0	0	GCS__ cycle: Command/Status
1	1	1	0	X	0	0	GCS__ cycle: ROM *
1	1	1	1	X	1	0	SRAM cycle
1	1	0	X	1	0	1	82561 cycle
1	0	X	X	X	0	0	GCS__ cycle: POS registers

* ROM refers to either PROM or EPROM. Bit 1 of the command register determines which.



5.1.5.1 Nonpipeline Cycles

In all nonpipeline cycles (SRAM or otherwise) CDCHRDY is pulled low within 30 ns after the status (S0__ or S1__) becomes active. It remains low until the 82561 HRDY output goes to 1, then it goes to 1.

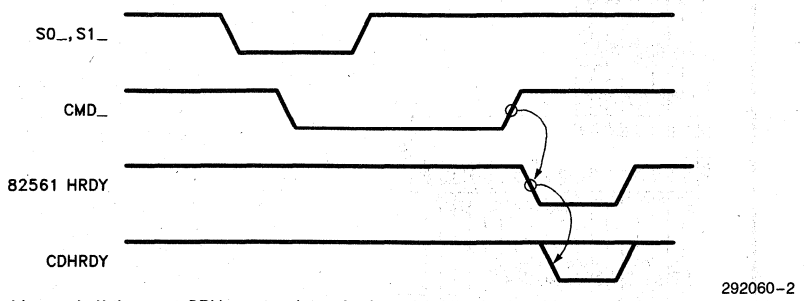
The memory address provided by the 82561 in the nonpipline mode is the same as the host CPU address except that its three most significant bits (12, 13, and 14) are logically ORed with the three least significant bits of the 82561 host address register. The low bank of memory is accessed for even-word addresses and the high bank for odd-word addresses.

```
BE0__   = ! ((LRDWR & LCDSFDBK & !CMD__ &
              !L561WIN__)
             # ( LCDSFDBK & LRDWR & !CMD__ &
             !(RAM & LA1))
             # (!LCDSETUP# ));
BE1__   = ! ((LRDWR & LCDSFDBK & !CMD__) &
             (L561WIN__ & RAM & LA1));
CYCLE__ = ! ((!(LREAD__ & LWRITE__ & CMD__) &
             !LCDSETUP__) #
             (!CMD__ & LCDSFDBK));
```

5.1.5.2 Pipeline Cycles

During SRAM pipeline cycles CDCHRDY stays high. It goes low after the cycle is over (HF__ removed). This is when the 82561 performs read ahead or write behind operations. Read ahead means that the next double word of data is copied from the local memory into the data latches in anticipation of the next two requests. Write behind means that two words of data are first latched in the data latches and then the 82561 copies them (two words at a time) into the local memory. The memory address in this mode is provided by the host address register, which is incremented by one after every 82561 transfer. To change the direction of the transfer the 82561 Host mode register should be accessed first and its bit 1 changed.

```
LPIPECYC = (LCDSFDBK & L561WIN__ & RAM &
           PIPELINE);
```



*CDHRDY is Pulled low only if the next CPU request is to the board (while the local bus cycle is being completed).

5.2 Memory Subsystem

The memory subsystem consists of the network address PROM (U2), the low bank of SRAM (U17 and U27),the high bank of SRAM (U18 and U28),the data bus transcievers (U11 and U6), an optional EPROM (U10), part of a PAL (U15), and the controller (U7).

All controls are generated by the 82561 except for chip select for the PROM and the EPROM. Note that the second term of the "PROMCS" is for recognizing the host's request to access the POS identification registers that reside in locations 16 and 17 of the PROM.

```
EPROMCS_  = ! (!GCS_&!IORD_&!PROM&L561WIN_);
PROMCS_   = ! ((!GCS_&!IORD_) & (PROM&L561WIN_ #
              !LA1&!LA2&!LCDSETUP_));
LCDSETUP  = ! (LCDSETUP_);
```

The data bus transceivers isolate the low- and high-word data paths of the local SRAM. This is needed because during pipeline read ahead or write behind operations the accesses to the SRAM are double word wide.

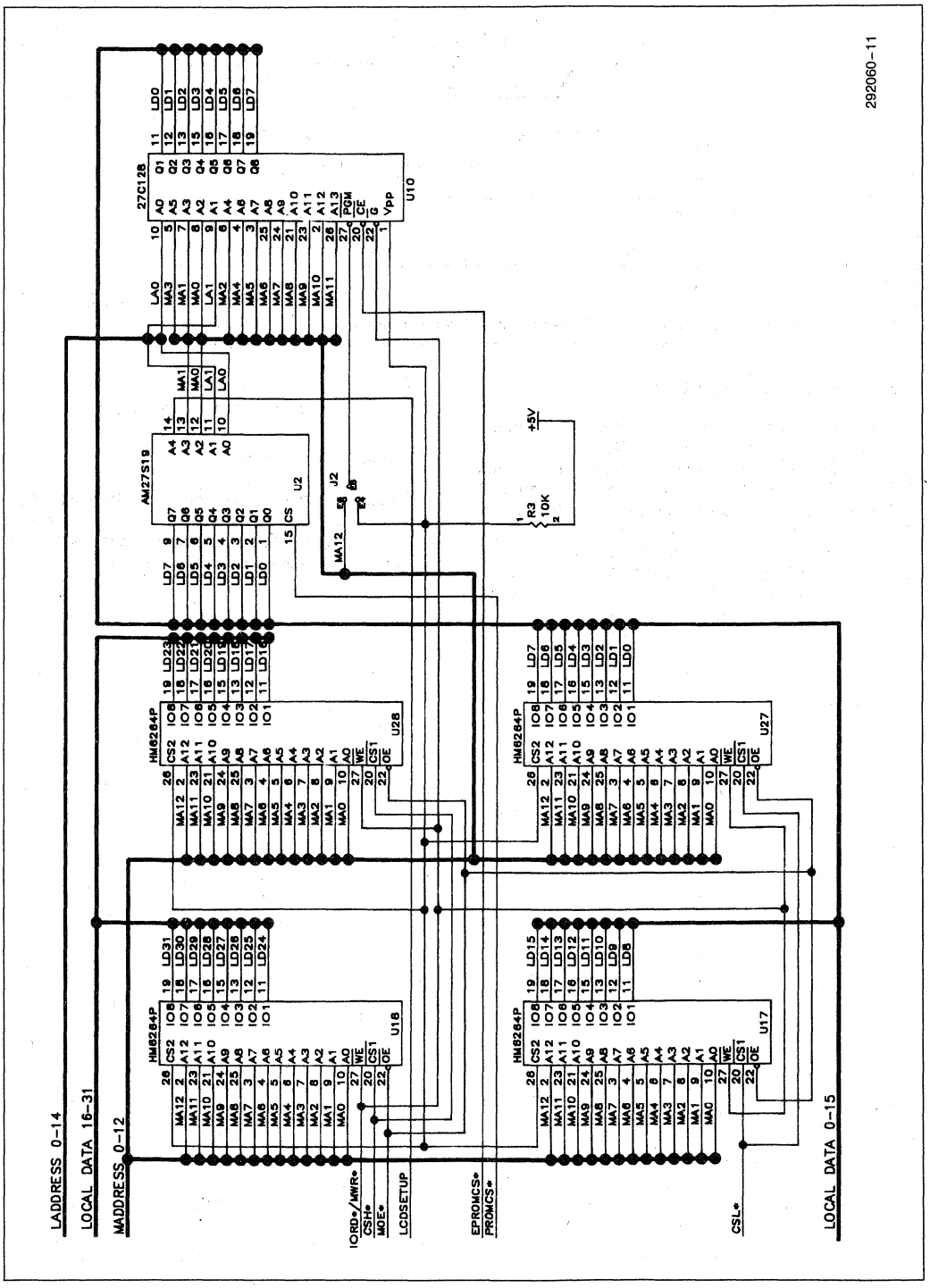


Figure 2h. PS592E-16 Digital Assembly Schematics (Continued)

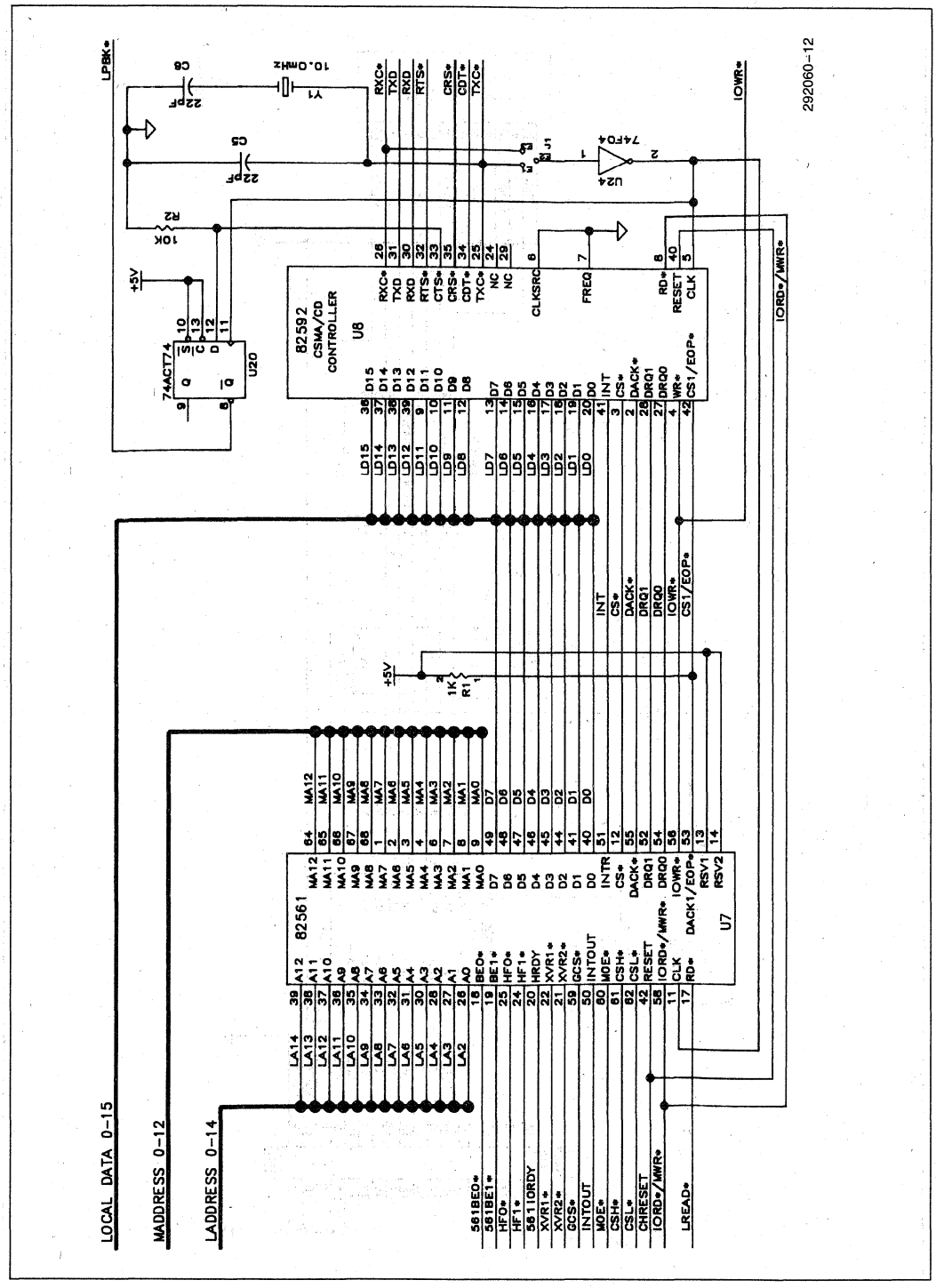


Figure 2i. PS592E-16 Digital Assembly Schematics (Continued)

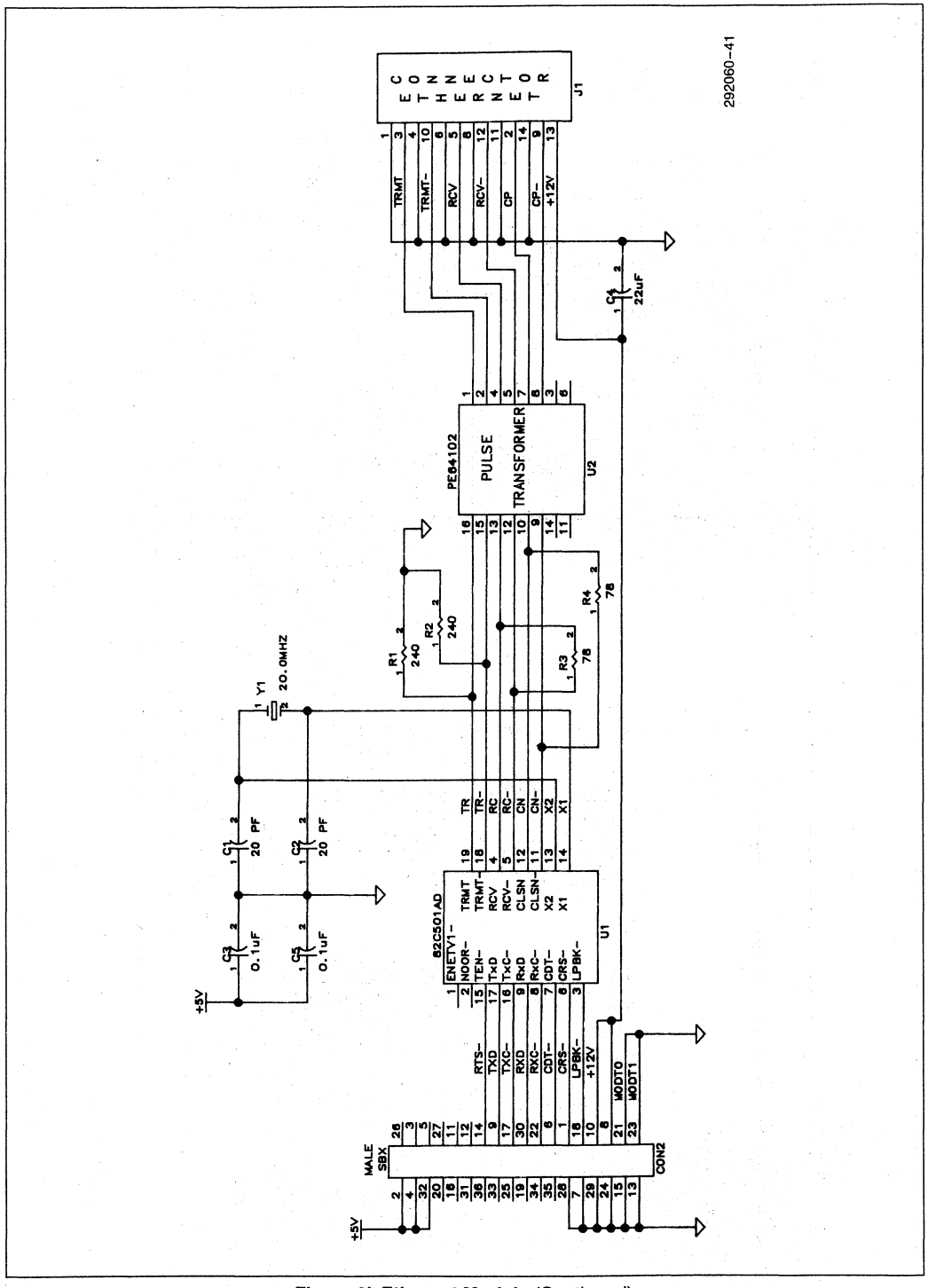


Figure 2j. Ethernet Module (Continued)

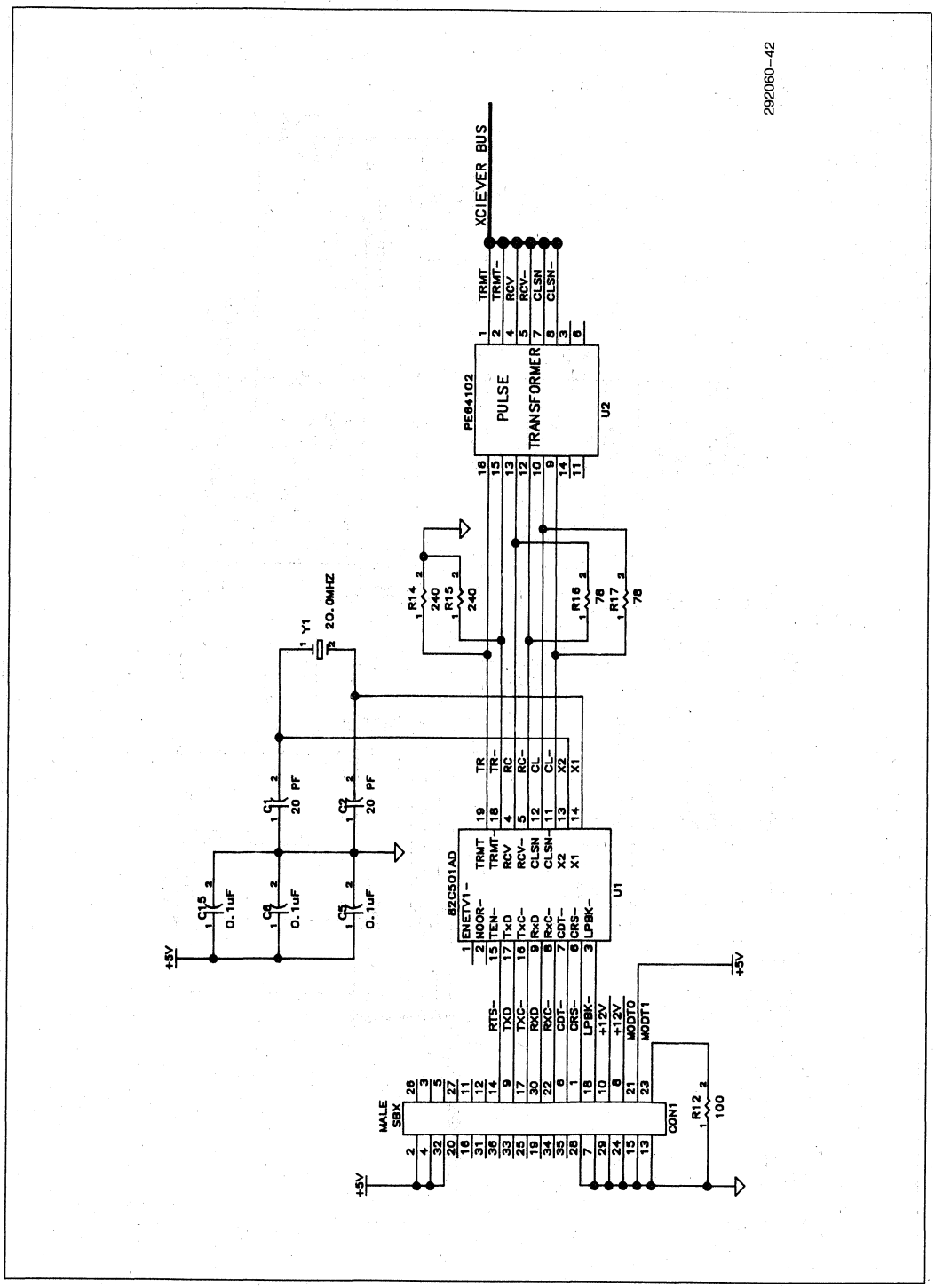


Figure 2k. Cheapernet Module

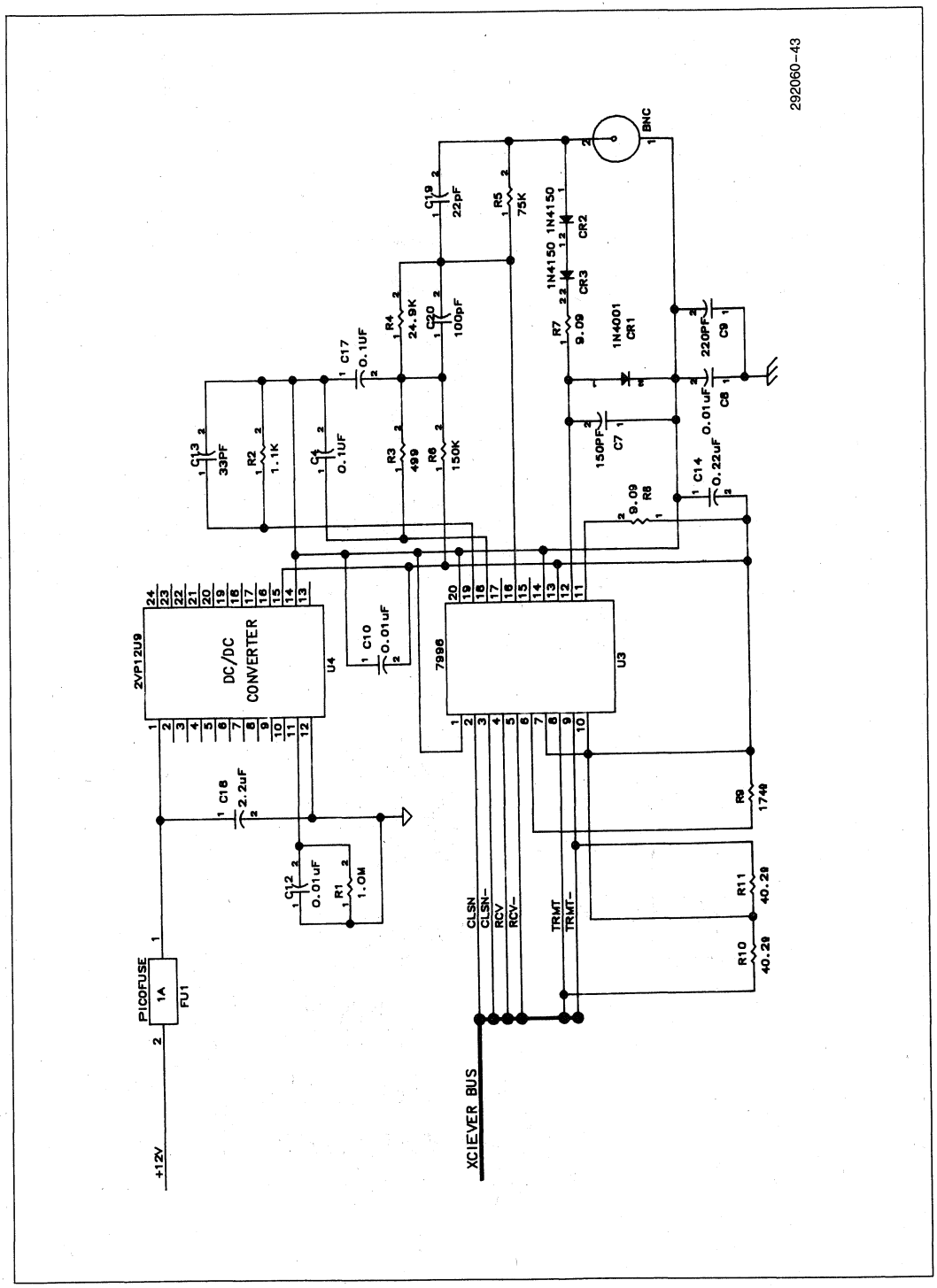


Figure 2l. Cheapernet Module (Continued)

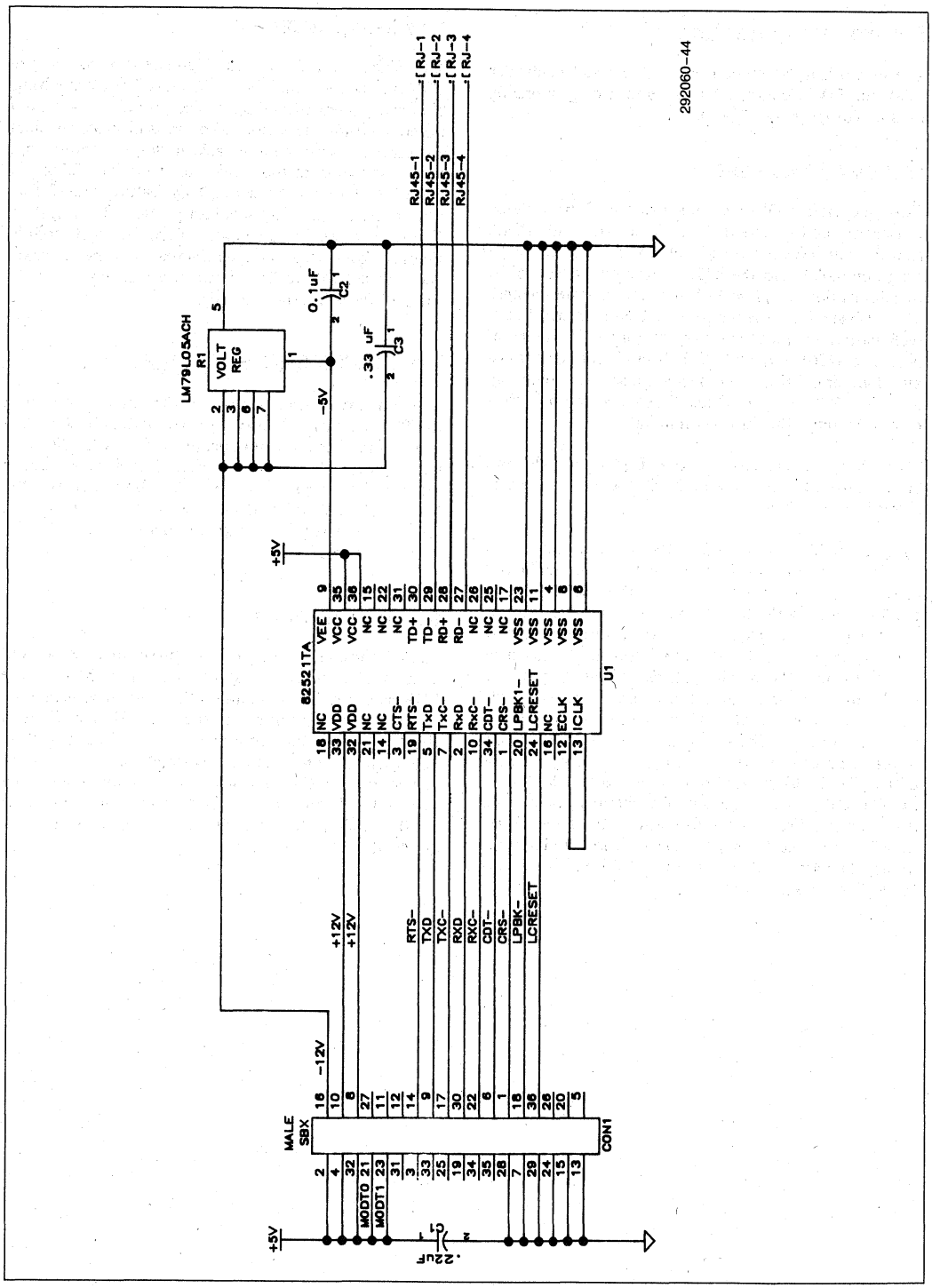


Figure 2m. Twisted Pair Ethernet Module

5.3 Network Interface

The network interface consists of the LAN controller (U8), the DMA controller (U7), and a plug-in analog module (Ethernet or C-Net).

5.3.1 DMA TRANSFERS

Two independent DMA channels of the 82561 are used to transmit and receive frames. Each word-wide DMA transfer can have a duration of 200 to 500 ns (82561 programmable), and the 82561 alternately accesses the two banks (low and high) of memory. Autoretransmit, back-to-back frame reception, and bad receive buffer reclamation are all performed without CPU intervention. The 82561 in the 82592 TCI mode supports transmit chanining. DMA channels are also used to configure the 82592 and to read its 69 bytes of internal information through the dump command.

The arbitration between the two DMA requests, and between the host request and the DMA request are performed by the 82561.

Transmit buffer size (including the configuration block) is about four kilobytes. It stops 256 bytes before the end of the adapter memory. The last 256 bytes of address are for accessing the 82561 and the command register. The receive memory space is 28 kB, arranged as a ring and managed by the 82561. The lower limit register of the 82561 hold the starting address of the ring and its Upper limit register holds the ending address of the ring. The 82561 performs the wraparound without CPU intervention. The stop register of the 82561 points to the last receive buffer location processed by the host CPU. The 82561 generates an interrupt to the host if the Current address register of a channel reaches the stop register. For more information about the DMA transfers and the format of the transmit and receive frame, the user is referred to the *82560/82561 Technical Reference Manual.*

5.3.2 SERIAL INTERFACE

The 82592 is used in the high-speed mode as the CSMA/CD controller. The 82C501AD performs Manchester encoding/decoding; it also provides a watchdog timer, collision detection, and transmit/receive clock generation. Using the loopback modes, the transmitted data is rerouted to the receive path (at the 82592, the 82C501AD, or on the wire). This feature is useful for testing the nonphysical medium portion of the system. Figure 2j shows the schematics of the Ethernet module. Figures 2k and 2l show the schematics of the Cheapernet module. Figure 2m shows the schematics of the Twisted Pair Ethernet module.

6.0 SOFTWARE EXAMPLES

The software examples given in this chapter are from the PS592E Novell Netware driver written by Joe Dragony, Intel Data Communications Technical Marketing Engineer. A brief description of each procedure is followed by exerpts from the code. The driver uses the Internetwork Packet Exchange (IPX) protocol and serves as the software interface between the PS592E hardware and IPX.

6.1 Declarations

Table 1 shows the declaration of program variables and equates of program constants including those of the POS registers. It also includes the data structures. One of these data structures is the ECB (Event Control Block). Information concerning transmit and receive operations is communicated between the IPX and the driver by using this data structure. Another data structure is the 82561 register ports which start at offset 7F00 h. This section is included to help the reader understand the rest of the code.

Table 1

```
$mod186

;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;
;       !!!!!!!!!!!!! FOR EVALUATION PURPOSES ONLY !!!!!!!!!!!!!
;
;       This code is given free of charge as an example of a driver for the
;       PS592E Demonstration Board. No warranties are given as to its
;       suitability for any purpose other than demonstration of the
;       PS592E Demonstration Board. This code is specifically NOT
;       represented as a commercial quality driver.
;
;       !!!!!!!!!!!!! FOR EVALUATION PURPOSES ONLY !!!!!!!!!!!!!
;
;       NetWare(R) Driver for the PS592E Evaluation Board
;
;       Written by Joe Dragony  DFG Technical Marketing
;
;       REVISION 0.00
;
;       Last revision: Date 03-08-89   Time 12:30
;
;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv


%*define(slow) local label (
    jmp    short %label
%label:
)


%*define(wordcopy) (
    shr  cx, 1
    rep  movsw
)


%*define(inc32 m) (
    add word ptr %m[0], 1
    adc word ptr %m[2], 0
)


%*define(validate r) local label (
    cmp  %r, 7000h
    jb   %label
    sub  %r, 7000h
%label:
)

    name    PS592EDriver

CGroup      group    Code, PSinit

    assume    cs: CGroup, ds: CGroup

Code    segment word public 'CODE'

    public    DriverSendPacket
    public    DriverBroadcastPacket
    public    DriverOpenSocket
    public    DriverCloseSocket
    public    DriverPoll
    public    DriverCancelRequest
    public    DriverDisconnect
    public    SDriverConfiguration

    public    LANOptionName
```

292060-13

Table 1 (Continued)

```
    extrn     IPXGetECB: NEAR
    extrn     IPXReturnECB: NEAR
    extrn     IPXReceivePacket: NEAR
    extrn     IPXReceivePacketEnabled: NEAR
    extrn     IPXHoldEvent: NEAR
    extrn     IPXServiceEvents: NEAR
    extrn     IPXIntervalMarker: word
    extrn     MaxPhysPacketSize: word
    extrn     ReadWriteCycles: byte
    extrn     IPXStartCriticalSection: NEAR
    extrn     IPXEndCriticalSection: NEAR


;^^^^^^^^^^^^^^^^^^^^;
;                    ;
;    Equates         ;
;                    ;
;vvvvvvvvvvvvvvvvvvvv;

CR                          equ  0Dh
LF                          equ  0Ah
TRUE                        equ  1
FALSE                       equ  0
TransmitHardwareFailure     equ  0FFh
PacketUnDeliverable         equ  0FEh
PacketOverflow              equ  0FDh
ECBProcessing               equ  0FAh
TxTimeOutTicks              equ  10
ExtInterrupt                equ  01h     ;mask for checking for Ext 592 events
TxChannel                   equ  30h     ;mask for checking for Tx DMA events
RxChannel                   equ  0Ch     ;mask for checking for Rx DMA events
HimmIntMask                 equ  03Dh    ;mask to check for any 561 interrupt


;  8259  definitions

PriIntControlPort       equ     020h
PriIntMaskPort          equ     021h    ;for primary 8259A
SecIntControlPort       equ     0A0h
SecIntMaskPort          equ     0A1h    ;for secondary 8259A
EOI                     equ     020h


; 82592 Commands

C_NOP     equ  00h
C_SWP1    equ  10h
C_SELRST  equ  0Fh
C_SWP0    equ  01h
C_IASET   equ  11h
C_CONFIG  equ  12h
C_MCSET   equ  13h
C_TX      equ  14h
C_TDR     equ  05h
C_DUMP    equ  16h
C_DIAG    equ  07h
C_RXENB   equ  08h
C_ALTBUF  equ  09h
C_RXDISB  equ  0Ah
C_STPRX   equ  0Bh
C_RETX    equ  1Ch
C_ABORT   equ  0Dh
C_RST     equ  0Eh
C_RLSPTR  equ  0Fh
C_FIXPTR  equ  1Fh
C_INTACK  equ  80h

;Adapter Setup Constants
```

292060-14

Table 1 (Continued)

```
POSPort    equ  96h            ;This port enables cardsetup for each slot
CardIDLo   equ  100h           ;Card ID low byte address
CardIDHi   equ  101h           ;Card ID high byte address
IDValLo    equ  0F9h           ;Card ID low byte value
IDValHi    equ  60h            ;Card ID high byte value
POSCnf     equ  102h           ;POS configuration byte address


;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;                               ;
;    Data Structures            ;
;                               ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

even
hardware_structure    struc
    io_addr1          dw    ?
    io_range1         dw    ?
    io_addr2          dw    ?
    io_range2         dw    ?
    mem_addr1         dw    ?
    mem_range1        dw    ?
    mem_addr2         dw    ?
    mem_range2        dw    ?
    int_used1         db    ?
    int_line1         db    ?
    int_used2         db    ?
    int_line2         db    ?
    dma_used1         db    ?
    dma_chan1         db    ?
    dma_used2         db    ?
    dma_chan2         db    ?
hardware_structure    ends

ecb_structure     struc
    link                          dd   0
    esr_address                   dd   0
    in_use                        db   0
    completion_code               db   0
    socket_number                 dw   0
    ipx_workspace                 db   4   dup (0)
    transmitting                  db   0
    driver_workspace              db   11  dup (0)
    immediate_address             db   6   dup (0)
    fragment_count                dw   1
    fragment_descriptor_list      db   6   dup (?)
ecb_structure     ends

fragment_descriptor    struc
    fragment_address   dd   ?
    fragment_length    dw   ?
fragment_descriptor    ends

rx_buf_structure     struc
    rx_dest_addr          db   6 dup (?)
    rx_source_addr        db   6 dup (?)
    rx_physical_length    dw   ?
    rx_checksum           dw   ?
    rx_length             dw   ?
    rx_tran_control       db   ?
    rx_hdr_type           db   ?
    rx_dest_net           db   4 dup (?)
    rx_dest_node          db   6 dup (?)
    rx_dest_socket        dw   ?
    rx_source_net         db   4 dup (?)
    rx_source_node        db   6 dup (?)
    rx_source_socket      dw   ?
rx_buf_structure     ends
```

292060-15

Table 1 (Continued)

```
ipx_header_structure    struc
    checksum             dw   ?
    packet_length        dw   ?
    transport_control    db   ?
    packet_type          db   ?
    destination_network  db   4 dup (?)
    destination_node     db   6 dup (?)
    destination_socket   dw   ?
    source_network       db   4 dup (?)
    source_node          db   6 dup (?)
    source_socket        dw   ?
ipx_header_structure    ends

adapter_structure       struc
    mem_space            db  07F00h dup (?)
    res_loc_0            dd  3 dup (?)
    port0                db  ?
    command_reg          db  3 dup (?)
    res_loc_4            dd  ?
    port1                dd  ?
    res_loc_6            dd  2 dup (?)
    semaphore            dd  8 dup (?)
    master_mode          dd  ?
    id_reg               dd  ?
    control_reg          dd  ?
    res_loc_13           dd  ?
    stat1_588_reg        dd  ?
    stat2_588_reg        dd  ?
    int_ctrl_stat_reg    dd  ?
    himm_int_mask_reg    dd  ?
    res_loc_18           dd  ?
    res_loc_19           dd  ?
    select_reg           dd  2 dup (?)
    host_addr_reg        dd  3 dup (?)
    dma_mode_reg         dd  ?
    b_c_addr0_reg        dd  3 dup (?)
    dma_ctrl0_reg        dd  ?
    lo_limit0_reg        dd  3 dup (?)
    rx_temp0_reg         dd  ?
    up_limit0_reg        dd  3 dup (?)
    res_loc_2b           dd  ?
    stop0_reg            dd  3 dup (?)
    host_mode_reg        dd  ?
    b_c_addr1_reg        dd  3 dup (?)
    dma_ctrl1_reg        dd  ?
    lo_limit1_reg        dd  3 dup (?)
    rx_temp1_reg         dd  ?
    up_limit1_reg        dd  3 dup (?)
    res_loc_3b           dd  ?
    stop1_reg            dd  3 dup (?)
    res_loc_3f           dd  ?
adapter_structure       ends
```

292060-16

Table 1 (Continued)

```
;^^^^^^^^^^^^^^^^^^^^^^^^^;
;                         ;
;    Variables            ;
;                         ;
;vvvvvvvvvvvvvvvvvvvvvvvvv;


;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;        TRANSMIT                ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

even
send_list        dd  0         ;points to list of ECBs to be sent
tx_start_time    dw  0
tx_byte_cnt      dw  0     ;IPX packet length plus header length
tx_buf_offset    dw  0
tx_buf_head      dw  0
padding          dw  0
loop_frag_cnt    dw  0
even_up_tx       db  0
tx_active_flag   db  0

;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;        RECEIVE                 ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

even
rx_ecb           dd  0
rx_buf_head      dw  0     ;used during rx list generation
new_stop_val     dw  0
rx_buf_ptr       dw  0
curr_rx_length   dw  0
rx_frag_count    dw  0
wrapped_frame    dw  0
pre_wrap_count   dw  0
misaligned       dw  0
split_fragment   dw  0

;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;        GENERAL                 ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

adapter_base        dw  0
config              dw  0
config_block        db  0Fh,00h,48h,00h,0A6h,00h,60h,00h,0F2h
                    db  00h,00h,40h,0FFh,00h,3Fh,87h,014h,0FFh
at_flag             dw  0
irq_array           db  0Ch, 09h, 07h, 03h
mem_array           dw  0C000h, 0C800h, 0D000h, 0D800h
int_mask_reg        dw  0
old_irq_vector      dd  0
int_vector_addr     dw  0
irq_channel         db  0
int_mask            db  0
int_unmask          db  0
num_of_slots        db  0
curr_slot           db  0
pos_byte            db  0
himm_int_mask       db  0
config_irq_loc      db  0
config_mem_loc      dw  0
event_code          db  0
result1             db  0
result2             db  0
result3             db  0
byte_cnt_hi         db  0
byte_cnt_lo         db  0
```

292060-17

Table 1 (Continued)

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;                                       ;
;    Define Hardware Configuration      ;
;                                       ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

ConfigurationID         db    'NetWareDriverLAN WS   '

SDriverConfiguration     LABEL      byte

reserved1                db  4 dup (0)
node_addr                db  6 dup (0)
reserved2                db  1            ;non-zero means is a real driver.
node_addr_type           db  0            ;address is determined at initialization
max_data_size            dw  1024         ;largest read data request will handle
lan_desc_offset          dw  LANOptionName
lan_hardware_id          db  0AAh         ;Bogus Type Code
transport_time           dw  1            ;transport time
reserved_3               db  11 dup (0)
major_version            db  01h          ;Bogus version number
minor_version            db  00h
flag_bits                db  0
selected_configuration   db  0            ;board configuration (int., IO add., etc.)
number_of_configs        db  01
config_pointers          dw  configuration0

LANOptionName       db    'Intel PS592E Evaluation Driver$'

configuration0  dw   4 dup (0)    ;IO ports and ranges
        db  0
        dw  0, 0           ;memory decode
        db  0
        dw  2 dup (0)          ;memory decode (secondary, not used)
        db  4 dup (0)          ;interrupt level
        db  4 dup (0)          ;DMA
        db  2 dup (0)

Count  dw  0FFFFh

        db  'Self Configuring Adapter            ', 0


;^^^^^^^^^^^^^^^^^^^^^^;
;                      ;
;    Error Counters    ;
;                      ;
;vvvvvvvvvvvvvvvvvvvvvv;

Public DriverDiagnosticTable,DriverDiagnosticText

DriverDiagnosticTable  LABEL  byte

DriverDebugCount         dw    DriverDebugEnd-DriverDiagnosticTable
DriverVersion            db   01,00
StatisticsVersion        db   01,00
TotalTxPacketCount       dw   0,0
TotalRxPacketCount       dw   0,0
NoECBAvailableCount      dw   0
PacketTxTooBigCount      dw   -1       ;not used
PacketTxTooSmallCount    dw   -1       ;not used
PacketRxOverflowCount    dw   0
PacketRxTooBigCount      dw   0
PacketRxTooSmallCount    dw   0
PacketTxMiscErrorCount   dw   -1       ;not used
PacketRxMiscErrorCount   dw   -1       ;not used
RetryTxCount             dw   0
ChecksumErrorCount       dw   -1       ;not used
HardwareRxMismatchCount  dw   0
NumberOfCustomVariables  dw   (DriverDiagnosticText-DriverDebugEnd1)/2

DriverDebugEnd1  LABEL    byte
```

292060-18

Table 1 (Continued)

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;                                       ;
;    Driver Specific Error Counters     ;
;                                       ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

rx_errors          dw  0
underruns          dw  0
no_cts             dw  0
no_crs             dw  0
rx_aborts          dw  0
no_590_int         dw  0
false_590_int      dw  0
false_rx_int       dw  0
false_tx_int       dw  0
lost_rx            dw  0
stop_tx            dw  0
rx_disb_failure    dw  0
tx_int_count       dw  0
rx_buff_ovflw      dw  0
tx_timeout         dw  0

DriverDiagnosticText     LABEL     byte

    db   'RxErrorCount',0
    db   'UnderrunCount',0
    db   'LostCTSCount',0
    db   'LostCRSCount',0
    db   'RxAbortCount',0
    db   'No590InterruptCount',0
    db   'False590InterruptCount',0
    db   'FalseRxInterruptCount',0
    db   'FalseTxInterruptCount',0
    db   'LostOurReceiverCount',0
    db   'QuitTransmittingCount',0
    db   'RxDisableFailureCount',0
    db   'TxIntCount',0
    db   'ReceiveBufferOverflow',0
    db   'TxTimeoutErrorCount',0

    db   0,0

DriverDebugEnd     LABEL   word
```

292060-19

6.2 Identification

The first step in the initialization is the identification of the system and the board. If no mismatch is found, then the POS configuration register is read. The PS592E identification number (assigned by IBM) is 60F9. If mismatch is found or no card enable bit is found, then an error message is given and the initialization routine is exited. If no problem is found, then the "Set Interrupt Vector" routine is called to set the vector to the address of the interrupt routine and save the old vector. Table 2 contains the code for the identification process.

Table 2

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;                                                                        ;
;   Driver Initialize                                                    ;
;                                                                        ;
;   assumes:                                                             ;
;     DS, ES are set to CGroup (== CS)                                   ;
;     DI points to where to stuff node address                           ;
;     Interrupts are ENABLED                                             ;
;     The Real Time Ticks variable is being set, and the                 ;
;     entire AES system is initialized.                                  ;
;     No registers or flags need to be preserved                         ;
;   returns:                                                             ;
;     If initialization is done OK:                                      ;
;          AX has a 0                                                    ;
;     If board malfunction:                                              ;
;          AX gets offset (in CGroup) of '$'-terminated error string     ;
;                                                                        ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

DriverInitialize PROC     NEAR
    mov  MaxPhysPacketSize, 1024
    cld

;First, find out the system we are in....
;Model 50 = 0FC04h  (3 slots)
;Model 60 = 0FC05h  (8 slots)
;Model 70 = ?????h  (3 slots)
;Model 80 = 0F800h  (8 slots)

    mov  ah, 0C0h
    int  15h
    cli
    jc   cant_get_id
    mov  ah, es:[bx+1]      ;model byte
    mov  al, es:[bx+2]      ;submodel byte
    cmp  ax, 0FC04h         ;is it a model 50?
    jnz  eight_slots
    mov  num_of_slots, 03h ;model 50 just has three slots
    jmp  slots_set
eight_slots:
    mov  num_of_slots, 08h ;models 60 and 80
cant_get_id:                ;for now, default to eight slots
slots_set:
    add  num_of_slots, 08h ;add 08h so we can use this in compare operation
    mov  dl, num_of_slots

;Next, find out which slot of the system we are in....

    mov  curr_slot, 08h     ;slot number 1
next_slot:
    mov  al, curr_slot
    out  POSPort, al        ;Output slot ID to Channel Position Select Reg.
    mov  dx, CardIDLo
    in   al, dx             ;Read the low byte of card ID.
    cmp  al, IDValLo        ;If it matches our low byte...
    je   next_byte          ;check the next byte
    inc  curr_slot          ;Otherwise, index to the next slot and check there.
    cmp  curr_slot, dl      ;Make sure we don't check beyond num_of_slots.
    je   card_not_found     ;
    jmp  next_slot          ;Check next slot.
next_byte:
    mov  dx, CardIDHi
    in   al, dx             ;Read high byte.
    cmp  al, IDValHi        ;Compare with expected high byte.
    je   its_us             ;If it's a match we can continue.
    inc  curr_slot          ;Otherwise, index to the next slot and check there.
    cmp  curr_slot, dl      ;Make sure we don't check beyond nim_of_slots.
    je   card_not_found     ;
    jmp  next_slot          ;Check next slot.
```

292060-20

Table 2 (Continued)

```
card_not_found:
    xor  al, al
    out  POSPort, al        ;take system out of setup
    mov  ax, offset cgroup:no_card_message
    jmp  init_exit

;Next, read the POS register on the PS592E to determine setup and
;fill in the variables for later use...

bogus_pos_data:
    mov  ax, offset cgroup:pos_data_error_message
    jmp  init_exit

its_us:
    mov  dx, POSCnf
    in   al, dx
    mov  pos_byte, al       ;save the value in a register
    xor  al, al
    out  POSPort, al
    mov  al, pos_byte
    mov  bx, 0
try_next_loc:
    shl  al, 1
    jc   set_int
    inc  bx
    cmp  bx, 03h            ;if bx reaches 3 without finding a set POS bit
    ja   bogus_pos_data     ;then POS register was 1) not initialized or
    jmp  try_next_loc       ;2) not read correctly so abort
set_int:
    mov  al, irq_array[bx]
    mov  irq_channel, al
    mov  al, pos_byte
    cbw
    and  ax, 06h              ;remove all extraneous bits
    mov  bx, ax               ;bx will index into array of memory offsets
    mov  ax, mem_array[bx]    ;get the value into ax
    mov  adapter_base, ax  ;set the variable


;set up registers then call set_vector
    push di
    mov  al, irq_channel
    mov  bx, OFFSET CGroup:DriverISR
    call SetInterruptVector
    pop  di
```

292060-21

6.3 Hardware Initialization

The 82561 configuration registers are set appropriately (enable memory mapped accesses, DMA 82588 TCI, double host bus mode, etc.). The command register is set to read the address PROM and store its value. Then the command register is set to enable RAM access. Then the transmit channel is set and 82592 is issued two configure commands. The first one puts the 82592 into 16 bit mode and the second one does the following:

— Puts the 82592 in High Speed Mode to support Ethernet serial bit rates.

— All network parameters are set up for default Ethernet values.

Table 3 shows the intialization code.

Table 3

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;
;    82561 Initialization section
;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

    mov  ax, adapter_base
    mov  es, ax
    mov  es:byte ptr id_reg, 00h            ;reset the 82561
%slow
    mov  es:byte ptr master_mode, 031h  ;DMA has priority, double bus width
    mov  es:byte ptr control_reg, 01h   ;late write, 0 i/o/mem wait state
    mov  es:byte ptr himm_int_mask_reg, 082h ;high assert,edge trig,drive,int on tx
    mov  es:byte ptr dma_mode_reg, 040h ; 588 TCI, discard bad frames
    mov  es:byte ptr select_reg+4, 0C0h ;enable access, memory mapped,

    mov  es:byte ptr command_reg , 02h  ;enable address PROM
    mov  al, es:byte ptr mem_space
    mov  byte ptr ds:[di], al     ;read address from PROM and store in
    mov  byte ptr node_addr, al
    mov  al, es:byte ptr mem_space+1  ;local variable "node_addr"
    mov  byte ptr ds:[di+1], al
    mov  byte ptr node_addr[1], al
    mov  al, es:byte ptr mem_space+2
    mov  byte ptr ds:[di+2], al
    mov  byte ptr node_addr[2], al
    mov  al, es:byte ptr mem_space+3
    mov  byte ptr ds:[di+3], al
    mov  byte ptr node_addr[3], al
    mov  al, es:byte ptr mem_space+4
    mov  byte ptr ds:[di+4], al
    mov  byte ptr node_addr[4], al
    mov  al, es:byte ptr mem_space+5
    mov  byte ptr ds:[di+5], al
    mov  byte ptr node_addr[5], al
    mov  es:byte ptr command_reg , 01h  ;enable RAM

;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;
;    Receive Channel Initialization
;
;    Receive is set up to use channel 0. Receive buffer is 28K
;    starting at location zero of adapter memory
;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

    mov  es:byte ptr b_c_addr0_reg, 00h     ;receive buffer starts at location
    mov  es:byte ptr b_c_addr0_reg+4, 00h   ;zero in the adapter memory
    mov  es:byte ptr b_c_addr0_reg+8, 00h
    mov  es:byte ptr lo_limit0_reg, 00h     ;lo_limit0_reg points to beginning
    mov  es:byte ptr lo_limit0_reg+4, 00h   ;of adapter memory
    mov  es:byte ptr lo_limit0_reg+8, 00h
    mov  es:byte ptr up_limit0_reg, 0FFh    ;up_limit0_reg points to the last
    mov  es:byte ptr up_limit0_reg+4, 01Bh  ;word of the 28K receive buffer
    mov  es:byte ptr up_limit0_reg+8, 00h
    mov  es:byte ptr stop0_reg, 0FDh        ;stop0_reg points to the location
    mov  es:byte ptr stop0_reg+4, 01Bh      ;two words before the end of the
    mov  es:byte ptr stop0_reg+8, 00h       ;receive buffer
```

292060-22

Table 3 (Continued)

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;
;    Transmit Channel Initialization
;
;    Transmit is set up to use channel 1. Transmit buffer is 4k
;    starting directly above the receive buffer. Transmit buffer
;    stops 256 bytes before the end of adapter memory because the
;    82561 registers and control registers are mapped there.
;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

    mov  es:byte ptr b_c_addr1_reg, 00h     ;transmit buffer starts at location
    mov  es:byte ptr b_c_addr1_reg+4, 01Ch  ;7000h (word 3800h) in the adapter
    mov  es:byte ptr b_c_addr1_reg+8, 00h   ;memory
    mov  es:byte ptr lo_limit1_reg, 00h     ;lo_limit1_reg points to location
    mov  es:byte ptr lo_limit1_reg+4, 01Ch  ;7000h in the adapter memory
    mov  es:byte ptr lo_limit1_reg+8, 00h
    mov  es:byte ptr up_limit1_reg, 07Fh    ;up_limit1_reg points to the last
    mov  es:byte ptr up_limit1_reg+4, 01Fh  ;adapter memory location before the
    mov  es:byte ptr up_limit1_reg+8, 00h   ;561 mapping begins
    mov  es:byte ptr stop1_reg, 07Dh        ;stop register points to the spot
    mov  es:byte ptr stop1_reg+4, 01Fh      ;2 words before the 561 space
    mov  es:byte ptr stop1_reg+8, 00h
    mov  es:byte ptr dma_ctrl1_reg, 14h      ;enable channel


    mov  tx_buf_head, 7000h
    mov  rx_buf_head, 0000h

    mov ax, 00h
    mov di, rx_buf_head
    mov cx, 3F7Fh
    rep  stosw

    ;set up for configure command
        mov  es:byte ptr port0, C_RST
        mov  di, tx_buf_head    ;the 82592 must be given a configure command
        xor  ax, ax             ;with zero in the byte count field to put
        stosw                   ;the 82592 into 16 bit mode
        stosw
        mov  es:byte ptr port0, C_CONFIG ;configure the 82592 for 16 bit mode
        xor  cx, cx
    wide_mode_wait_loop:
        mov  es:byte ptr port0, 00h
    %slow
        mov  al, es:byte ptr port0   ;read register 0
        and  al,0DFh                 ;disregard exec bit
        cmp  al, 92h                 ;is configure finished?
        jz   do_config
        loop wide_mode_wait_loop
        mov  ax, offset cgroup:no_response_message
        jmp  init_exit

    do_config:
        mov  es:byte ptr port0, C_INTACK         ;clear interrupt in 592
        mov  es:byte ptr dma_ctrl1_reg, 04h      ;disable channel
        mov  es:byte ptr b_c_addr1_reg, 00h     ;transmit buffer starts at location
        mov  es:byte ptr b_c_addr1_reg+4, 01Ch  ;7000h (word 3800h) in the adapter
        mov  es:byte ptr b_c_addr1_reg+8, 00h   ;memory
        mov  es:byte ptr dma_ctrl1_reg, 14h      ;enable channel

        mov  si, offset cgroup:config_block
        mov  di, tx_buf_head
        mov  cx, 9
    rep movsw
        mov  es:byte ptr port0, C_CONFIG             ;configure the 82592
        xor  cx, cx
```

292060-23

Table 3 (Continued)

```
config_wait_loop:
    mov  es:byte ptr port0, 00h
%slow
    mov  al, es:byte ptr port0      ;read register 0
    and  al, 0DFh                   ;discard extraneous bits
    cmp  al, 92h                    ;is configure finished?
    jz   config_done
    loop config_wait_loop
    mov  ah, al
    mov  ax, offset cgroup:config_failure_message
    jmp  init_exit

config_done:
; clear interrupt caused by configuration
    mov  es:byte ptr port0, C_INTACK
    mov  es:byte ptr int_ctrl_stat_reg, 01h  ;clear 561 external interrupt

; do an IA_setup
    mov  es:byte ptr dma_ctrl1_reg, 04h      ;disable channel
    mov  es:byte ptr b_c_addrl_reg, 00h      ;transmit buffer starts at location
    mov  es:byte ptr b_c_addrl_reg+4, 01Ch   ;7000h (word 3800h) in the adapter
    mov  es:byte ptr b_c_addrl_reg+8, 00h    ;memory
    mov  es:byte ptr dma_ctrl1_reg, 16h      ;enable channel

    mov  di, tx_buf_head
    mov  ax, 06h    ;address byte count
    stosw
    mov  si, OFFSET CGROUP:node_addr
    mov  cx, 03h
rep movsw
    mov  es:byte ptr port0, C_IASET    ;set up the 82592 individual address
    xor  cx, cx                        ;cx is used by the loop instruction below. this
                                       ;causes the loop to be executed 64k times max
ia_wait_loop:
    mov  es:byte ptr port0, 00h
%slow
    mov  al, es:byte ptr port0
    and  al, 0DFh    ;discard extraneous bits
    cmp  al, 91h     ; is command finished?
    jz   ia_done
    loop ia_wait_loop
    mov  ah, al
    mov  ax, offset cgroup:iaset_failure_message
    jmp  init_exit

ia_done:
    mov  es:byte ptr port0, C_INTACK
    mov  es:byte ptr int_ctrl_stat_reg, 01h  ;clear 561 external interrupt
;unmask our interrupt channel
    mov  dx, int_mask_reg
    in   al, dx
    mov  bl, int_unmask
    and  al, bl
%slow
    out  dx, al
    mov  es:byte ptr himm_int_mask_reg, 04h  ;enable slave interrupt

;enable the receiver
    mov  es:byte ptr dma_ctrl0_reg, 3Eh     ;enable channel
    mov  es:byte ptr port0, C_RXENB
    xor  ax, ax
    mov  cx, 1

init_exit:
    sti                          ;make sure interrupts are enabled
    ret                          ;return control to IPX


DriverInitialize    endp
```

292060-24

6.4 Interrupt Routines

First the current status of the machine is saved and the interrupt mask bits are set appropriately. Then the type of Interrupt is identified and the control is transferred to the appropriate routine. Transmit and receive interrupts are handled by the 82561, other interrupts are passed to the CPU. Table 4 shows the Interrupt routines.

Table 4

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;
;    Interrupt Service Procedure
;
;    DMA channel 0 is the transmit channel
;
;    DMA channel 1 is the receive channel
;
;    BX is set to point to the current receive or transmit buffer head
;    after the cause of the interrupt has been determined.
;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv


DriverISR  PROC    far
public     DriverISR

    pusha
    push ds
    push es
    call IPXStartCriticalSection  ;tell AES we're busy
    mov  ax, cs
    mov  ds, ax                ;DS points to C/DGroup
    mov  es, adapter_base      ;segment of adapter memory base
    mov  dx, int_mask_reg
    in   al, dx                ;get mask state of 8259
    or   al, int_mask          ;set mask bit for our channel
    out  dx, al                ;write new mask to 8259
    mov  al, EOI
    out  PriIntControlPort, al
    cmp  at_flag, 0h           ;is our assigned interrupt in the secondary 8259?
    jz   below_8
    out  SecIntControlPort, al ;if so, clear secondary too
below_8:
    cld
    mov  al, es:byte ptr int_ctrl_stat_reg
int_poll_loop:
;   sti                        ;enable interrupts to be friendly
    test al, RxChannel         ;did I receive a frame?
    jnz  rcvd_packet_jmp
    test al, TxChannel         ;did I finish a transmit?
    jnz  sent_packet_jmp
    test al, ExtInterrupt      ;is there an error condition?
    jnz  other_588_int
    inc  false_590_int          ;unwanted interrupt
    jmp  int_exit

sent_packet_jmp:
    jmp  sent_packet

rcvd_packet_jmp:
    jmp  rcvd_packet

other_588_int:
    inc  false_590_int
    mov  es:byte ptr port0, C_INTACK
    mov  es:byte ptr int_ctrl_stat_reg, 01h
    mov  es:byte ptr port1, C_INTACK
    jmp int_exit ;
```

292060-25

6.4.1 RECEIVE

An interrupt on the receive channel is either due to receiving a frame or due to hitting the stop register. The location pointed to by the receive buffer head is examined to check if a complete frame was received. This location is initalized to FF h (by the 82561). After receiving a frame the byte count is copied to this location. Assuming the value read is not FF h (a full frame was received), the routine checks the size of the frame. If the frame is too short, too long or does not match the phsical length, "buffer crash" routine is called. In this routine the receive error count is increased, the DMA channel is disabled, the DMA address registers are reprogrammed and then the channel is reenabled. If the frame length is O.K., then routine that processes the received frame is called. At the end the buffer head is incremented to point to the beginning of the next frame in the buffer. The stop register is then updated. A check is performed to findout if a wrap around was done during the reception of the last frame. If so, then the buffer head is appropriately modified. Table 5 shows the driver code for this section.

Table 5

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;       RECEIVE EVENT              ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

buffer_crash:
    inc  rx_errors
    mov  es:byte ptr port0, C_RXDISB
    mov  es:byte ptr port1, C_SELRST
    mov  es:byte ptr dma_ctrl0_reg, 2Eh     ;disable channel
    mov  es:byte ptr b_c_addr0_reg, 00h     ;receive buffer starts at location
    mov  es:byte ptr b_c_addr0_reg+4, 00h   ;zero in the adapter memory
    mov  es:byte ptr b_c_addr0_reg+8, 00h
    mov  es:byte ptr lo_limit0_reg, 00h     ;lo_limit0_reg points to beginning
    mov  es:byte ptr lo_limit0_reg+4, 00h   ;of adapter memory
    mov  es:byte ptr lo_limit0_reg+8, 00h
    mov  es:byte ptr up_limit0_reg, 0FFh    ;up_limit0_reg points to the last
    mov  es:byte ptr up_limit0_reg+4, 01Bh  ;word of the 28K receive buffer
    mov  es:byte ptr up_limit0_reg+8, 00h
    mov  es:byte ptr stop0_reg, 0FDh        ;stop0_reg points to the location
    mov  es:byte ptr stop0_reg+4, 01Bh      ;two words before the end of the
    mov  es:byte ptr stop0_reg+8, 00h       ;receive buffer
    mov  rx_buf_head, 0000h
    mov  es:byte ptr dma_ctrl0_reg, 3Eh     ;enable channel
    mov  es:byte ptr port0, C_RXENB
    mov  es:byte ptr port0, C_SWP1
    mov  es:byte ptr port0, C_SWP0
    jmp  int_exit

false_rx:                          ;if not, increment counter and
    inc  false_rx_int
    jmp  int_exit

rcvd_packet:
    mov  ax, rx_buf_head               ;get index into rx buffer
    mov  rx_buf_ptr, ax             ;get index into rx buffer
    mov  es:byte ptr int_ctrl_stat_reg, 0Ch ;ack 561 interrupt
    mov  bx, rx_buf_head               ;get index into rx buffer
    mov  ax, es:word ptr mem_space[bx]  ;word moves are required when accessing
    mov  cx, es:word ptr mem_space[bx] + 2
    mov  ah, cl
    cmp  ax, 0FFFFh                 ;make sure we really received a frame
    jz   false_rx
do_next_frame:
    mov  curr_rx_length, ax         ;ax contains total length of the frame buffer
    add  bx, 4                      ;index bx to point to beginning of data
    dec  ax                         ;toss byte count & status
    and  al, 0feh                   ;round up
    sub  ax, 14                     ;sub length of 802.3 header
    cmp  ax, 1024 + 64
    jbe  not_too_big
    inc  PacketRxTooBigCount
    jmp  buffer_crash
not_too_big:
    cmp  ax, 30
    jae  not_too_small
    inc  PacketRxTooSmallCount
    jmp  buffer_crash
not_too_small:
    mov  dx, es:[bx].rx_length  ;get IPX length
    xchg dl, dh
    inc  dx
    and  dl, 0feh
    xchg dl, dh
    cmp  dx, es:[bx].rx_physical_length  ;same as 802.3 length ?
    je   fields_match
    jmp  buffer_crash
 fields_match:
    xchg dl, dh
    cmp  dx, 60 - 14          ; at least min length minus header
    ja   len_ok               ; yes, continue
    mov  dx, 60 - 14          ; no, round up
```

292060–26

Table 5 (Continued)

```
len_ok:
    cmp  ax, dx              ; match physical length
    jz   not_inconsistent    ; yes, continue
    inc  HardwareRxMismatchCount
    jmp  buffer_crash
not_inconsistent:
    %inc32  TotalRxPacketCount      ; Double Word Increment
    call ProcessRxFrame
    mov  ax, curr_rx_length     ;get original byte count back
    add  ax, 10                 ;add overhead bytes to receive length
    add  ax, 3                  ;round up to nearest double word
    and  al, 0FCh               ;boundary
    add  rx_buf_head, ax        ;rx_buf_head points to next frame buffer
%slow
    mov  ax, rx_buf_head
    cmp  ax, 7000h
    jb   no_wrap
    sub  ax, 7000h
    mov  rx_buf_head, ax        ;ax contains new rx_buf_head
no_wrap:
    shr  ax, 2          ;convert byte address to doubleword address
    sub  ax, 4          ;calculate the stop value
    jns  set_stop       ;check for negative value (stop was at 1st 16 bytes)
    and  ax, 1BFFh      ;mask to generate required stop value
set_stop:
    mov  new_stop_val, ax
    mov  bx, rx_buf_head                ;load bx for use as pointer
    mov  ax, word ptr es:mem_space[bx]  ;get byte count LSB
    mov  cx, word ptr es:mem_space[bx] + 2  ;get byte count MSB
    mov  ch, al                             ;combine them in cxi
    mov  ax, new_stop_val
    mov  es:byte ptr stop0_reg, al     ;update the stop register
    mov  es:byte ptr stop0_reg + 4, ah   ;by writing new values
    mov  es:byte ptr stop0_reg + 8, 00h  ;to all three bytes
    cmp  cx, 0FFFFh             ;Is there another received frame to be
    je   int_exit               ;processed?
    mov  ax, cx                 ;receive loop expects count to be in ax
    jmp  do_next_frame


int_exit:
    push cs
    pop  ds

finish_exit:
    mov  es, adapter_base
    mov  al, es:byte ptr int_ctrl_stat_reg
    test al, himm_int_mask    ;check for new interrupt
    jnz  int_pending          ;if we do, service it
    cli                         ;mask interrupts so we can unmask our
    mov  dx, int_mask_reg       ;channel without reentrancy problems
    in   al, dx                 ;get mask state of 8259
    and al, int_unmask          ;clear mask bit for our channel to
%slow                           ;enable new interrupts from adapter
    out  dx, al                 ;write new mask to 8259
    call IPXEndCriticalSection  ;tell IPX it can run
    pop  es                     ;restore machine state
    pop  ds
    popa
    iret                        ;return

too_big:
    inc  PacketRxOverflowCount
    jmp  int_exit

int_pending:
    jmp  int_poll_loop
```

292060-27

6.4.2 TRANSMIT

After acknowledging the 82561 transmit interrupt, the routine checks if the transmit flag is set. If it is set, the routine reads the status registers of the 82561. If the transmit OK bit is not set, there was a problem with the transmit. In this case, other bits are read to determine the exact cause of the problem so that the appropriate action can take place. If transmit was successfull the routine updates the retry count and returns the TX ECB to IPX with a good completion code. Table 6 shows the code for transmit.

Table 6

```
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^;
;       TRANSMIT EVENT           ;
;vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv;

sent_packet:
    cli
    mov  es:byte ptr int_ctrl_stat_reg, 30h ;ack 561 interrupt
    cmp  tx_active_flag, 0
    jz   bogus_tx_int     ;shouldn't have been transmitting
    inc  tx_int_count
    mov  al, es:byte ptr stat1_588_reg
    mov  result1, al
    mov  al, es:byte ptr stat2_588_reg
    mov  result2, al
    test al, 20h
    jz   tx_error
    mov  al, result1       ;extract the total number of retries from
    and  ax, 0Fh          ;the status register and add to retry count
    add  RetryTxCount, ax
    xor  ax, ax           ;status = 0, good transmit

FinishUpTransmit:
    les  si, send_list
    cmp  es:byte ptr [si].transmitting, TRUE
    jnz  not_active
    mov  es: [si].completion_code, al
    mov  ax, es: word ptr [si].link
    mov  word ptr send_list, ax
    mov  ax, es: word ptr [si].link + 2
    mov  word ptr send_list + 2, ax
;finish the transmit
    mov  es: [si].in_use, 0
    call IPXHoldEvent
not_active:
    push cs
    pop  ds
    mov  cx, word ptr send_list + 2
    mov  tx_active_flag, cl
    jcxz int_exit_jmp1
    mov  es, cx            ;segment of next SCB in list
    mov  si, word ptr send list    ;offset of next SCB in list
    call start_send
    jmp  finish_exit

int_exit_jmp1:
    jmp  int_exit

bogus_tx_int:
    inc  false_tx_int
    jmp  int_exit

tx_error:
    test result1, 20h    ;Max collisions??
    jnz  QuitTransmitting
    test result2, 01h    ;Tx underrun??
    jz   lost_cts
    inc  underruns
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit
lost_cts:
    test result2, 02h    ;did we lose clear to send??
    jz   lost_crs
    inc  no_cts
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit
lost_crs:
    test result2, 04h    ;did we lose carrier sense??
    jz   late_coll
    inc  no_crs
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit
```

292060-28

Table 6 (Continued)

```
late_coll:
    test result2, 08h     ;did we have a late collision?
    jz   hmmm
    inc  no_crs
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit
hmmm:
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit

QuitTransmitting:
    add  RetryTxCount, 0Fh
    inc  stop_tx
    mov  al, TransmitHardwareFailure
    jmp  FinishUpTransmit

DriverISR    endp
```

292060-29

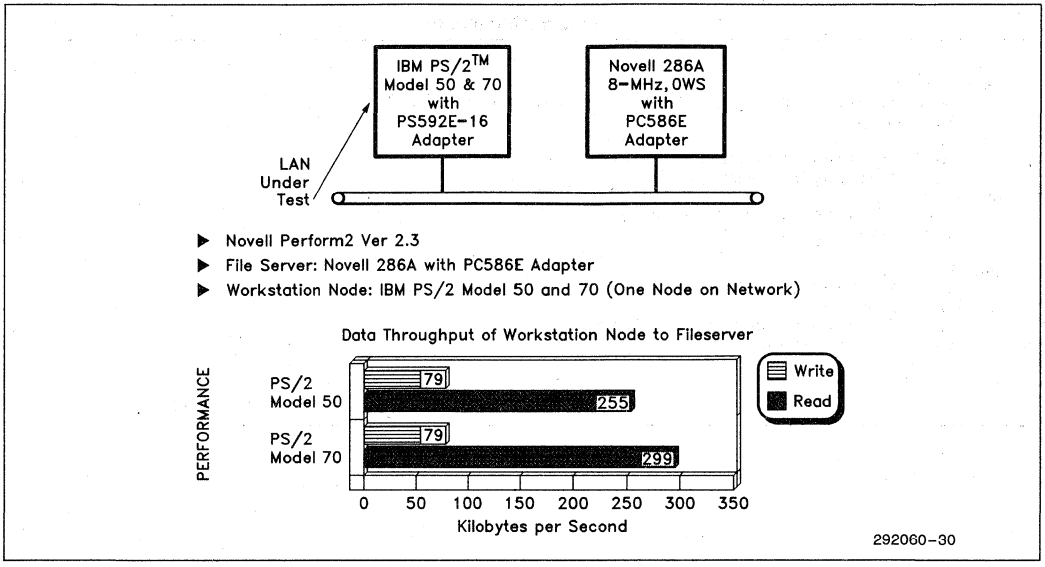


Figure 3. PS5292E/16 Performance Benchmarks

7.0 PERFORMANCE

The PS592E provides very high performance compared to many commercial adapters. The result of the performance experiment is shown in Figure 2. The PS592E can perform default cycles (200 ns) on the Micro Channel. This is done when the board is configured for pipeline mode. Without this feature each host access would take about 500 ns, resulting in less efficient use of the host bus bandwidth. In addition to reducing the host bandwidth consumption, performing the default cycles (200 ns) on the Micro Channel improves the network performance by about 10% over non-default cycles. The experiment was performed using the standard Perform2 with one station and the file server (no collision).

8.0 PS592E-32

This chapter shows how the PS592E-16 design can be modified to provide an adapter for the 32-bit Micro Channel I/O slots (Models 70 and 80). The design is so similar to the 16-bit design that only the differences (PAL equations and schematics) are specified.

8.1 Architecture

The data path between the Micro Channel and the data latches/transceivers is 32-bit wide. The low word (D0–D15) goes to U32 and U22 and the high word (D16–D31) goes to U26 and U16. The component count is the same as in the 16-bit design. The complete set of the PS592E-32 schematic is given.

8.2 Schematics

Besides the 32-bit data path, two other minor modifications are made to the schematics.

1. U5 is changed from a 3-input NAND gate to a 4-input NAND gate. Half of it is used to generate the CDDS32__ signal (indicating 32-bit data transfers) to the Micro Channel. An extra input is added to the CDDS16__ logic (PIPELINE__, pin 11 of U1). Only the memory transfers in the pipeline mode can be 32-bit wide. Nonpipeline memory transfers remain 16-bit wide on the low word (D0–D15), and non-memory transfers remain 8-bit wide on the low Byte (D0–D7).

NOTE:

1. Our Netware driver uses the pipeline mode to transfer the frame data and nonpipeline mode to transfer other frame information (byte count, status, etc.). Since most of the time is spent in transfering frame data, the restriction (pipeline mode for data transfer only), does not reduce performance. However, if a designer would like to extend the 32-bit transfers to nonpipeline cycles he can do so. To do this, (BE0__ # BE1__) and (BE2__ + BE3__) should be generated on the board and latched (BEn__ are the Byte enable signals on the Micro Channel).

2. CBA2__ signal (which was generated in U31) is removed and CBA1__ is renamed CBA__. CBA2__ was the latch signal for the high word data latch/transciever (U26, U16) in the 16-bit version. It is not needed because the data is latched 32 bits at a time. The extra pins of U31 are used to generate SWEN__ in the PAL. SWEN__ is the enable signal to the local bus transceivers (U6 and U11).

3. Only the low 24 bits of address are decoded (no change from the PS592E-16). If the software code is that of the 80386, then the higher 8 bits should also be decoded and the result should be latched.

8.3 PAL Equations

8.3.1 HOST REQUEST

The equation for unqualified HF0 is modified. A request to the 82561 is generated each time the host CPU requests access to local memory (double word). Thus, LA1 is removed as the qualifier.

```
HF0U = ((LRDWR & LCDSFDBK) & ((!L561WIN__ &
       !LSBHE__) # (L561WIN__)) #
       (!LCDSETUP__ & LRDWR);
```

8.3.2 DATA LATCH/TRANSCEIVER CONTROLS

CBA1__ is renamed CBA__, and LA1 is removed from the equation, because the data is latched 32 bits at a time (independent of LA1). CBA2__ is removed.

```
CBA__ = ! (!CMD__ & PIPELINE & !LWRITE__ & RAM &
          L561WIN__ & LCDSFDBK);
```

For the same reason, data latch/transciever enable signals are modified to be independent of LA1 during pipeline memory accesses.

```
G1__ = ! ((!MEMREQ__ & PIPELINE & !LREAD__ &
          !CMD__)                                  #
          (!XVR1__ & !CMD__)                       #
          (!IORD__ & PIPELINE & !XVR2__));

G2__ = ! ((!HF0__ & HF1__ & PIPELINE & !LREAD__ &
          !CMD__)                                  #
          (!HF0__ & HF1__ & !PIPELINE & !XVR2__ &
          !CMD__)                                  #
          (!IORD__ & PIPELINE & !XVR2__));
```

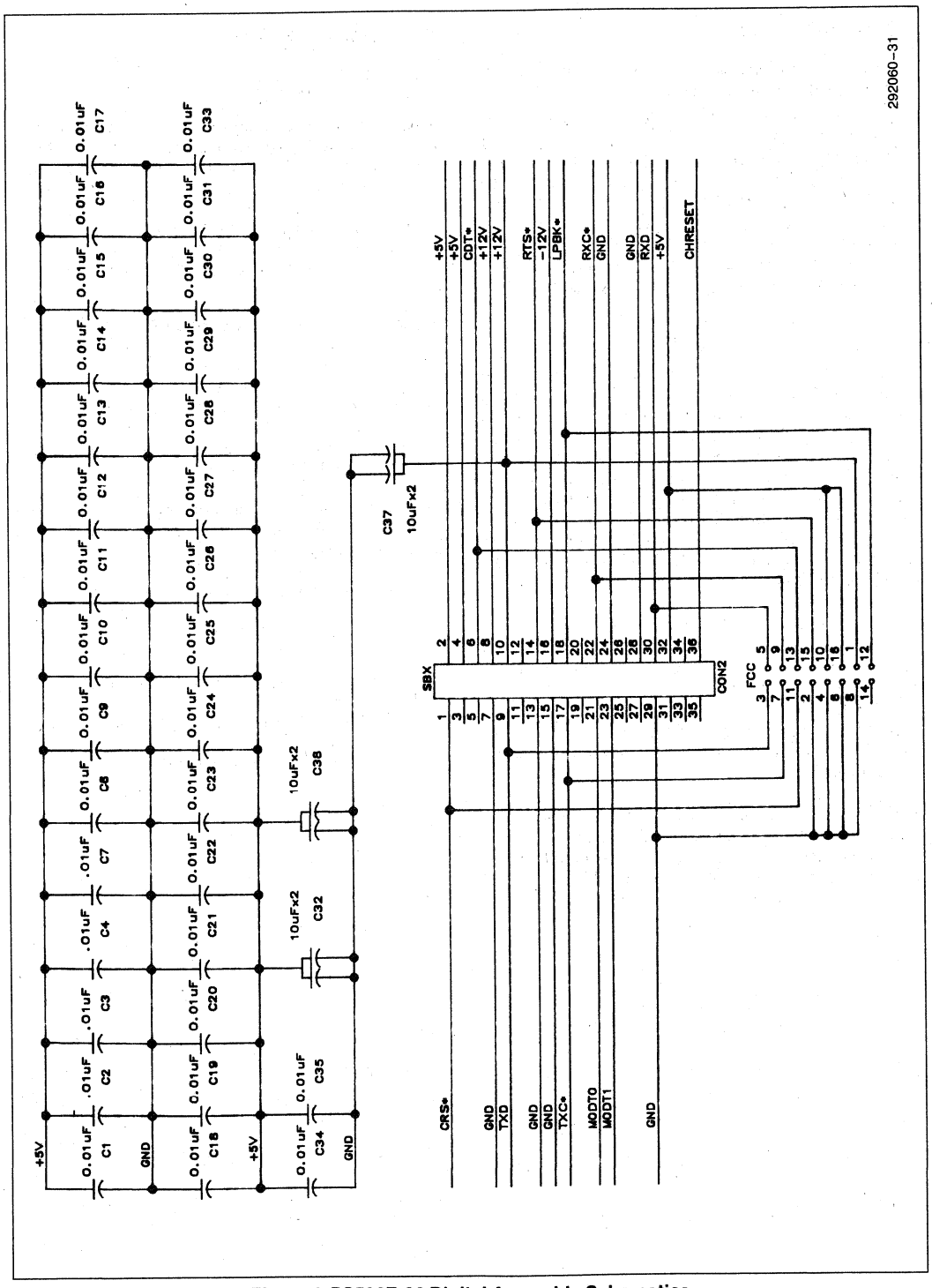


Figure 4. PS592E-32 Digital Assembly Schematics

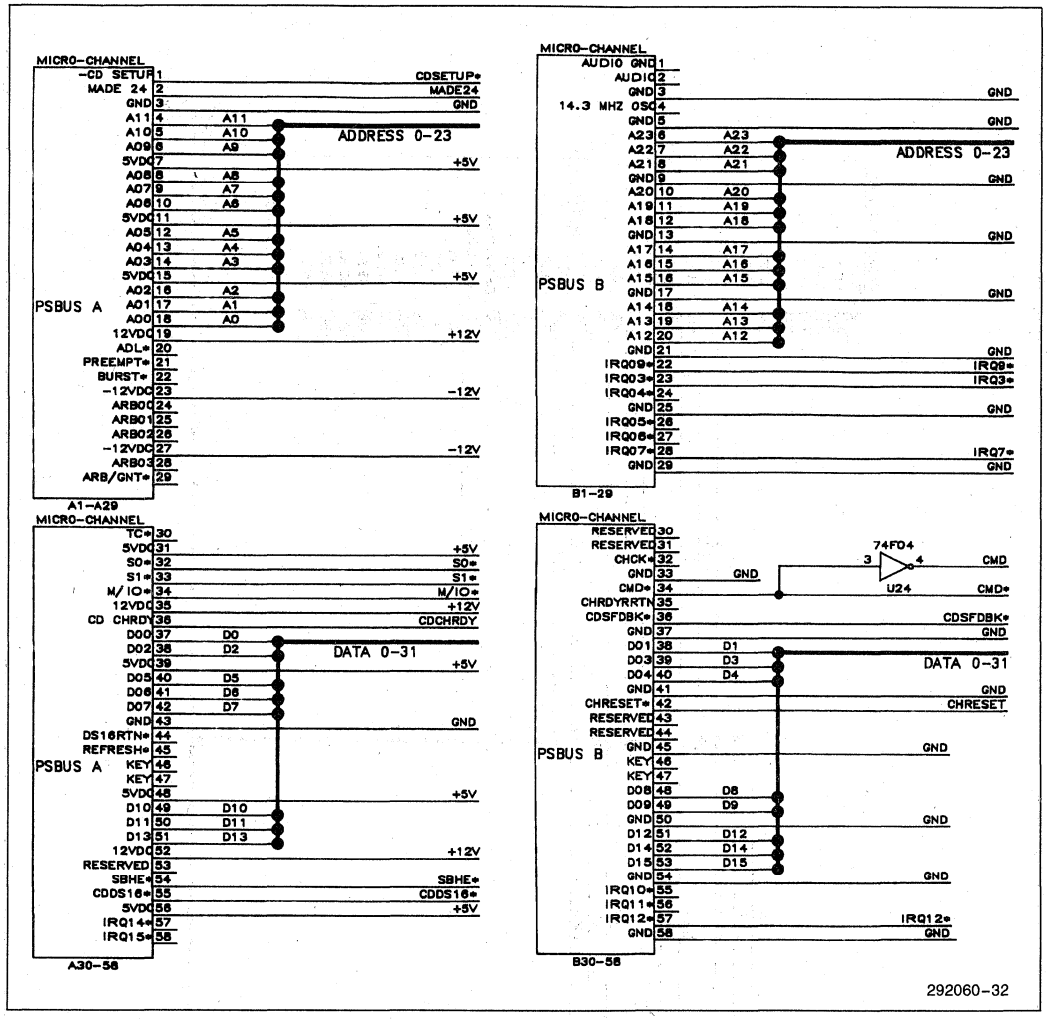


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

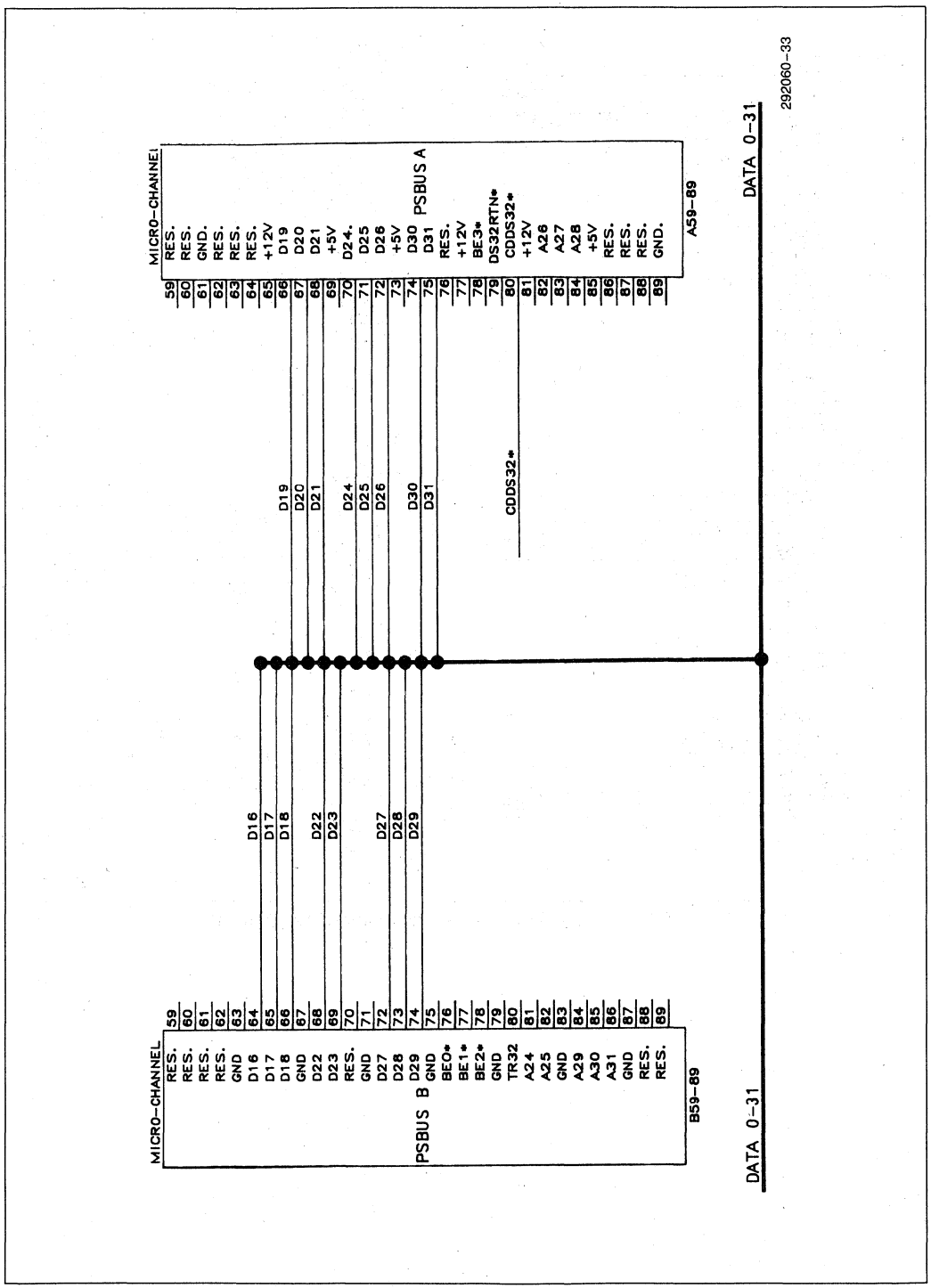


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

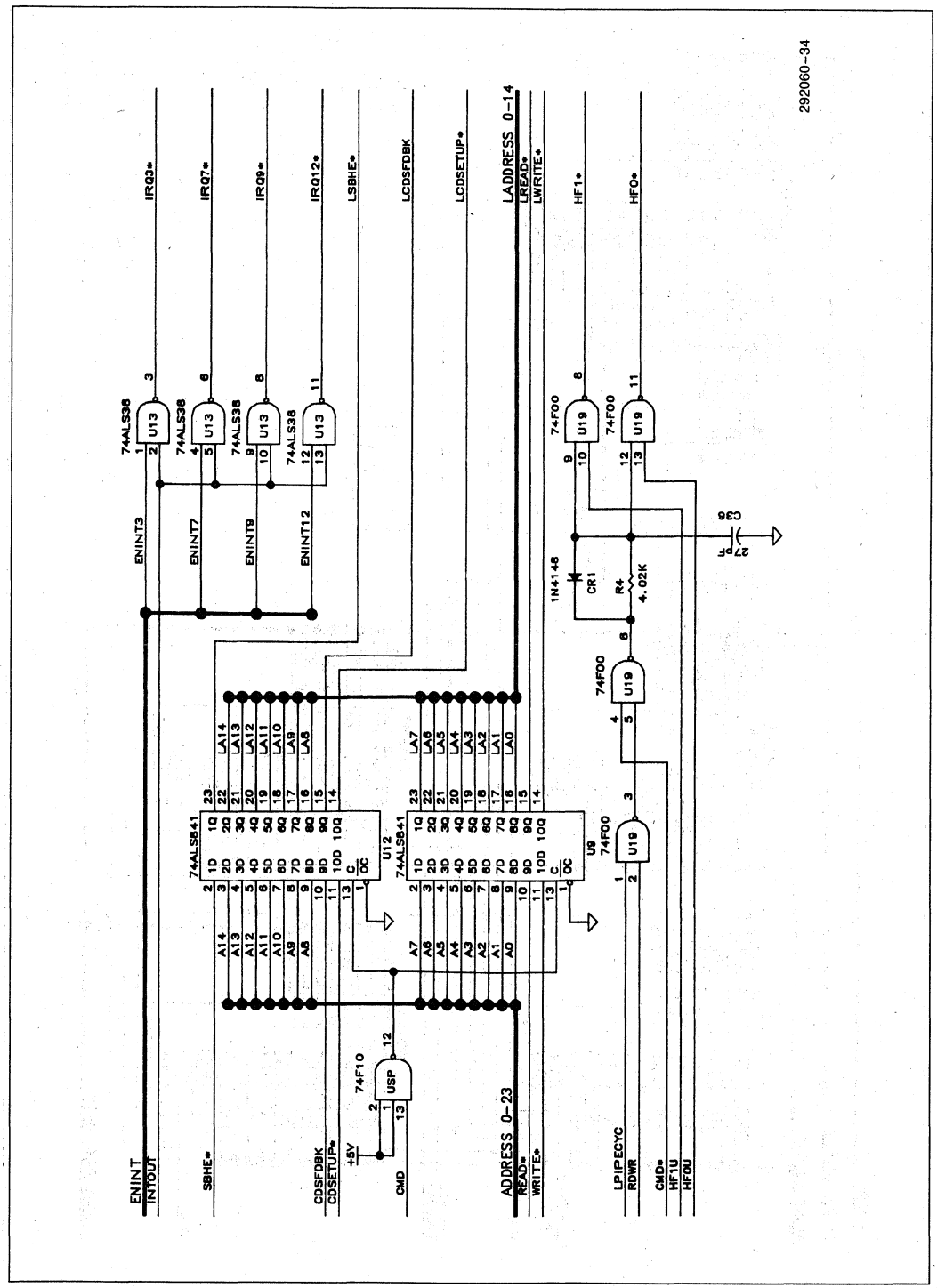


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

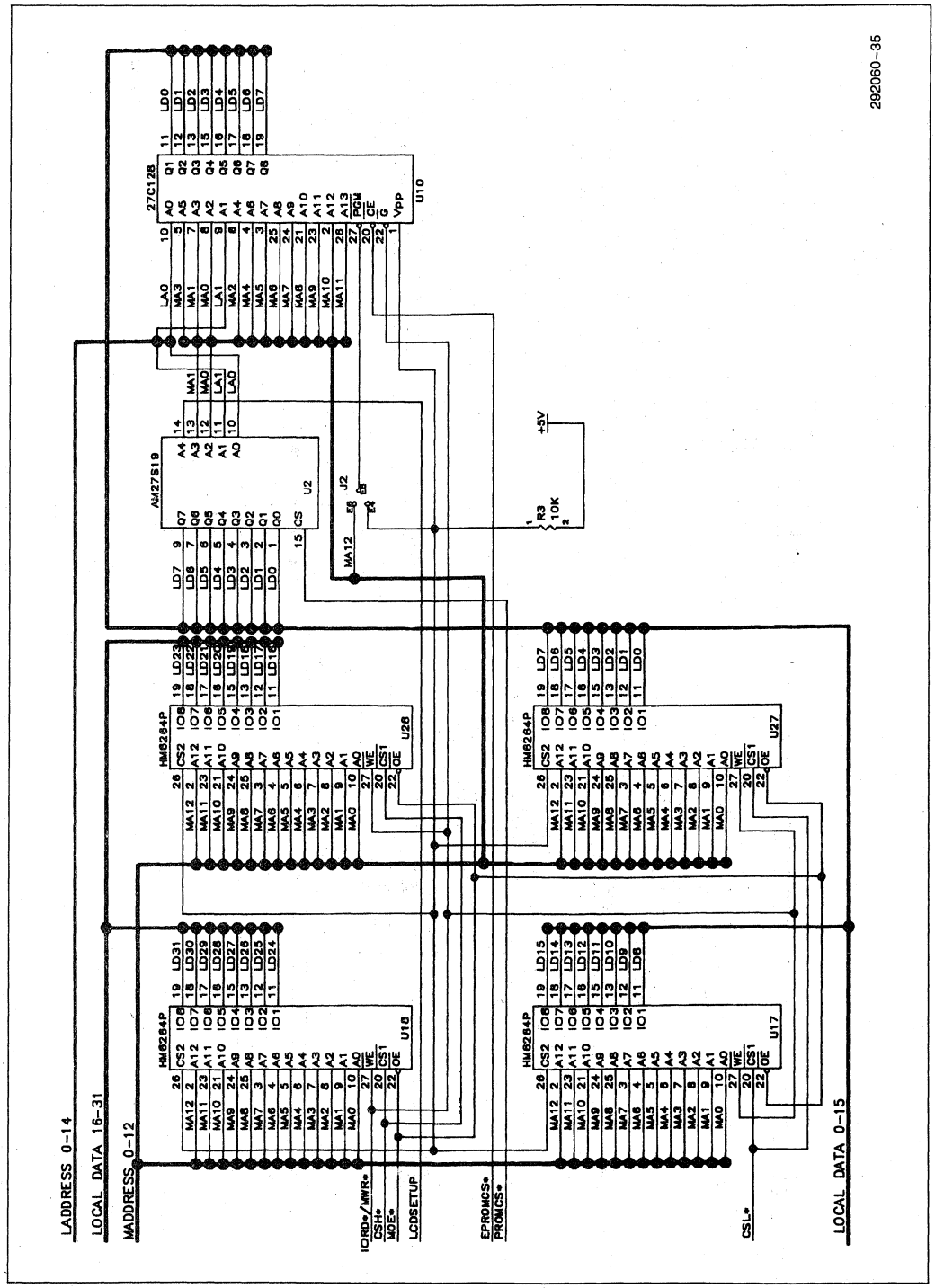


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

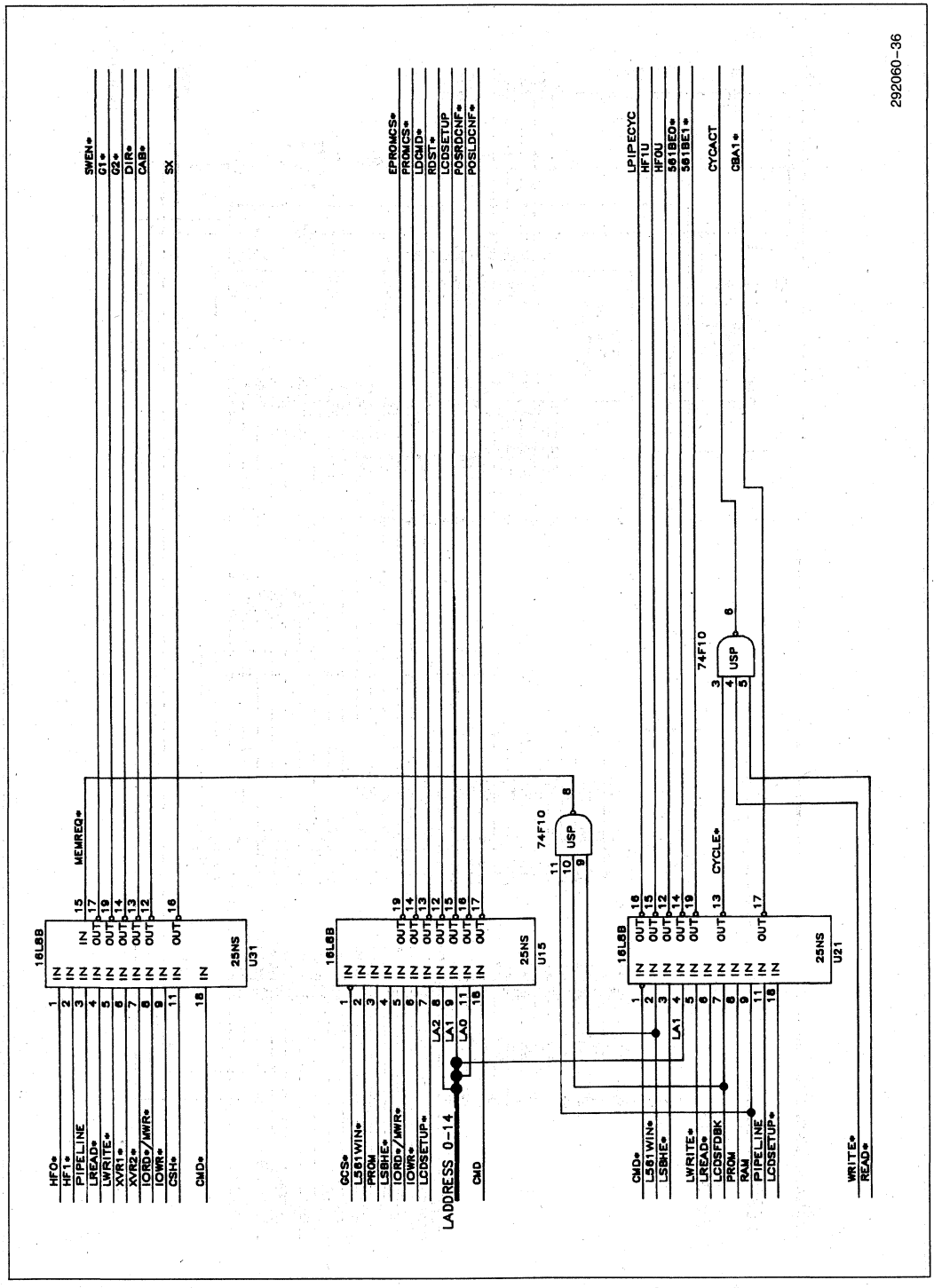


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

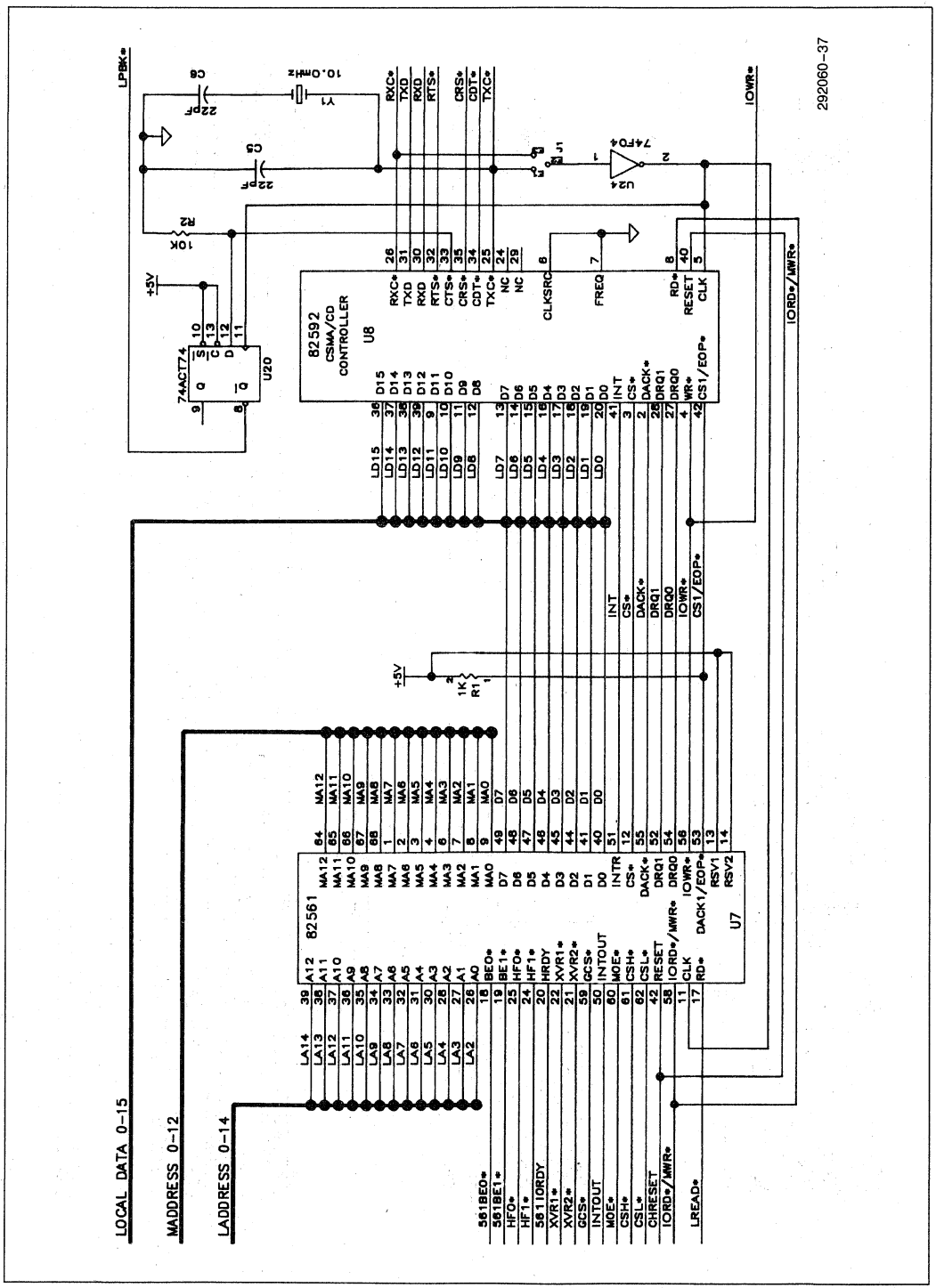


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

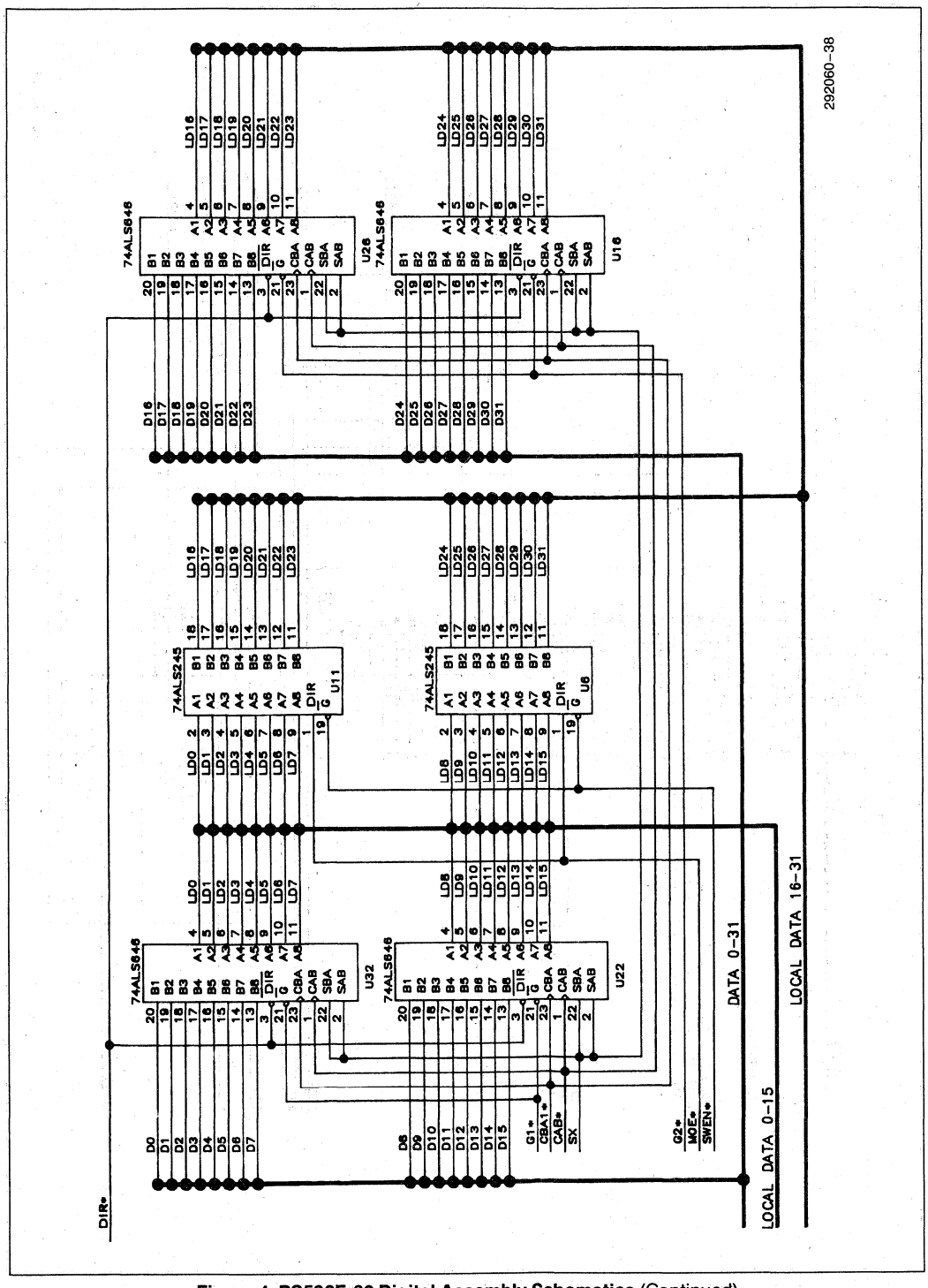


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

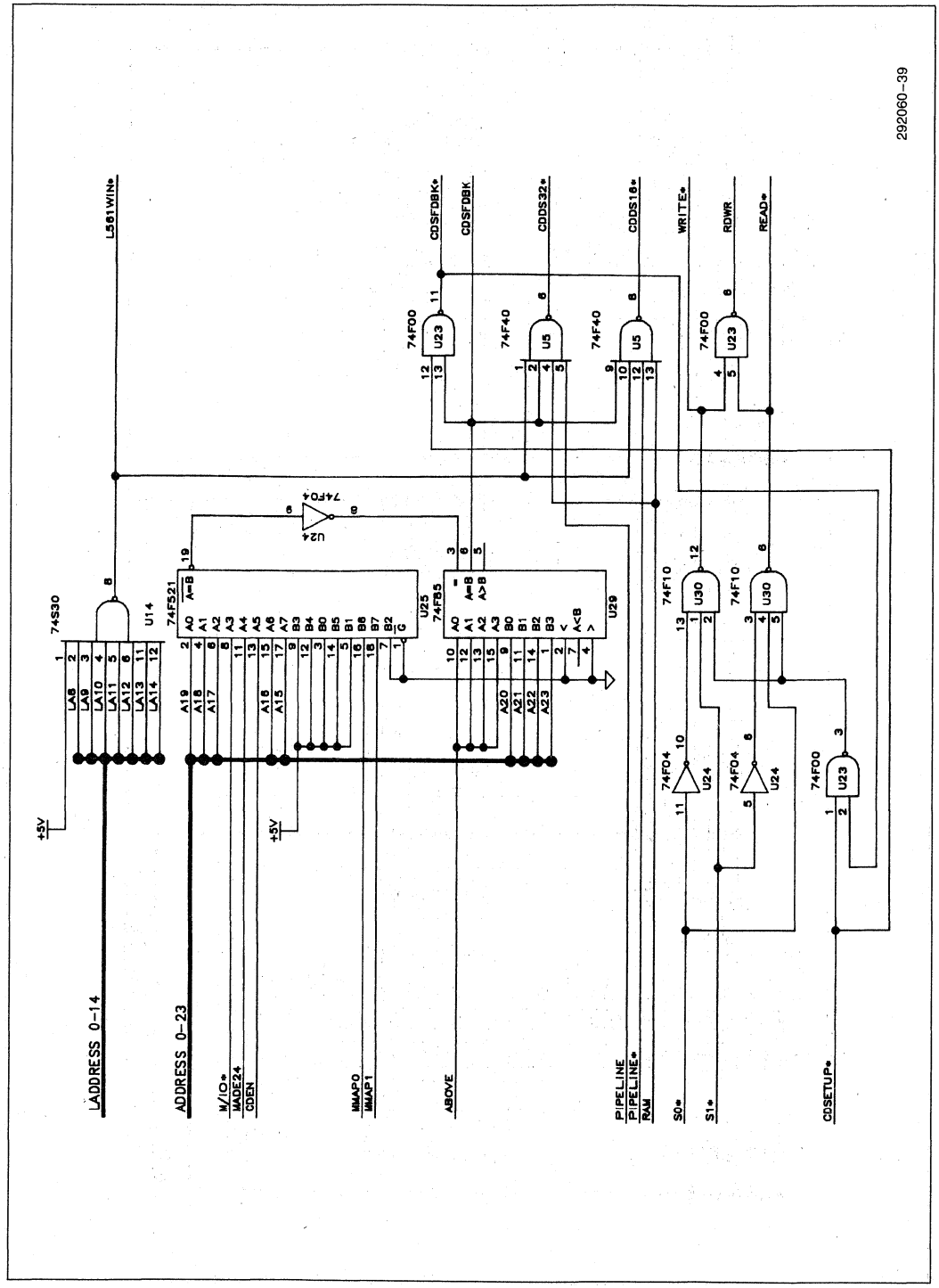


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

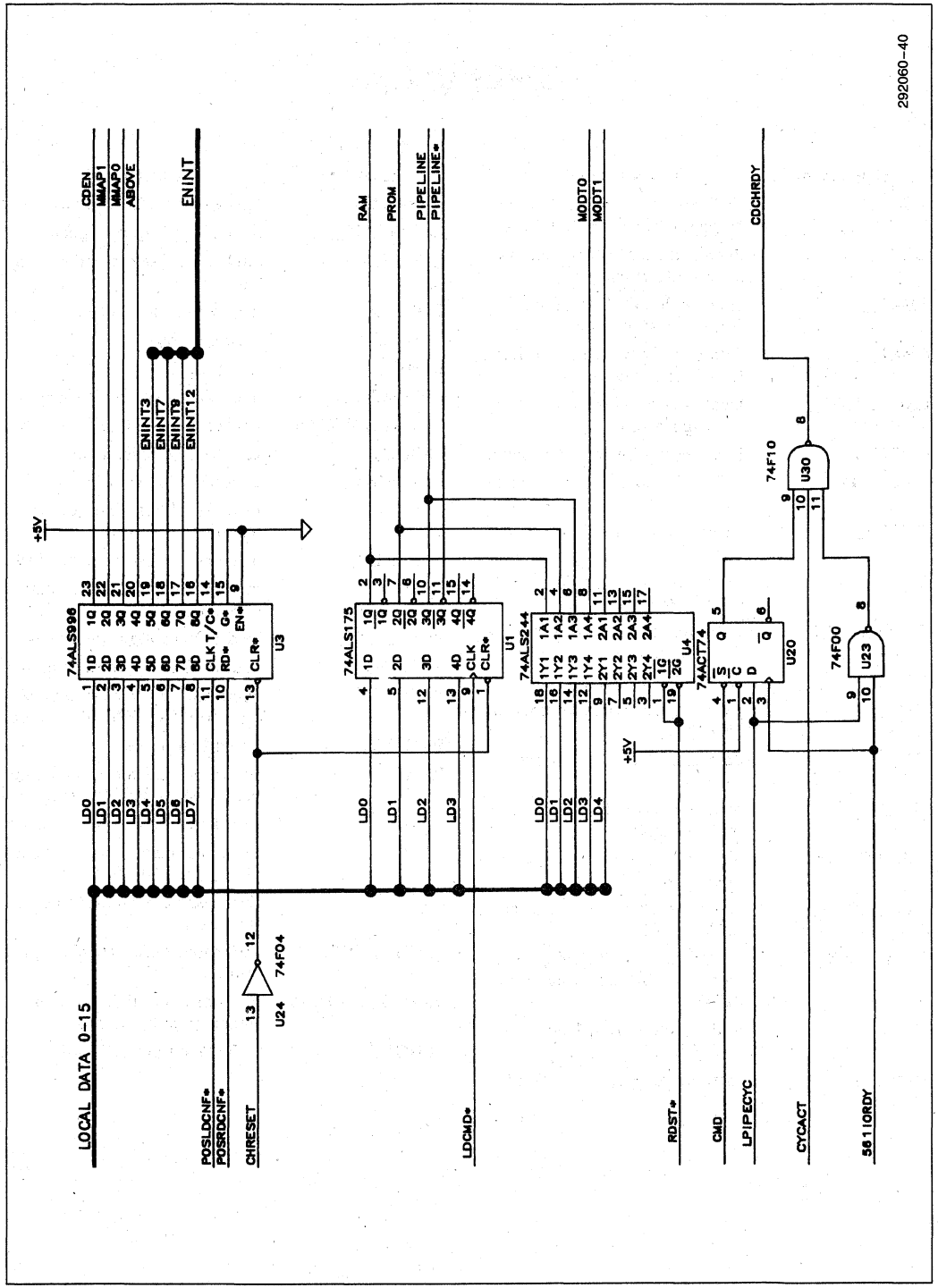


Figure 4. PS592E-32 Digital Assembly Schematics (Continued)

APPENDIX A

Signal Name	Description
ABOVE	POS register bit that determines mapping below or above 1 Meg.
CAB__	data latch clock (direction: local memory to latch)
CBA1__	data latch D0–D15 clock (direction: host to latch)
CBA2__	data latch D16–31 clock (direction: host to latch)
CSL__, CSH__	local SRAM chip selects (low and high banks)
CDEN	POS register bit for card enable
CYCACT	active host cycle enveloping signal
CYCLE__	partial active host cycle enveloping signal
DIR__	direction signal for data latches/trasceivers
ENINT3,7,9,12	POS register bits determining the selected interrupt signal
EPROMCS__	EPROM chip select signal
GCS__	general chip select (output of 82561)
G1__	data latch D0–D15 transceiver tristate enable
G2__	data latch D16–D31 transceiver tristate enable
HF0U,HF1U	unqualified, active high host request signal
HF0__,HF1__	active low host request signals to the 82561
IORD__ / MWR__	read from non-SRAM port or write to SRAM (output of 82561)
IOWR__	write to non-SRAM port (output of 82561)
INTOUT	Interrupt output of the 82561

Signal Name	Description
LA0–LA14	latched host address lines
LCDSETUP__	latched card set up signal
LCDSFDBK	latched card select feedback signal
LDCMD__	load command register
LD0–LD31	local data bus
LPBK__	loopback signal
LPIPECYC	latched pipeline cycle signal
LREAD,LWRITE	latched read and write signals
LSBHE__	latched byte high enable
L561WIN__	latched 82561 subwindow
MA0–MA12	memory address lines (output of 82561)
MEMREQ__	memory request from the host
MOE__	memory output enable (output of 82561)
POSLDCNF__	load POS configuration register
POSRDCNF__	read POS configuration register
PROM	command register bit determining PROM vs EPROM paging
PROMCS__	PROM chip select
RAM	command register bit determining RAM vs. ROM paging
RDST__	read from the status register
RDWR	unlatched read or write (decode of S0__ and S1__)
SX	select between latched and real time data
XVR1__,XVR2__	transceiver/latch enable signals (output of 82561)
561BE0__, 561BE1__	low word and high word enable signals (input to 82561)
561IORDY	IORDY output of 82561

intel®

APPLICATION NOTE

AP-328

August 1989

PC592E Buffered LAN Adapter Solution for the IBM PC-XT and PC-AT*

DARYOOSH KHALILOLAHI
TECHNICAL MARKETING ENGINEER

TSVIKA KURTS
SYSTEM VALIDATION GROUP

*IBM, PC-XT and PC-AT are trademarks of International Business Machines.

Order Number: 292063-001

1.0 INTRODUCTION

In recent years IBM PC-AT*s and compatibles have become the most popular personal computers. Judged by the amount of adapter hardware and application software developed for them, the trend seems likely to continue in the near future. Introduction of the Extended Industry Standard Architecture (EISA) is another reason supporting this prediction.

The role of local area networks (LANs) expands as the role of the personal computers in the office environment increases. Some examples of the benefits provided by networking are sharing expensive peripherals (to reduce the cost); sharing a single data base (to improves data control and security), and having electronic mail capabilities (to improve communication).

The best choices for Local Area Networks are those that provide low cost, reliable operation, ease of expansion, and the backing of major VLSI manufacturers.

The 82592/82560 is an ideal choice for 16-bit, buffered adapter applications. The high level of integration reduces the component count and the design cycle. The combination provides high performance and competitive cost.

The PC592E is a 16-bit, nonintelligent, buffered slave adapter design. It interfaces the IBM PC-AT or PC-XT* or compatibles to an Ethernet network. The 82592 LAN Controller and 82560 host interface and DMA Controller are used to receive and transmit frames between the network and local memory (16 kB). The PC592E can perform zero wait state memory data transfers on the PC bus. The design permits the use of interchangeable network serial interface modules for Ethernet**, Cheapernet and TPE applications.

2.0 OBJECTIVE

This application note demonstrates how to use the Intel 82560 and 82592 to build a high-performance, cost-effective PC-AT LAN adapter that implements the traditional buffered architecture.

2.1 Acknowledgements

We acknowledge and thank Yosi Mazor and Joe Dragony, of Intel's (Folsom, Calif.) Data Communications Focus Group, for their work in developing the hardware and the software and their contribution to this application note.

2.2 Terminology

The following table shows the terminology used in this document.

Symbol	Description
-	at the end of a signal name indicates active low
#	logical OR
&	logical AND
!	logical INVERSION

3.0 ORGANIZATION

Section 4 provides an overview of the 82560 and 82592 functionality. The reader needs a basic knowledge of these components to better understand the following chapters. Section 5 provides a functional description of the PC592E. In this section, the design is divided into three architectural subsections (host interface, memory subsystem, and network interface). PAL equations and schematics are broken down according to the architectural division. The last section provides the performance benchmarks for the board. Appendix A provides a brief description of most PC592E internal signals. Appendix B provides the complete sets of PAL equations.

4.0 COMPONENT OVERVIEW

4.1 82592 LAN Controller

The CHMOS 82592 is a CSMA/CD controller with a 16-bit data path. It can be configured to support a wide variety of industry standard networks, including Ethernet, Cheapernet, TPE, PCNet, and STARLAN***. The 82592 also supports Deterministic Collision Resolution (DCR) applications. The 82592 consists of three subsystems: parallel, serial, and FIFO. The parallel subsystem provides an 8- or 16-bit interface to the external bus. The 82592 supports memory transfers (at up to 16 MB/s); it accepts commands from the processor that controls the bus and provides it with status information. The 82592 can support simultaneous transmission and reception including autoretransmit, transmit frame chaining, and back-to-back frame reception. The serial subsystem consists of a highly flexible CSMA/CD unit, a data encoder/decoder, collision detect and carrier sense logic, and a clock generator. In high integration mode it supports NRZI, Manchester, or Differential Manchester encoding and decoding at bit rates up to 4 Mb/s. In high speed mode the 82592 is capable of 20-Mb/s Manchester or NRZI encoding. The FIFO subsystem consists of a transmit FIFO, a receive FIFO, and control logic (with programmable threshold). A total of 64 bytes of FIFO can be divided between receive and transmit.

*IBM, PC-AT and PC-XT are trademarks of International Business Machines Corp.
**Ethernet is a trademark of Xerox Corp.
***STARLAN is a trademark of AT&T.

4.2 82560 Host Interface and Memory Controller

The CHMOS 82560 is a high-performance DMA controller designed to work in a tightly coupled fashion with the 82592 in a PC-XT, PC-AT or MCA adapter application.

Two independent DMA channels support a transfer rate of up to 10 MB/s to/from the local SRAM. Up to 16 kB of ring buffer memory can be I/O or memory mapped into the address space. Host accesses to the local memory can be made with zero wait states. These accesses can be byte or word wide. The 82560 implements all of the 82592 tightly coupled functions: back to back frame reception, bad receive buffer reclaimation, auto retransmit upon collision, and transmit chaining.

5.0 IMPLEMENTATION

The board is divided into three sections (see figure 1), the host interface, the memory subsystem, and the network subsystem. Both the 82592 and 82560 operate on the 10-MHz Ethernet clock generated by the serial side. In the rest of this section a component (designated by its unit No. on the board and the schematic) is defined as part of a subsystem if one or more of its output pins are in that subsystem. To access any port on the board (including SRAM), the host CPU generates a request to the 82560. When in a 16-bit slot the board supports 16-bit hosts (PC-AT), otherwise it supports 8-bit hosts (PC-XT). To enable the board the most significant bit (bit 3) of the command register should be set to 1. In the 16-bit configuration SRAM accesses can be word (zero wait state or nonzero wait state) or byte wide (nonzero wait state). All other transfers (non-SRAM) are byte wide. In the 8-bit configuration SRAM accesses can be either with zero added wait state (pipeline mode) or with 3 to 4 added wait states (nonpipeline mode). The reader is referred to the table and listing in Section 5.2.3 for a complete set of host memory cycles.

The data transfers between the local memory and the 82592 are 16 bits wide and are controlled by the 82560 DMA channels.

5.1 Hardware Configurable Options

The board has seven jumpers and one switch. These can be set to change the board configuration.

5.1.1 MEMORY MAPPING WINDOW

E1 through E6 are used to select the lower portion of the address window. The letter E followed by a number refers to a unique jumper node. These two jumpers are used to select the lower portion of the memory address window. E1, E2, E3 select the least significant bit (MAP0) of the three bit address selector. E4, E5, E6 select the next bit (MAP1).

E7 through E9 select the most significant bit (MAP2) of the address selector. MAP2 determines the upper portion of the address window; namely below 1 Megabyte or above 1 Meg. Together with MAP0 and MAP1, these three jumper selections determine which one of the eight 16-kB windows of host memory address is chosen.

a) Below 1 Megabyte

memory window	MAP2	MAP1	MAP0	
0C0000 to 0C3FFF	0 E9/E8	0 E6/E5	0 E3/E2	(factory default)
0C8000 to 0CBFFF	0 E9/E8	0 E6/E5	1 E1/E2	
0D0000 to 0D3FFF	0 E9/E8	1 E4/E5	0 E3/E2	
0D8000 to 0DBFFF	0 E9/E8	1 E4/E5	1 E1/E2	

b) Above 1 Megabyte

memory window	MAP2	MAP1	MAP0
FC0000 to FC3FFF	1 E7/E8	0 E6/E5	0 E3/E2
FC8000 to FCBFFF	1 E7/E8	0 E6/E5	1 E1/E2
FD0000 to FD3FFF	1 E7/E8	1 E4/E5	0 E3/E2
FD8000 to FDBFFF	1 E7/E8	1 E4/E5	1 E1/E2

5.1.2 INTERRUPT

A switch is used to determine the interrupt line selected as shown below.

PC bus interrupt	Switch position
IRQ5	1 (factory default)
IRQ7	2
IRQ9	3
IRQ12	4

5.1.3 16- OR 8-BIT HOST SUPPORT

Jumper E16 through E18 is used to select the source of the BHE__. In a 16-bit slot E16/E17 should be selected. This connects the SBHE__ from the PC bus to the BHE__ of the board. In an 8-bit slot E18/E17 should be selected. This connects the inverted SA0 to the BHE__ of the board. The factory default is E16/E17 (PC-AT).

PCXT is a signal on the board that can be read by the host (through the status register) to determine if the board is plugged into a 16- or 8-bit slot. This signal is connected to the GND pin of the D connector (pin 18) of the 16-bit I/O channel. A 1-kohm pull-up resistor connects the node to the +5-V supply. Therefore in a 16-bit slot PCXT will be read as 0, otherwise it will be read as 1.

5.1.4 CLOCK SOURCE

Jumper E19 , E20, E21 selects the source of the clock for the 82592 and the 82560.

Jumper position	Clock source
E19/E21	10-MHz serial clock from the serial unit connected to TXC of 82592 (factory default)
E20/E21	external crystal connected to X1 of 82592

5.2 Host Interface

This subsystem consists of the command register (U4), the status register (U12), and their control (U11); the address decoder (U1 and U2) and its latch (U4); the data latches/transceivers (U8 and U9), the high memory bank to D7–0 data transceiver (U10), and their control (U5); the host request generator (U2); and the memory and peripheral controller (U6).

5.2.1 COMMAND AND STATUS REGISTERS

The command register is four bits wide.

3	2	1	0
BOARDEN	PIPELINE	PROM	RAM

After power up, or reset, the board is disabled. When disabled, the board recongnizes only host requests to access its command and status registers.

Bit 0, Bit 1: Determine the memory port being accessed.

Bit 1	Bit 0	
0	0	EPROM access
0	1	RAM access
1	0	PROM access
1	1	Reserved

Bit 2: Determines the mode for accessing the buffer memory. When 0 the host accesses to the local SRAM are made with added wait states. When 1, and the 82560 is configured for the pipeline mode, the host accesses to the local SRAM are made with no added wait states (pipelined data transfers). This bit and the bit 0 of the 82560 host mode register should always be set to the same value before an SRAM memory access is attempted.

Bit 3: Is the board enable bit. After power up, or reset, this bit should be set to 1 in order to access the on-board ports (SRAM, PROM, EPROM, 82560 registers, and 82592 registers).

The following is the PAL (U11) equation for writing to the command register.

```
LDCMD_ = !( !GCS_ & !L560WIN_ & !IOWR_)
```

The status is seven bits wide read only register.

6	5	4	3	2	1	0
PCXT	MODT1	MODT0	BOARDEN	PIPELINE	PROM	RAM

The four least significant bits are the contents of the command register.

Bit 4, Bit 5: Are generated by the network module and determine the type of network module installed.

Bit 5	Bit 4	
0	0	Ethernet
0	1	Cheapernet
1	0	Reserved
1	1	TPE

The reserved combination is for future use.

Bit 6: Is read to determine if the board is plugged into an 8- or 16-bit slot. In a 16-bit slot a 0 is read, otherwise a 1 is read.

The following is the PAL (U11) equation for reading the status register.

```
RDST_ = !( !GCS_ & !L560WIN_ & !IORD_)
```

5.2.2 ADDRESS DECODER

The highest bits of the address (LA23–20) are decoded in U2. The output is the MSBDEC signal, which is latched on the falling edge of the BALE in U4. The latched signal (LMSBDEC) is input to the other PAL (U1). SA19–14 are decoded in this PAL for the lower portion (within 1 Megabyte) of the address mapping. The output is LSBDEC.

Three signals from the board provide the handshake required by the host:

IOCHRDY, MEMCS16_, and ZEROWS_.

The relevant PAL equations (of U1 and U2) are given below. The reader is referred to the appendix A for the decription of internal signals.

```
RAMEN      = (RAM & BOARDEN)
MSBDEC     = ( !LA20 & !LA21 & !LA22 &
               !LA23 & !MAP2) #
             (LA 20 & LA21 & LA22 & LA23
             & MAP2)
!LSBDEC_ = ((SA19 & SA18 & !SA17 &
             !SA16 & !SA15 & !SA14 &
             !MAP1 & !MAP0) #
             (SA19 & SA18 & !SA17 & !SA16
             & SA15 & !SA14 & !MAP1 &
             MAP0) #
             (SA19 & SA18 & !SA17 & SA16
             & !SA15 & !SA14 & MAP1 &
             !MAP0) #
             (SA19 & SA18 & !SA17 & SA16
             & SA15 & !SA14 & MAP1 &
             MAP0))
!L560WIN_  = (SA13 & SA12 & SA11
             & SA10 & SA9 & SA8 &
             SA7)
!MEMCS16_  = ( !LSBDEC_ & LMSBDEC
             & RAMEN & L560WIN_ &
             !SA0 & !BHE_)
enable (ZEROWS_) = (RAM & !LSBDEC_ &
             LMSBDEC & BOARDEN &
             PIPELINE & L560WIN_)
ZEROWS_    = !L560IORDY_
```

5.2.3 DATA TRANSCEIVERS/LATCHES

Bit 2 of the command register determines the mode in which the host accesses the local SRAM.

In the nonpipeline mode the data transceivers/latches act as simple transceivers. The cycles are extended by pulling IOCHRDY low until the transfer to/from local memory is completed. All non-SRAM accesses are in nonpipeline mode.

In pipeline mode the data transciever/latches act as data latches. In this mode a read ahead or write behind operation is performed after every host request to the 82560. These accesses are to sequential locations in the local SRAM. In this mode no wait states are asserted. The direction of the pipeline transfer is determined by the value of bit 1 of the 82560 host mode register. In read cycles, after the current transfer, the 82560 updates the buffer with the contents of the next local memory address. This is called "read ahead" (in anticipation of the next host read request). In write cycles the data is copied from the data latch to the local memory after the host has finished writing to the data latch. This is called "write behind."

In the 8-bit configuration, during odd byte memory accesses the transceiver U10 becomes transparent. Thus the data path would be from/to the high bank of SRAM (U13), through U10, to the low byte of data bus (D7–D0).

In the nonpipeline mode U8 and U9 serve as simple data transceivers. All non-SRAM accesses use the even byte (D7–D0). In the 16-bit configuration the value of SA0 and SBHE. determines which bank of memory (or both) should be accessed.

The following table shows different types of memory cycles and the expected value of the transceiver and data latch (74ALS646) control signals. B is the host side and A is the local bus side. When SX is 1, stored data is selected. When 0, real time data is selected. X stands for don't care.

cycle	Pipeline	PCXT	BHE_	SA0	MEMW_	MEMR_	G1L_	G1H_	G2_	CBA1_	CBA2_	SX
1	1	0	0	0	0	1	0	0	1	0	0	1
2	1	0	0	0	1	0	0	0	1	1	1	1
3	0	0	1	0	X	X	0	1	1	1	1	0
4	0	0	0	1	X	X	1	0	1	1	1	0
5	0	0	0	0	X	X	0	0	1	1	1	0
6	1	1	1	0	0	1	0	1	1	0	1	1
7	1	1	1	0	1	0	0	1	1	1	1	1
8	1	1	0	1	0	1	0	0	0	1	0	1
9	1	1	0	1	1	0	1	0	0	1	1	1
10	0	1	1	0	X	X	0	1	1	1	1	0
11	0	1	0	1	X	X	1	0	0	1	1	0

cycle 1 : PCAT pipelined (word-wide) memory write
cycle 2 : PCAT pipelined (word-wide) memory read
cycle 3 : PCAT nonpipelined even-byte memory access
cycle 4 : PCAT nonpipelined odd-byte memory access
cycle 5 : PCAT nonpipelined word-wide memory access
cycle 6 : PCXT pipelined even-byte memory write
cycle 7 : PCXT pipelined even-byte memory read
cycle 8 : PCXT pipelined odd-byte memory write
cycle 9 : PCXT pipelined odd-byte memory read
cycle 10 : PCXT nonpipelined even-byte memory access
cycle 11 : PCXT nonpipelined odd-byte memory access

The control signals to the transcievers/latches are generated by the PAL U5, which realizes the above table.

```
!G1L_  = (MEMREQ & PCXT & SA0_ & !MEMR_) # ( !PCXT & PIPECYC & MEMR_) #
         ( !XCV1_ & SA0_) # ( !HF0_ & !HF1_) # ( !XCV2_ & HF1_)

!G1H_  = (PIPECYC & !MEMR_) # ( !XCV1_ & !BHE_) # ( !XCV2_ & HF1_)

!G2_   = (PCXT & MEMREQ & !SA0_)

!CAB_  = ( !IOWR_ & !XCV2_ & HF1_)

!CBA1_ = (PIPECYC & !MEMW_ & !PCXT) # (MEMREQ & PIPELINE & !MEMW_ & PCXT &
         SA0_)

!CBA2_ = (PIPECYC & !MEMW_)

SX     = (MEMREQ & PIPELINE # !IORD_ & PIPELINE & !XCV2_ & HF1_)

!DIR_  = ( !XCV2_ & HF1_ & !IORD_) # ( !MEMW_)
```

5.2.4 HOST REQUEST GENERATOR

The following table shows how different ports on the board are accessed. BOARDSEL refers to the board address decoded, that is the logical AND of LSBDEC and LMSBDEC. X stands for don't care.

BOARDSEL	L560WIN__	BOARDEN	RAM	SBHE__*	HF1__	HF0__	
0	X	X	X	X	1	1	Idle
1	0	X	X	0	0	0	GCS__ access: Command/Status
1	1	1	0	X	0	0	GCS__ access: ROM **
1	1	1	1	X	1	0	SRAM access
1	0	1	X	1	0	1	82560 register access

* In the 8-bit configuration, inverted SA0 serves as SBHE__.
** ROM refers to either PROM or EPROM. Bit 1 of the command register determines which.

The following PAL equations realize the above table. HF0__ and HF1__ are the request lines to the 82560.

```
!HF0_   =  ( !MEMW_ & L560WIN_ & BOARDSEL & BOARDEN & REQEN)    #
           ( !MEMW_ & L560WIN_ & BOARDSEL & BOARDEN & !PIPELINE) #
           ( !MEMW_ &!L560WIN_ & BOARDSEL & !BHE_) #
           ( !MEMR_ & L560WIN_ & BOARDSEL & BOARDEN & REQEN)   #
           ( !MEMR_ & L560WIN_ & BOARDSEL & BOARDEN & !PIPELINE)#
           ( !MEMR_ &!L560WIN_ & BOARDSEL & !BHE_)

!HF1_   =  ( !MEMW_ & !RAM & BOARDSEL & BOARDEN) #
           ( !MEMW_ & BOARDSEL & !L560WIN_) #
           ( !MEMR_ & !RAM & BOARDSEL & BOARDEN) #
           ( !MEMR_ & BOARDSEL & !L560WIN_)
```

5.2.5 MEMORY AND PERIPHERAL CONTROLLER

To access any port on the board the host generates a request to the 82560. The 82560 will then synchronize the request and perform arbitration (with any active local DMA request). It then asserts the proper memory or peripheral control signals.

The 82560 also supports the interrupt function. The interrupt initiated by either the 82592 or the 82560 is passed to the host system through one of the four interrupt lines (depending on the position of the switch).

In nonpipeline mode IOCHRDY is pulled low immediately after the board address is decoded. It goes high a programmable number of clock transitions after the host cycle starts. The 82560 register bit that can affect it are: HRDY delay, HRDY delay reference source, IO and access delays. The reader is referred to the 82560 data sheet for more details. In pipeline mode the IOCHRDY remains high until after the request is removed. Then the 82560 HRDY output goes low while the local bus cycle is being completed. IO CHRDY will be pulled low while the board address decode is active and 82560 HRDY output is low. In either case the IOCHRDY is tristated until the board address is decoded.

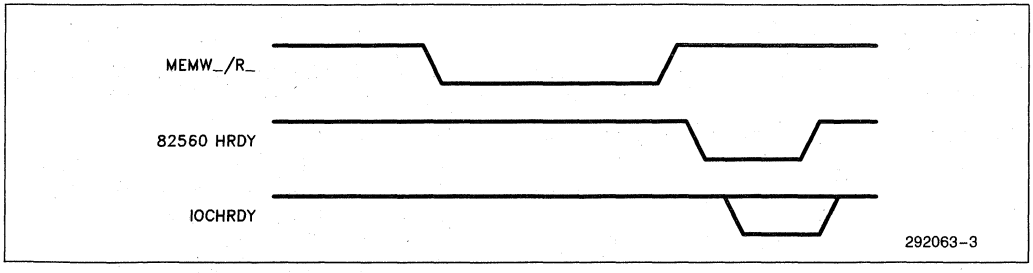
enable (IOCHRDY) = (LSBDEC & LMSBDEC)
IOCHRDY = !560IORDY__
# (!LSBDEC__ & LMSBDEC & !BOARDEN & L56OWIN__)
# (!LSBDEC & LMSBDEC & !BOARDEN & !BSA0)

5.2.5.1 Nonpipeline cycles

In ALL nonpipeline cycles (SRAM or otherwise) IOCHRDY is pulled low within 30 ns after the address on the PC bus becomes valid. It remains low until the 82560 HRDY output goes high, then it goes high.

The memory address input to the 82560 in the nonpipline mode is the same as the host CPU address shifted by one bit (SA1 goes to A0 input of 82560). During byte-wide memory accesses byte enable inputs of the 82560 (BE0__ and BE1__) determine which bank of SRAM is being accessed.

The output address of the 82560 is the same as its input address, except that its three most significant bits (12, 13, and 14) are logically OR'd with the three least significant bits of the 82560 host address register. The low bank of memory is accessed for even-byte addresses and the high bank or odd-byte addresses.



5.2.5.2 Pipeline Cycles

During SRAM pipeline cycles IOCHRDY stays high. It goes low after the cycle is over (HF__ removed). This is when the 82560 performs read ahead or write behind operations. Read ahead means that the next word (or byte in the case of 8-bit configuration) of data is copied from the local memory into the data latches in anticipation of the next request. Write behind means that data is first latched in the data latch(s) and then the 82560 copies the data into the local memory. The memory address in this mode is provided by the host address register, which is incremented by one after each 82560 transfer. To change the direction of the transfer the 82560 Host mode register should be accessed first and its bit 1 changed.

5.3 Memory Subsystem

The memory subsystem consists of the network address PROM (U15), the low bank of SRAM (U14), the high bank of SRAM (U13), an optional EPROM (U16), control PAL (U11, U5), and the 82560 memory controller (U6).

All controls are generated by the 82560, except for chip select for the EPROM and the PROM.

```
EPROMCS_ = !( !GCS_ & !IORD_ & !PROM &
           L560WIN_)
PROMCS_  = !( !GCS_ & !IORD_ & PROM &
           L560WIN_)
```

5.4 Network Interface

The network interface consists of the 82592 LAN controller (U7), the 82560 DMA controller (U6), and a plug-in analog module (Ethernet, CNet, or TPE).

5.4.1 DMA TRANSFERS

Two independent DMA channels of the 82560 are used to transmit and receive frames. Each word-wide DMA transfer can have a duration of 200 to 500 ns (82560 programmable). Autoretransmit, back-to-back frame reception, and bad receive buffer reclamation are all performed without CPU intervention.

The 82560 in the 82592 TCI mode supports transmit chaining. DMA channels are also used to configure the 82592 and to read its 69 bytes of internal information through the dump command.

The arbitration between the two DMA requests, and between the host request and the DMA request, are performed by the 82560 local bus arbiter function.

Transmit buffer size (including the configuration block) can be about two kilobytes. It should stop 128 bytes before the end of the adapter memory. The last 128 bytes of address (highest addresses) are for accessing the 82560 and the command and the status registers.

The receive memory buffer can occupy the rest of the local memory. It can be arranged as a ring buffer and managed by the 82560. The lower limit register of the 82560 holds the starting address of the ring, and its upper limit register holds the ending address of the ring. The 82560 performs the wraparound without CPU intervention. The stop register of the 82560 points to the last receive buffer location processed by the host CPU. The 82560 generates an interrupt to the host if the current address register of the channel reaches the stop register of that channel.

For more information about DMA transfers and the format of the transmit and receive frame, the user is referred to the 82560/82561 Technical Reference Manual Order #290198.

5.4.2 SERIAL INTERFACE

The 82592 CSMA/CD controller is used in the high-speed mode. The 82C501AD performs Manchester encoding and decoding; it also provides a watchdog timer, collision detection, and transmit/receive clock generation. Using the loopback modes, the transmitted data is routed to the receive path (at the 82592, the 82C501AD, or on the wire). This feature is useful for controller and physical layer diagnostics.

5.5 Schematics

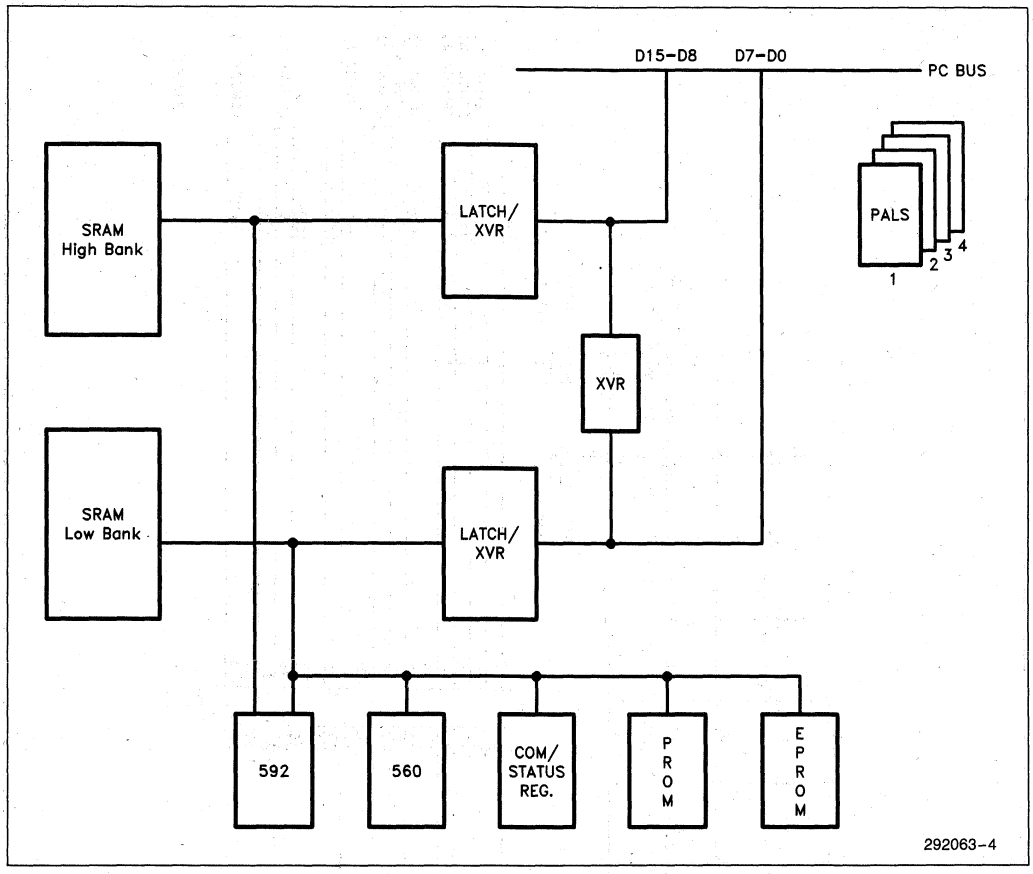


Figure 1. PC592 Block Diagram

5.5 Schematics (Continued)

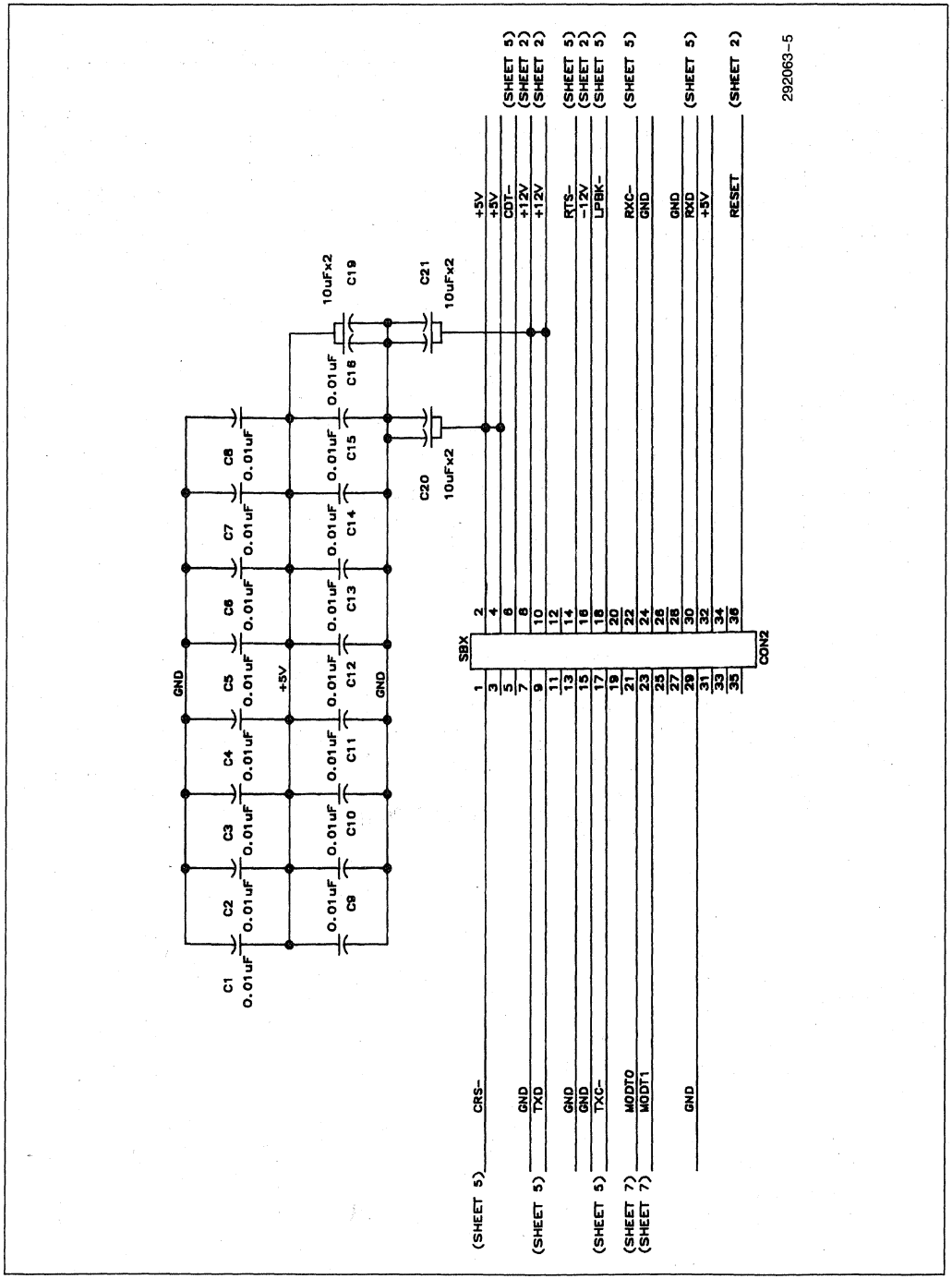


Figure 2. PC592 Digital Assembly Schematics

5.5 Schematics (Continued)

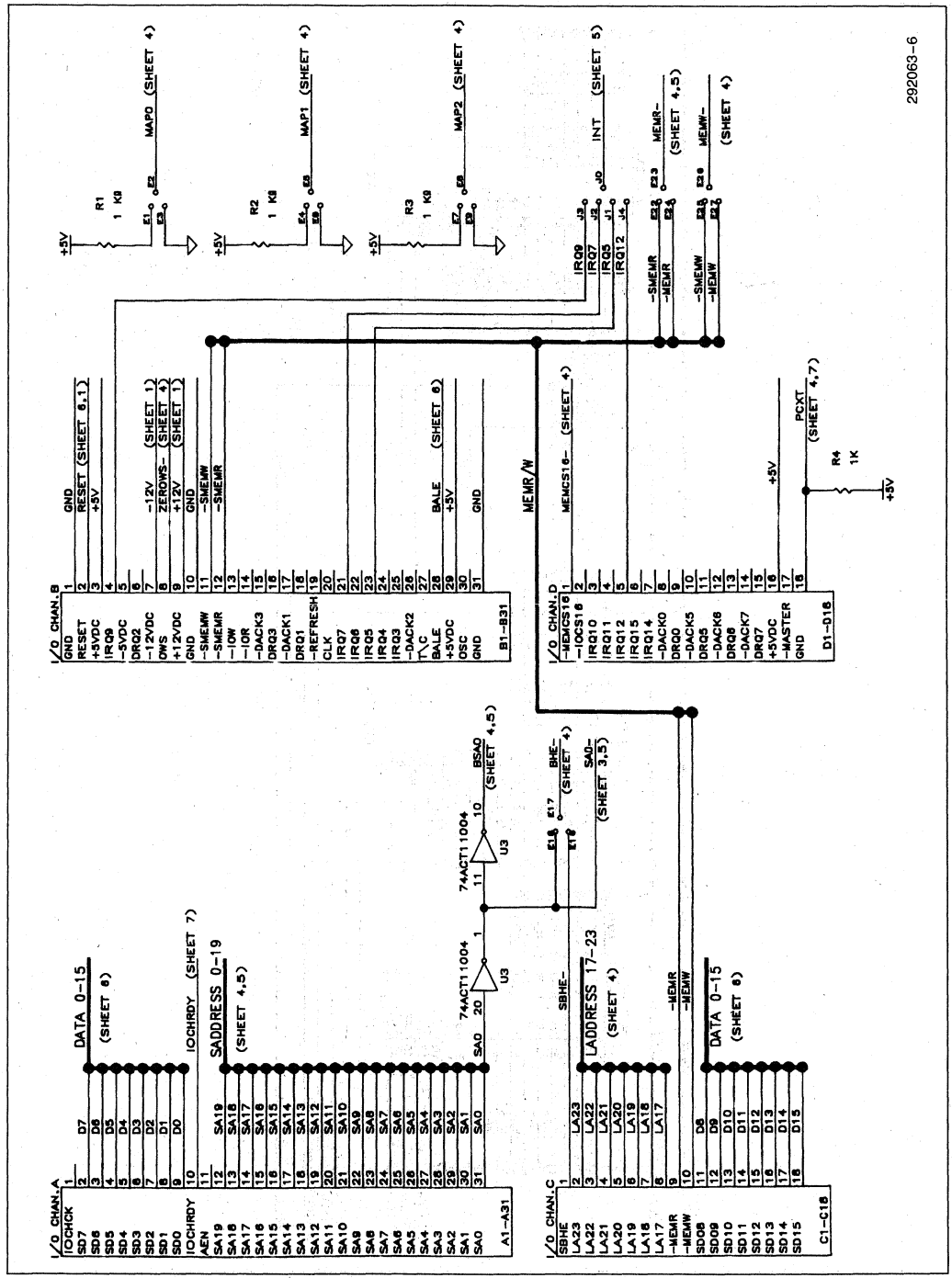


Figure 2. PC592 Digital Assembly Schematics (Continued)

5.5 Schematics (Continued)

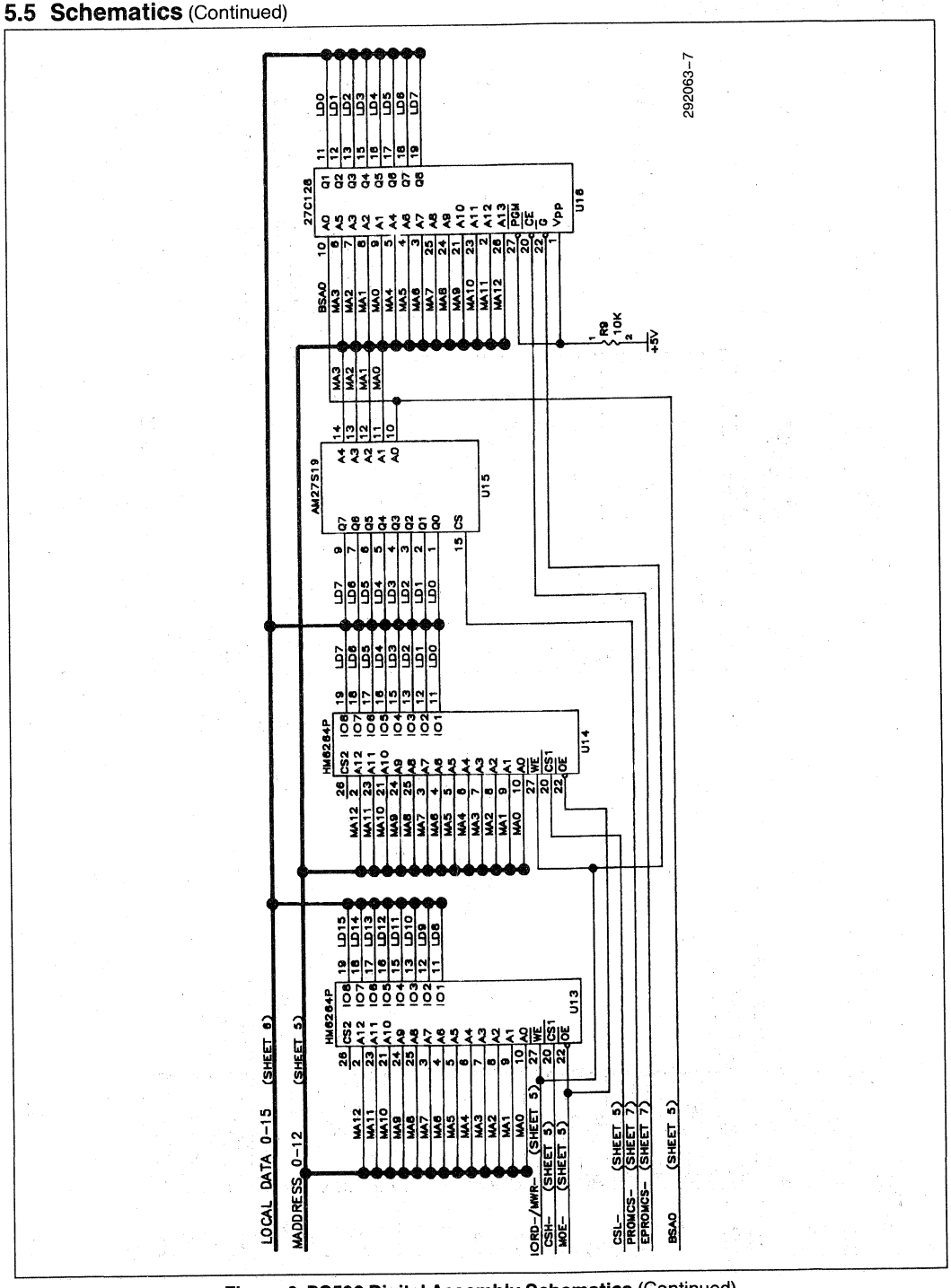


Figure 2. PC592 Digital Assembly Schematics (Continued)

5.5 Schematics (Continued)

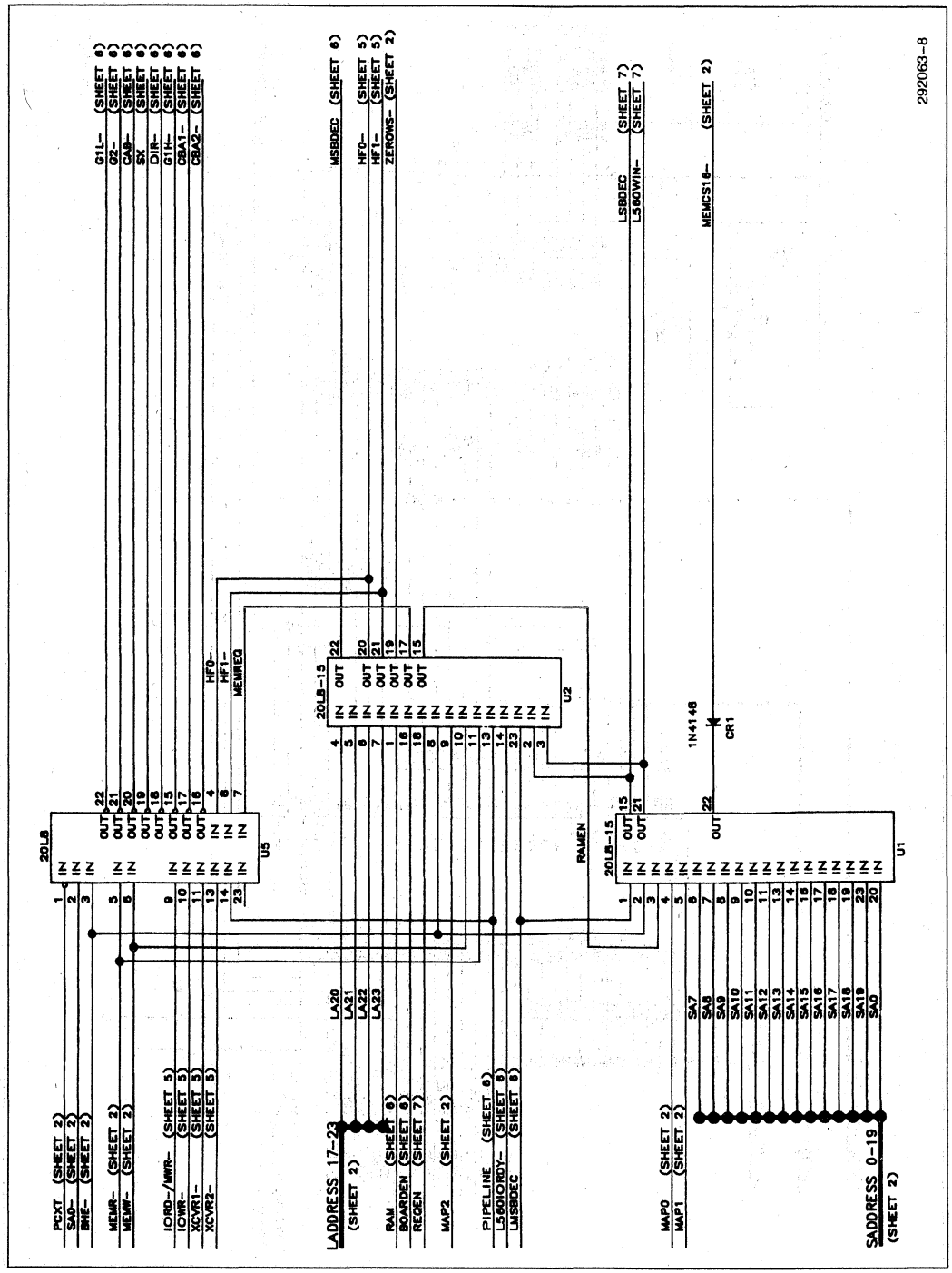


Figure 2. PC592 Digital Assembly Schematics (Continued)

5.5 Schematics (Continued)

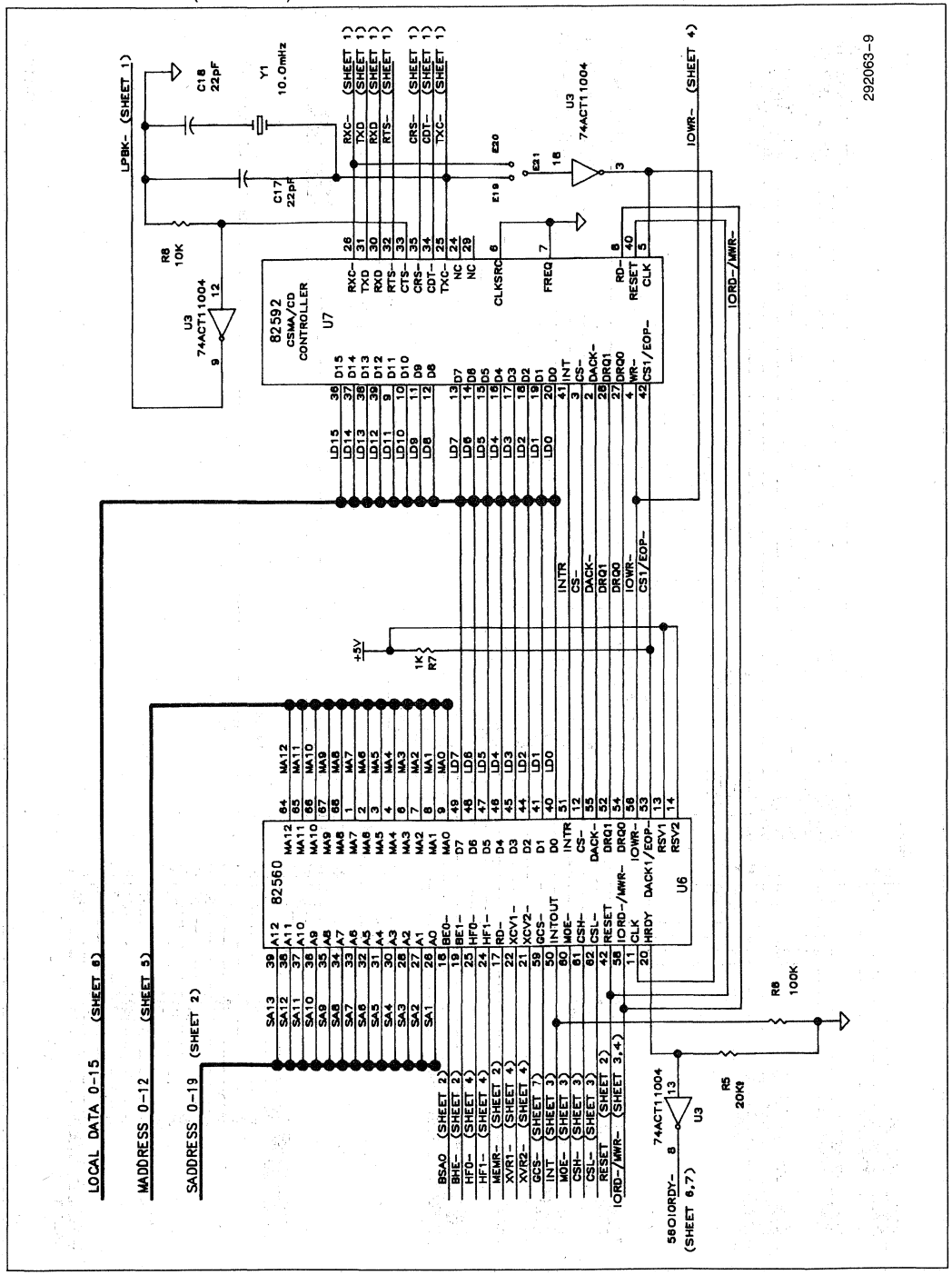


Figure 2. PC592 Digital Assembly Schematics (Continued)

5.5 Schematics (Continued)

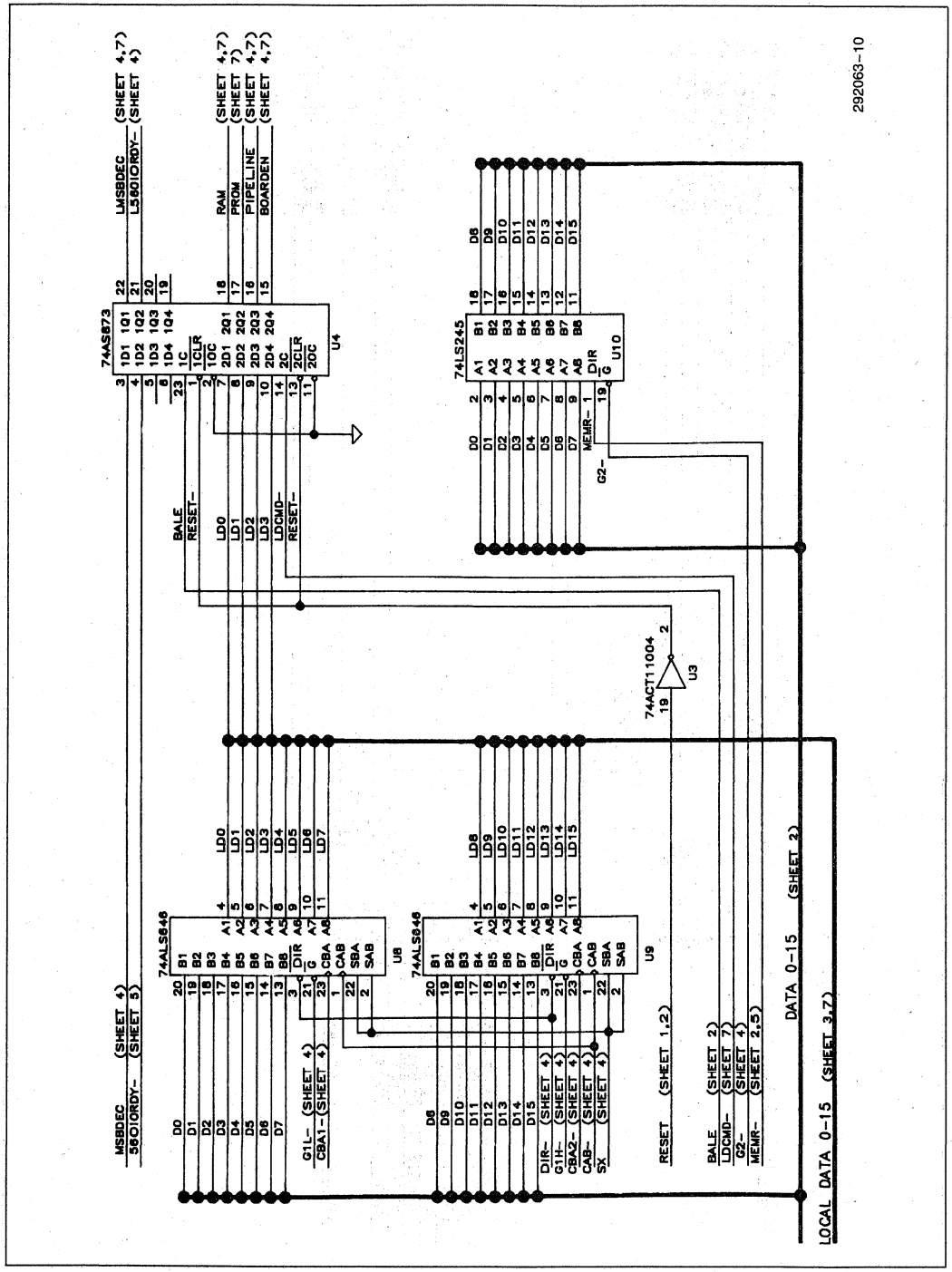


Figure 2. PC592 Digital Assembly Schematics (Continued)

5.5 Schematics (Continued)

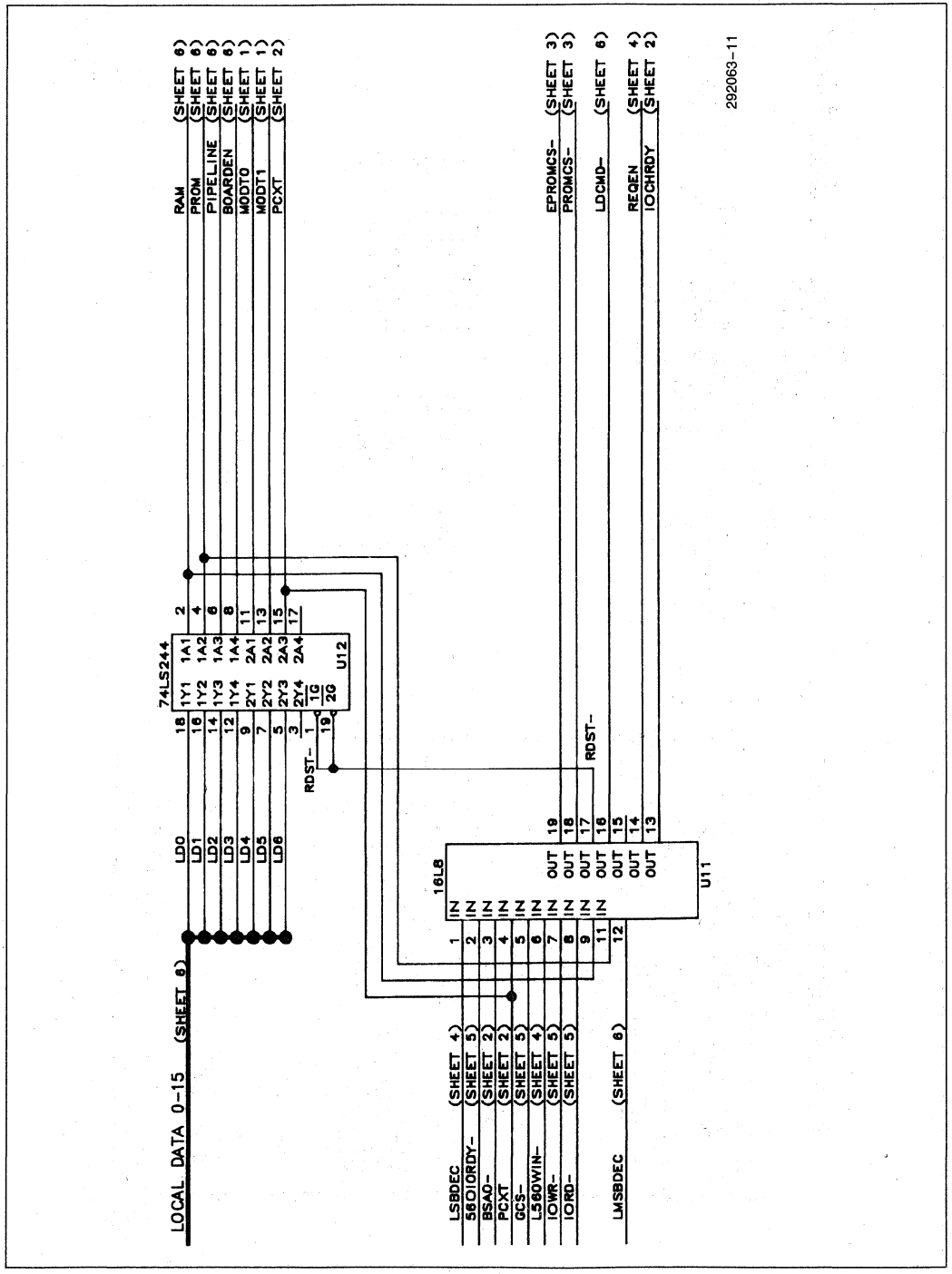


Figure 2. PC592 Digital Assembly Schematics (Continued)

5.5 Schematics (Continued)

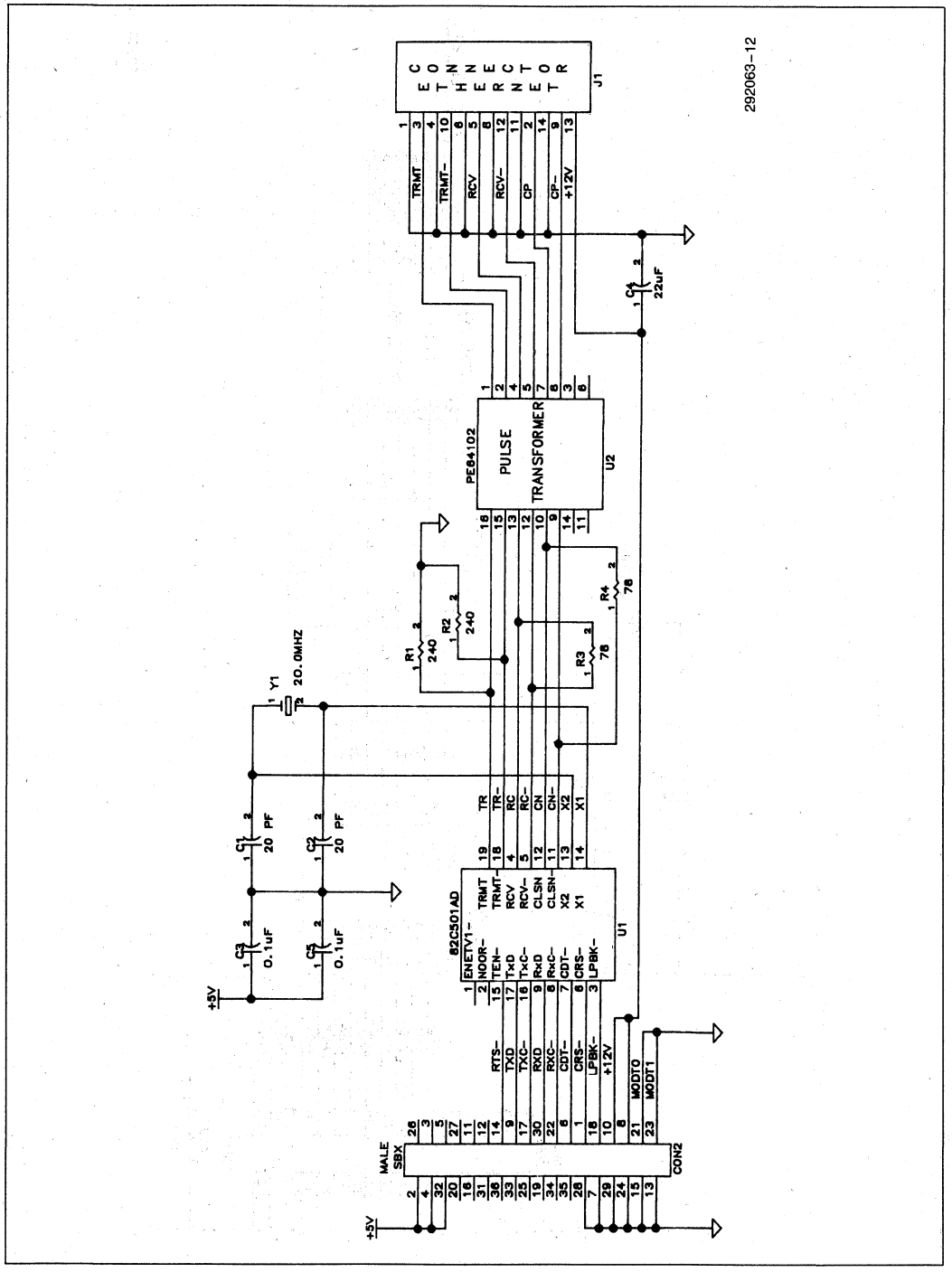


Figure 3. PC592 Schematic (Ethernet Module)

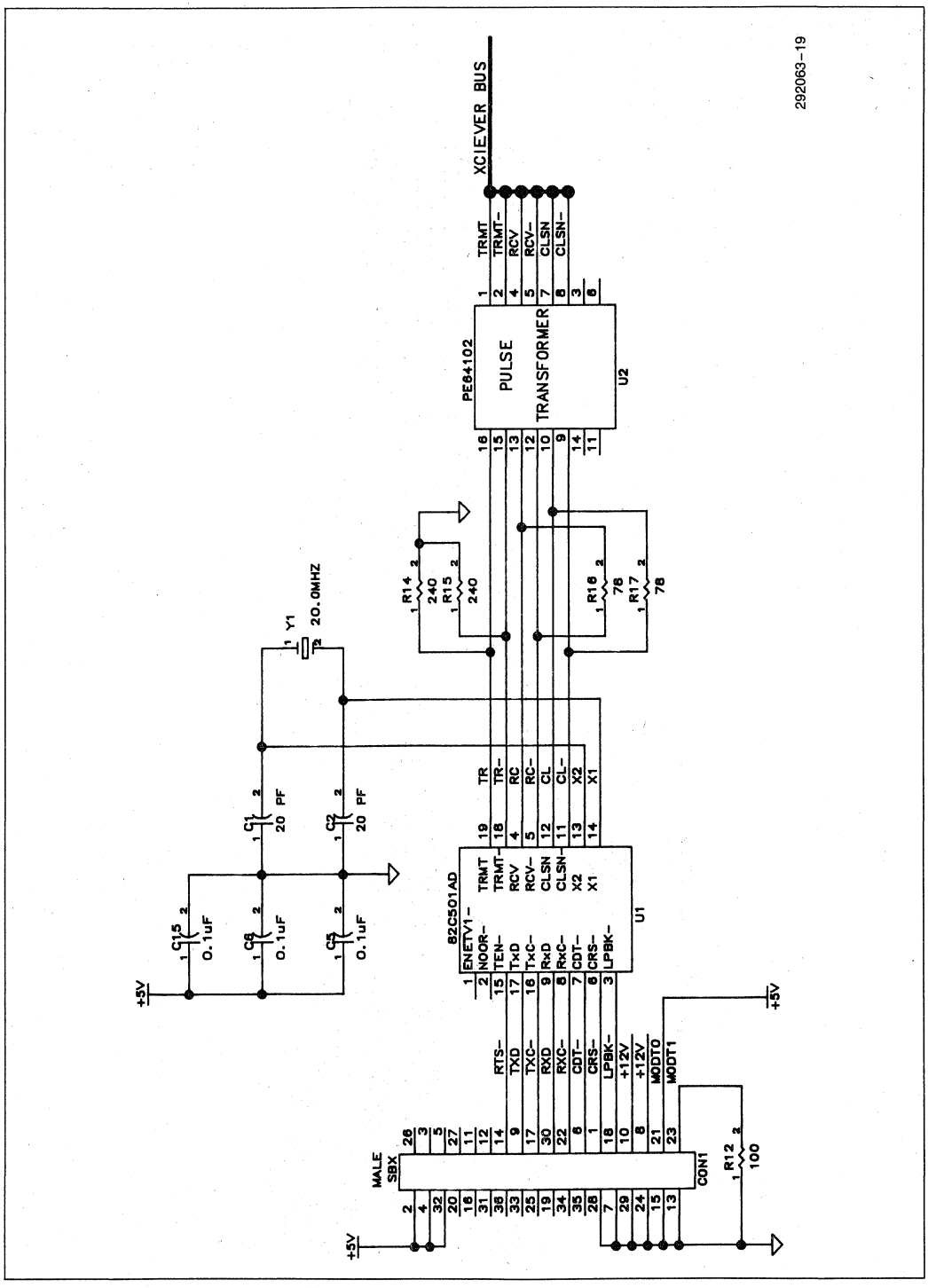


Figure 4. PC592 Schematics (Cheapernet Module)

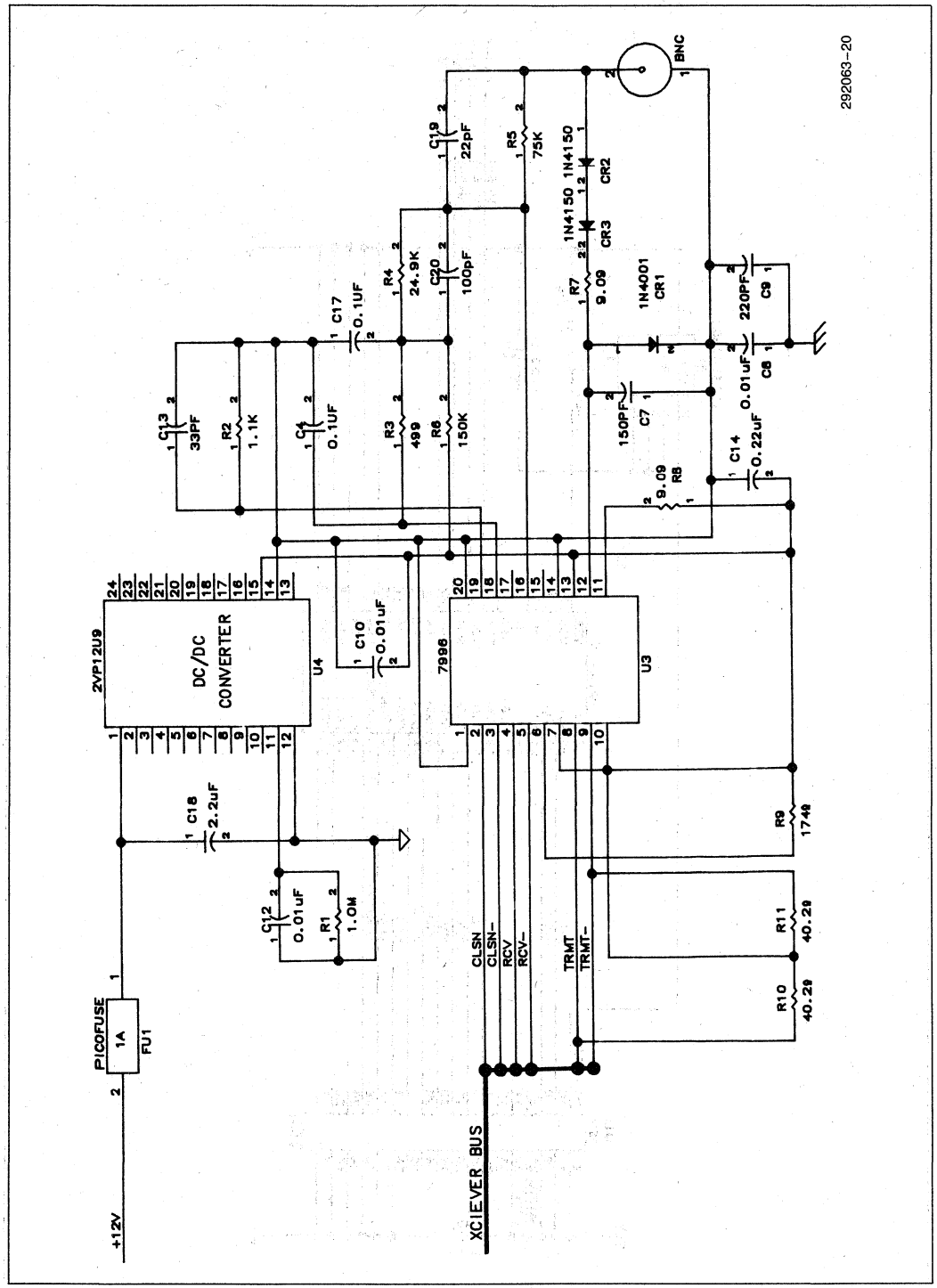


Figure 4. PC592 Schematics (Cheapernet Module) (Continued)

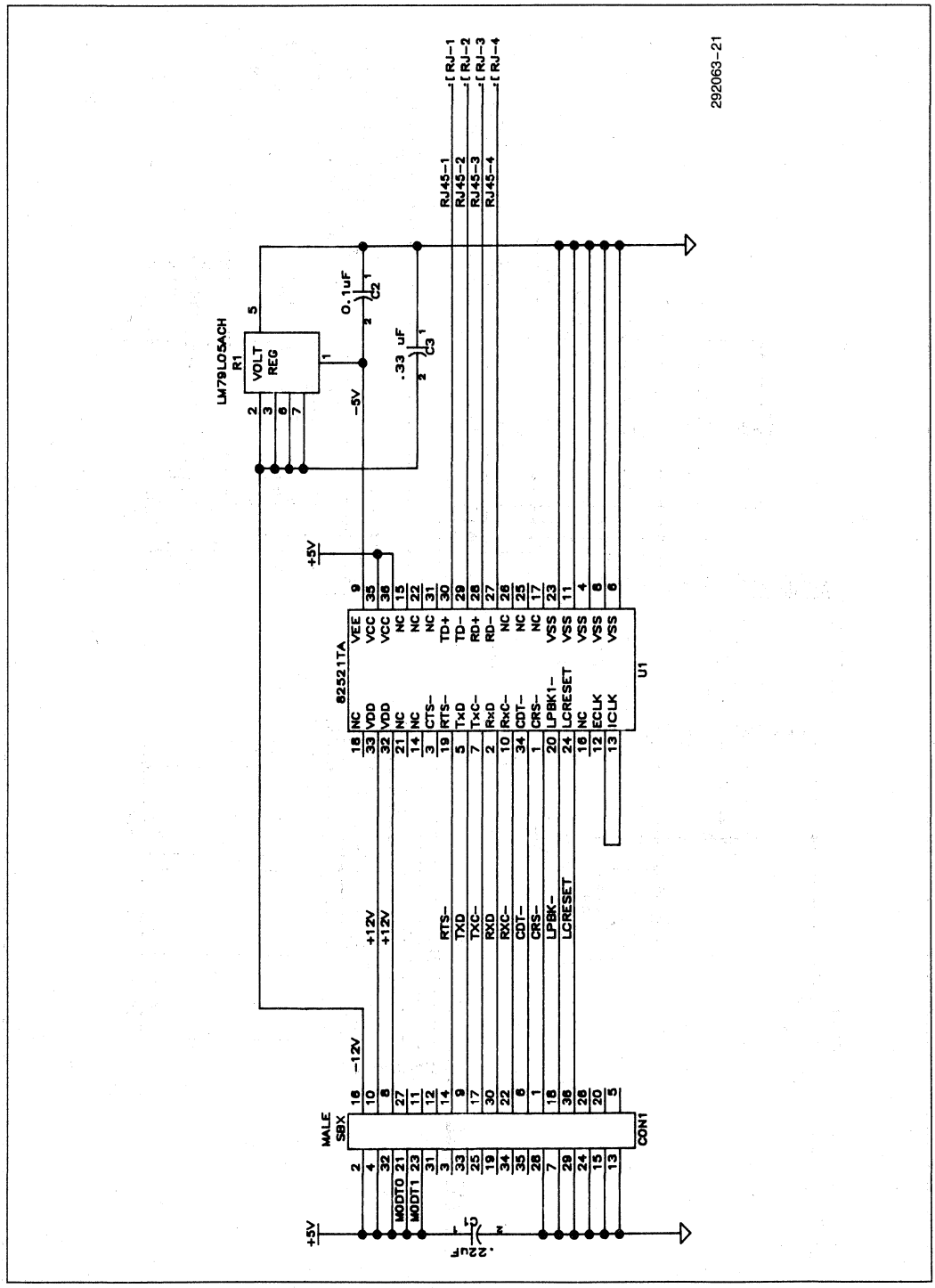


Figure 5. PC592 Schematics (TPE Module)

6.0 PERFORMANCE

The PC592E design is expected to provide very high performance compared to many commercial adapters. When the board is configured for pipeline mode, it can perform zero wait state cycles on the PC-AT bus. Without this feature each host access would be with three or four additional wait states, resulting in a less efficient use of the host bus bandwidth. In addition to reducing the host bandwidth consumption, performing zero wait state cycles should improve the network performance by about 12% over wait state cycles.

Network performance can be influenced by how fast the station can retransmit after a collision. Measurements show that an 82592/82560 design in the 82588 TCI mode can retransmit 4.8 μSec after collision. In the 82592 TCI mode this time is reduced to 2.1 μSec. The 82592 built in FIFOs reduce the possibility of underruns and overruns.

APPENDIX A

APPENDIX A

Signal name	Description
BALE	address latch enable (from the PC bus)
BHE_	byte high enable (input of 82560)
BOARDEN	board enable (command register bit 3)
CAB_	data latch clock (direction: local memory to latch)
CBA1_	data latch D7-D0 clock (direction: host to latch)
CBA2_	data latch D15-D7 clock (direction: host to latch)
CSL_, CSH_	SRAM chip select outputs of 82560 (low and high bank)
DIR_	direction signal for data latches/transceivers
EPROMCS_	EPROM chip select signal
GCS_	general chip select (output of 82560)
G1L_	data D7-D0 latch/transceiver tristate enable
G1H_	data D15-D8 latch/transceiver tristate enable
G2_	byte swap transceiver tristate enable
HF0_,HF1_	active low host request signals to the 82560
IOCHRDY	IOCHRDY to the PC bus
IORD_ / MWR_	read from non-SRAM port or write to SRAM (output of 82560)
IOWR_	write to non-SRAM port (output of 82560)
INT	interrupt output of the 82560
LA23-LA20	unlatched host address lines
LDCMD_	load command register
LD0-LD15	local data bus
LMSBDEC	latched most significant bit decode
LPBK_	loopback signal
LSBDEC	least significant bit address deccode
L560IORDY_	latched (falling edge of BALE), inverted 560 HRDY output
L560WIN_	latched 82560 subwindow
MAP0,MAP1	Jumper selections for lower portion of the address mapping
MAP2	Jumper selection that determines mapping below or above 1 Meg.
MA0-MA12	memory address lines (output of 82560)
MEMR_	memory read (from the PC bus)
MEMREQ	memory access requested by the host CPU
MEMW_	memory write (from the PC bus)
MODT0,MODT1	network module identifier (bits 4 and 5 of the status register
MOE_	memory output enable (output of 82560)
MSBDEC	most significant bit decode (output of U2)
MEMCS16_	memory chip select 16 (to the PC bus)
PCXT	8-bit vs. 16-bit configuration
PIPECYC	pipeline cycle
PIPELINE	pipeline mode (comamnd register bit 2)
PROM	command register bit determining PROM vs EPROM paging
PROMCS_	PROM chip select
RAM	command register bit determining RAM vs. ROM paging
RAMEN	RAM and board enabled
RDST_	read from the status register
REQEN	memory request enabled (output of U11)
RESET	reset (from the PC bus)
SBHE_	byte high enable (from the PC bus)
SX	select between latched and real time data
SA19-SA0	latched PC address lines
XVR1_,XVR2_	transceiver/latch enable signals (output of 82560)
ZEROWS_	zero wait state cycle (to the PC bus)
560IORDY_	buffered (inverted) HRDY output of 82560

292063-13

APPENDIX B

```
*******************************************************************************
Module U1
Title 'MEMCS16     REV 4.0   5/3/89
Daryoosh Khalilolahi,  Intel Corporation
PAL20L8B (15ns)'

PC592P1          Device  'P20L8';

LMSBDEC     Pin      1;        "I"          VCC         Pin      24;
BHE_        Pin      2;        "I"          MEMCS16_    Pin      22;      "O"
RAMEN       pin      3;        "I"          LSBDEC_     Pin      15;      "O"
MAP0        Pin      4;        "I"          L560WIN_    Pin      21;      "O"
MAP1        Pin      5;        "I"          SA0         Pin      20;      "I"
SA7         Pin      6;        "I"          SA19        Pin      23;      "I"
SA8         Pin      7;        "I"          SA18        Pin      19;      "I"
SA9         Pin      8;        "I"          SA17        Pin      18;      "I"
SA10        Pin      9;        "I"          SA16        Pin      17;      "I"
SA11        Pin     10;        "I"          SA15        Pin      16;      "I"
SA12        Pin     11;        "I"          SA14        Pin      14;      "I"
GND         Pin     12;                     SA13        Pin      13;      "I"


" SIGNAL DEFINITIONS:

"INPUTS: LMSBDEC       % latched most significant bits (LA23-LA20) decode
"        SA0           % host address input
"        RAMEN         % RAM selection anded with board enable
"        MAP0,MAP1     % Jumper selection for the  mapping window.
"        SA7-SA19      % Latched address lines from the PC bus
"        BHE_          % Byte high enable

"OUTPUTS: MEMCS16_     % MEMCS16_ for the below 1 Meg. windows
"         LSBDEC_      % Least signifiacnt bit address decode
"         L560WIN_     % 82560/command subwindow address decode

EQUATIONS


" the corresponding address lines are compared with MAP0 and MAP1 to generate
" the least significant bits address decode.


!LSBDEC_   = ((SA19 & SA18 & !SA17 & !SA16 & !SA15 & !SA14 & !MAP1 & !MAP0) #
              (SA19 & SA18 & !SA17 & !SA16 &  SA15 & !SA14 & !MAP1 &  MAP0) #
              (SA19 & SA18 & !SA17 &  SA16 & !SA15 & !SA14 &  MAP1 & !MAP0) #
              (SA19 & SA18 & !SA17 &  SA16 &  SA15 & !SA14 &  MAP1 &  MAP0));
              (SA19 & SA18 & !SA17 &  SA16 &  SA15 & !SA14 &  MAP1 &  MAP0));

"If SA7 through SA13 are all one, assuming that higher bits of address match,
" the address is for accessing the 82560 or command/status register.


!L560WIN_ =  (SA13 & SA12 & SA11 & SA10 & SA9 & SA8 & SA7);

" a 16 bit memory transfer is recognized if the address for SRAM access is
" matched and both SA0 and BHE_ are low.


!MEMCS16_ =  (LMSBDEC & RAMEN & !SA0 & !BHE_ & L560WIN_) &
             ((SA19 & SA18 & !SA17 & !SA16 & !SA15 & !SA14 & !MAP1 & !MAP0) #
              (SA19 & SA18 & !SA17 & !SA16 &  SA15 & !SA14 & !MAP1 &  MAP0) #
              (SA19 & SA18 & !SA17 &  SA16 & !SA15 & !SA14 &  MAP1 & !MAP0) #
              (SA19 & SA18 & !SA17 &  SA16 &  SA15 & !SA14 &  MAP1 &  MAP0));
end U1;
```

292063-22

```
************************************************************************
Module U2
Title ' unlatched address decode and host interface     REV 5.0   5/10/89
Daryoosh Khalilolahi,  Intel Corporation
PAL20L8B (15ns)'

PC592P2        Device   'P20L8';

RAM         Pin      1;      "I"          VCC         Pin      24;
LSBDEC_     Pin      2;      "I"          REQEN       Pin      18;     "I"
L560WIN_    pin      3;      "I"          MEMREQ      Pin      17;     "O"
LA20        Pin      4;      "I"          BOARDEN     pin      16;     "I"
LA21        Pin      5;      "I"          PIPELINE    Pin      13;     "I"
LA22        Pin      6;      "I"          LMSBDEC     Pin      23;     "I"
LA23        Pin      7;      "I"          MSBDEC      Pin      22;     "O"
BHE_        Pin      8;      "I"          RAMEN       Pin      15;     "O"
MAP2        Pin      9;      "I"          HF0_        Pin      20;     "O"
MEMW_       Pin     10;      "I"          HF1_        Pin      21;     "O"
MEMR_       Pin     11;      "I"          ZEROWS_     Pin      19;     "O"
GND         Pin     12;                   L560IORDY_Pin      14;     "I"


"Definitions

BOARDSEL = !LSBDEC_ & LMSBDEC;


" SIGNAL DEFINITIONS:

"INPUTS: RAM             % RAM selected
"        LSBDEC          % least significant bits address decode
"        LMSBDEC         % latched most significant bit address decode
"        L560WIN_        % 560/command register subwindow
"        LA20-LA23       % unlatched address lines
"        BHE_            % byte high enable
"        MAP2            % jumper selection for below/above 1 Meg.
"        MEMW_           % memory write command
"        MEMR_           % memory read command
"        PIPELINE        % command register bit 2, indicating pipeline mode
"        BOARDEN         % command register bit 3, board enabled
"        L560IORDY_      % latched and inverted 82560 HRDY output

"OUTPUTS: MSBDEC         % Most significant bits address decode
"         HF0_,HF1_      % host request to the 82560
"         ZEROWS_        % 0 wait state output to the PC bus
"         IOCHRDY        % output to the PC bus
"         MEMREQ         % host request to access SRAM

EQUATIONS

" HF0_ is asserted during SRAM and GCS (general purpose chip select) cycles.

!HF0_ = (!MEMW_ & L560WIN_ & BOARDSEL & BOARDEN & REQEN)    #
        (!MEMW_ & L560WIN_ & BOARDSEL & BOARDEN & !PIPELINE)#
        (!MEMW_ &!L560WIN_ & BOARDSEL & !BHE_) #
        (!MEMR_ & L560WIN_ & BOARDSEL & BOARDEN & REQEN)    #
        (!MEMR_ & L560WIN_ & BOARDSEL & BOARDEN & !PIPELINE)#
        (!MEMR_ &!L560WIN_ & BOARDSEL & !BHE_);

" HF1_ is asserted during 560/592 register accesses and also GCS accesses.
" That is all non-SRAM cycles.

!HF1_ = (!MEMW_ & !RAM & BOARDSEL & BOARDEN) #
        (!MEMW_ & BOARDSEL & !L560WIN_) #
```

292063-23

```
        (!MEMR_ & !RAM & BOARDSEL & BOARDEN) #
        (!MEMR_ & BOARDSEL & !L560WIN_);

" 0 Wait states cycles are performed only when accessing the SRAM and in the
" pipeline mode.

enable ZEROWS_ =  (RAM & BOARDSEL & BOARDEN & PIPELINE & L560WIN_);

ZEROWS_  = L560IORDY_;


MEMREQ = L560WIN_ & BOARDSEL & BOARDEN & RAM;

" most significant bits of address are compared with jumper node MAP2.

MSBDEC  = ((!LA20 & !LA21 & !LA22 & !LA23 & !MAP2) #
           (LA20 & LA21 & LA22 & LA23 & MAP2));

RAMEN = RAM & BOARDEN;

end U2;

*****************************************************************************
Module U5
Title 'Local bus control    REV 5.0  7/7/89
Daryoosh Khalilolahi,  Intel Corporation
PAL20L8 (25 ns)'

PC592P3        Device  'P20L8';

PCXT      Pin      1;      "I"          VCC       Pin     24;
SA0_      Pin      2;      "I"          G1L_      Pin     22;      "O"
BHE_      pin      3;      "I"          G2_       Pin     21;      "O"
HF0_      Pin      4;      "I"          CAB_      Pin     20;      "O"
MEMR_     Pin      5;      "I"          SX        Pin     19;      "O"
MEMW_     Pin      6;      "I"          DIR_      Pin     18;      "O"
MEMREQ    Pin      7;      "I"          G1H_      Pin     15;      "O"
HF1_      Pin      8;      "I"          CBA1_     Pin     17;      "O"
IORD_     Pin      9;      "I"          CBA2_     Pin     16;      "O"
IOWR_     Pin     10;      "I"          PIPELINE  Pin     14;      "I"
XCV1_     Pin     11;      "I"          unused    Pin     23;      "I"
GND       Pin     12;                   XCV2_     Pin     13;      "I"

"Definition

PIPECYC = (HF1_ & !HF0_ & PIPELINE);


"The following table shows different types of memory cycles and the expected
"value of the transceiver and data latch (74ALS646) control signals. B is the
"host side and A is the local bus side.  When SX is 1, stored data is selected.
"When 0, real time data is selected. X stands for don't care.

"cycle  Pipeline  PCXT  BHE_  SA0  MEMW_  MEMR_  G1L_  G1H_  G2_  CBA1_  CBA2_  SX
"------ --------  ----  ----  ---  -----  -----  ----  ----  ---  -----  -----  ---
"  1       1       0     0     0    0      1      0     0     1    0      0      1
"  2       1       0     0     0    1      0      0     0     1    1      1      1
"  3       0       0     1     0    X      X      0     1     1    1      1      0
"  4       0       0     0     1    X      X      1     0     1    1      1      0
"  5       0       0     0     0    X      X      0     0     1    1      1      0
"  6       1       1     1     0    0      1      0     1     1    0      1      1
"  7       1       1     1     0    1      0      0     1     1    1      1      1
"  8       1       1     0     1    0      1      0     0     0    1      0      1
"  9       1       1     0     1    1      0      1     0     0    1      1      1
"  10      0       1     1     0    X      X      0     1     1    1      1      0
"  11      0       1     0     1    X      X      1     0     0    1      1      0
```

292063-24

```
" cycle 1 : PCAT pipelined (word-wide) memory write
" cycle 2 : PCAT pipelined (word-wide) memory read
" cycle 3 : PCAT nonpipelined even-byte memory access
" cycle 4 : PCAT nonpipelined odd-byte memory access
" cycle 5 : PCAT nonpipelined word-wide memory access
" cycle 6 : PCXT pipelined even-byte memory write
" cycle 7 : PCXT pipelined even-byte memory read
" cycle 8 : PCXT pipelined odd-byte memory write
" cycle 9 : PCXT pipelined odd-byte memory read
" cycle 10 : PCXT nonpipelined even-byte memory access
" cycle 11 : PCXT nonpipelined odd-byte memory access


" SIGNAL DEFINITION

"INPUTS:  PCXT            % Jumper slecting 8 bit or 16 bit machine
"         SA0_            % inverted address line 0
"         MEMR_           % MEMR_ from the PC bus
"         MEMW_           % MEMW_ from the PC bus
"         IORD_           % IORD_/MEMWR_ output of the 82560
"         IOWR_           % IOWR_ output of the 82560
"         XCV1_           % XCV1_ output of the 82560
"         XCV2_           % XCV2_ output of the 82560
"         HF0_,HF1_       % host requests to the 82560
"         PIPELINE        % command register bit for 0 added W.S. cycles

"OUTPUTS: G1L_            % Data transceiver/latch tristate enable (D7-D0)
"         G1H_            % Data transceiver/latch tristate enable (D15-D8)
"         G2_             % local bus transceiver enable signal
"         CAB_            % Local data latch clock
"         CBA1_           % latch host data (D7-D0)
"         CBA2_           % latch host data (D15-D8)
"         SX              % Real time vs. latched data select
"         DIR_            % Data transceiver/latch direction

EQUATIONS

!G1L_   = (MEMREQ & PCXT & SA0_ & !MEMR_) # (!PCXT & PIPECYC & MEMR_) #
          (!XCV1_ & SA0_) # ( !HF0_ & !HF1_) # (!XCV2_ & HF1_);

!G1H_   = (PIPECYC & !MEMR_) # (!XCV1_ & !BHE_) # (!XCV2_ & HF1_);

!G2_    = (PCXT & MEMREQ & !SA0_);

!CAB_ = (!IOWR_ & !XCV2_ & HF1_);

!CBA1_ = (PIPECYC & !MEMW_ & !PCXT) # (MEMREQ & PIPELINE &!MEMW_ & PCXT & SA0_)

!CBA2_ = (PIPECYC & !MEMW_);

SX    =  (MEMREQ & PIPELINE #  !IORD_ & PIPELINE & !XCV2_ & HF1_);

!DIR_ = (!XCV2_ & HF1_ & !IORD_) # (!MEMW_);


end U5;


*****************************************************************************
Module U11
Title 'Local ports control    REV 3.0   5/3/89
Daryoosh Khalilolahi,  Intel Corporation
PAL16L8B-2 (25ns)'

PC592P4       Device  'P16L8';
```

292063-25

```
LSBDEC_    Pin     1;      "I"          VCC        Pin     20;
IORDY_     Pin     2;      "I"          EPROMCS_   Pin     19;     "O"
BSA0       pin     3;      "I"          PROMCS_    Pin     18;     "O"
PCXT       Pin     4;      "I"          RDST_      Pin     17;     "O"
GCS_       Pin     5;      "I"          LDCMD      Pin     16;     "O"
L560WIN_   Pin     6;      "I"          LMSBDEC    Pin     15;     "I"
IOWR_      Pin     7;      "I"          REQEN      Pin     14;     "O"
IORD_      Pin     8;      "I"          IOCHRDY    Pin     13;     "O"
BOARDEN    Pin     9;      "I"          unuseda    Pin     12;     "O"
GND        Pin     10;     "I"          PROM       Pin     11;     "I"


" SIGNAL DEFINITION

"INPUTS:   GCS_            % General chip select ouput of 82560
"          L560WIN_        % 82560/command register subwindow
"          IORD_           % IORD_/MEMWR_ output of the 82560
"          IOWR_           % IOWR_ output of the 82560
"          PROM            % PROM selected
"          LSBDEC_         % least significant bits address decode
"          LMSBDEC         % latched most significant bits address decode
"          IORDY_          % inverted 82560 HRDY output
"          BSA0            % buffered address line 1
"          BOARDEN         % Board enabled

"OUTPUTS:
"          EPROMCS_        % EPROM chip select
"          PROMCS_         % PROM chip select
"          LDCMD           % load command register
"          RDST_           % read status register
"          IOCHRDY         % IOCHRDY to the PC bus
"          REQEN           % memory request enabled or a non-SRAM access

EQUATIONS

EPROMCS_ = !(!GCS_ & !IORD_ & !PROM & L560WIN_);

PROMCS_  = !(!GCS_ & !IORD_ & PROM & L560WIN_);

LDCMD    = !GCS_ & !L560WIN_ & !IOWR_;

RDST_    = !(!GCS_ & !L560WIN_ & !IORD_);

REQEN    = (PCXT & BSA0) # (!PCXT) # (!L560WIN_);

enable IOCHRDY = (!LSBDEC_ & LMSBDEC);

IOCHRDY = (!IORDY_) # (!LSBDEC_ & LMSBDEC & !BOARDEN & L560WIN_)
                    # (!LSBDEC_ & LMSBDEC & !BOARDEN & !BSA0);

end;
```

292063-26

Wide Area Networks

2

intel®

APPLICATION NOTE

AP-401

December 1986

Designing With the 82510 Asynchronous Serial Controller

FAISAL IMDAD-HAQUE
APPLICATIONS ENGINEER

Order Number: 231928-001

1.0 INTRODUCTION

The emergence of asynchronous communications as the most widely used protocol (it commands the largest installed base of nodes, exceeding HDLC/SDLC, the second most popular protocol, by a factor of 10 to 1) for point to point serial links has led to the need for an asynchronous communications component with high integration to reduce component count and decrease the cost of a serial port. The trend towards higher data rates and multiple job, multiple user systems has underscored the need for an intelligent serial controller to improve system throughput and decrease the CPU load normally associated with asynchronous serial communications.

The 82510 CMOS Asynchronous Serial Controller is designed to improve asynchronous communications throughput and reduce system cost by integrating functions and simplifying the system interface. Two independent FIFOs, and Control Character Recognition (CCR), provide data buffering and increase software efficiency. Two Baud Rate Generators/Timers, an On-Chip Crystal Oscillator and seven Programmable I/O pins provide a high degree of integration and reduce system component count. This application note will demonstrate the use of these features in an Asynchronous Communications Environment.

1.1 Goal

The goal of this application note is to demonstrate the use of the major 82510 features in an asynchronous communications environment and to depict basic hardware and software design techniques for the 82510. It will discuss interfaces using both polling and interrupt techniques, as well as the impact of FIFOs using either scheme. An application example covering the application of Error Free File Transfer is also provided.

1.2 Scope

The application note describes the operation of the 82510 ASC in a normal (non 8051 9-bit) asynchronous communications mode. The majority of the discussion is focused towards the systems aspects of the Controller. The use of the 82510 in a multidrop or 8051 9-bit asynchronous environment is not covered. This application note assumes that the reader is familiar with the 82510 in terms of pin description, register architecture and interrupt structure. It is also assumed that the reader is familiar with the information provided in the 82510 Data Sheet.

The initial sections of the application note provide an overview of the 82510 and its major functional blocks. This is followed by a discussion of the hardware design and system interface considerations in sections three and four. The fifth section provides some software techniques for transmitting and receiving data as well as the use of timers. Section seven briefly discusses the file transfer application based on the XMODEM protocol and includes the software listings.

2.0 82510 DESCRIPTION

2.1 Overview

The 82510 can be divided into seven functional blocks (See Figure 1): Bus Interface Unit (BIU), Timing Unit, Modem Interface Module, Tx FIFO, Rx FIFO, Tx Machine and Rx Machine. All blocks, except BIU can generate a block interrupt request to the 82510 interrupt logic. In the case of the Rx Machine, Timing Unit and Modem Interface Module, multiple sources (errors and status events) within the block cause the block interrupt request to become active. All of the blocks have registers associated with them. The registers, allow configuration, provide status information about events/errors, and may also be used to send commands to each block.

2.2 Bus Interface Unit (BIU)

The Bus Interface Unit (BIU) interfaces the 82510 functional blocks to the system or CPU bus. It provides read and write access to the 82510 registers and controls the generation of interrupts to the external world. The interrupt logic resolves contention between block interrupt requests, on a priority basis. The BIU also has the Hardware Reset circuitry, which is driven by the RESET pin. The reset signal clears all internal Flip Flops, and Registers and puts them in a predefined state. All activities on the Bus interface, including register accesses by the CPU, are synchronized to an internal (82510) system clock, supplied via the CLK pin.

2.3 Receive Machine (RxM)

The Rx Machine (RxM) converts the serial data to parallel and writes it to the Rx FIFO, along with the appropriate flags (available in the *Receive Flags Register*). The Rx Machine can be configured for control character recognition, data sampling and DPLL operation. The software can check for noise, control character, break, address or parity and framing errors by reading the status or character flags. Optionally, the Receive Status bits (in RST), when enabled, can generate interrupt requests. The Rx Machine block Interrupt request is reflected in the *General Status Register* and is set when an enabled interrupt request within the Rx Ma-

chine (i.e. RST bits) becomes active. The Rx Machine has eight registers associated with it:

Receive Data (RXD)—Receive Data Character

Receive Flags (RXF)—Receive Character Flags

Receive Status (RST)—Receive Events and Receive Errors

Receive Interrupt Enable (RIE)—Enables Interrupts on corresponding bits in RST

Receive Mode (RMD)—Receive Machine Configuration

Receive Command (RCM)—Receiver Command Register

Line Control (LCR)—16450 Register, Character Attribute Configuration

Line Status (LSR)—16450 Status Register, Tx and Rx status

2.4 Transmit Machine (TxM)

The Tx Machine reads characters from the Tx FIFO and transmits them serially over the TXD line. The Tx Machine can also transmit additional character attributes (9th bit of Data, Address Marker, Software Parity) available from the Transmit Flags, if configured in the appropriate mode. The Tx Machine Idle interrupt request is reflected in the GSR and LSR registers to indicate that the Transmitter is either Empty or Disabled. The Tx Machine has six registers associated with it:

Line Control (LCR)—16450 Register, Character Attribute Configuration

Line Status (LSR)—16450 Status Register, Tx and Rx status

Transmit Mode (TMD)—Tx Machine Configuration

Transmit Command—(TCM)—Transmit Command Register

Transmit Flags (TXF)—Transmit Character Flags

Transmit Data (TXD)—Transmit Data Character

2.5 Modem Interface Module

The Modem Interface module is responsible for the modem interface and general purpose I/O pins. It will generate Interrupts (if enabled) upon transitions in the modem input pins ($\overline{DCD}$, $\overline{CTS}$, $\overline{RI}$, and $\overline{DSR}$). The modem output pins can be controlled by the CPU, also the RTS pin can be used to provide flow control, in the automatic transmission mode. It is the source of the Modem Interrupt bit in GSR. This bit is set whenever there is a state change in the $\overline{DCD}$, $\overline{RI}$, $\overline{DSR}$ or $\overline{CTS}$ inputs (reflected in *Modem Status Register*) and the corresponding enable bits are set. The function and direction of the multifunction pins can be reprogrammed and is available as a configuration option. Multifunction pins, when configured as outputs, can be controlled by the CPU through the *Modem Control Register.* The Modem module has four registers associated with it:

Modem Status
Register (MSR)—State transitions on modem input pins, and State of the modem input pins

Modem Control (MCR)—Control state of Modem Output pins

Modem Interrupt
Enable (MIE)—Enable Interrupt on State transitions in modem input pins

I/O Pin Mode (PMD)—Functions and Directions of Multifunction pins

2.6 Timing Unit

The Timing Unit is responsible for the generation of the System Clock, using either its Crystal Oscillator or an externally generated clock, and generation of the Tx and Rx clocks from either the On-Chip Baud Rate Generators or the SCLK pin. It is also responsible for generating Timer Expired interrupts when the BRGs/Timers are configured for use as Timers. There are ten registers associated with the Timing Unit, four of these are used in the Timer mode only.

Timer Status (TMST)—Timer A and/or Timer B expired

Timer Interrupt
Enable (TMIE)—Enables Interrupts upon Expiration of Timers A or B in TMST

Timer Control (TMCR)—Start and Disable Timers

Clock Configure (CLCF)—Select source and mode for Tx and Rx clocks

BRG B Configuration (BBCF)—Mode and Clock source of BRG B

BRG B LSB of Divisor (BBL)—Least Significant Byte of BRG B Divisor/Count

BRG A MSB of Divisor (BBH)—Most Significant Byte of BRG B Divisor/Count

BRG A Configuration (BACF)—Mode and Clock source of BRG A

BRG A LSB of Divisor (BAL)—Least Significant Byte of BRG A Divisor/Count

BRG A MSB of Divisor (BAH)—Most Significant Byte of BRG A Divisor/Count

2.7 FIFOs, Rx and Tx

The Dual FIFOs (transmit and receive), serve as buffers for the 82510. They buffer the transmitter and Receiver from the CPU. Each of the FIFOs has a programmable threshold. The threshold is the FIFO level which will generate an interrupt. The threshold is used to optimize the CPU throughput and provide increased interrupt to service latency for higher baud rates. It can be configured through the FIFO Mode Register. Each FIFO character has flags associated with it (TxF and RxF). As each character is read from the Rx FIFO its flags are put into the RxF register. Before a write to TXD (if character configuration requires) the character flags are written to the TXF register. The two FIFOs are totally independent of each other and each FIFO can generate an interrupt request which indicates that the configured threshold has been met.

3.0 HARDWARE DESIGN

3.1 System Interface

The 82510 has a standard I/O peripheral interface, it has a demultiplexed Bus, which consists of a bidirectional eight bit Data Bus, and three Address lines. Interrupt, Read, Write, Chip Select and Reset pins complete the system interface. The three address lines along with the *Bank register* are used to select a particular register.

3.1.1 REGISTER ACCESS

The 82510 registers are logically divided into four banks. Only one bank can be accessed at any one time. Each register bank occupies eight I/O addresses. To select a register, the correct Bank must first be selected by writing to the GIR/Bank register (the *GIR/Bank register* I/O address is two (A_0 = 0, A_1 = 1, A_2 = 0). Then one of the eight I/O space addresses is selected by outputting a value (between zero and seven) to the 82510 address pins A_0–A_2.

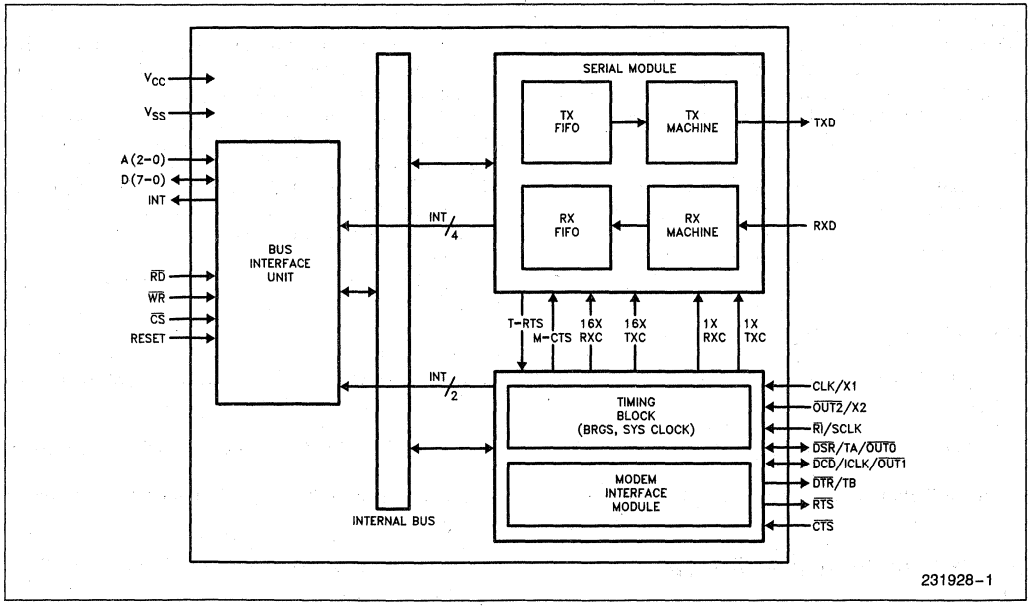


Figure 1. 82510 Block Diagram

BANK ZERO 8250—COMPATIBLE BANK										
Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Address**	**Default**
TxD	Tx Data bit 7	Tx Data bit 6	Tx Data bit 5	Tx Data bit 4	Tx Data bit 3	Tx Data bit 2	Tx Data bit 1	Tx Data bit 0	0	—
RxD	Rx Data bit 7	Rx Data bit 6	Rx Data bit 5	Rx Data bit 4	Rx Data bit 3	Rx Data bit 2	Rx Data bit 1	Rx Data bit 0	0	—
BAL	BRGA LSB Divide Count (DLAB = 1)								0	02H
BAH	BRGA MSB Divide Count (DLAB = 1)								1	00H
GER	0	0	Timer Interrupt Enable	Tx Machine Interrupt Enable	Modem Interrupt Enable	Rx Machine Interrupt Enable	Tx FIFO Interrupt Enable	Rx FIFO Interrupt Enable	1	00H
GIR/BANK	0	BANK Pointer bit 1	BANK Pointer bit 0	0	Active Block Int bit 2	Active Block Int bit 1	Active Block Int bit 0	Interrupt Pending	2	01H
LCR	DLAB Divisor Latch Access bit	Set Break	Parity Mode bit 2	Parity Mode bit 1	Parity Mode bit 0	Stop bit Length bit 0	Character Length bit 1	Character Length bit 0	3	00H
MCR	0	0	OUT 0 Complement	Loopback Control bit	OUT 2 Complement	OUT 1 Complement	RTS Complement	DTR Complement	4	00H
LSR	0	TxM Idle	Tx FIFO Interrupt	Break Detected	Framing Error	Parity Error	Overrun Error	Rx FIFO Int Reg	5	60H
MSR	DCD Input Inverted	RI Input Inverted	DSR Input Inverted	CTS Input Inverted	State Change in DCD	State (H → L) Change in RI	State Change in DSR	State Change in CTS	6	00H
ACR0	Address or Control Character Zero								7	00H

BANK ONE—GENERAL WORK BANK										
Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Address**	**Default**
TxD	Tx Data bit 7	Tx Data bit 6	Tx Data bit 5	Tx Data bit 4	Tx Data bit 3	Tx Data bit 2	Tx Data bit 1	Tx Data bit 0	0	—
RxD	Rx Data bit 7	Rx Data bit 6	Rx Data bit 5	Rx Data bit 4	Rx Data bit 3	Rx Data bit 2	Rx Data bit 1	Rx Data bit 0	0	—
RxF	—	Rx Char OK	Rx Char Noisy	Rx Char Parity Error	Address or Control Character	Break Flag	Rx Char Framing Error	Ninth Data bit of Rx Char	1	—
TxF	Address Marker bit	Software Parity bit	Ninth bit of Data Char	0	0	0	0	0	1	—
GIR/BANK	0	BANK Pointer bit 1	BANK Pointer bit 0	0	Active Block Int bit 2	Active Block Int bit 1	Active Block Int bit 0	Interrupt Pending	2	01H
TMST	—	—	Gate B State	Gate A State	—	—	Timer B Expired	Timer A Expired	3	30H
TMCR	0	0	Trigger Gate B	Trigger Gate A	0	0	Start Timer B	Start Timer A	3	—
MCR	0	0	OUT 0 Complement	Loopback Control bit	OUT 2 Complement	OUT 1 Complement	RTS Complement	DTR Complement	4	00H

Figure 2. 82510 Register Map

BANK ONE—GENERAL WORK BANK (Continued)										
Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Address**	**Default**
FLR	—	Rx FIFO Level			—	Tx FIFO Level			4	00H
RST	Address/ Control Character Received	Address/ Control Character Match	Break Terminated	Break Detected	Framing Error	Parity Error	Overrun Error	Rx FIFO Interrupt Requested	5	00H
RCM	Rx Enable	Rx Disable	Flush RxM	Flush Rx FIFO	Lock Rx FIFO	Open Rx FIFO	0	0	5	—
MSR	DCD Complement	RI Input Inverted	DSR Input Inverted	CTS Input Inverted	State Change in DCD	State Change in RI	State Change in DSR	State Change in CTS	6	00H
TCM	0	0	0	0	Flush Tx Machine	Flush Tx FIFO	Tx Enable	Tx Disable	6	—
GSR	—	—	Timer Interrupt	TxM Interrupt	Modem Interrupt	RxM Interrupt	Tx FIFO Interrupt	Rx FIFO Interrupt	7	12H
ICM	0	0	0	Software Reset	Manual Int Acknowledge Command	Status Clear	Power Down Mode	0	7	—

BANK TWO—GENERAL CONFIGURATION										
Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Address**	**Default**
FMD	0	0	Rx FIFO Threshold		0	0	Tx FIFO Threshold		1	00H
GIR/BANK	0	BANK Pointer bit 1	BANK Pointer bit 0	0	Active Block Int bit 2	Active Block Int bit 1	Active Block Int bit 0	Interrupt Pending	2	01H
TMD	Error Echo Disable	Control Character Echo Disable	9-bit Character Length	Transmit Mode		Software Parity Mode	Stop Bit Length		3	00H
IMD	0	0	0	0	Interrupt Acknowledge Mode	Rx FIFO Depth	ulan Mode Select	Loopback or Echo Mode of Operation	4	0CH
ACR1	Address or Control Character 1								5	00H
RIE	Address/ Control Character Recognition Interrupt Enable	Address/ Control Character Match Interrupt Enable	Break Terminate Interrupt Enable	Break Detect Interrupt Enable	Framing Error Interrupt Enable	Parity Error Interrupt Enable	Overrun Error Interrupt Enable	0	6	1EH
RMD	Address/Control Character Mode		Disable DPLL	Sampling Window Mode	Start bit Sampling Mode	0	0	0	7	00H

BANK THREE—MODEM CONFIGURATION										
Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Address**	**Default**
CLCF	Rx Clock Mode	Rx Clock Source	Tx Clock Mode	Tx Clock Source	0	0	0	0	0	00H
BACF	0	BRGA Clock Source	0	0	0	BRGA Mode	0	0	1	04H
BBL	BRGB LSB Divide Count (DLAB = 1)								0	05H
BBH	BRGB MSB Divide Count (DLAB = 1)								1	00H

Figure 2. 82510 Register Map (Continued)

BANK THREE—MODEM CONFIGURATION (Continued)										
Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Address**	**Default**
GIR/BANK	0	BANK Pointer bit 1	BANK Pointer bit 0	0	Active Block Int bit 2	Active Block Int bit 1	Active Block Int bit 0	Interrupt Pending	2	01H
BBCF	BRGB Clock Source		0	0	0	BRGB Mode	0	0	3	84
PMD	DCD/ICLK/ OUT 1 Direction	DCD/ICLK/ OUT 1 Function	DSR/TA/ OUT 0 Direction	DSR/TA/ OUT 0 Function	RI/SCLK Function	DTR/TB Function	0	0	4	FCH
MIE	0	0	0	0	DCD State Change Int Enable	RI State Change Int Enable	DSR State Change Int Enable	CTS State Change Int Enable	5	0FH
TMIE	0	0	0	0	0	0	Timer B Interrupt Enable	Timer A Interrupt Enable	6	00H

Figure 2. 82510 Register Map (Continued)

3.1.2 READ AND WRITE CYCLES

Like most other I/O based peripherals the Read and Write pins are used to access data in the 82510. Each read or write cycle has specified setup and hold times in order for the data to be transferred correctly to/from the 82510. The critical timings for the read cycle are:

1. Address Valid to Read Active (Tavrl)
2. Command Access Time to Data Valid (Trldv)
3. Command Active Width (Trlrh)

The less critical parameters are:

4. Address Hold to Read Inactive (Trhax)
5. Data Out Float Delay after Read Inactive (Trhdz)

The critical timings for the write cycle are:

1. Address Valid to Write Low (Tavwl)
2. Write Active Time (Twlwh)
3. Data Valid to Write Inactive (Tdvwh)

The less critical parameters are:

4. Address and Chip Select Hold Time After Write Inactive (Twhax)
5. Data Hold Time After Write Inactive (Twhdx)

These timings determine the number of wait states required for the 82510 and the CPU interface. The interfaces for some popular microprocessors are discussed in the following sections.

3.1.3 80186 INTERFACE

The exact interface is shown in Figure 3. The schematic shows the 80186 interface to the 82510 on a local bus. Although the Data Bus is buffered, it is possible to directly connect the 82510 to the 80186 data bus; because the Data Float Delay after read inactive is 40 ns for the 82510, which is well under the 85 ns requirement of the 80186. The timing equations for the interface are given below.

Read Cycle:

$$\text{Address to Read Low} = \text{Tclcl} - \text{Tclav}_{max} + \text{Tclrl}_{min} - \text{Latch Delay}_{max}$$

$$\text{Read Access Time} = 2\text{Tclcl} - \text{Tclrl}_{max} - \text{Tdvcl} - \text{Transceiver Delay}_{max}$$

$$\text{Read Active Time} = \text{Trlrh} = 2\text{Tclcl} - 46$$

Write Cycle:

$$\text{Address Valid to Write Active} = \text{Tclcl} + \text{Tcvctv}_{min} - \text{Tclav}_{max} - \text{Latch Prop. Delay}_{max}$$

$$\text{Write Active Time} = \text{Twlwh} = 2\text{Tclcl} - 40$$

$$\text{Data Valid to Write Inactive} = 2\,\text{Tclcl} - \text{Tcldv}_{max} - \text{Transceiver Delay}_{max} + \text{Tcvctx}_{min}$$

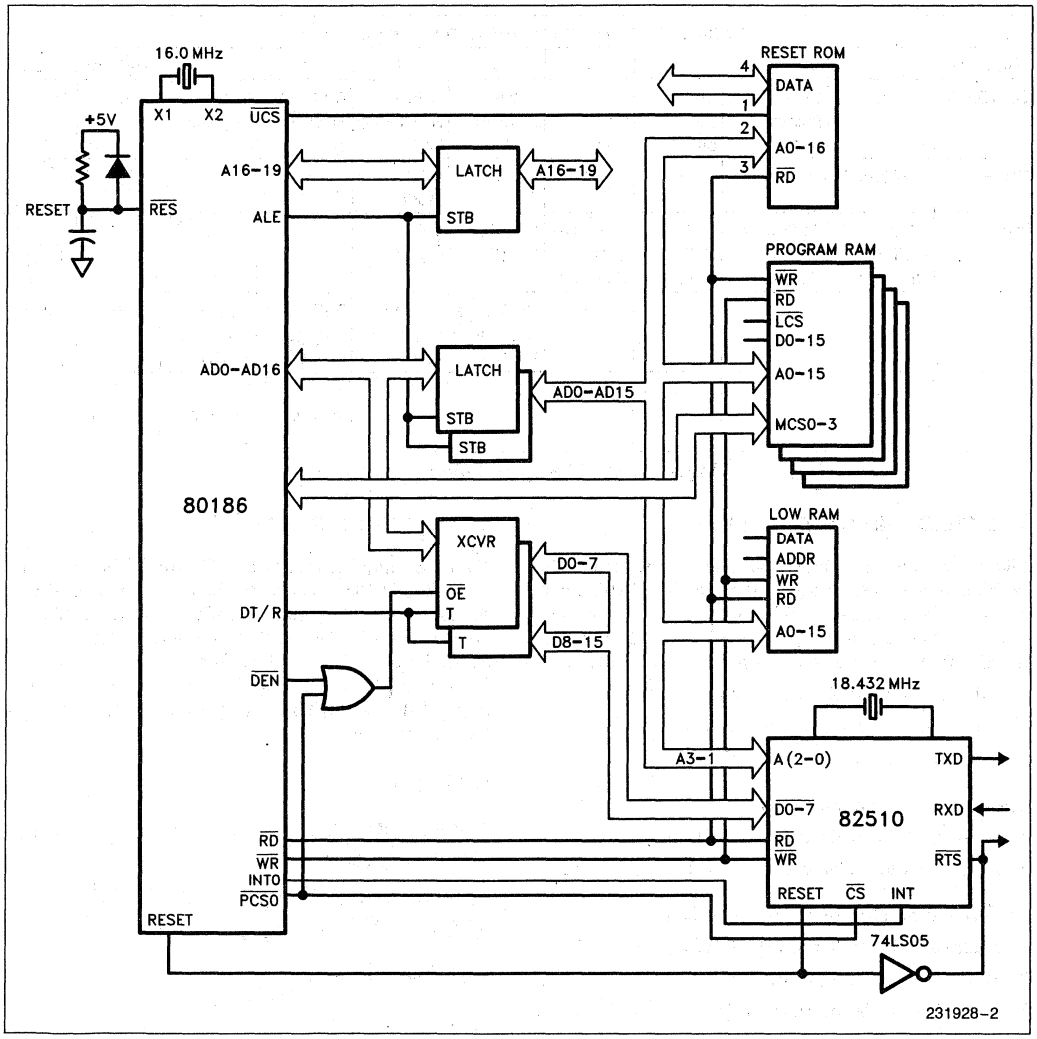


Figure 3. 82510 Interface to 80186

The user can transfer data to the 82510, using the DMA capabilities of the 80186, by using the $\overline{RTS}$ pin, in automatic modem control mode, as a DMA request line. The $\overline{RTS}$ pin, in automatic mode, will go inactive as soon as the Tx FIFO and the Tx shift register are empty. It will become active once a data character is written to the *TXD register.* In most 80186 DMA transfers the user has to make sure that the DMA request line goes inactive at least two clock cycles from the end of the DMA deposit cycle. In this case, the extra DMA cycle is not a problem, because the Tx FIFO will buffer the data to prevent an overrun (Since the Tx FIFO can buffer up to four characters, the $\overline{RTS}$ pin only needs to go inactive two clocks before the end of the deposit phase of the fourth DMA). Typically $\overline{RTS}$ will go inactive five (82510) system clocks after the rising edge of write.

3.1.4 80286 INTERFACE

The 80286 interface is shown in Figure 4. The 82510 is on the local bus, and is using the control signals from the 82288 Bus Controller. The Data Enable (OE) is qualified by the 82510 Chip Select, to avoid Data Bus contention between the 82510 and the CPU. The timing equations for the Read and Write Cycles are given below.

Read Cycle:

Address Valid to Read Active = T1 (CLK period) + $T29_{min}$ (CLK to cmd active) − $T16_{max}$ (ALE active delay) − Latch Prop. $Delay_{max}$

Read Access to Data Valid = 2T1 (CLK period) − $T29_{max}$ (CLK to cmd active) − T8 (Read Data Setup Time) − Transceiver $Delay_{max}$

Read Active Time = 2T1 (CLK period) − $T29_{max}$ (CLK to cmd active) + $T30_{min}$ (CLK to cmd inactive)

Write Cycle:

Address to Write Low = T1 (CLK period) + $T29_{min}$ (CLK to cmd active) − $T16_{max}$ (ALE active delay) − Latch $Delay_{max}$

Write Active Time = 2 T1 (CLK period) − $T29_{max}$ (CLK to cmd active) + $T30_{min}$ (CLK to cmd inactive)

Data to Write High = 3 T1 − $T14_{min}$ (Write Data Valid Delay) + $T30_{min}$ (CLK to cmd inactive) − Xcvr. $Delay_{max}$

Using an 8 MHz 80286 with the 82510 at 18.432 MHz (divide by two—9.216 MHz) requires two wait states. The critical timings are the read cycle timings—Read Access Time and Read Active Width. Inserting two wait states means that the access times for the relevant parameters will be increased by 250 ns.

NOTE:
The address decoding scheme of the 80286 interface is different from the IBM PC/PC AT I/O addresses for the serial ports, therefore the interface shown in Figure 4 cannot be used in PC/PC AT oriented designs.

3.1.5 80386 INTERFACE

The 80386 interface to the 82510 is given in Figure 5. The example uses the Basic I/O interface given in the 80386 Hardware Reference Manual section 8.3. The only differences are in the specific address lines used for chip select generation, and the additional wait states in the wait state generation logic. The address lines A3, A4 and A5 are used to select one of the eight register address spaces in the 82510, therefore, A6 and A7, rather than A4 and A5, are used in the I/O decoder. This causes a granularity of four in the 82510's I/O address space, i.e., the addresses of two consecutive registers in the 82510 differ by four.

The 82510 requires one additional wait state (as currently specified), the design assumes that the PAL equations are modified for that purpose. The user may also externally generate the wait states and connect to the "other ready logic" input ORed with the RDY pin of PAL 2. The two read timings Read Active width and Read Access time to Data Valid each require one additional wait state in order to meet the 82510 timing requirements. The timings are given below. (82510 times are at 9.216 MHz)

Read Cycle:

Read Access to Data Valid	= 253.25 ns
82510 Trldv	= 308
additional time reqd.	= 308-253.25
	= 54.75 ns
Read Active Width	= 269.25
82510 Trlrh	= 308
additional time reqd.	= 308-269.25
	= 38.75 ns
Address Valid to Read Active	= 132.75 ns
82510 T_{AVRL}	= 7 ns

Since each additional wait state adds 62.5 ns at 16 MHz, the 82510 requires one additional wait state.

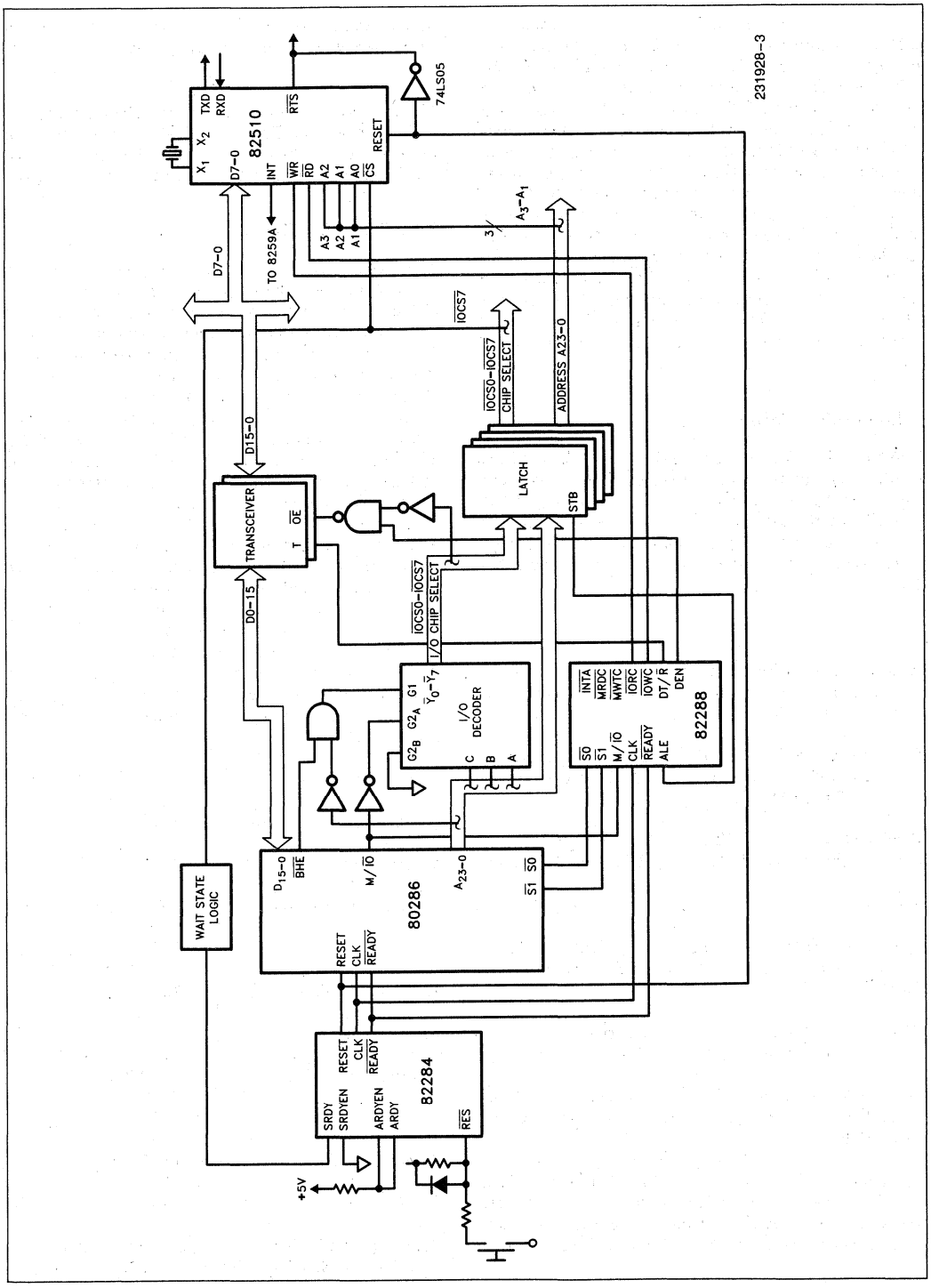


Figure 4. 80286 Interface to the 82510

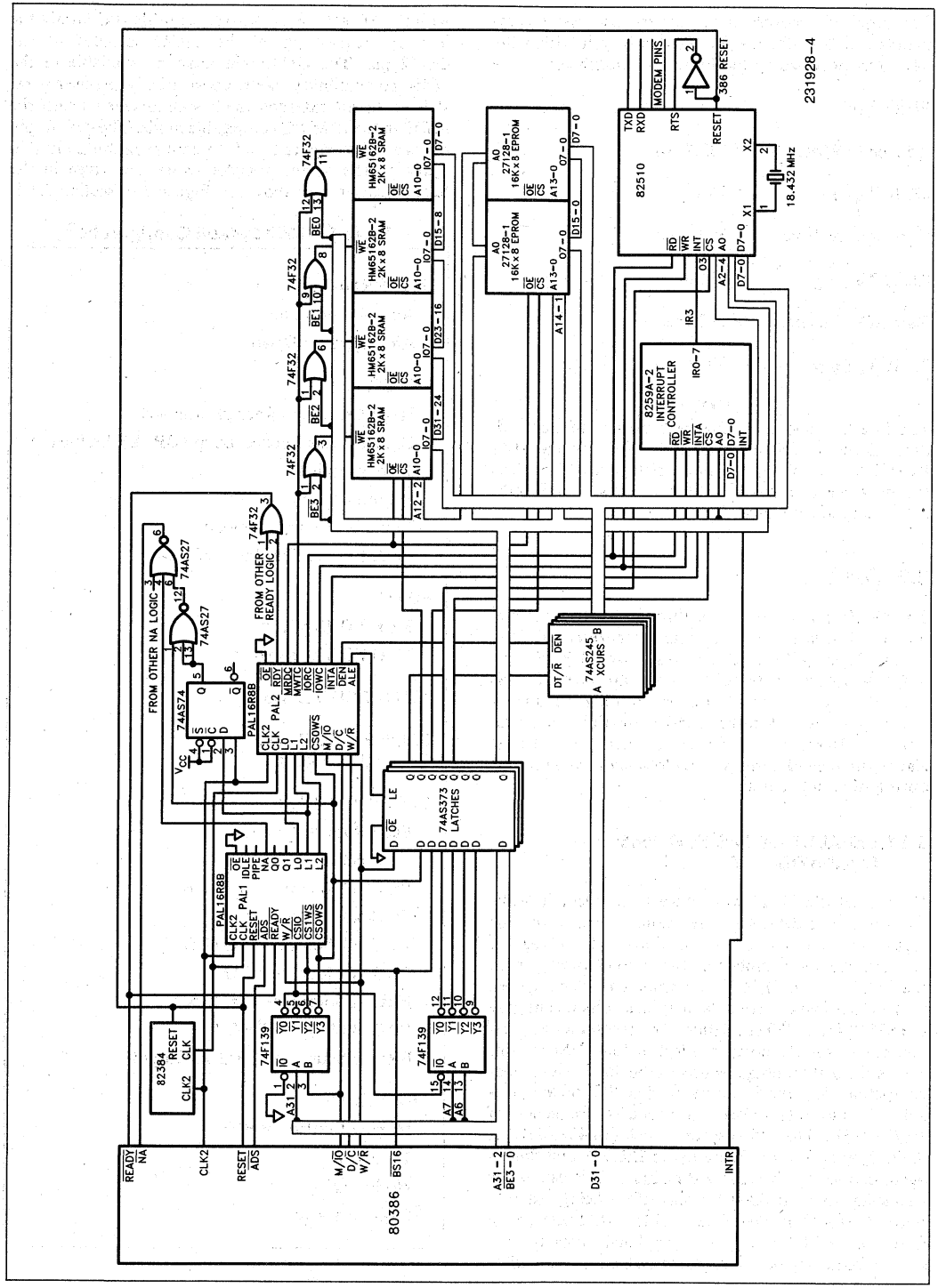


Figure 5. 80386 Interface to the 82510

The required recovery time between successive commands is 123 ns for the 82510, this is well within the 331.75 ns provided by the Basic I/O interface.

Write Cycle:

Addven to Write Low = 132.75 ns

82510 T_{AVWL} = 7 ns

Write Active Time = 300.5 ns

82510 T_{WLWH} = 231 ns

Data to Write High = 289.5 ns

82510 T_{DVWH} = 90 ns

NOTE:
The interface shown in Figure 5 uses a different address decoding scheme than that used for the IBM PC/PC AT families, for the serial ports. Therefore, the interface in Figure 5 can not be used in PC/PC AT compatible designs.

3.2 Reset

The 82510 can be reset either through hardware (Reset pin) or Software (reset command via *Internal Command Register-ICM*). Either reset would cause the 82510 to return to its default wake up mode. In this mode the register contents are reset to their default values and the device is in the 16450 compatible configuration. The Reset pulse must be held active for at least eight system clocks, the system clock should be running during reset active time.

3.2.1 DEFAULT MODES FOR 16450 COMPATIBILITY

Upon reset the 82510 will return to its Default Wake Up mode. The default register bank is bank zero. The registers in bank zero are identical to the 16450 register set, and provide complete software compatibility with the 16450* in the IBM PC environment. The registers in the other banks have default values, which configure the 82510 for 16450 emulation. The recommended system clock (for PC compatibility) is 18.432 MHz, this allows the baud rates generation to be done in a manner compatible with the PC software. The PC software calculates baud rates based on a source frequency of 1.8432 MHz. The 82510 system clock (18.432 MHz) is divided by two before being fed to BRG A and then is again divided by five (BRG B default). This causes the frequency to be divided by ten before being fed into BRG A. 18.432 divided by ten yields 1.8432 MHz, so in effect the BRG A is generating baud rates from a source frequency of 1.8432 MHz (which is compatible with the PC software). Also since in the PC family the interrupt request pin of the UART is gated by the OUT2 pin, The OUT2 pin must be available in the 16450 compatibility mode, consequently the user is restricted to an external clock source when using the 82510 in the IBM PC compatible mode. The default pin out is given in Figure 6 and the configuration is given in Table 1. The default register values are given in the 82510 register map shown in Figure 2 in section 3.1.1.

Table 1. 82510 Default Configuration

INTERRUPTS
- Auto Acknowledge
- All Interrupts Disabled

RECEIVE
- Stand Ctl. Char. Recogn. disabled
- Digital Phase Locked Loop (DPLL) disabled
- 3/16 Sampling
- Majority Vote Start bit
- Non μlan (Normal) mode
- BkD, FE, OE, PE Int. enabled

FIFO
- Rx FIFO Depth = 1
- Tx FIFO Threshold = 0

AUTO ECHO Disabled

LOOP BACK Configured for Local Loopback

CLOCK OPTIONS
- Baud Rate = 57.6K
- Rx Clock = 16 x
- Rx Clock Source = BRG B
- Tx Clock = 16 x
- Tx Clock Source = BRG B
- BRG A Mode = BRG
- BRG A Source = Sys. Clock
- BRG B Mode = BRG
- BRG B Source = BRG A Output

TRANSMIT
- Manual Control of RTS
- 1 Stop Bit
- No Parity
- 5 Bit Character

*16450 is the PC AT version of the INS 8250A.

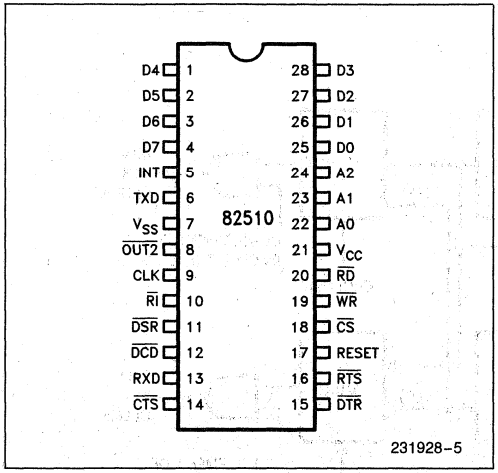


Figure 6. Default Pin Out Configuration of the 82510

3.3 System Clock Options

The term "System Clock" refers to the clock which provides timings for most of the 82510 circuitry. The 82510 has two modes of system clock usage. It can generate its system clock from its On-Chip Crystal Oscillator and an external crystal, or it can use an externally generated clock, input to the device through the CLK pin. The selection of the system clock option is done during reset. The default system clock source is an externally generated clock, which can be reconfigured by a strapping option on the $\overline{RTS}$ pin. During Reset, the $\overline{RTS}$ pin is an input; it is internally pulled high, if it is externally driven low, then the 82510 expects to use the Crystal Oscillator for system clock generation, otherwise it is set up for using an external clock source. This can be done by using an open collector inverter to $\overline{RTS}$, the input of the inverter is the Reset signal. The 82510 has a pull up resistor in the $\overline{RTS}$ circuitry so no external pull up is needed. In the crystal oscillator mode the CLK/X1 pin is automatically configured to X1, and the $\overline{OUT2}$/X2 pin is configured to X2. In the External Clock mode, the CLK/X1 is configured to CLK and the $\overline{OUT2}$/X2 is configured to $\overline{OUT2}$.

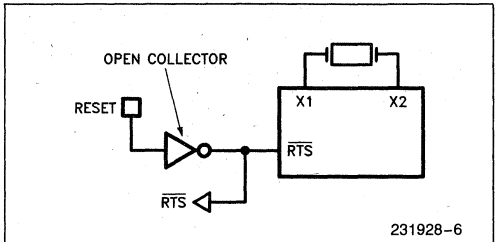


Figure 7. Crystal Oscillator Strapping Option

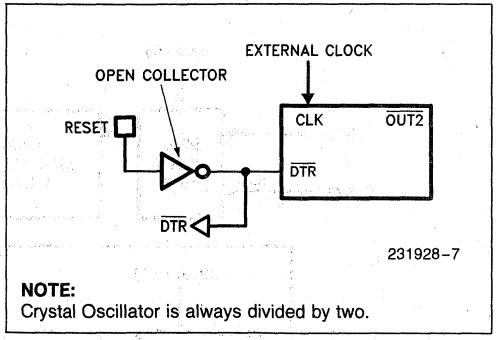


Figure 8. Disable Divide by Two

If the Crystal Oscillator is being used to supply the system clock, then the clock frequency is always divided by two before being fed into the rest of the 82510 circuitry. If, however an external clock source is being used to supply the system clock, then the user has two options:

1. Use the System Clock after **division by two,** e.g. if a 8 MHz clock is being fed into the CLK pin, then the actual frequency of the 82510 system clock will be 4 MHz (default).
2. **Disable Division by two** and use the direct undivided clock, e.g. if an 8 MHz clock is being fed into the CLK pin, then the actual frequency of the 82510 system clock is also 8 MHz.

The divide by two option is the default mode of operation in the External Clock mode of the 82510. A strapping option can be used to disable the Divide By Two operation (For Crystal Oscillator Mode Divide By Two must always be active). During Reset, the $\overline{DTR}$ pin is an input; it is internally pulled high, if it is externally driven low then the Divide By Two operation is disabled. The strapping option is identical to the one used on $\overline{RTS}$ for selection of the System Clock source.

The 82510 system clock must be chosen with care since it influences the wait state performance, Baud Rate Generation (if being used as source frequency for the BRGs), the power consumption, and the Timer counting period. The power consumption of the 82510 is dependent upon the system clock frequency. If using the system clock as a source for the Baud Rate Generator(s), then the system clock frequency must be a baud rate multiple in order to minimize frequency deviation. For standard baud rates a multiple of 1.8432 MHz can be used, in fact the 18.432 MHz maximum frequency was chosen with this particular criteria in mind.

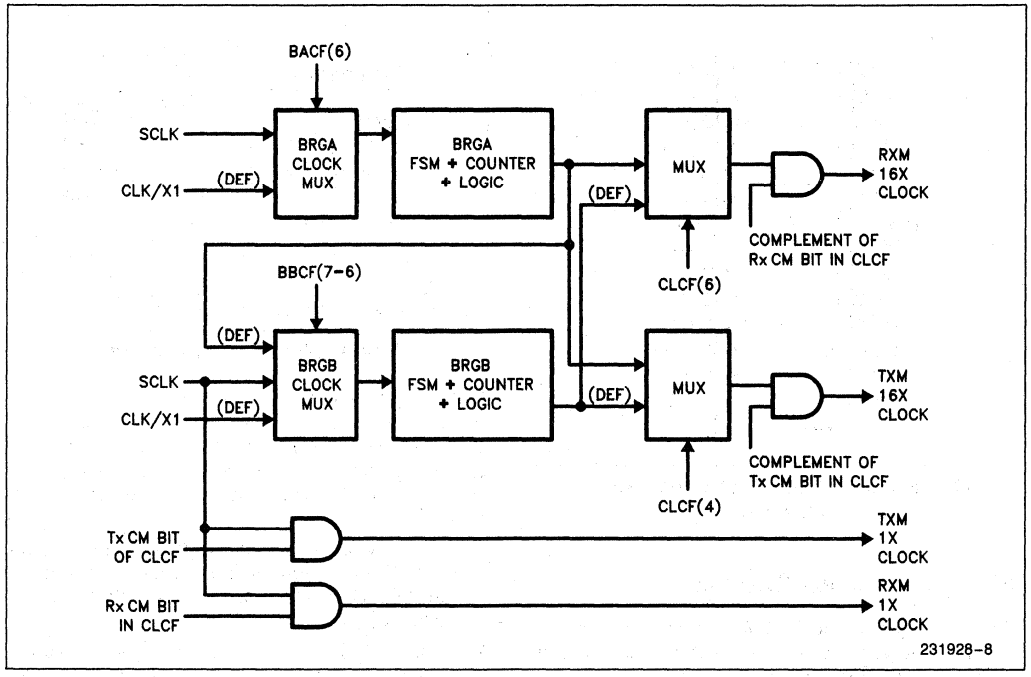


Figure 9. Timing Flow of the 82510

3.3.1 POWER DOWN MODE

The 82510 has a "power down" mode to reduce power consumption when the device is not in use. The 82510 powers down when the power down command is issued via the *Internal Command Register* (ICM). There are two modes of power down, Power Down Sleep and Power Down Idle.

3.3.1.1 Sleep Mode

This is the mode when even the system clock of the 82510 is shut down. The system clock source of the 82510 can either be the Crystal Oscillator or an external clock source. If the Crystal Oscillator is being used and the power down command is issued, then the 82510 will automatically enter the Sleep mode. If an external clock is being used, then the user must disable the external clock in addition to issuing the Power Down command, to enter the Sleep mode. The benefit of this mode is the increased savings in power consumption (typical power consumption in the Sleep mode is in the range of hundreds of microAmps. However, upon wake up, if using a crystal oscillator, the user must reprogram the device. The data is preserved if the external clock is disabled after the power down command, and enabled prior to exiting the power down mode. To exit this mode the user can either issue a Hardware reset, or read the *FIFO Level Register (FLR)* and then issue a software reset (if using a Crystal Oscillator). In either case the contents of the 82510 registers are not preserved and the device must be reprogrammed prior to operation.

NOTE:
If the Crystal Oscillator is being used then the user must allow about 1 ms for the oscillator to wake up before issuing the software reset.

3.3.1.2 Idle Mode

The 82510 is said to be in the Idle mode when the Power Down command is issued and the system clock is still running (i.e. the system clock is generated externally and not disabled by the user). In this mode the contents of all registers and memory cells are preserved, however, the power consumption in this mode is greater than in the Sleep mode. Reading FLR will take the 82510 out of this mode.

NOTE:
The data read from FLR when exiting Power Down is incorrect and must be ignored.

4.0 INTERRUPT BEHAVIOR

4.1 FIFO Usage

The 82510 has two independent four bytes transmit and receive FIFOs. Each FIFO can generate an interrupt request, when the FIFO level meets the Threshold requirements. The FIFOs can have a considerable impact on the performance of an asynchronous communications system. For systems using high baud rates they can provide increased interrupt-to-service latency reducing the chances of an overrun occurring. In systems constrained for CPU time, the FIFOs can increase the CPU Bandwidth by reducing the number of interrupt requests generated during asynchronous communications. It can reduce the interrupt load on the CPU by up to 75%. By choosing the FIFO thresholds which reflect the system bandwidth or service latency requirements, the user can achieve data rates and system throughput, unattainable with traditional UARTs.

Table 2. The Power Down Modes

Mode	Clock Source	Exit Procedure	Power Consumption	Data Preservation
Sleep	Crystal Oscill. Automatically Disabled	H/W Reset or Read FLR and Issue S/W Reset	100–900 μA	Not Preserved Must be Reprogrammed
	External Clock Must be Disabled by User	Enable External Clock, Read FLR and Issue S/W Reset H/W Reset	100–900 μA	Not Preserved Must be Reprogrammed
Idle	External Clock Running	H/W Reset Read FLR	1–3 mA	All Data Preserved Does Not Need to be Reprogrammed

4.1.1 INTERRUPT-TO-SERVICE LATENCY

The interrupt-to-service latency is the time delay from the generation of an interrupt request, to when the interrupt source in the 82510 is actually serviced. Its primary application is in the reception of data. In traditional UARTs the CPU must read the current character in the Receive Buffer before it is overrun by the next incoming character. The Rx FIFO in the 82510 can buffer up to four characters, allowing an interrupt-to-service latency of up to four character transmission times. The character transmission time is the time period required to transmit one full character at the given Baud Rate. It is dependent upon the baud rate and is given by equation (1):

(1) Character Transmission Time =

$$\frac{\text{Num. of Bits per Character Frame}}{\text{Baud Rate}}$$

The Transmit and Receive FIFO thresholds should be selected with consideration to two factors the Baud rate, and the (CPU Bandwidth allocated for Asynchronous Channels is dependent upon the number of channels supported since it does not include the overhead of supporting other peripherals) number of Asynchronous Serial ports being supported by the CPU. In order to avoid overrun, the interrupt-to-service delay must be less than the time it takes to fill the 82510 Rx FIFO. The relationship is given by equation (2):

(2) Int_to_service-latency < FIFO Size × Character Transmission Time

Example

Calculate the maximum baud rate that can be supported by a 6 MHz PC AT to support four Full Duplex Asynchronous channels using

a) The 82510 with four byte FIFO.

b) The 82510 with one byte FIFO.

Assumptions:

- CPU dedicated to Asynchronous communications.
- UART Interrupts limited to Transmission and Reception only.
- Interrupt Routines are optimized for fast throughput.
- 10 bits per character frame.

Going back to equation (2):

Int._to_service latency < *Buffer size x 10/baud rate*

Int_to_service latency = # of Channels × (# of int. sources per channel) × Time required to service interrupt

Int_to_service latency = 4 × 2 × Time required to service interrupt

The Time required to service interrupt has been calculated to be 100 μs for a slightly optimized service routine. RMX86 interrupt service time is given as 250 μs and for other operating systems it should be slightly higher.

Int_to_service latency = 4x2x100 s
= 800 μs

82510 max Baud Rate = 4 × 10/800 μs
(four byte FIFO) = 50K bits/sec

82510 max Baud Rate = 1 × 10/800 μs
(one byte FIFO) = 12.5K bits/sec

4.2 Interrupt Handling

The 82510 has 16 different sources of interrupt, each of these sources, when set and enabled, will cause their respective block interrupt requests to go active. The block interrupt request, if enabled, will set the 82510's INT pin high, and will be reflected as a pending interrupt in the *General Interrupt Register (GIR)* if no other higher priority block is requesting service. If a higher priority block interrupt is also active at the same time, then the *General Interrupt Register* will reflect the higher priority request as the source of the 82510 interrupt. The lower priority interrupt will issue a new edge on the interrupt pin only after the higher priority interrupt is acknowledged and if no other priority block requests are present. Both the block interrupts and the individual sources within the blocks are maskable. The block interrupts are enabled through the *General Enable Register (GER)* which prevents masked bits in the *General Status Register (GSR)* from being decoded into the *General Interrupt Register.* This does not prevent the block request from being set in the *General Status Register*, it only prevents the masked GSR bits from being decoded into the *General Interrupt Register*, and thus generating any interrupts. The individual sources within the block are masked out via the corresponding interrupt enable register associated with the specific block (Rx Machine; Timing Unit and the Modem I/O module each have an Interrupt Enable register).

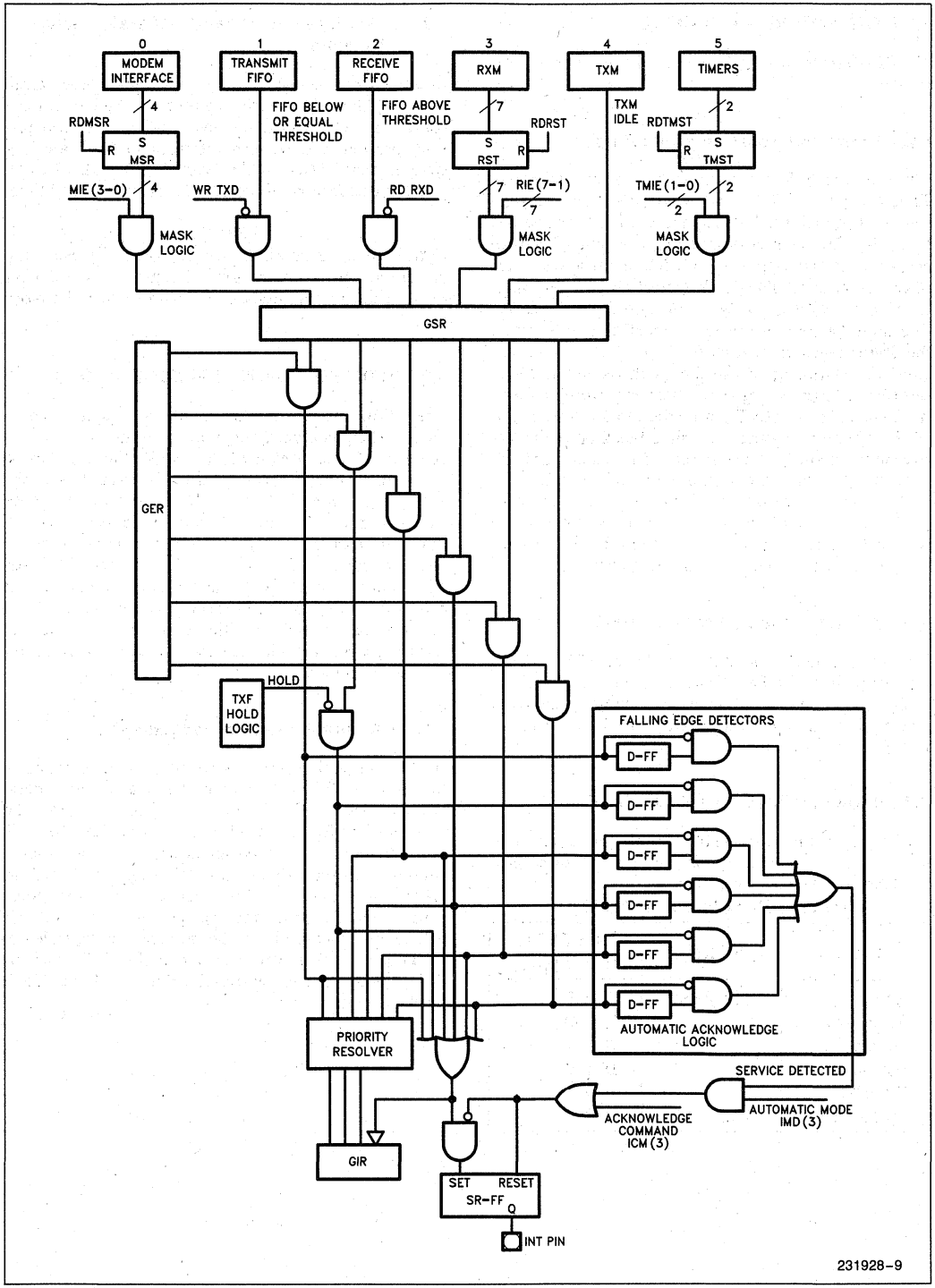


Figure 9. 82510's Interrupt Scheme

4.2.1 THE INTERRUPT SCHEME

The 82510 interrupt logic consists of the following elements:

4.2.1.1 Interrupt Sources Within Blocks

Three of the 82510 functional blocks (Rx Machine, Timer, Modem I/O) have more than one possible source of interrupts, for instance the Rx Machine has seven different sources of interrupts—standard control character recognition (Std. CCR), control character Match (special CCR), Break Detect, Break Terminated, Overrun Error, Parity Error, and Framing Error. The multiple sources are represented as Status bits in the Status registers of each of these blocks. When enabled the Status bits cause the block request to set in the *General Status Register.* There is no difference in the behavior of the INT pin or the block status bits in GSR, for multiple sources within a block becoming active simultaneously. The corresponding block status bit in GSR is set when one or more interrupt sources within the block become active. When the status register for the block is read all the active interrupt sources within the block are reset. Each source within the three blocks can be masked through its respective enable register.

4.2.1.2 General Status Register (GSR)

This register holds the status of the six 82510 blocks (all except Bus Interface Unit). Each bit when set indicates that the particular block is requesting interrupt service, and if enabled via the *General Enable Register,* will cause an interrupt.

4.2.1.3 General Enable Register (GER)

This register is used to enable/disable the corresponding bits in the *General Status Register.* It can be programmed by the CPU at any time.

Table 3. Block Interrupt Priority

Block	Priority	GIR CODE 3 2 1 (Bits)
Timers	5 (highest)	1 0 1
Tx Machine	4	1 0 0
Rx Machine	3	0 1 1
Rx FIFO	2	0 1 0
Tx FIFO	1	0 0 1
Modem I/O	0 (lowest)	0 0 0

4.2.1.4 Priority Resolver and General Interrupt Register

If more than one enabled Interrupt request from GSR is active, then the priority resolver is used to resolve contention. The priority resolver finds the highest priority pending and enabled interrupt in GSR and decodes it into the *General Interrupt Register* (bits 3 to 1). The *General Interrupt Register* can be read at any time.

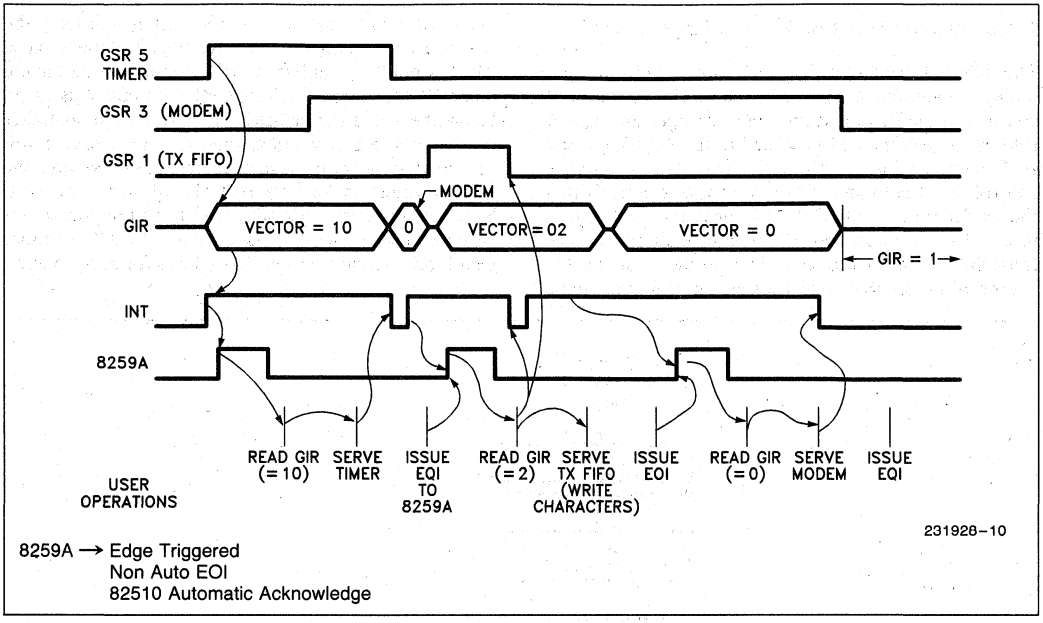
NOTE:
GIR is updated continuously, so while the user may be serving one interrupt source, a new interrupt with higher priority may update GIR and replace the older one.

4.2.2 INTERRUPT ACKNOWLEDGE MODES

The 82510 has two modes of interrupt acknowledgement—Manual acknowledge and Automatic acknowledge. In Manual Acknowledge mode, the user has to issue an explicit Acknowledge Command via the *Internal Command Register (ICM)* in order to cause the INT pin to go low. In Automatic Acknowledge mode the INT pin will go low as soon as an active or pending interrupt request is serviced by the CPU. An operation is considered to be a service operation if it causes the source of the interrupt (within the 82510) to become inactive (the specific status bit is reset). The service procedures for each source vary, see section 4.2.3.2 for details.

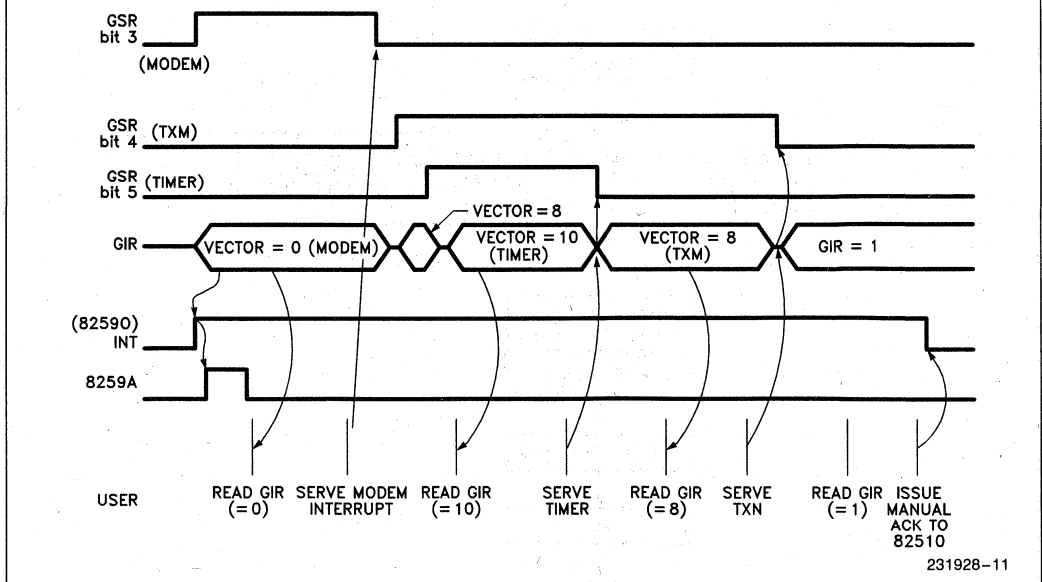
4.2.2.1 Automatic Acknowledgement

In the automatic acknowledge mode, a service operation by the CPU will be considered as an automatic acknowledgement of the interrupt. This will force the INT pin low for two clock cycles, after that the INT pin is updated i.e. if there is an active enabled source pending then the INT pin is set high again (reflected in GIR). This mode is useful in an edge triggered Interrupt system. Servicing any enabled and active GSR bit will cause Auto Acknowledge to occur (independently of the source currently decoded in the *GIR register).* This can be used to rearrange priorities of the 82510 block requests.



8259A → Edge Triggered
Non Auto EOI
82510 Automatic Acknowledge

Figure 10. Automatic Acknowledge Mode Operation



NOTE:
Vector refers to GIR bit (3-0)
82510: Manual Ack. Mode
8259A: Edge Triggered Non AEOI

Figure 11. Manual Acknowledge Mode Operation

4.2.2.2 Manual Mode of Acknowledgement

The Manual Acknowledgement Mode requires that, unlike the automatic mode where a service operation is considered as an automatic acknowledge, an explicit acknowledge command be issued to the 82510 to cause INT to go inactive. In this mode the CPU has complete control over the timing of the Interrupts. Before exiting the service routine, the CPU can check the *GIR register* to see if other interrupts are pending and can service those interrupts in the same invocation, avoiding the overhead of another interrupt as in the Automatic mode. Of course the user has the option of issuing the acknowledge command immediately after the service, which would be similiar in behavior to the automatic mode. If the manual acknowledge command is given before the active source has been serviced and no higher priority request is pending, then the same source will immediately generate a new interrupt. Therefore, the software must make sure that the Manual Acknowledge command is issued after the interrupt source has been serviced by the CPU (see section 4.2.3.2. for more details on interrupt service procedures for each source).

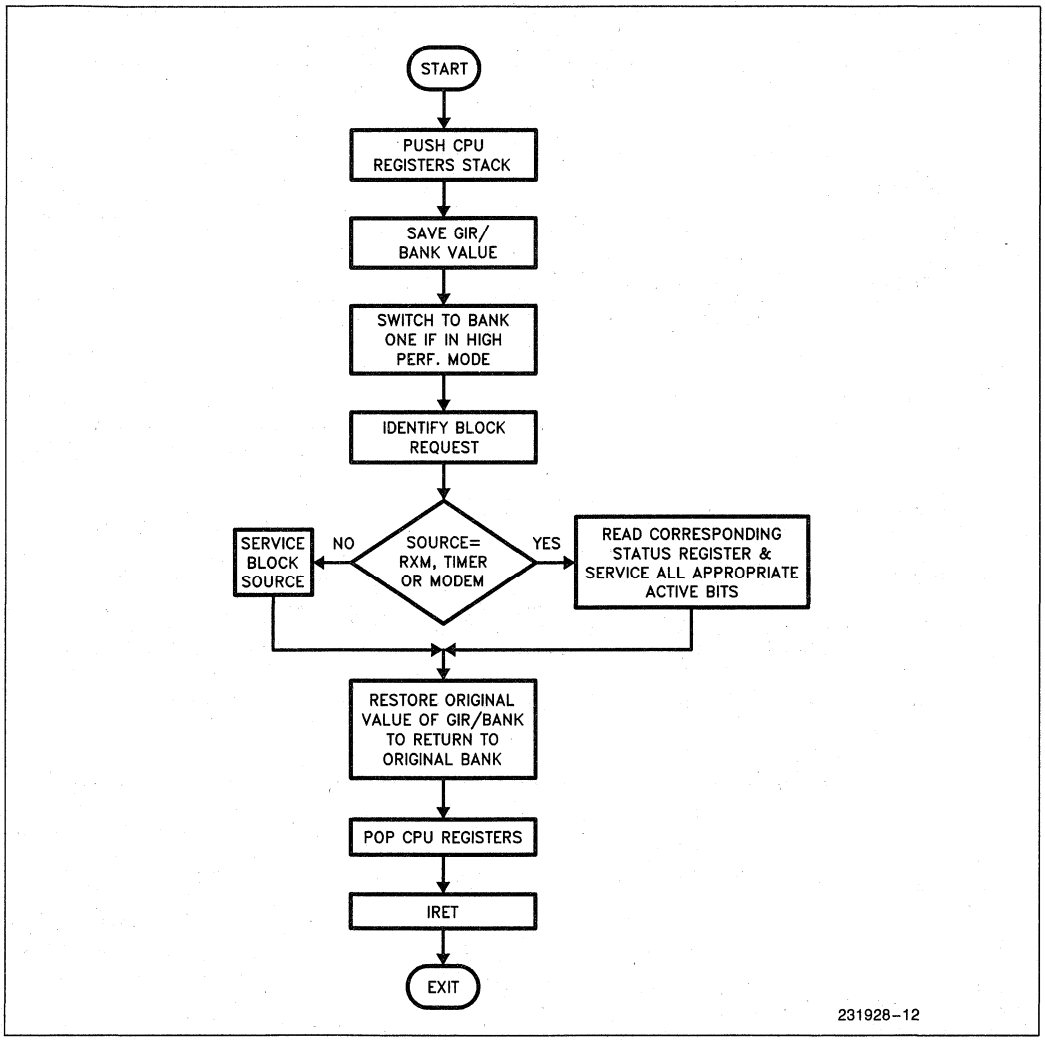


Figure 12. Typical Interrupt Handler

4.2.3 GENERAL INTERRUPT HANDLER

In general an interrupt handler for the 82510 must first identify the interrupt source within the 82510, transfer control to the appropriate service routine and then service the active source. The active source can be identified from two registers—*General Interrupt Register,* or *General Status Register.* The *GIR register* identifies the highest priority active block interrupt request. The *GSR register* identifies all active (pending or in service) Block Interrupt Requests. The typical operation of the 82510 interrupt handler is given in Figure 12. The two major issues of concern are the source identification and Control Transfer to the appropriate service routine.

Since the 82510 registers are divided into banks, and the interrupt handler may change register banks during service, it is best to save the bank being used by the main program and then do the interrupt processing. Upon completion of service, the original bank value is restored to the *GIR/Bank register.*

4.2.3.1 Source Identification

The 82510 has 16 interrupt sources, and the CPU must identify the source before performing any service. Although the procedure varies, the typical method would be to identify the block requesting service by reading

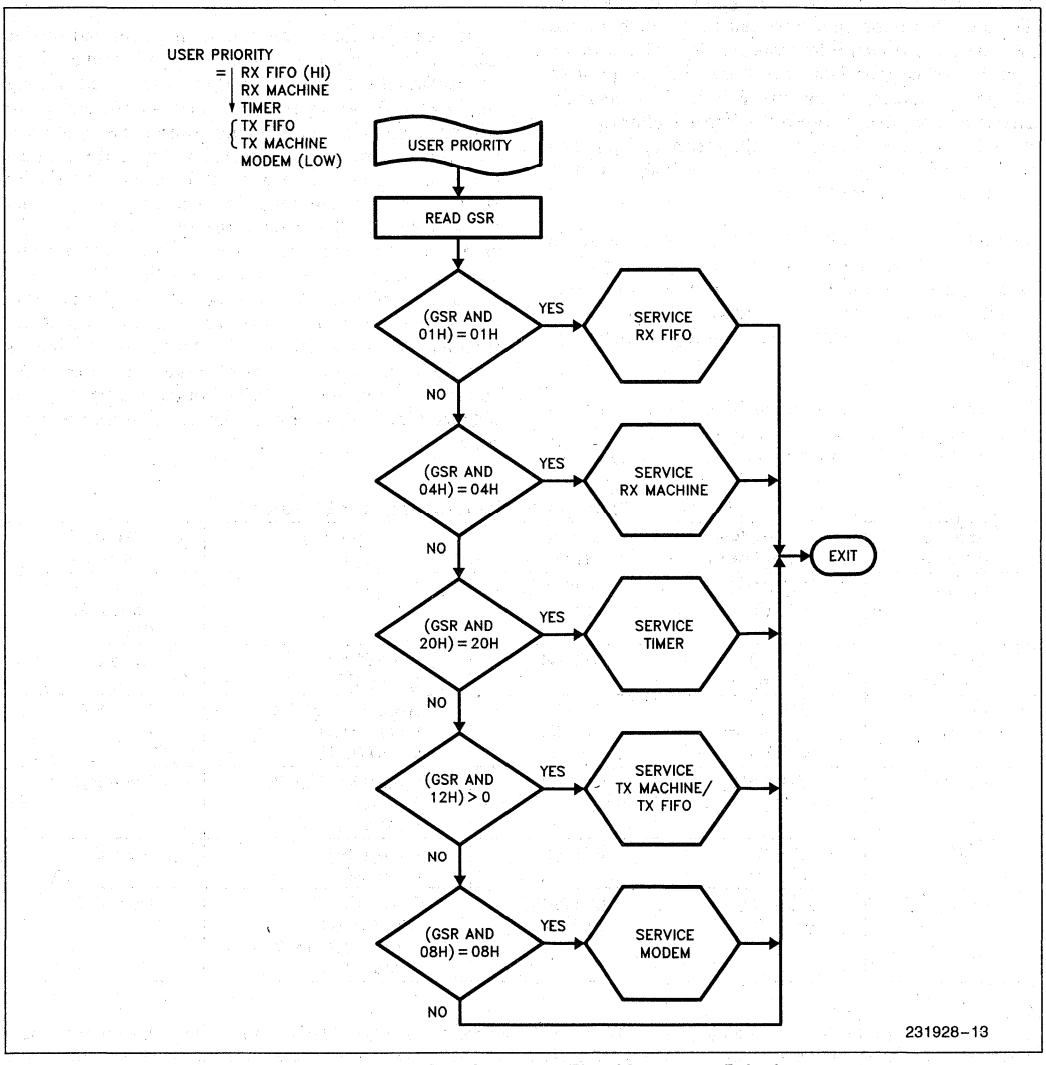


Figure 13. Bypassing the 82510 Fixed Interrupt Priority

GIR bits 3-1. If the source is either Tx Machine, Tx FIFO, or Rx FIFO, no further indentification is needed, the user can transfer control to the service routine (in most cases, only one Timer will be used, therefore the Timer Routine can also be directly invoked). All modem I/O interrupts can be handled via one routine as all the modem interrupt sources are supplementary to the modem handshaking function. The Rx Machine, however, has two different types of interrupt sources, event indications (CCR/Address recognition CCR/Address Match, Break Detect, Break Terminate, and Overrun Error), and error indications (Parity Error, Framing Error, these error indications do not refer to any particular character, they just indicate that the specific error was detected during reception). For most applications, the error indicators can be masked off, and only the event driven interrupts enabled. The error indicators can be read from the Receive Flags prior to reading a character from the FIFO. This interrupt scheme can be used, because the Receive character error indicators are available in the Receive Flags, and can be checked by the Receive routine before reading the character from the Rx FIFO.

Since all active status bits (except Rx FIFO interrupt in LSR and RST) are reset when the corresponding block status register is read, the interrupt routine must check for all possible active sources within the block, and service each active source before exiting the interrupt handler.

The 82510 interrupt contention is resolved on a fixed priority basis. In some applications the fixed priority may not be suitable for the user. For these cases the user can bypass the 82510's priority resolution by using the *General Status Register* (rather than GIR) to determine the block interrupt sources requesting service. Each source is checked in order of user priority and serviced when identified (There will be no problem with using this algorithm in auto acknowledge mode because the INT pin will go low as soon as a pending and enabled interrupt request goes low). The user will be trading some service latency time for additional source identification time, this algorithm's efficiency will improve as the number of block sources to verify is reduced. See Figure 13 for the algorithm.

4.2.3.2 Interrupt Service

A service operation is an operation performed by the CPU, which causes the source of the 82510 interrupt to go inactive (it will reset the particular status bit causing the interrupt). An interrupt request within the 82510 will not reset until the interrupt source has been serviced. Each source can be serviced in two or three different ways; one general way is to disable the particular status bit causing the interrupt, via the corresponding block enable register. Setting the appropriate bit of the enable register to zero will mask off the corresponding bit in the status register, thus causing the INT pin to go inactive. The same effect can be achieved by masking off the particular block interrupt request in GSR via the *General Enable Register.* Another method, which is applicable to all sources, is to issue the Status Clear command from the *Internal Command Register.* The detailed service requirements for each source are given below:

Table 4. Service Procedures For Each Interrupt Source

Interrupt Source	Status Bits & Registers	Interrupt Masking	Specific Service	General Service
Timers	TMST (1-0) GSR (5)	TMIE (1-0) GER (5)	Read TMST	Issue Status Clear (StC)
Tx Machine	GSR (4) LSR (6)	GER (4)	Write Character to Tx FIFO	Issue StC
Rx Machine	LSR (4-1) RST (7-1) GSR (2)	RIE (7-1) GER (2)	Read RST or LSR Write 0 to bit in RST/LSR	Issue StC
Rx FIFO	RST/LSR (0) GSR (0)	GER (0)	Write 0 to LSR/RST Bit zero. Read Character(s)	Issue StC
Tx FIFO	LSR (5) GSR (1)	GER (1)	Write to FIFO Read GIR	Issue StC
Modem	MSR (3-0) GSR (3)	MIE (3-0) GER (3)	Read MSR write 0 into the appropriate bits of of MSR (3-0)	Issue StC

NOTE:
The procedures listed in Table 4 will cause the INT pin to go low only if the 82510 is in the automatic acknowledge mode. Otherwise, only the internal source(s) are decoded, the INT pin will go low only when the Manual Acknowlege command is issued.

4.3 Polling

The 82510 can be used in a polling mode by using the *General Status Register* to determine the status of the various 82510 blocks, this is useful when the software must manage all the blocks at once. If the software is dedicated to performing one function at a time, then the specific status registers for the block can be used, e.g. if the software is only going to be Transmitting, it can monitor the Tx FIFO level by polling the *FIFO Level Register,* and write data whenever the Tx FIFO level decreases. Reception of data can be done in the same manner.

5.0 SOFTWARE CONSIDERATIONS

5.1 Configuration

The 82510 must be configured for the appropriate modes before it can be used to transmit or receive data. Configuration is done via read and write registers, each functional block (except for BIU) has a configuration register. Typically the configuration is done once after start up, however, the FIFO thresholds and the interrupt masks can be reconfigured dynamically. If the 82510 configuration is not known at start up it is best to bring the device to a known state by issuing a software reset command (ICM register, bank one). At this point all block interrupts are masked out in GER and all configuration and status registers have default values. The bank register is pointing to bank zero. The 82510 can now be configured as follows:

1. **If BRG A is being issued as a baud rate generator then load the baud rate count into BAL and BAH registers.**
2. **Configure the character attributes in LCR register (Parity, Stop Bit Length, and Character Length).**

(Note if interrupts are being used, steps 1 and 2 can also be done at the end, since the user will have to return to bank zero to set the interrupt masks in GER)

3. **Load ACR0 register with the appropriate Control or Address character (if using the Control Character Match or Address Match capability of the 82510).**
4. **Switch to Bank two.**

   (In this Bank the configuration can be done in any order)
5. **Configure the Receive and Transmit FIFO thresholds if using different thresholds than the default).**
6. **Configure the Transmit Mode Register for the Stop Bit length, modem control, and if using echo or 9 bit length or software parity, configure the appropriate bits of the register. The default mode of the modem control is Manual, if using the FIFO then the automatic mode would be most useful).**
7. **Configure the Rx FIFO depth, interrupt acknowledge mode, μlan or normal mode and echo modes in IMD register.**
8. **Load ACR1 if necessary**
9. **Enable Rx Machine Interrupts as necessary via RIE.**
10. **Configure RMD for CCR, DPLL operation, Sampling Window, and start bit.**
11. **Switch to Bank 3.**
12. **Configure CLCF register for Tx and Rx clocks and or Sources**
13. **Configure BACF register for BRG A mode and source.**
14. **Load BBL and BBH if BRGB is being used (as either a BRG or a Timer).**
15. **Configure BBCF register if necessary.**
16. **If reconfiguration of the modem pin is necessary then program the PMD register.**
17. **Enable any modem interrupt sources, if required, via MIE register.**
18. **Enable Timer interrupts, if necessary, via TMIE.**
19. **If using interrupts**

    then

    i) **Switch to Bank zero.**

    Disable Interrupts at CPU (either by masking the request at the interrupt controller or executing the CLI instruction).

    ii) **Enable the appropriate 82510 Block interrupts by setting bits in the GER register. (CPU interrupts can now be reenabled, but it is recommended to switch banks before enabling the CPU interrupts).**

NOTE:
At this stage it is best to leave the TxM and Tx FIFO interrupt disabled. See section 6.3 Transmit Operation for details)

20. **Switch to Bank One. Load Transmit Flags if using 9-bit characters, or 8051 9-bit mode or software parity. If using interrupts CPU interrupts can now be enabled.**

Bank One is used for general operation, the 82510 can now be used to transmit or receive characters.

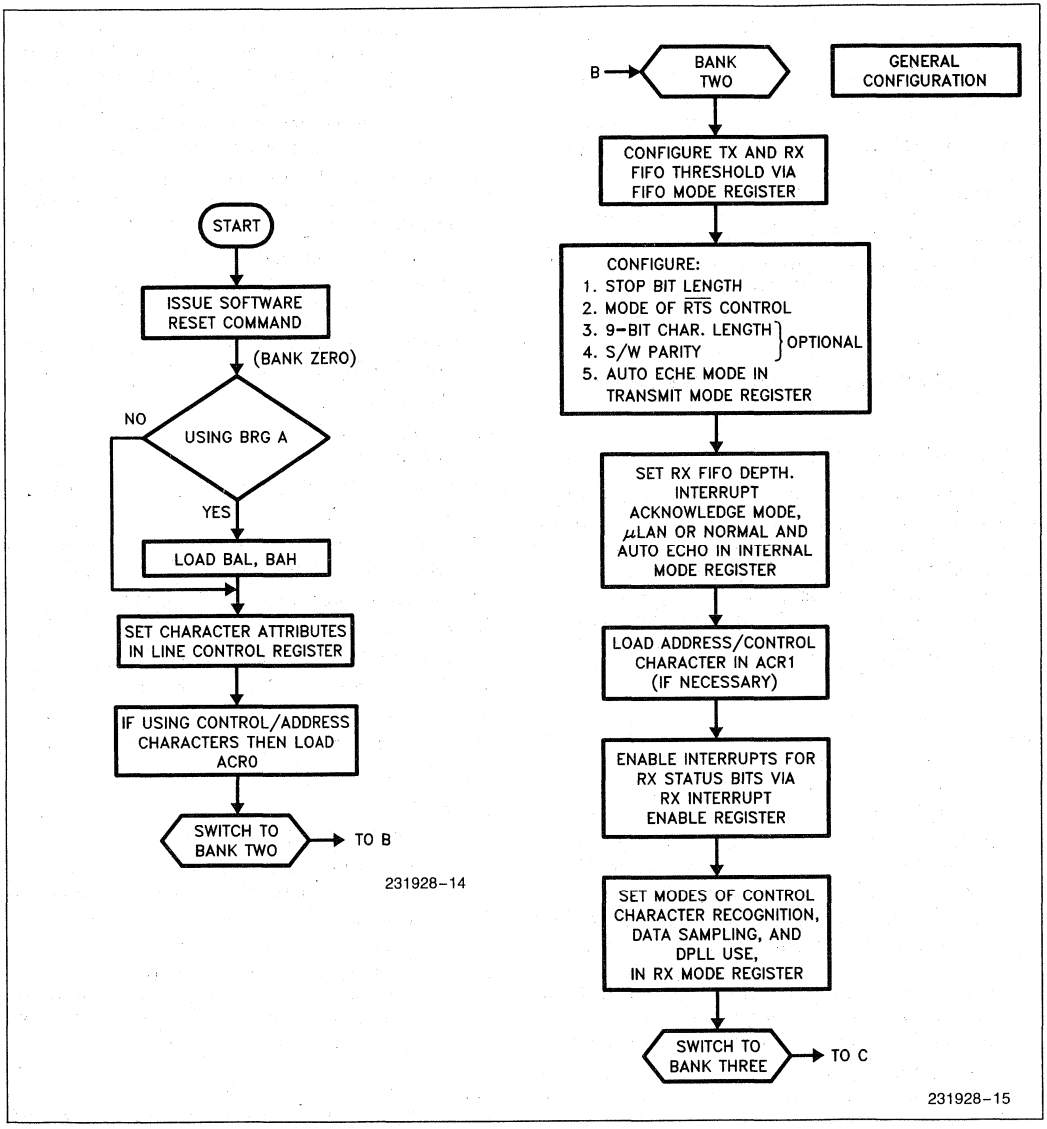


Figure 14. Configuration Flow Chart

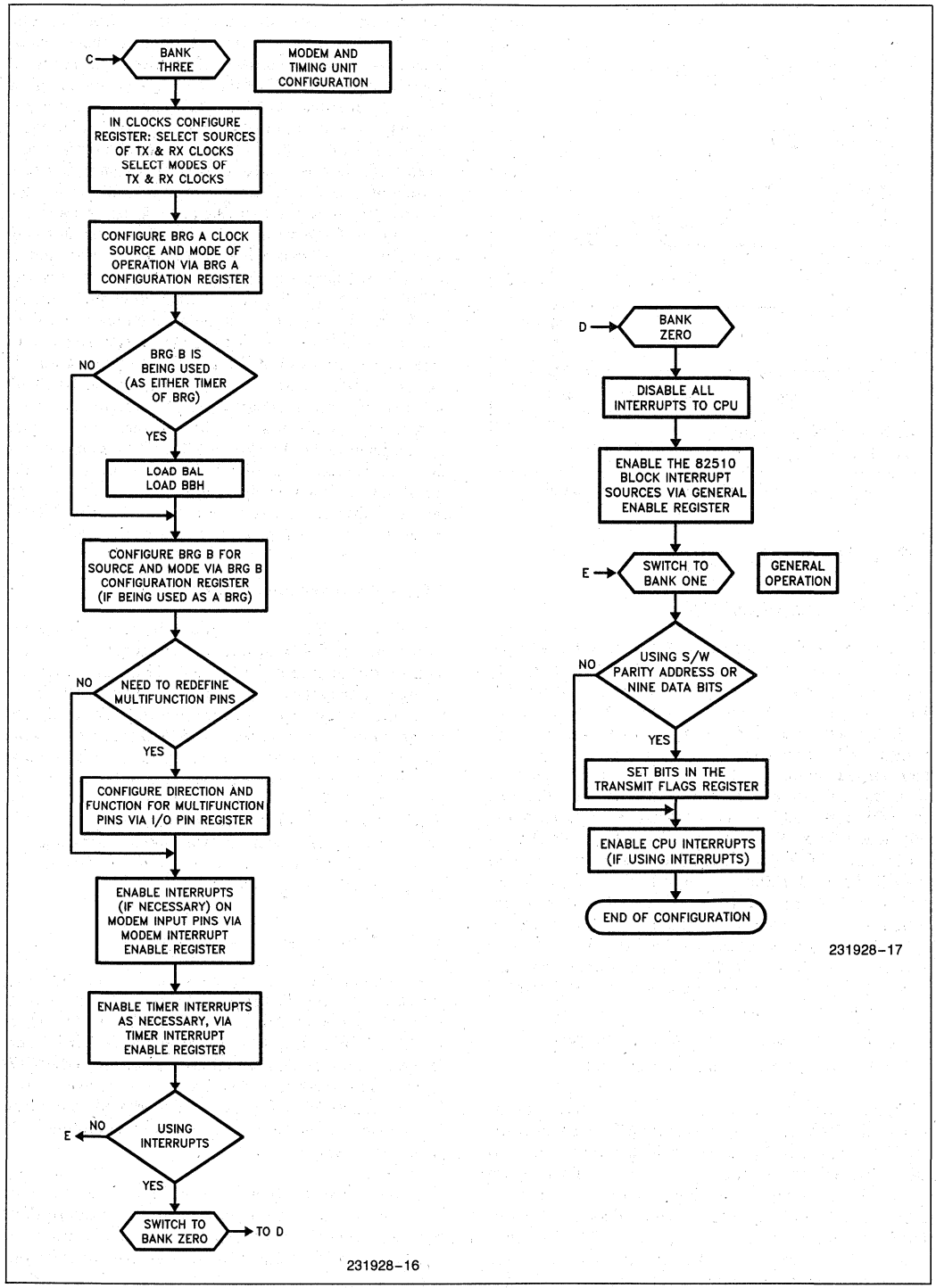


Figure 14. Configuration Flow Chart (Continued)

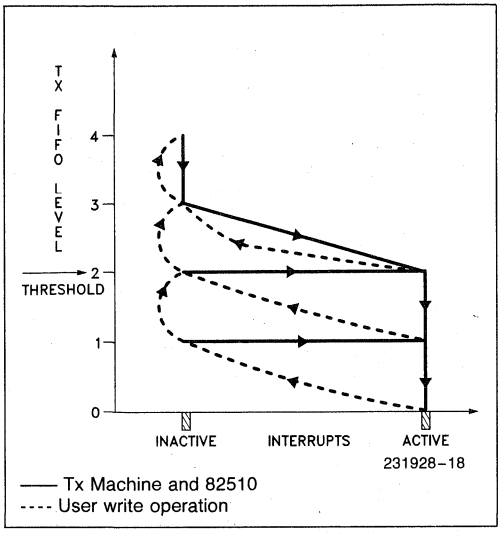


Figure 15. Tx FIFO Interrupt Hysteresis

5.2 Transmit Operation

5.2.1 GENERAL OPERATION

To transmit a character the CPU must write it to the *TXD register*, this character along with the flags from the *Tx Flags register* is loaded to the top of the TX FIFO. If the Tx Machine is empty, then the character is loaded into the shift register, where it is serially transmitted out via the TXD pin (the flags are not transmitted unless the 82510's configuration requires their transmission e.g. if software parity is selected then the S/W parity bit is transmitted as the parity bit of the character). The CPU may write more than one character into the FIFO, it can write four characters in a burst (five if the Tx Machine is empty) or it can check the FIFO level before each write, to avoid an overrun condition to the transmitter. In the case of the latter, the software overhead of checking the FIFO level must be less than the time required to transmit a character, otherwise the transmit routine may not exit until another exit condition has been met.

e.g. at 288,000 bps for an 8-bit char no parity

It takes 34.7 μs to transmit one character.

If the time, from the write to TXD to the reading of the Transmit FIFO level, is greater than 34.7 μs then the Tx FIFO level will never reach higher than zero, and the FIFO will always appear to be empty. Therefore, if the transmit routine is checking for a higher level in the FIFO it may not be able to return until some other exit condition—such as no more data available—is met. This can be a problem in the interrupt handler, where the service routine is required to be efficient and fast.

The transmitter has two status flags. Tx Machine Idle and Tx FIFO interrupt request, each of these conditions may cause an interrupt, if enabled. The Transmit Idle condition indicates that the Tx Machine is either empty or disabled. The Tx FIFO interrupt bit is set only when the level of the Tx FIFO is less than or equal to the threshold. These interrupts should remain disabled until data is available for transmission. Because outside of disabling the corresponding GSR status bits, the only way to service Tx Idle is by writing data to the Transmitter. Otherwise, the Tx Machine interrupt may occur when no data is available for transmission, and as a result will keep the INT pin active, preventing the 82510 from generating any further interrupts (unless the Transmit Interrupt routine automatically disables the Tx Machine Idle and Tx FIFO interrupt requests in GSR). The threshold of the Tx FIFO is programmable from three to zero, at a threshold of three the Tx FIFO will generate an interrupt after a character has been transmitted. While at a threshold of zero the interrupt will be generated only when the Tx FIFO is empty. For most applications a threshold of zero can be used. If the threshold is dynamically configured, i.e. it is being modified during operation, then the Tx FIFO level must be checked before writing data to the transmitter.

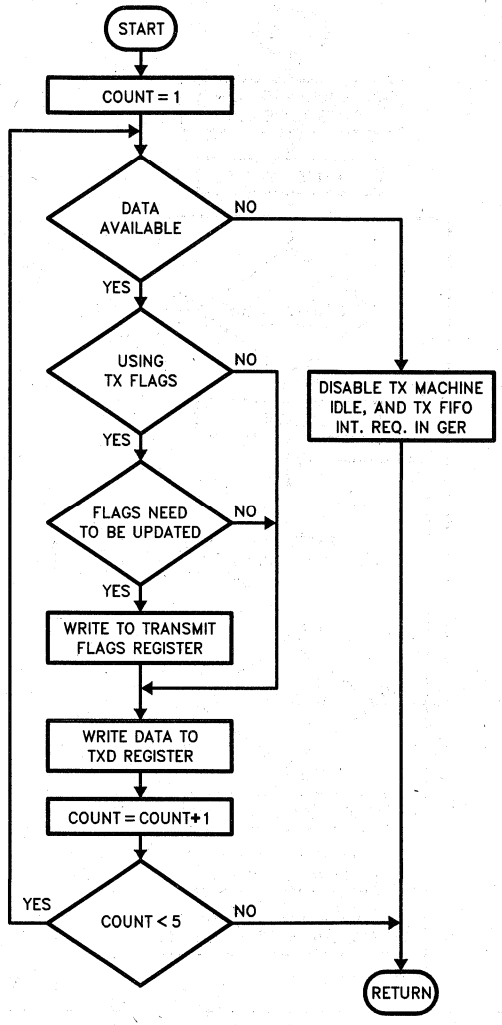
5.2.1.1 Transmit Interrupt Handler

The Transmit Interrupt Handler will be invoked when either the Tx FIFO threshold has been met or if the Transmitter is empty. Since the Tx Machine interrupt is high priority (second highest priority, with Timer being the highest), the interrupt line will not be released to other lower priority, pending 82510 sources until the Tx Machine interrupt has been serviced. If no data is available for transmission, then the only way to acknowledge the interrupt is by disabling it in the *General Enable Register.* Thus the Tx Machine interrupt should not be enabled until there is data available for transmission. The Tx Machine interrupt should be disabled after transmission is completed.

5.2.1.2 Transmission By Polling

Transmission on a polling basis can be done by using the *General Status Register* and/or the *FIFO Level Register.* The software can wait until the Tx FIFO and/or the Tx Machine Idle bits are set in the *General Status Register,* and then do a set number of writes to the *TXD register.* This method is useful when the software is trying to manage other functions such as modem control, timer management and data reception, simultaneously with transmission.

If management of other functions is not needed while transmitting, then continuous transmission can be done by monitoring the Tx FIFO level. A new character is written to TXD as soon as the FIFO level drops by one level.



Tx FIFO Threshold = 0

231928-19

NOTE:
TxM Idle and Tx FIFO Empty interrupts are enabled by the Main Program, when data transmission is required.

Figure 16. 16 Tx Interrupt Handler Flow Chart

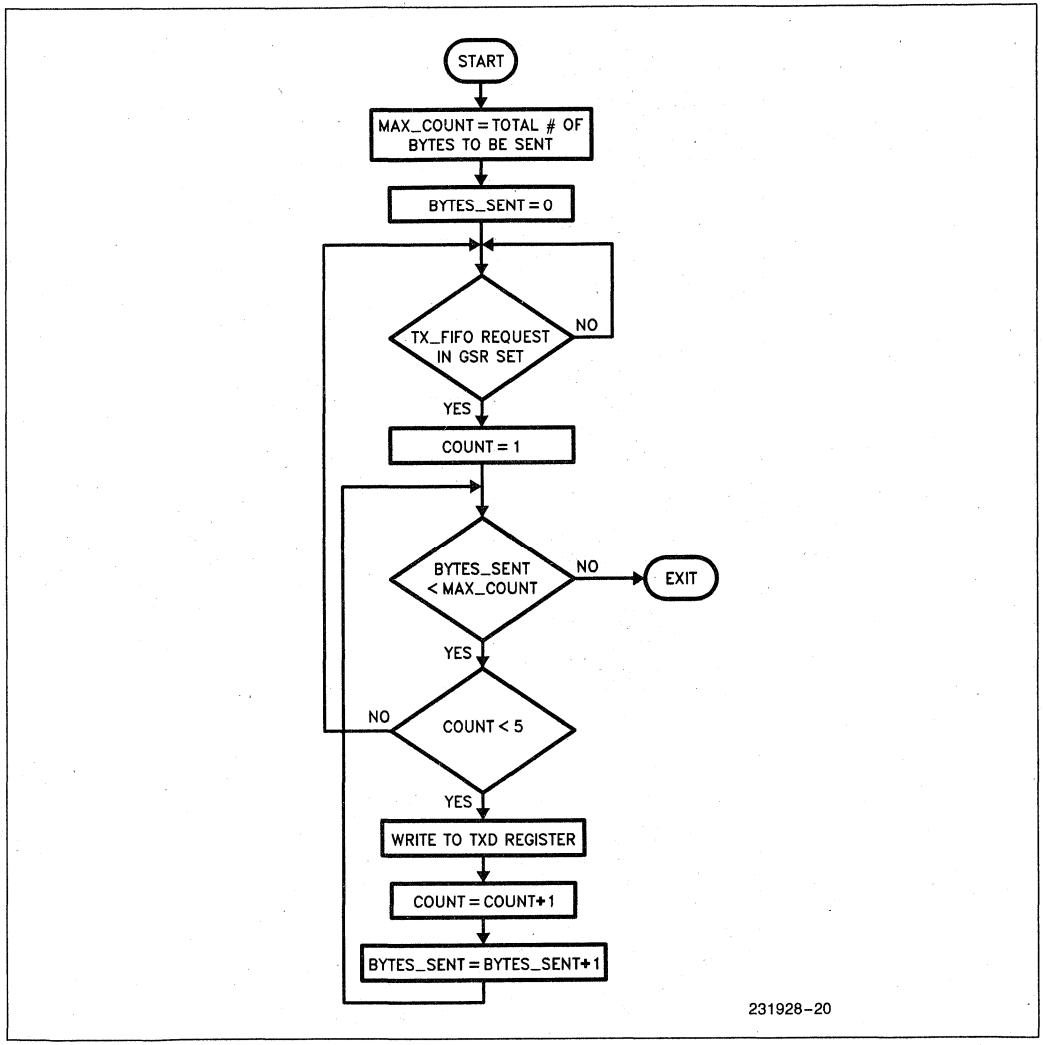


Figure 17. Using GSR for Polling

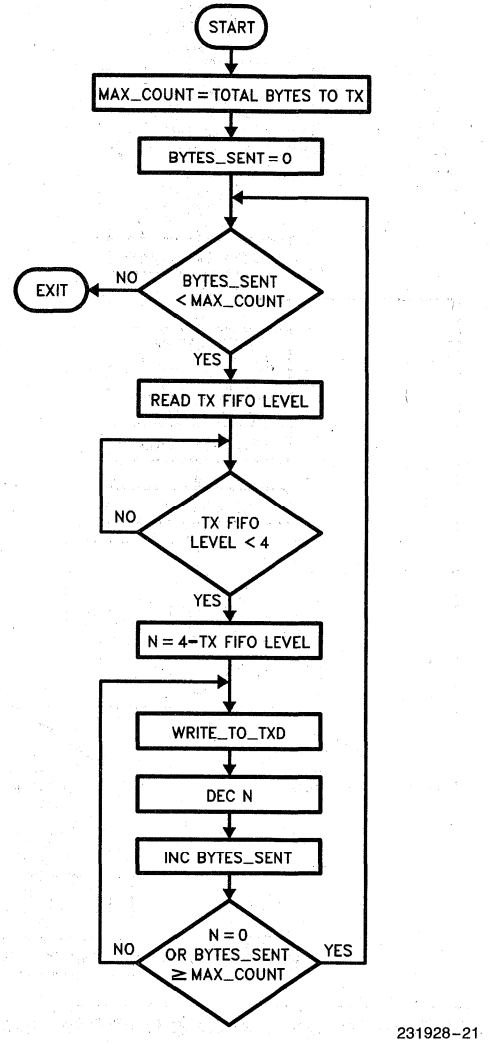


Figure 18. Data Transmission by Monitoring FIFO Level

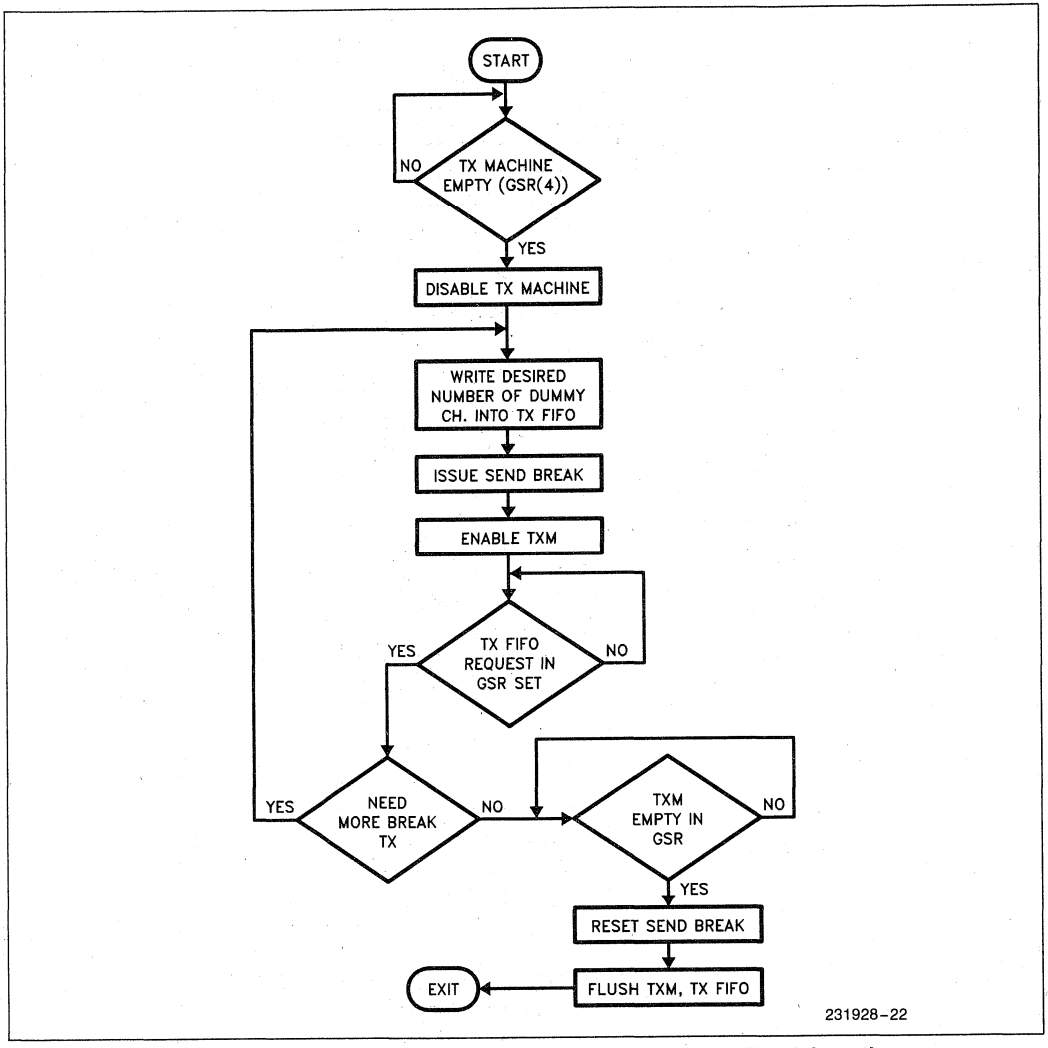


Figure 19. Break Transmission Using Tx FIFO to Measure Break Length

5.2.1.3 Break Transmission

The 82510 will transmit a break when bit six of the *Line Control Register* is set high. This will cause the TXD pin to be held at Mark for one or more character time. The Tx FIFO can be used to program a variable length break, see Figure 19 for details. If the break command is issued in the midst of character transmission the TXD pin will go low, but the transmitter will not be disabled. The characters from the Tx FIFO will be shifted out on to the Tx Machine and lost. To prevent the erroneous transmission of data, The CPU must make sure the Transmitter is empty or disabled before issuing the Send Break command.

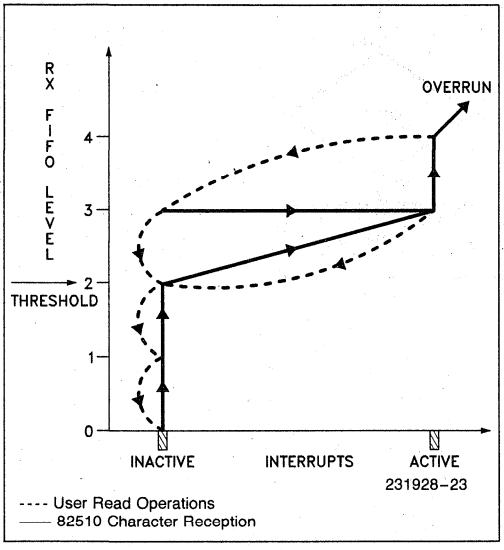


Figure 20. Rx FIFO Hysteresis

5.3 Data Reception

The receiver provides the 82510 with three types of information:

a) Data characters received

b) Rx Flags for each data character

c) Status information on events within the Rx Machine.

The Rx FIFO interrupt request goes active when the Rx FIFO level is greater than the threshold, if the interrupt for this bit is enabled then it will generate an interrupt to the CPU. This is a request for the CPU to read characters from the 82510. Each character on the Rx FIFO has flags associated with it, all of these flags are generated by the Rx Machine during reception of the character. These flags provide information on the integrity of the character, e.g. whether the character was received OK, or if there were any errors. The receiver status is provided via the *Receive Status Register (RST),* which provides information on events occurring within the Rx Machine, since the last time RST was read. The information may or may not apply to the current character being read from the *RXD register.* The CPU may read one or more characters from the Rx FIFO. After each read, if the FIFO contains more than a single character, a new character is loaded into the *RXD register* and the flags for that character are placed into the *RXF register.* The software can check for the Rx character OK bit in the flags to make sure that the character was received without any problems.

5.3.1 RECEIVE INTERRUPT HANDLER

The Receiver will generate two types of interrupts, Rx FIFO interrupt and Rx Machine Interrupt. The Rx FIFO interrupt requires that the CPU read data characters from the Rx FIFO. If the Rx Machine interrupts are disabled then the CPU should also check for errors in the character before moving it to a valid buffer. The interrupts generated by the Rx Machine can be divided into two categories—occurrence of errors during reception of data (parity error, framing error, overrun error), or the occurrence of certain events (Control/Address character received, Break detected, Break Terminated). For typical applications, the error status of each received character can be checked via the *Receive Flags,* and the events can be handled via interrupts.

5.3.2 RECEIVING DATA BY POLLING

To receive data through polling, the 82510 can use the *General Status or the Receive Status Registers* to check for the Rx FIFO request. If the Receive routine does not generate time outs or modem pin transitions, then the data can also be received by monitoring the Rx FIFO level in the *FIFO Level Register.* The implementation using GSR would be useful in applications where the software routine must monitor the timer for time outs or the modem pins for change in status. The example polling routine illustrates the use of the *FIFO Level Register* in receiving data. It waits for the Rx FIFO request before beginning data reception. The procedure Rx_Data_Poll will receive the number of characters requested in Char_count and place them in the Receive buffer.

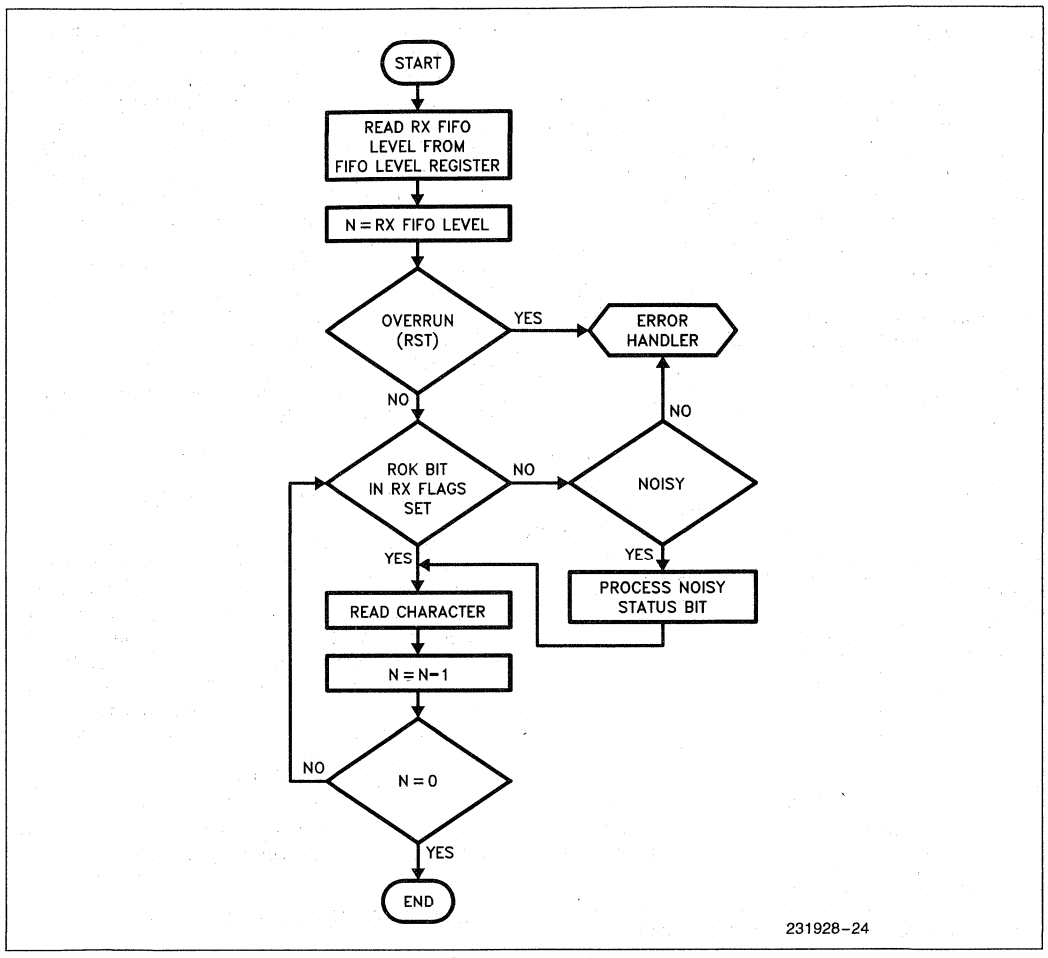


Figure 20. Rx FIFO Interrupt Handler

```
#define base 0x3F8;      /* base address of 82510 */
#define buff__size 128;

Rx__Data__Poll (Char__count, Rxbuffer)
int Char__count;         /* Total # of bytes to be received */
char *Rxbuffer [buffsize];
{
int count = 0;
int status, IvI, Rok;

While (((status = (Inp(base+7) & 0x05)) == 0x01) /* If Rx FIFO Req in GSR set */
{                                                /* Assume in bank one */
   /* If Rx FIFO is not empty */
   While ((IvI = ((Inp (base+4) & 0x70)/0x10)0&&(count < (Char__count))
   {
      /* If Character Received OK */
      if (((Rok = (Inp (base+1) & 0x60)) == 0x40)
      {
         Rxbuffer [count] = Inp (base);
         ++count;
      {
   {
{
{
```

Figure 21. Example Polling Routine

5.4.3 CONTROL CHARACTER HANDLING

The 82510 has two modes of control character recognition. It can recognize either standard ASCII or standard EBCDIC control characters, or it can recognize a match with two user programmed control (or Address Characters in MCS-51 9-bit mode, for Automatic Wake up) characters. Each mode generates an interrupt through the *Receive Status Register.* The *Receive Flags* also indicate whether the character being read is a control character. The usage of CCR depends on the maximum number of possible control characters that can be received at any one time. Applications such as Terminal Drivers, which have no more than two control characters outstanding, such as XON and Ctl-C, or XOFF and Ctl-C, can use just the Control Character Match mode by programming the registers ACR0 and ACR1. If the CPU needs to process text on a line by line basis, the standard Control Character recognition capability can be used to determine when an end of line has occurred e.g. a whole line has been received when a Carriage Return (CR) or Line Feed (LF) is received by the UART.

Implementation of a character oriented asynchronous file transfer protocol can be done using both standard and specific Control Character Recognition. In such protocols most control characters such as Start of Header (SOH), can only be received during certain states, these characters can be received via Standard Control Character Recognition. A few Control Characters (e.g. abort) can be received at any stage of communication, these can be received by using the Control Character Matching capabilities of the 82510.

5.3.3 BREAK RECEPTION

The 82510 has two status indications of break reception, Break Detect indicates that a break has been detected on the RXD pin. Break Terminated indicates that the Break previously detected on the RXD line has terminated and normal Data reception can resume. Each of these status bits can generate an interrupt request through the Rx Machine Interrupt request. Normal consequence of break is to abort the data reception or to introduce a line idle delay in the middle of data reception. In the case of the former, the Break Detect interrupt can be used to reset the 82510 Receive Machine and the Rx routine flags; in the case of the latter, the break terminated interrupt can be used to filter out the break characters and resume normal reception. Each break character is identified by a break flag in the *Rx Flags Register* (the CCR flag, Framing error, and CCR Match flag also may become active when a break character is received) and is loaded onto the Rx FIFO as a NULL character. If break continues even after the Rx FIFO is full, then an overrun error will occur but no further break characters will be loaded on to the Rx FIFO. The user can also measure the length of the break character stream by using the Timer.

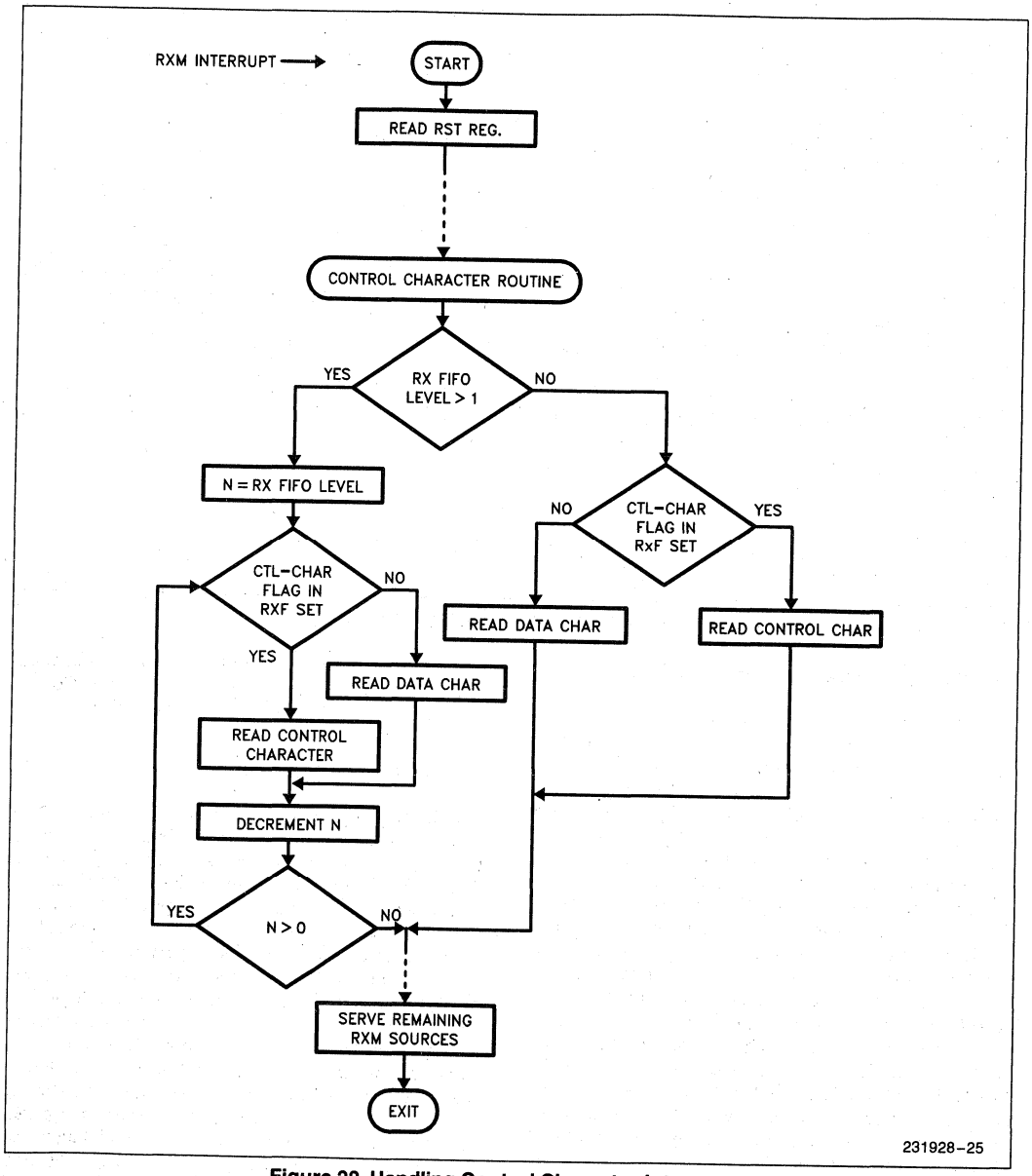


Figure 22. Handling Control Character Interrupts

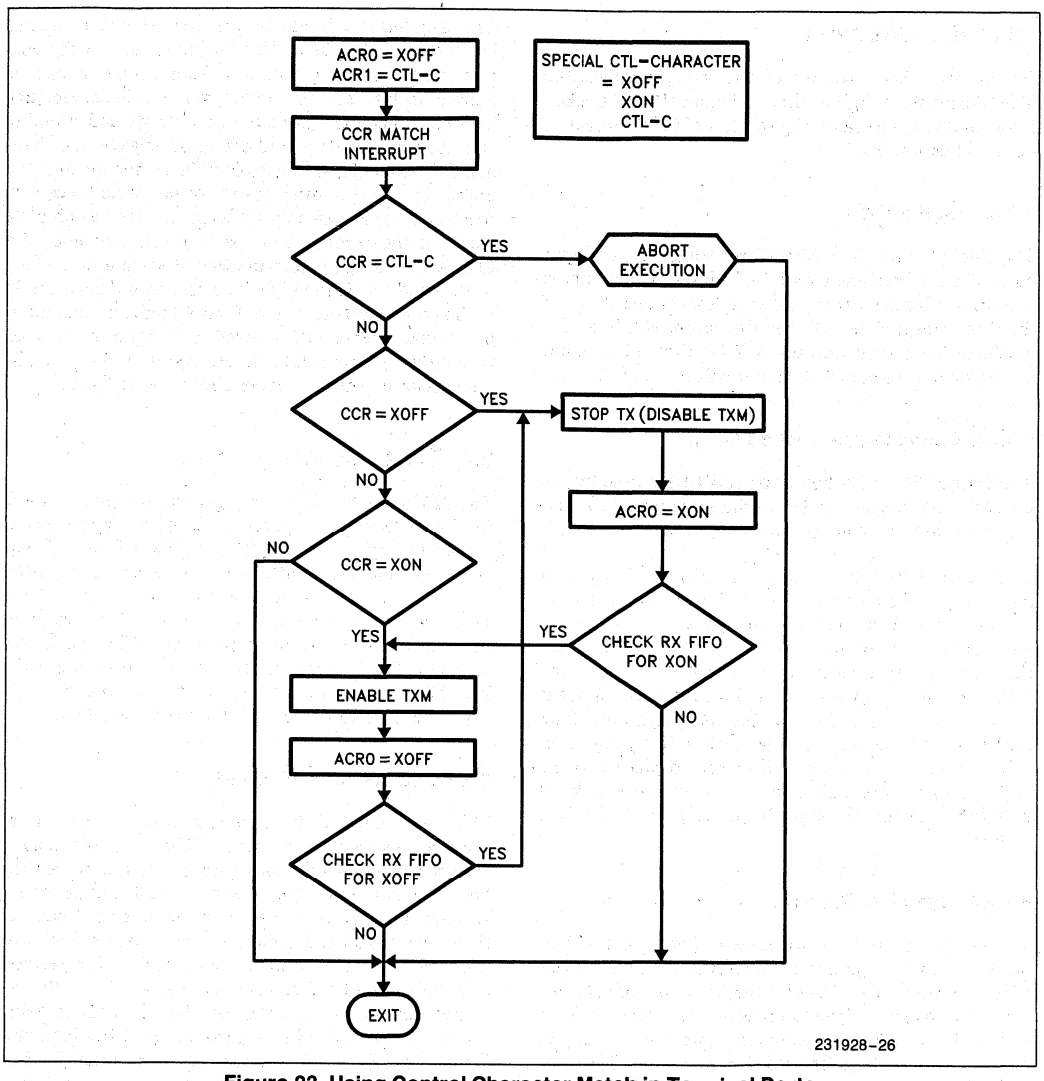


Figure 23. Using Control Character Match in Terminal Ports

5.3.4 DATA INTEGRITY

To improve the reliability of the incoming data the 82510 provides a digital filter, a Digital Phase Locked Loop, and multiple sampling windows (which provide a noise indication bit).

5.3.4.1 Digital Filter

The Digital Filter is used to filter spikes in the input data. The Rx Machine uses a 2 of 3 filter. The output is determined by the majority of samples. If at least two of the three samples are "1" then the output will be a "1". Spikes of one sample duration will be filtered but spikes of two or more samples duration will not be filtered.

5.3.4.2 Digital Phase Locked Loop

The Digital Phase Locked Loop (DPLL) is used by the Rx Machine to synchronize to the incoming data, and adjust for any jitter in the incoming data.

The 82510 DPLL operates on the assumption that a transition in the incoming data indicates the beginning of a new bit cell. A valid asynchronous character frame will contain one or more transitions depending upon the data. If upon occurrence of the transition, the DPLL phase expectation is different from the sampled phase, then there is jitter in the incoming data. The DPLL will compensate for the phase shift by adjusting its phase expectations, until the expected phase and the sampled phase are locked in. The user can enable or disable the DPLL through the *Receive Mode Register (RMD).*

5.3.4.3 Sampling Windows

The sampling windows are used to generate the data bit, by repeated sampling of the RXD line. The bit polarity decision is based upon a majority vote of the samples. If a majority of the samples are "1" then the bit is a "1". If all samples are not in agreement then the Noisy Character bit in the *RXF register* is set. The sampling windows are programmable for either 3 of 16 or 7 of 16. The 3/16 mode improves the jitter tolerance of the medium. While the 7/16 window improves the impulse noise tolerance of the channel.

The sampling windows also provide a Noisy character bit in the *RXF register.* This bit indicates that the current character being read had some noise in one or more of its bits (all the samples were not in agreement). This bit can be used along with the Parity and Framing error bits to provide an indication of noise on the channel. For example, if the Noisy Character bit and the Parity or the Framing errors occur simultaneously, then the noise is probably sufficient to merit a complete check of the communications channel. The noisy bit can also be used to determine when the cable is too long or the baud rate is too high. The user would keep a tally of the noisy characters, and if more than a certain number of characters were received with noise indications, then either the baud rate should be lowered or the distance between the two nodes should be reduced.

5.4 Timer Usage

The 82510 has two baud rate generators, each of these can be configured to operate as Timers. Typical applications use BRG A as a BRG and BRG B as a Timer. Since both the Transmitter and the Receiver may need to generate time outs, it is best to use the Timer as a Time Base to decrement ticks (upon a Timer Expired Interrupt) from (software implemented) Tx and/or Rx counters. The Timer can also be used to time out the Rx FIFO and read characters that otherwise may not have been able to exceed the Rx FIFO threshold.

5.4.1 USE AS A TIME BASE

The transmitter and the receiver routines use a software variable which acts as a counter. The variable is loaded with the required number of ticks that are needed for the Time Out period. Once started the Timer generates an interrupt each time it expires, the interrupt handler then decrements the counters. Once loaded the software monitors the counters until their value reaches zero, this would indicate to the software that the required time period has elapsed. The Time Base value should be selected with regards to the CPU interrupt load. The CPU load will increase substantially when the Timer is used as a Time Base, therefore using the Timer in this mode at very high baud rates may cause character overruns. A time base of 5 or 1 ms is probably the most useful. An additional benefit of the Time Base is that it can support more than two counters if required.

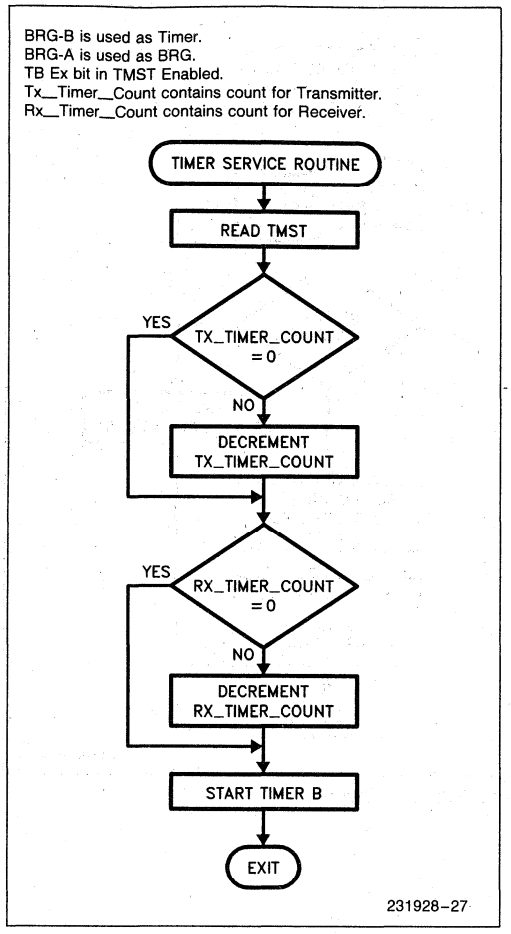


Figure 24. Timer use as Time Base for Transmit and Receive

5.4.2 USE FOR RX FIFO TIME OUT

In the 82510, Rx FIFO interrupts will occur only after the FIFO level has exceeded the threshold. Due to this mechanism and the nonuniform arrival rate of characters in asynchronous communications, there is a chance that characters will be "trapped" in the Rx FIFO for an extended period of time.

For example, assume the 82510 is a serial port on a system and is connected to a terminal. The user is entering a command line. The Rx FIFO Threshold = 3, and at the end only two bytes are received. Since the FIFO threshold has not been exceeded, the Rx FIFO interrupt is not generated. No other characters are received for 30 minutes, if the characters (in the Rx FIFO) are a line feed and carriage return, respectively, the CPU may be waiting for the CR to process the characters it has received. Consequently the characters will not be processed for 30 minutes.

In order to avoid such situations, a Rx FIFO Time Out mechanism can be implemented by using the 82510 Timer. The time out indicates that a certain amount of time has elapsed since the last read operation was performed. It causes the CPU to check the Rx FIFO and read any characters that are present.

In applications where the character reception occurs in a spurious manner (the exact number of characters cannot be guaranteed), the Rx FIFO Time Out is the only way to prevent characters from being trapped. The time out period is measured from the last read operation, every read operation resets the Rx FIFO Timer. To synchronize with the beginning of the data reception, initially the Rx FIFO threshold is set to zero. After the first character has been received, the threshold is adjusted to the desired value. When a Rx FIFO time out occurs and no data is available, the threshold is reset to zero. In error free data transmission, the beginning of data transmission is signaled by the reception of a control character, such as SOH or STX, the Rx FIFO time out mechanism should be triggered to the reception of these control characters.

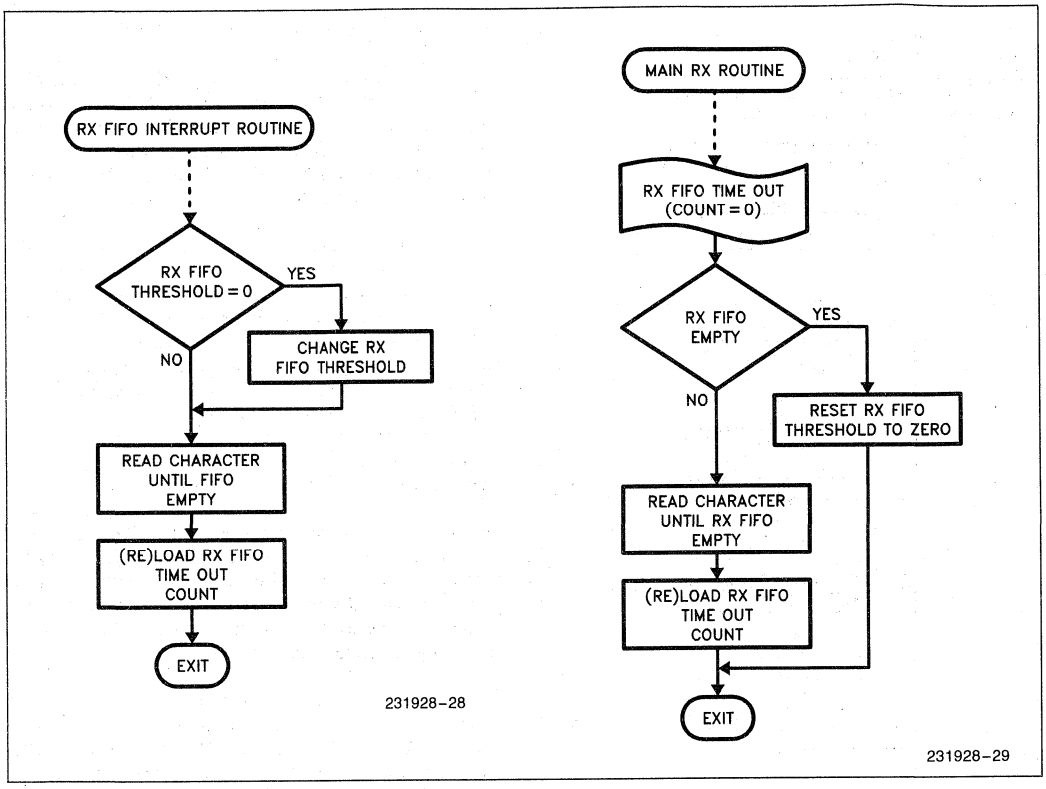


Figure 25. Rx FIFO Time Out Flow Chart

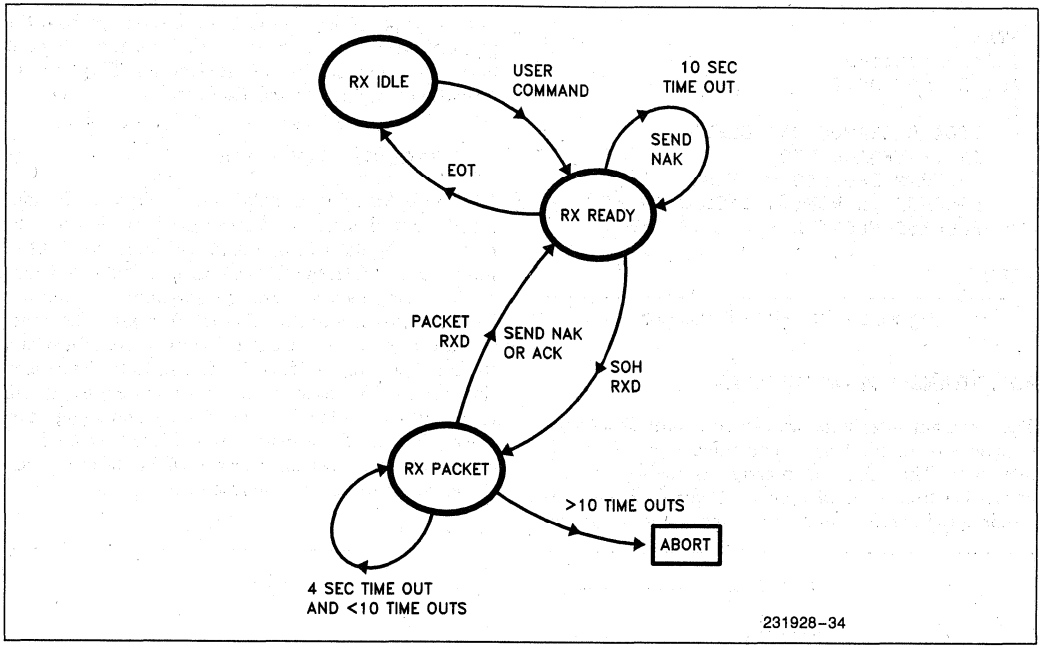


Figure 30. Rx State Machine

```
START
Initialization
WHILE (NOT QUIT)
{
   UPDATE STATUS ON SCREEN
   IF (KEYBOARD HIT)
      THEN PROCESS COMMAND
   PROCESS TRANSMIT STATE MACHINE
   PROCESS RECEIVE STATE MACHINE
}
END
```

Figure 31. Software Structure

6.2.1 TRANSMISSION OF DATA

The Transmit interrupts are disabled until data transmission is required, this prevents unnecessary Transmit interrupts. The Transmit interrupt is enabled when a packet has been assembled or if a Control Character is required to be transmitted. Upon invocation the Transmit interrupt service routine reads characters from the packet buffer and writes it to the Tx FIFO. Since it does not require the use of the Transmit Flags, no information is written to the *TXF register.*

6.2.2 RECEPTION OF DATA

Data reception begins only after a Start of Header (SOH) control character is received. This control character puts the receiver in a data reception mode. After receiving the SOH, the CCR interrupt is disabled (since all data being received now is transparent and can not be interpreted as a control character). After 132 characters are received, the CCR interrupt is reenabled and the corresponding ACK or NAK sent to the Transmitting system. The receiver has a time out feature, which causes it to check the Rx FIFO for any remaining characters. End of Transmission is indicated by an EOT control character, which causes the file to be closed and the Receiver to go into the Idle state.

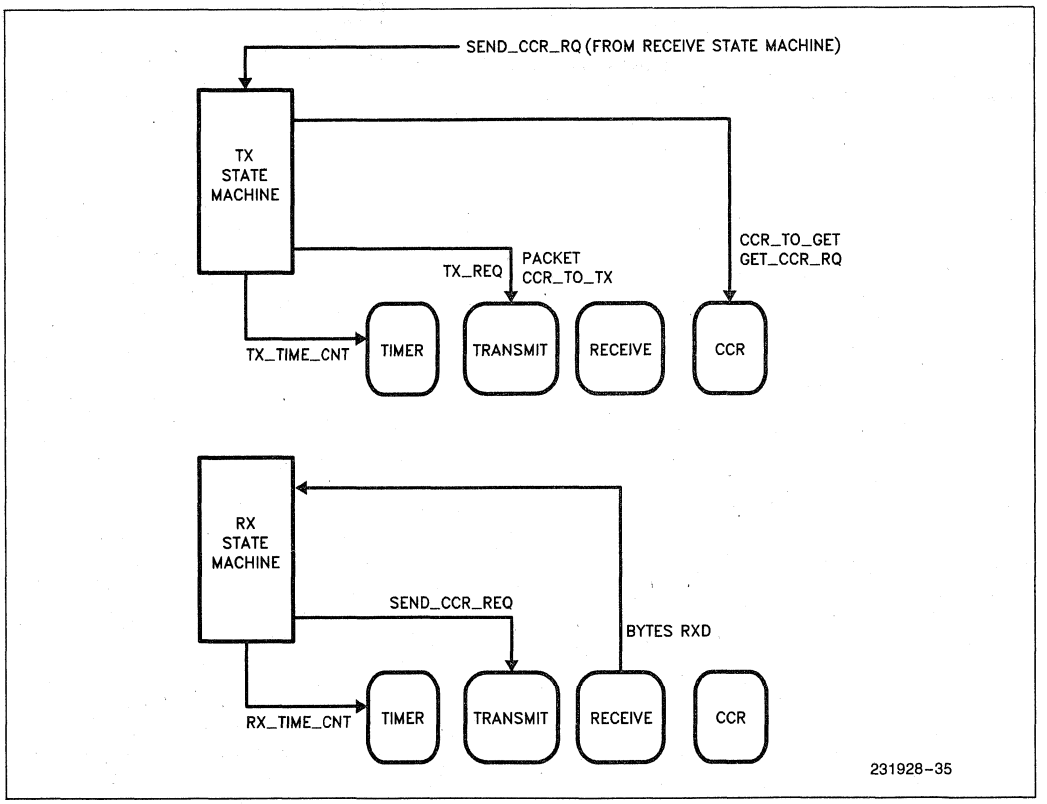


Figure 32. Using Flags for Communications with Interrupt Routine

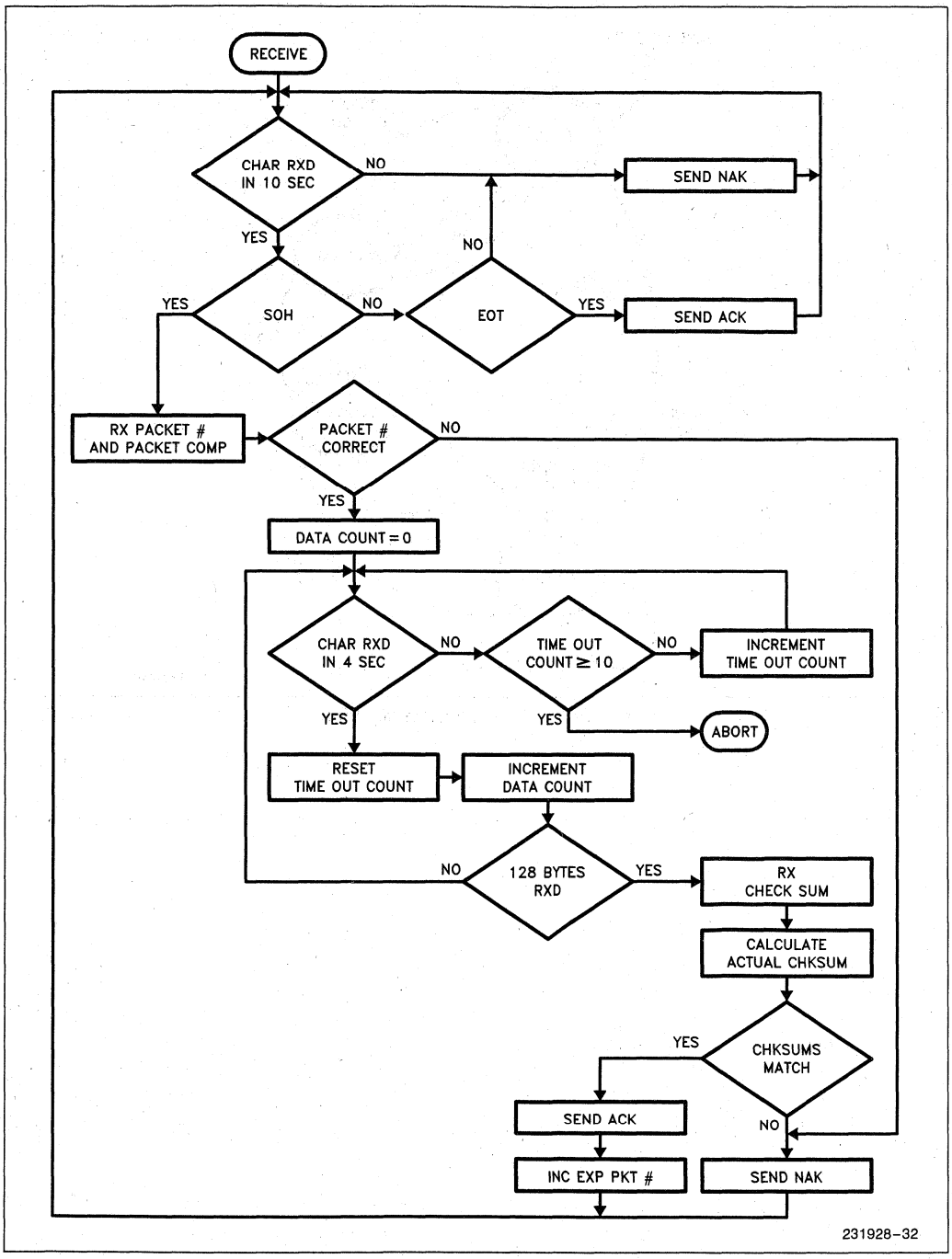


Figure 28. Protocol Flow for Receive Side of XMODEM

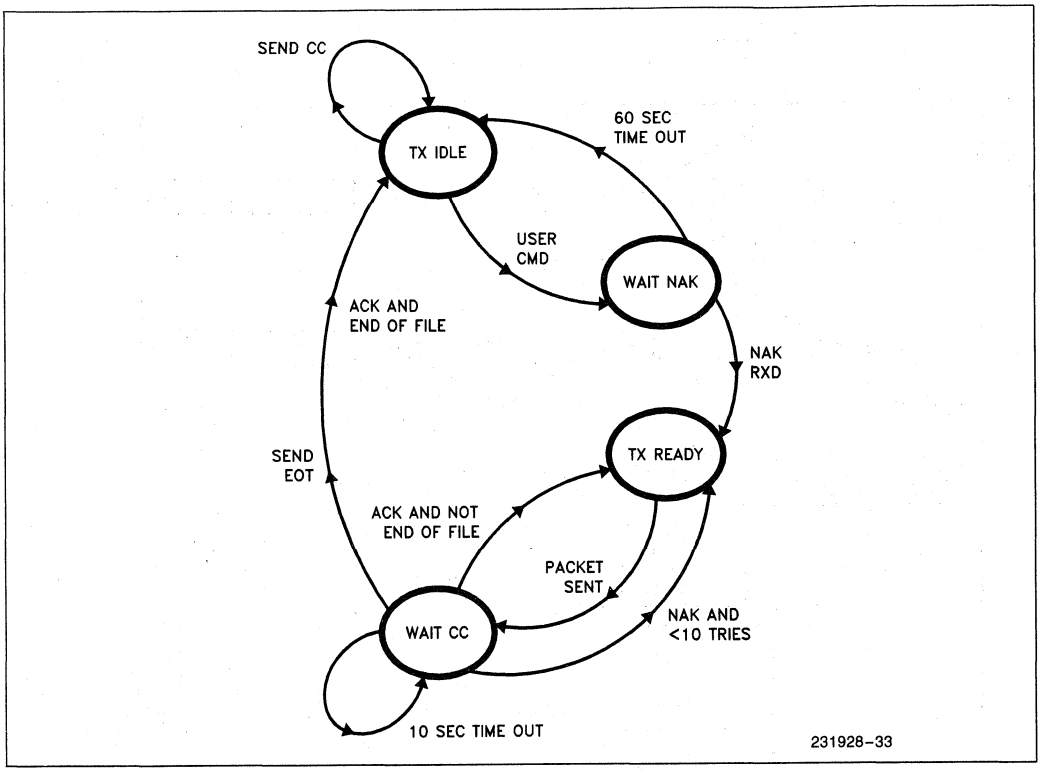


Figure 29. Transmit State Machine

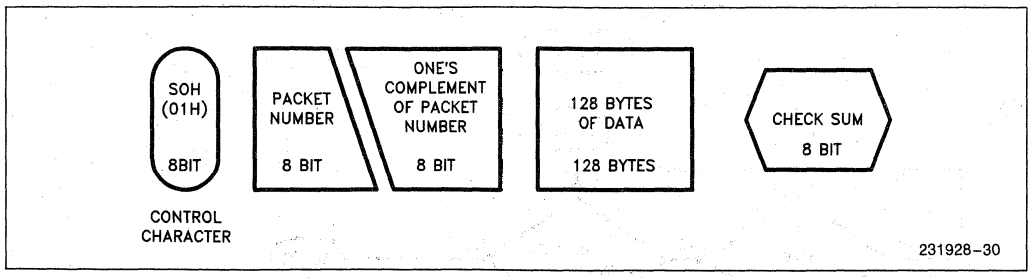


Figure 26. Packet Structure of XMODEM

6.0 82510 IMPLEMENTATION OF XMODEM

The 82510 XMODEM implementation is a file transfer program for the 82510 based on the XMODEM protocol. The software runs on the PC AT on a especially designed adapter board (the adapter board design is shown in Figure 33). The software uses most of the 82510 features including the baud rate generator, Timer, Control Character Recognition and FIFOs. The software uses an interrupt driven implementation, written in both assembly and C languages.

6.1 XMODEM Protocol

XMODEM is a popular error free data transfer protocol for asynchronous communications. Data is transferred in fixed length 128 byte packets, each packet has a checksum for error checking. The packets are delineated by control characters, which act as flags between the Receiver and the Transmitter. There are four control characters, SOH, EOT, ACK, and NAK. SOH indicates the Start of a Packet, EOT indicates the End Of Transmission; ACK and NAK are positive or negative acknowledgements of the packet respectively. The packet structure and protocol flow of XMODEM is provided in the figures given below.

6.2 Software

Interrupts are used to transmit and receive data. The software is implemented as two independent finite state machines—Transmit State Machine and Receive State Machine. Each state machine is triggered by external events such as user commands and data or Control Character reception. The state machines communicate with the 82510 interrupt service routines through software flags. The overall structure of the main routine is given in Figure 31. The major modules of the software are given in the hierarchy Chart, Figure 34, which lists the different modules in order.

The interface between the main program and the interrupt service routine is done through global flags. The interrupt handler services four sources—Transmit, Timer, Receive, and Control Characters. Each of the interrupt sources communicates with each of the state machines through the global flags. The state machines keep track of their individual states through state variables. The interface between the individual states within a state machine is done through state flags. The state machine diagrams are given in Figure 29 and Figure 30.

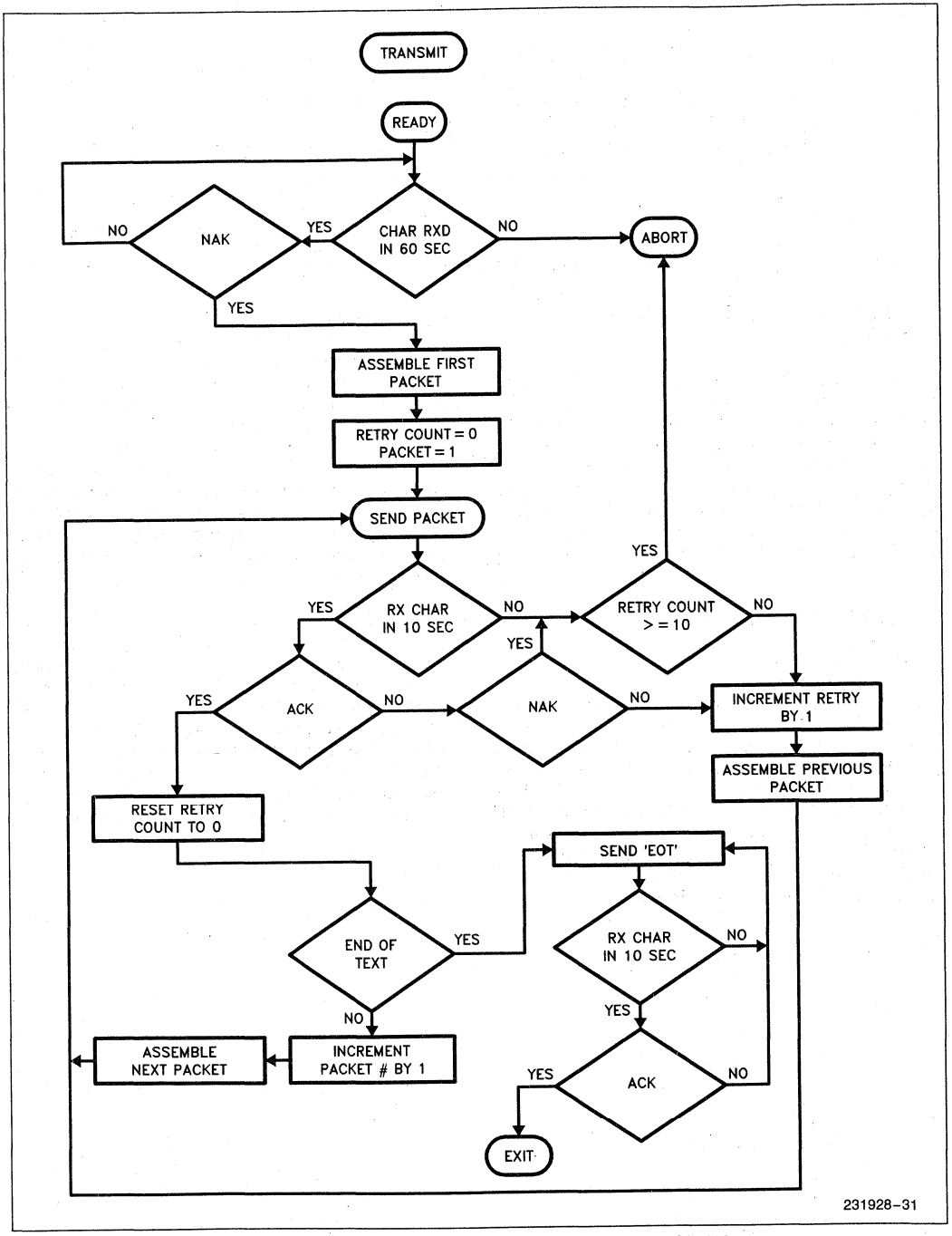


Figure 27. Protocol Flow for Transmit Side of XMODEM

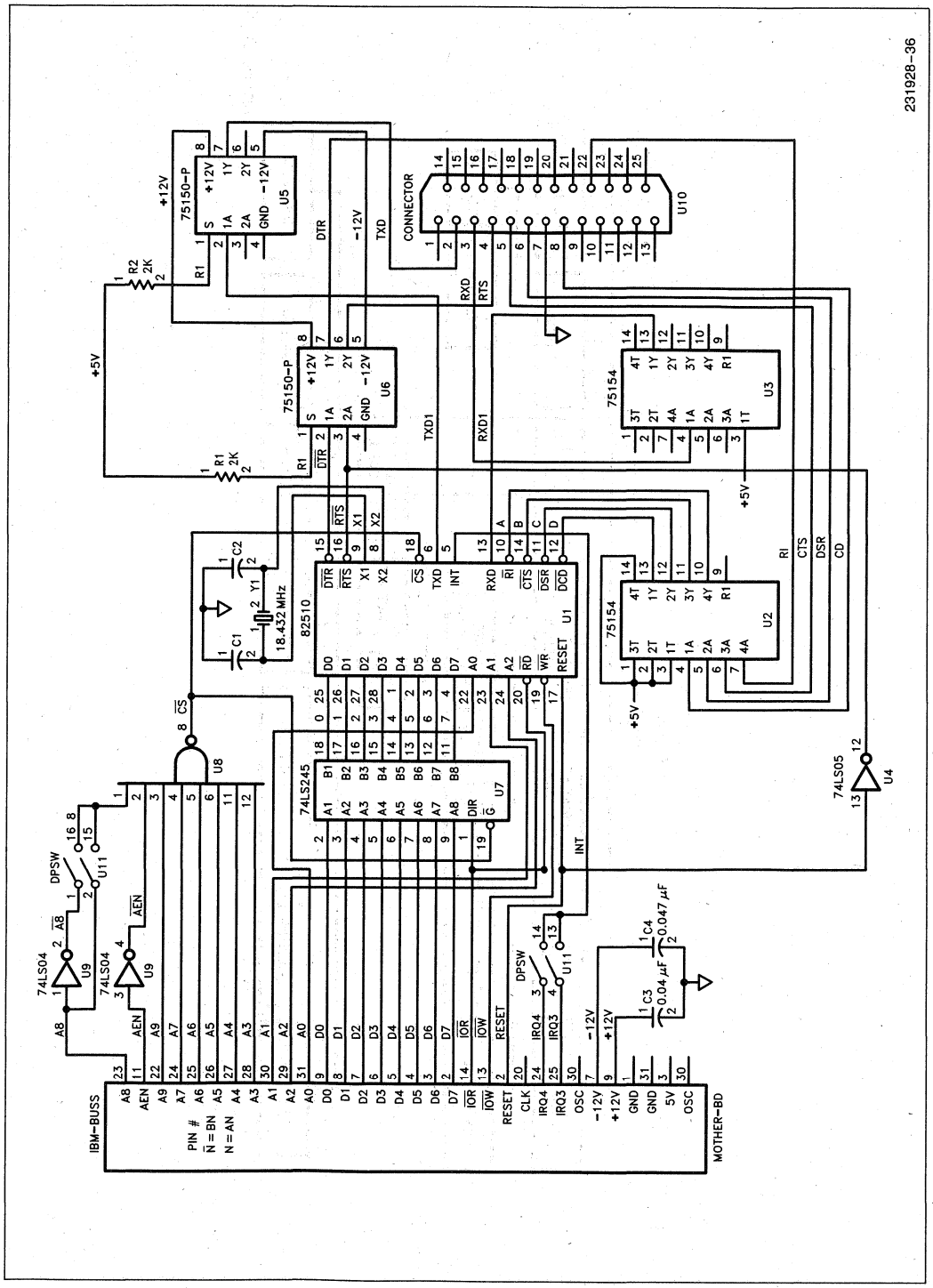


Figure 33. PC AT Adapter

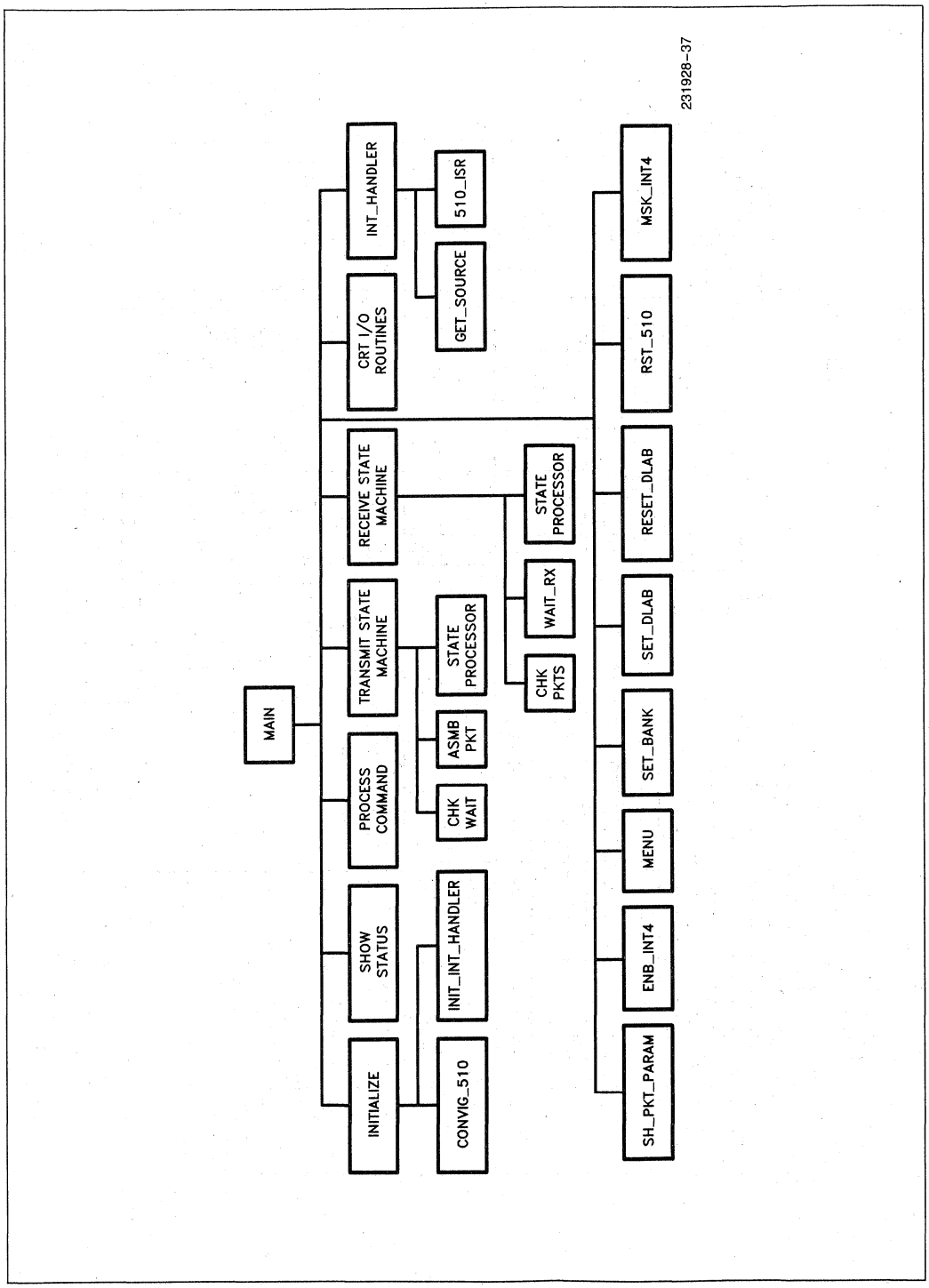


Figure 34. Hierarchy Chart

6.3 Software Listings

```
PAGE  1       MAIN PROGRAM   ftp.c  82510 XMODEM


  1. #include "C:\ftp\ftp.def"
  2. #include "C:\lc\fcntl.h"
  3. #include "C:\lc\stdlib.h"
  4. #include "C:\lc\stdio.h"
  5. /*************************************************/
  6. /***                                          *****/
  7. /***    SEPTEMBER 1986                        *****/
  8. /***                                          *****/
  9. /***    82510 XMODEM IMPLEMENTATION           *****/
 10. /*************************************************/
 11. int     eof =false;          /* end of file flag */
 12. int     rxpk =0;
 13. int     txrflg;
 14. int     rxrflg;
 15. int     exp_pkt_num = 1;     /* next packet number expected by receiver */
 16. int     pkst;
 17. int     rxtocnt;             /* Time Out counter for receiver */
 18. int     quit =false;
 19. int     key = 0;
 20. int     sohcnt =0;           /* # of SOH characters received */
 21. int     rxfcnt =0;           /* # of Rx FIFO Interrupts */
 22. int     ccrcnt =0;           /* # of Ctl-Char. Interrupts */
 23. int     tx_state =tx_idle;   /* Transmitter State Variable */
 24. int     rx_state = rx_idle; /* Receiver State Variable */
 25. int     tx_cmd = inactive;   /* Indicates a Valid Tx Command was given */
 26. int     rx_cmd = inactive;   /* Indicates a valid Rx Command was issued */
 27.
 28. /* File to be Transmitted */
 29. char    tx_file_name[40] = "                                        ";
 30.
 31. /* File to be Received */
 32. char    rx_file_name[40] = "                                        ";
 33.
 34. int     send_ccr_req = inactive;    /* Flag - Request to Tx Ctl-Char */
 35. int     intvec =0;                  /* contains the  GIR vector */
 36. int     l;
 37. char    txdata [132];               /* Tx Buffer */
 38. char    rxbuf [128];                /* Rx Buffer */
 39. char    rxdata [131];
 40. char    rx_f_buf [32000];           /* Rx File Stored in this buffer */
 41.
 42. /*****************************/
 43. /*** tx state variables *****/
 44. /*****************************/
 45. int     tx_indx;                    /* Pointer to the next character in the
 46.                                        buffer */
 47.
 48. struct  packet   {
 49.          char         head;
 50.          char         pack_num;
 51.          char         pack_cmpl;
 52.          char         buffer [128];
 53.          char         chksm;
 54.       } ;
 55.
 56. struct packet rxpack, txpack;
 57.
```

231928-38

82510 XMODEM Implementation

```
PAGE  2       MAIN PROGRAM   ftp.c  82510 XMODEM


 58.
 59. /*****************************************************/
 60. /****       tx State machine and interrupt       ******/
 61. /****       handler  flags                       ******/
 62. /*****************************************************/
 63.
 64. /** txm and tx fifo  **/
 65. int     tx_req =0;       /* Flag - indicates a request for transmission to
 66.                             82510 Interrupt Handler */
 67. int     ccr_to_tx = 0;   /* Actual Ctl-char to Transmit */
 68. int     tx_byte_cnt =0; /* Total # of Bytes Transmitted */
 69. int     pkts_sent =0;    /* # of Packets sent */
 70.
 71. /** Timer   **/
 72. int     tx_time_cnt =0; /* Transmitter Timer Counter */
 73.
 74. /** CCR **/
 75. int     get_ccr_rq =0;   /* Flag - Request to Receive Ctl-character */
 76. int     ccr_to_get  =0; /* Received Ctl-char value */
 77.
 78.
 79. /*************************************************************/
 80. /****                                                 *****/
 81. /****       RX STATE VARIABLES                        *****/
 82. /****                                                 *****/
 83. /****                                                 *****/
 84. /*************************************************************/
 85.
 86. char    pk_chksm;        /* Calculated Chksum */
 87. int     eot_cnt =0;      /* # of EOTs Received */
 88. int     bad_pkt_cnt;     /* # of Bad Packets Received */
 89.
 90. /*****************************************************/
 91. /****       rx state machine and Interrupt       ******/
 92. /****       handler flags                        ******/
 93. /*****************************************************/
 94.
 95. /** rx fifo **/
 96. int     rx_byte_cnt =0;      /* # of Bytes Received */
 97.
 98. /** CCR **/
 99.
100. int     ctl_rxd_flg =0;      /* Flag-Indicating that a Ctl-Char. has been
101.                                 receivd*/
102. int     rx_ctl_chr=0;        /* Actual Ctl-char received */
103.
104. /** Timer   **/
105. int     rx_time_cnt =0;      /* Receive Timer Count */
```

231928-39

82510 XMODEM Implementation (Continued)

```
PAGE  3       MAIN PROGRAM   ftp.c  82510 XMODEM


106.
107. /*****************************************/
108. /**          MAIN ROUTINE                ***/
109. /*****************************************/
110.
111. main ()
112. {
113. int     q,txfl,rxfl;
114. int     rval,v =0;
115. int     cmd = 0;
116. int     wn_status =0;
117. int     ecode = 0;
118. FILE    *fp;
119. FILE    *rxfp;
120. int     rwst;
121. int     wcst;
122. int     retx_cnt =0;            /* Retransmit count */
123. int     tocnt =0;               /* Time Out Count */
124. int     tx_secs,rx_secs;
125. int     i,s, lpcnt = 0;
126.
127.     CLR ();                     /* Clear Screen */
128.     MV_CURS (so_r,so_c);        /* Sign On Message */
129.     printf (s1);
130.     init ();                    /* Initialize 82510 and Variables */
131.     MENU ();                    /* Print Menu */
132.     enbint4 ();                 /* Enable Interrupts in 8259A */
133.     outp (ip00,eoi);            /* issue EOI    */
134.     outp ((bpa+3),0x22);        /* start timer B */
135.     lpcnt =0;                   /* Keeps Track of # of Loops */
136.
137. /*****************************************************/
138. /***            main while loop                  ****/
139. /*****************************************************/
140.     while (quit==false)
141.     {
142.
143.     /*************************************/
144.     /** display protocol parameters     **/
145.     /*************************************/
146.
147.       ++ lpcnt;
148.       mv_curs (4,30);
149.       printf ("loop # = %u",lpcnt);
150.       mv_curs (4,50);
151.       printf ("rx int. cnt = %u",rxfcnt);
152.       mv_curs (5,50);
153.       printf ("ccr int cnt = %u",ccrcnt);
154.       mv_curs (4,1);
155.       printf ("interrupt vector = %u \n", intvec);
156.       q = inp (bpa+4);
157.       txfl = q & 0x07;
158.       mv_curs (5,1);
159.       printf ("TX FIFO = %u ",txfl);
160.       q = inp (bpa+4);
161.       rxfl = q & 0x70;
162.       mv_curs (6,1);
163.       printf ("RX FIFO = %u \n",rxfl/16);
164.       mv_curs (6,50);
165.       printf ("SOH count = %3u",sohcnt);
```

231928-40

82510 XMODEM Implementation (Continued)

```
PAGE  4       MAIN PROGRAM   ftp.c  82510 XMODEM


166.        mv_curs (7,1);
167.        printf ("bytes received %3u",rx_byte_cnt);
168.        mv_curs (7,30);
169.        printf ("Bytes Sent = %3u",tx_byte_cnt);
170.        mv_curs (7,50);
171.        printf  ("EOT count %3u", eot_cnt);
172.        mv_curs (5,30);
173.        printf ("pkts rxd = %3u",(exp_pkt_num-1));
174.        mv_curs (6,30);
175.        printf ("pkts sent = %3u", pkts_sent);
176.        tx_secs = tx_time_cnt/200;
177.        rx_secs = rx_time_cnt/200;
178.        open_wind (3,1,"Tx Timer");
179.        printf (" = %2u secs",tx_secs);
180.        open_wind (3,50,"Rx Timer");
181.        printf (" = %2u secs",rx_secs);
182.        mv_curs (8,1);
183.        printf ("Bad Packets Rxd = %3u",bad_pkt_cnt);
184.        mv_curs(8,30);
185.        printf ("# of ReTx packets = %3u",retx_cnt);
186.
187.  /* If Command Issued then process the Command */
188.        if ((key =kbhit()) ) 0)
189.           quit = process_cmd ();
190.
191.        else
192.        {
193.
194.          /*************************************************************/
195.          /*****          Process Tx STATE MACHINE             *****/
196.          /*****          revision 0                           *****/
197.          /*****                                               *****/
198.          /*************************************************************/
199.
200.          switch (tx_state) {
201.              case tx_idle:
202.
203.          /*************************************************************/
204.          /****                                                 *****/
205.          /****  TRANSMITTER IDLE STATE                         *****/
206.          /****                                                 *****/
207.          /****              Checks for a Send Ctl-Char.        *****/
208.          /****              Checks for the Transmit Command    *****/
209.          /****                                                 *****/
210.          /****                                                 *****/
211.          /*************************************************************/
212.
213.
214.                  /* If Control Character to be Transmitted Then Transmit the
215.                     Control Character by setting the Tx_req Flag and enabling
216.                     the TxM and Tx FIFO interrupts */
217.
218.                  if ((send_ccr_req ==  active) && (!tx_req))
219.                  {
220.                      tx_req =ctl_chr;
221.                      tx_i_enb ();
222.                      while ( tx_req>0);
223.                      tx_i_dis ();
224.                      send_ccr_req=inactive;
225.                  }
```

231928-41

82510 XMODEM Implementation (Continued)

```
PAGE  5       MAIN PROGRAM   ftp.c  82510 XMODEM


226.                /* If the Transmit Command is issued then Wait for a NAK */
227.                if (tx_cmd ==  active)
228.                 {
229.                    tx_cmd = inactive;
230.                    get_ccr_rq =active;
231.                    tx_time_cnt =200*60; /* 60 sec. Time Out */
232.                    tx_state = wait_NAK;
233.                 }
234.                break;
235.
236.            case wait_NAK : /* Waiting for a NAK character to begin Tx */
237.
238.        /*********************************************************/
239.        /****  TRANSMITTER WAITING FOR A NAK TO BEGIN       *****/
240.        /****  TRANSMISSION.                                *****/
241.        /****                                               *****/
242.        /****              Checks For Time Out              *****/
243.        /****                    or  NAK Received           *****/
244.        /*********************************************************/
245.
246.                wn_status = check_wait ();      /* Time Out or NAK Rcvd? */
247.                switch (wn_status)  {
248.
249.                    case time_out :             /* If Time Out then Abort
250.                                                   Transmission */
251.                        tx_state =tx_idle;
252.                        beep ();
253.                        prmsg ("Time OUT !!!! receiver not ready");
254.                        cll (tx_r, tx_c);
255.                        open_wind (tx_r,tx_c,"NONE");
256.                    break;
257.
258.                    case waiting :              /* if no Time Out and no NAK
259.                                                   rcvd then do nothing */
260.                    break;
261.
262.                    case rx_NAK :
263.                                                /* If NAK received then Open
264.                                                   file and advance to
265.                                                   Transmit Packet State */
266.
267.                        fp =fopen (tx_file_name,"rb" );
268.                        if (fp== NULL)
269.                        {
270.                         beep ();
271.                         prmsg ("ERROR !!! file does not exist");
272.                         cll (tx_r,tx_c);
273.                         open_wind (tx_r,tx_c, "none");
274.                         tx_state =tx_idle;
275.                        }
276.                     else
277.                            {
278.                            tx_state = tx_rdy;
279.                            txrflg = mkpkt;     /* First task for Tx
280.                                                   is to Prepare Packet */
281.                            wn_status = 0;      /* Reset Wait_NAK Flag */
282.                            }
283.                    break;
284.                    }
285.            break;/* end  wait nak */
```

231928-42

82510 XMODEM Implementation (Continued)

```
PAGE  6       MAIN PROGRAM   ftp.c  82510 XMODEM


286.            case    tx_rdy :
287.            /*********************************************************/
288.            /****  TANSMITTER READY TO TRANSMIT                   *****/
289.            /****               three stages of transmission      *****/
290.            /****                      prepare packet             *****/
291.            /****                      Int. Handler Transmitting  *****/
292.            /****                      or retransmit request      *****/
293.            /*********************************************************/
294.
295.                /* Any Control Character To Transmit? */
296.                if ((send_ccr_req == active) && (tx_req==0))
297.                   tx_req =ctl_chr;
298.
299.                /* Which Stage of transmission ?*/
300.                switch (txrflg)
301.                {
302.                case mkpkt:                     /* Prepare Packet  */
303.                    if  (tx_req==0)
304.                    {
305.                     asmbpkt (pkts_sent,fp);    /* Assemble Packet */
306.                     cpy2buf ();
307.                     tx_req =pkt;               /* Request Int. Handler
308.                                                   to Tx data in buffer */
309.                     txrflg =txmtg;             /* Start Transmission */
310.                     tx_indx =0;
311.                     tx_i_enb ();               /* Enable TxM and Tx FIFO
312.                                                   Interrupts */
313.
314.                    }
315.                break;
316.                case txmtg :
317.                    if (tx_req == 0)            /* Interrupt Handler Resets
318.                                                   this flag to 0, when 132
319.                                                   bytes are transmitted */
320.                    {
321.                      tx_indx =0;
322.                      prmsg ("packet transmitted");
323.                      get_ccr_rq =active;       /* Wait for ACK or NAK */
324.                      tx_time_cnt = 200*10;     /* 10 sec Time Out */
325.                      tx_state = wait_CC;       /* Wait for ctl Character */
326.                      txrflg =mkpkt;
327.                      tx_i_dis ();              /* Disable TxM and Tx FIFO
328.                                                   Interrupts */
329.                    }
330.                    else                        /* Tx_req not reset then
331.                        prmsg ("transmitting");    still transmitting */
332.                break;
333.                case retx :                     /* The Retransmit request is
334.                                                   issued by the Wait _CC
335.                                                   state */
336.                     outp((bpa+6),txen);        /* enable txm, flush tx fifo
337.                                                   & txm **/
338.                     tx_req = pkt;              /* transmit Packet,pkt. in
339.                                                   buffer */
340.                     txrflg =txmtg;             /* next task - ReTransmit */
341.                     tx_i_enb ();               /* Enable TxM and Tx FIFO
342.                                                   Interrupts */
343.                break;
344.                }
345.            break; /* End tx rdy case */
```

231928-43

82510 XMODEM Implementation (Continued)

```
PAGE  7       MAIN PROGRAM   ftp.c  82510 XMODEM


346.     case wait_CC :
347. /*********************************************************/
348. /****                                                 *****/
349. /****  Transmitter State - Waiting For Ctl Char.       *****/
350. /****                                                 *****/
351. /****            NAK - requests retransmission        *****/
352. /****            ACK - Transmit Next Packet           *****/
353. /****                                                 *****/
354. /****                                                 *****/
355. /*********************************************************/
356.
357.             wcst = check_wait ();              /* Check for one of the
358.                                                   Following events:
359.                                                    Time Out
360.                                                    NAK Received
361.                                                    ACK Received
362.                                                   or Still Waiting */
363.             switch (wcst)
364.              {
365.               case time_out :
366.
367.                                                /* If Time Out, then restart
368.                                                   Tx Timer. Abort if Time
369.                                                   Out count is greater than
370.                                                   ten */
371.                  if (tocnt )10)
372.                      {
373.                      wcst =0;
374.                      abort_tx ();
375.                      prmsg ("receiver not responding");
376.                      }
377.                  else
378.                      {
379.                       ++tocnt;                 /* Inc. Time Out Count */
380.                       tx_time_cnt =200*10;
381.                      }
382.
383.               break;
384.
385.               case waiting :                   /* if waiting, do nothing */
386.               break;
387.
388.               case rx_NAK :                    /* If NAK or Corrupted
389.                                                   ctl-char. received */
390.               case rx_gen :
391.                 prmsg ("NAK received");
392.                 if (retx_cnt )10)              /* more than 10 attempts,
393.                                                   then Abort*/
394.                   {
395.                    retx_cnt =0;
396.                    tocnt =0;
397.                    abort_tx ();
398.                    prmsg ("Bad link transmission aborted");
399.                   }
```

231928-44

82510 XMODEM Implementation (Continued)

```
PAGE  8       MAIN PROGRAM   ftp.c  82510 XMODEM


400.                  else                              /* If Retransmit Count Not
401.                                                       exceeded then go back to
402.                                                       Transmit stage - task is
403.                                                       retransmit */
404.                   {
405.                    txrflg =retx;
406.                    ++ retx_cnt ;
407.                    tx_state =tx_rdy;
408.                   }
409.                break;
410.
411.               case rx_ACK:                          /* ACK Received*/
412.                   prmsg ("ACK received");
413.                   retx_cnt=0;
414.                   tocnt = 0;
415.                   ++pkts_sent;
416.                   printf ("pkts_sent = %3u", pkts_sent);
417.                   if (eof ==false)                  /* If more data to transmit
418.                                                       then retrun to mkpkt
419.                                                       stage and tx new pkt. */
420.                      {
421.                       txrflg =mkpkt;
422.                       tx_state =tx_rdy;
423.                      }
424.                   else
425.                    {
426.                     prmsg ("sending EOT");          /* if end of file , then
427.                                                       send EOT */
428.                     ccr_to_tx = EOT;
429.                     tx_req =ctl_chr;
430.                     tx_i_enb ();
431.                     while (tx_req != 0);            /* wait for Int. Handler
432.                                                       to reset flag */
433.                     tx_i_dis ();
434.                     get_ccr_rq =active;             /* wait for Ack */
435.                     while (get_ccr_rq ==active);
436.                     prmsg ("EOT acknowledgement received");
437.                     if (ccr_to_get == ACK)          /* ACK rxd , Close File */
438.                     {
439.                        s = fclose (fp);
440.                        abort_tx();
441.                        prmsg ("file transmission complete");
442.                     }
443.                     tx_state =tx_idle;              /* Return to Idle */
444.                    }
445.               break;
446.               } /* /end wait_cc case */
447.          break;
448.          } /* end switch tx state */
```

231928-45

82510 XMODEM Implementation (Continued)

```
PAGE  9       MAIN PROGRAM   ftp.c  82510 XMODEM


449. /*****************************************************/
450. /*****          Process Rx STATE MACHINE             *****/
451. /*****          revision 0                           *****/
452. /*****                                               *****/
453. /*****************************************************/
454.          switch (rx_state)
455.          {
456.              case rx_idle:
457.              /*************************************************************/
458.              /****                                                     *****/
459.              /**** RECEIVER IDLE:                                     *****/
460.              /****                                                     *****/
461.              /****                waits for user command             *****/
462.              /****                before sending NAKs                *****/
463.              /****                                                     *****/
464.              /****                                                     *****/
465.              /*************************************************************/
466.
467.                  if (rx_cmd == active)       /* If receive Command is issued
468.                                                 then start Rx timer and change
469.                                                  Receiver state to ready */
470.                      {
471.                          rx_state = rx_rdy;
472.                          rx_time_cnt =200*10;
473.                          rx_cmd = inactive;
474.                      }
475.              break;
476.
477.              case rx_rdy:
478.              /*************************************************************/
479.              /****                                                     *****/
480.              /**** RECEIVER READY:                                     *****/
481.              /****                sends NAK upon Time Out            *****/
482.              /****                or checks for SOH                  *****/
483.              /****                or EOT ctl-char.                   *****/
484.              /****                                                     *****/
485.              /****                                                     *****/
486.              /*************************************************************/
487.
488.                  rxrflg =wait_rx ();     /* Checks Rx Timer and returns -
489.                                             Time Out if expired
490.                                             waiting if not expired
491.                                             SOH if SOH ccr received
492.                                             EOT if EOT ccr received */
493.                  switch (rxrflg)
494.                  {
495.                   case waiting :         /* If waiting then do nothing */
496.                   break;
497.
498.                   case  SOH :            /* If SOH received, then go into
499.                                             data reception mode and change Rx
500.                                             Timer count to 4 secs */
501.                      ++ sohcnt;
502.                      rx_state =rx_pkt;
503.                      rx_time_cnt =200*4; /* four second time out */
504.                      rxtocnt =0;
505.                  break;
506.
```

231928-46

82510 XMODEM Implementation (Continued)

```
PAGE 10       MAIN PROGRAM   ftp.c  82510 XMODEM


                case time_out:          /* if time out & not in the midst of
                                           packet reception then send NAK */
                    if (( exp_pkt_num ==1) && (rx_byte_cnt ==0))
                    {
                        prmsg ("rx time out !!!!! sending NAK");
                        if (send_ccr_req ==inactive)
                        {
                            ccr_to_tx =NAK;
                            send_ccr_req =active;
                        }
                    }
                    rx_time_cnt =200*10;
                break;

                case EOT:                           /* If End Of Text rcvd,
                                                       and data rcvd then
                                                       send ACK and save all
                                                       packets received in
                                                       file */
                    ++ eot_cnt;
                    open_wind (22,50,"End of Text");
                    if (exp_pkt_num )1)
                    {
                      if (send_ccr_req == inactive) /* Send ACK */
                      { send_ccr_req =active;
                        ccr_to_tx =ACK;
                      }
                      rx_state =rx_idle;            /* Receiver Returns to
                                                       Idle */
                                                    /* create file */
                      rxfp =fopen (rx_file_name,"ab+");
                      rwst =fwrite (&rx_f_buf[0],128,exp_pkt_num-1,rxfp);
                      if (rwst (1)
                            {
                             prmsg ("Write file error ");
                             printf ("error = %4u",(rwst=ferror(rxfp)));
                            }
                      rval =fclose (rxfp);
                      if (rval ==0)
                          prmsg ("file received");
                      else
                        prmsg ("Error in closing file ");

                    }
                break;
                }
            break;

    /*.PA*/
            case rx_pkt: /* Packet reception */
            /*****************************************************/
            /*****                                           *****/
            /*****   RECEIVE PACKET STATE                    *****/
            /*****                                           *****/
            /*****               checks for Time Out         *****/
            /*****               or 131 bytes received       *****/
            /*****               which signals the end of packet *****/
            /*****                                           *****/
            /*****************************************************/
```

231928-47

82510 XMODEM Implementation (Continued)

```
PAGE 11       MAIN PROGRAM   ftp.c  82510 XMODEM


567.            /* If valid Rx Time Out , i.e. no data received for 4 secs then
568.               check Rx FIFO for characters and read if any available */
569.            if ((rx_time_cnt ==0) && (rx_byte_cnt (131))
570.            {
571.                rxfl = ((inp (bpa +4) & 0x70)/ 0x10);       /* check Rx FIFO
572.                                                               Level */
573.                if ((rxtocnt )= 10) && (rxfl (=0))          /* if more than
574.                                                               10 attempts
575.                                                               and no data
576.                                                               then abort
577.                                                               transmit */
578.                {
579.                    rx_state =rx_idle;
580.                    prmsg (" Receiver Time Out, no DATA");
581.                    rxtocnt =0;
582.                }
583.
584.                else
585.                /* otherwise restart Rx Timer, and read data from 510 */
586.                {
587.                 if (rxfl != 0)                         /* Rx FIFO level ) 0 */
588.                 {
589.                     rx_time_cnt =200*5;
590.                     rxfl = ((inp ( bpa +4) & 0x70)/0x10);
591.                     while ( rxfl != 0)                 /* Read from FIFO */
592.                     {
593.                         rxdata [rx_byte_cnt] = inp (bpa);
594.                         ++ rx_byte_cnt;
595.                         ++ rxfcnt;
596.                         -- rxfl;
597.                     }
598.                     rxtocnt = 0;
599.                 }
600.                 else
601.                 {
602.                     ++ rxtocnt;                        /* inc. receive TimeOut
603.                     rx_time_cnt =200*4;                   Count */
604.                 }
605.                }
606.            }
607.            else
608.            {
609.                if (rx_byte_cnt == 131)                 /* Packet Received */
610.                {
611.                    rx_byte_cnt =0;
612.                    rxtocnt =0;
613.                    pkst =chkpkt (exp_pkt_num);        /* Check Packet */
614.                                                       /* returns EOK if Packet
615.                                                          without errors */
616.        /* PA*/
617.                    if ((pkst == eok) || (pkst ==eold))
618.                    {
619.                        prmsg ("sending ACK");
620.                        for (i=0; i(128; i++)
621.                            rxbuf [i] = rxdata [i+2];
622.                        /** write packet to buffer  **/
623.                        if (pkst ==eok)
624.                        {
625.                                                       /* copy to main file
626.                                                          buffer */
```

231928-48

82510 XMODEM Implementation (Continued)

```
PAGE 12        MAIN PROGRAM    ftp.c  82510 XMODEM


627.                                buf_cpy (exp_pkt_num);
628.                                ++ exp_pkt_num;
629.                            }
630.                            else
631.                                prmsg ("old packet retransmitted");
632.                        }
633.
634.                        else
635.                            sh_pkt_param ();            /* If error then show
636.                                                           packet #, chksum and
637.                                                           packet complement */
638.                        rx_state = rx_rdy;
639.                        mskint4 ();                     /* Enable Ctl-Chr int*/
640.                        set_bank (00);
641.                        outp ((bpa+1),(inp(bpa+1)|ccien));
642.                        set_bank (01);
643.                        enbint4 ();
644.                        send_ccr_req =active;           /* Send ACK */
645.                        ccr_to_tx =ACK;
646.
647.                    }
648.
649.                }
650.                break;
651.
652.            } /** end switch rx state **/
653.
654.
655.        } /* end else */
656.
657.    } /* end while quit */
658.
659.    rst510 ();                                  /* reset 82510 */
660.    outp ((bpa + 1),00);                        /* disable 82510 interrupts */
661.
662.
663.    cmd = 0x10;                                 /* disable 8259A interrupt */
664.    v=inp (0x21);                               /*       00010000          */
665.    cmd = (v | cmd);
666.    outp  (0x21,cmd);
667.    clr ();
668.    ecode  = 0;
669.    _exit (ecode);
670.
671.    }/* end main */
672.
```

231928-49

82510 XMODEM Implementation (Continued)

```
PAGE 13       MAIN PROGRAM   ftp.c  82510 XMODEM


673.
674. rst510 ()
675. /*****************************************************/
676. /****                                             *****/
677. /****    RESET 82510 to default wake up mode      *****/
678. /****                                             *****/
679. /****                                             *****/
680. /****                                             *****/
681. /*****************************************************/
682.
683. {
684. set_bank (01);
685. outp ((bpa+7),0x10);
686. }
687.
688.
689. menu ()
690. /*************************************************/
691. /**     displays the menu on the                **/
692. /**     screen.                                 **/
693. /**                                             **/
694. /**                                             **/
695. /**                                             **/
696. /*************************************************/
697. {
698.
699.
700.     open_wind (1,1,"baud rate");
701.     printf (" = 1200  ");
702.     open_wind (1,22,"char. size");
703.     printf (" = 8 bits");
704.     open_wind (1,45 , "Parity");
705.     printf (" disabled");
706.     open_wind (1,68, "Stop Bits");
707.     printf (" = 2");
708.     mv_curs (2,1);
709.     printf ("user messages :");
710.     mv_curs (10,15);
711.     printf ("(1) TRANSMIT FILE : ");
712.     OPEN_WIND (tx_r,tx_c,"none");
713.     mv_curs (12,15);
714.     printf ("(2) RECEIVE FILE : ");
715.     OPEN_WIND (rx_r,rx_c,"none");
716. }
717.
```

231928-50

82510 XMODEM Implementation (Continued)

```
PAGE 14       MAIN PROGRAM   ftp.c  82510 XMODEM


718.
719. init    ()
720. /*************************************************/
721. /**  Intializes Software and Configures          **/
722. /**  the 82510. Also sets up the interrupt       **/
723. /**   Handler.                                   **/
724. /**                                              **/
725. /**                                              **/
726. /*************************************************/
727.
728. {
729. tx_time_cnt =200;
730. rx_time_cnt =2000;
731. initpack ();
732. clms ();
733. init_ih ();                          /* Set up interrupt handler */
734. config_510 ();                       /* Configure 82510 */
735. set_bank (01);                       /* Switch to Bank one for operation */
736.
737. }
738. initpack ()
739. /*************************************************/
740. /**                                              **/
741. /**      Intializes Tx Buffer to NULs            **/
742. /**                                              **/
743. /*************************************************/
744.
745. {
746. int      i;
747.
748. txpack.head = SOH;
749. rxpack.head =SOH;
750. txpack.pack_num =0;
751. rxpack.pack_num =0;
752. txpack.pack_cmpl = 0;
753. rxpack.pack_cmpl = 0;
754. for (i=0; i (129; i++)
755.     {
756.         rxpack.buffer[i] =NUL;
757.         txpack.buffer[i] =NUL;
758.     }
759. txpack.chksm =0;
760. rxpack.chksm =0;
761. }
762.
763. enbint4 ()
764. /*************************************************/
765. /**                                              **/
766. /**      Enables INT4  in the 8259A              **/
767. /**                                              **/
768. /*************************************************/
769.
770. {
771. int      int_enb = 0xEF;
772. int      v;
773.
774. v=inp (ip01);                 /* 11101111 */
775. int_enb = (v & int_enb);
776. outp  (ip01,int_enb);
777. }
```

231928-51

82510 XMODEM Implementation (Continued)

```
PAGE 15       MAIN PROGRAM   ftp.c  82510 XMODEM


mskint4 ()
/*************************************************/
/**                                             **/
/**     Masks INT4 in the 8259A                 **/
/**                                             **/
/*************************************************/
{
int     int_dis = 0xEF;
int     v;

v=inp (ip01);               /* 00010000 */
int_dis = (v ! 0x10);
outp  (ip01,int_dis);
}

config_510 ()
/*************************************************/
/**                                             **/
/**     Configure the 82510                     **/
/**                                             **/
/*************************************************/
{
int val;

set_bank (02);              /*******************************************/
val = 0x00;                 /* IMD - Rx FIFO depth =4, auto ack,normal */
outp ((bpa + 4), val);      /* local loopback                          */
val =0x78   ;               /* RMD - ASCII CCR,disable dpll,7/16 sampl */
outp ((bpa +7),val);        /* window,absolute start bit sampling      */
val =0x00       ;           /* TMD - manual  mode, 2 stop bits         */
outp ((bpa +3),val);        /* no 9-bit char, no s/w parity            */
val =0x30;                  /* FMD - Rx fifo Threshold = 3             */
outp ((bpa+1),val);         /* Tx fifo threshold =0                    */
val =0x80;                  /* RIE - Enable rx interrupts              */
outp ((bpa+6),val);         /*                                         */
set_bank (03);              /* MODEM CONFIGURATION                     */
val = 0x50;                 /* CLCF - 16X, BRGA                        */
outp ((bpa),val);           /*                                         */
val =0xd8;                  /* BBL  - for 5ms base                     */
set_dlab (03);              /*                                         */
outp ((bpa), val);          /*                                         */
val =0xb4;                  /*                                         */
outp ((bpa+1),val);         /* BBH - for 5 ms base                     */
reset_dlab (03);            /*                                         */
val = 0x00;                 /* BBCF - sys clk source, timer mode       */
outp ((bpa+3),val);         /*                                         */
val =0x02;                  /* TMIE - Timer B interrupt enable         */
outp ((bpa+6),val);         /*                                         */
set_bank (00);              /* BANK 0 FOR GENERAL CONFIG               */
val =blkenb;                /* GER -  enable timer, rx, CCR            */
outp ((bpa+1),val);         /* block interrupts                        */
val = 0x07;                 /* LCR - disable parity, 8 bit char        */
outp ((bpa+3),val);         /*                                         */
set_dlab (00);              /*                                         */
val = 0xE0;                 /* BRGA divisor =01E0H for 1200            */
outp (bpa,val);             /*******************************************/
val = 0x01;
outp ((bpa+1),val);
reset_dlab (00);
}
```

231928-52

82510 XMODEM Implementation (Continued)

```
PAGE 16        MAIN PROGRAM    ftp.c  82510 XMODEM


838. set_dlab (bank)
839. /*************************************************/
840. /**                                             **/
841. /**      Set DLAB bit to allow access to        **/
842. /**      Divisor Registers                      **/
843. /**                                             **/
844. /*************************************************/
845.
846. int bank;
847. {
848. int      inval;
849. set_bank (00);
850. inval = inp(bpa +3);
851. inval =inval | 0x80;              /* set dlab in LCR*/
852. outp ((bpa+3),inval);
853. set_bank (bank);
854. }
855.
856. reset_dlab (bank)
857. /*************************************************/
858. /**                                             **/
859. /**      Reset DLAB bit of LCR                  **/
860. /**                                             **/
861. /*************************************************/
862.
863. int bank;
864. {
865. int      inval;
866. set_bank (00);
867. inval = inp(bpa +3);
868. inval = (inval & 0x7f);          /* dlab = 0 in LCR*/
869. outp ((bpa+3),inval);
870. set_bank (bank);
871. }
```

231928-53

82510 XMODEM Implementation (Continued)

```
PAGE 17       MAIN PROGRAM   ftp.c  82510 XMODEM


872. /*********************************************************/
873. /****                                                 *****/
874. /****    82510 interrupt service routine              *****/
875. /****                                                 *****/
876. /****    82510 Interrupt sources:                     *****/
877. /****                          TxM    TX FIFO         *****/
878. /****                          CCR    RX FIFO         *****/
879. /****                                 TIMER B         *****/
880. /****                                                 *****/
881. /****   Identifies and services the 82510 interrupt   *****/
882. /****   source requesting service.                    *****/
883. /****                                                 *****/
884. /****                                                 *****/
885. /*********************************************************/
886.
887. isr_510 ()
888. {
889. int     source ;
890. int     cmd_b;
891. int     st_b;
892. int     i;
893. int     ctlc;
894. int     flgs;
895. int     girval;                          /* Stores Temp. Value of GIR */
896. int     rxflvl;
897. int     tx_char;
898.
899. girval =inp (bpa+2);                     /* Save Bank register in temp.
900.                                             location */
901. outp ((bpa+2),0x20)
902. source = getsrc ();                      /* Get Vector From GIR 123 */
903. intvec =source;
904. switch (source)  {                       /* Service the Source */
905.
906.     case timer :
907. /*********************************************************/
908. /****                                                 *****/
909. /****   TIMER SERVICE ROUTINE                         *****/
910. /****                    decrements tx counter        *****/
911. /****                    decrements rx counter        *****/
912. /****                                                 *****/
913. /*********************************************************/
914.
915.         st_b = inp (bpa+3);              /* Decrement Transmit Counter */
916.         if (tx_time_cnt )0)
917.             tx_time_cnt =tx_time_cnt - 1;
918.         if (rx_time_cnt )0)              /* Decrement Receive Counter */
919.             rx_time_cnt =rx_time_cnt - 1;
920.         cmd_b = 0x22;
921.         outp ( (bpa+3),cmd_b );          /* restart timer */
922.         outp ((bpa+7),0x08);             /* manual ack */
923.         break;
```

231928-54

82510 XMODEM Implementation (Continued)

```
PAGE 18       MAIN PROGRAM   ftp.c  82510 XMODEM


924.    case txm    :
925.    case txf    :
926.    /*********************************************************/
927.    /****                                                 *****/
928.    /****  TRANSMITTER SERVICE ROUTINE                    *****/
929.    /****                                                 *****/
930.    /****              transmits Four characters          *****/
931.    /****              and resets tx_req flag when        *****/
932.    /****              whole packet transmitted           *****/
933.    /****                                                 *****/
934.    /*********************************************************/
935.
936.            if (tx_req )0)                  /* If data to send */
937.            {
938.                if (tx_req == pkt)          /* request to send Packet */
939.                {
940.                    for (i =0 ; i(4 ;i++)
941.                    {
942.                        tx_char = txdata [i + tx_indx];
943.                        outp (bpa,tx_char);
944.                    }
945.
946.                    tx_indx +=4;
947.                    tx_byte_cnt +=4;
948.
949.                    if (tx_indx )=132)      /* if 132 char. sent then */
950.                       tx_req = 0;          /* reset Tx request */
951.
952.                }
953.                else
954.                {
955.                    if (tx_req ==ctl_chr)   /* if ctl char. transmission
956.                                               requested , then transmit the
957.                                               char. in ccr_to_tx */
958.                        outp (bpa, ccr_to_tx);
959.                    tx_req =0;
960.                }
961.            }
962.            else
963.                {                           /* if no data to transmit */
964.                                            /* then disable tx interrupts */
965.                    set_bank (00);
966.                    outp ((bpa+1), (inp(bpa) &txidb));
967.                    set_bank (01);
968.                }
969.        outp ((bpa+7),0x08);                /* issue manual acknowledge */
970.    break;
971.
```

231928-55

82510 XMODEM Implementation (Continued)

```
PAGE 19       MAIN PROGRAM   ftp.c  82510 XMODEM


972.    case ccr    :
973.     /*********************************************************/
974.     /****                                                 *****/
975.     /****    Control Character Service Routine            *****/
976.     /****                                                 *****/
977.     /****               if control char = NAK or ACK      *****/
978.     /****                  inform    transmitter          *****/
979.     /****               if SOH  or EOT                    *****/
980.     /****                  inform receiver                *****/
981.     /*********************************************************/
982.
983.               ++ccrcnt;
984.               flgs =inp (bpa +5);          /* read RST register to service
985.                                               RxM interrupt */
986.               flgs =inp (bpa+1);
987.               ctlc =inp (bpa);
988.               if ((flgs & 0xFF) ==0x48)  /* if no errors and ctl. char */
989.               {                          /* then process control char. */
990.                                          /* and send to tx or rx state */
991.               switch (ctlc)
992.                {
993.                   case NAK:
994.                   case ACK:
995.                       if (get_ccr_rq == active)
996.                           {                  /* inform transmitter that
997.                                                 ctl. char. received */
998.                               get_ccr_rq =inactive;
999.                               ccr_to_get =ctlc;
1000.                           }
1001.
1002.                    break;
1003.
1004.                    case SOH:
1005.                    case EOT:
1006.                        if (ctlc ==SOH)     /* if SOH disable CCR int. */
1007.                        {
1008.                          set_bank (00);
1009.                          outp ((bpa+1),(inp(bpa+1)& ccidb));
1010.                          set_bank (01);
1011.                        }
1012.                        if (rx_state == rx_rdy)  /* if receiver waiting for
1013.                                                    SOH and ready to rcv
1014.                                                    then inform receiver of
1015.                                                    a valid ctl. char. */
1016.                         {
1017.                            ctl_rxd_flg =active;
1018.                            rx_ctl_chr =ctlc;
1019.                         }
1020.
1021.                    break;
1022.
1023.
1024.              }
1025.              }
1026.              outp ((bpa+7),0x08);             /* issue manual ack. */
1027.    break;
```

231928-56

82510 XMODEM Implementation (Continued)

```
PAGE 20       MAIN PROGRAM   ftp.c  82510 XMODEM


1028.     case rxf    :
1029.     /*********************************************************/
1030.     /****                                                 *****/
1031.     /****  Rx FIFO SERVICE ROUTINE                        *****/
1032.     /****                                                 *****/
1033.     /****            Reads four bytes                     *****/
1034.     /****            Byte Count indicates packet rcvd.    *****/
1035.     /****                                                 *****/
1036.     /****                                                 *****/
1037.     /*********************************************************/
1038.
1039.
1040.         /* RXF not checked for errors, since checksum is already used*/
1041.
1042.                 rx_time_cnt =200*5;     /* reset Rx Timer  to indicate
1043.                                            char. received before time out */
1044.
1045.                 rxflvl = ((inp ( bpa +4) & 0x70)/0x10);
1046.                 while ( rxflvl != 0)    /* Check Rx FIFO level and read
1047.                                            data if FIFO not empty */
1048.                 {
1049.                     rxdata [rx_byte_cnt] = inp (bpa);
1050.                     ++ rx_byte_cnt;
1051.                     ++ rxfcnt;
1052.                     -- rxflvl;
1053.                 }
1054.                 outp ((bpa+7),0x08);    /* issue manual acknowledge */
1055.     break;
1056.     default    :
1057.                                         /* if invalid source then issue a
1058.                                            manual acknowledge */
1059.                 outp ((bpa+7),0x08);
1060.     break;
1061.     }
1062. outp ((bpa+1),girval);                 /* Restore Original value of Bank
1063.                                            register to return the 82510 to
1064.                                            original Bank  */
1065.
1066. outp (ip00,eoi);                       /* issue end of int. to 8259*/
1067. }
1068.
1069.
1070. Set_bank (bank_num)
1071. int     bank_num;
1072. *************************************************************/
1073. /****    PROCEDURE SET_BANK                               *****/
1074. /****    switches 82510 register bank to                  *****/
1075. /****    given value.                                     *****/
1076. /*************************************************************/
1077. {
1078. int     port;
1079. int     bank_reg_val;
1080.
1081. bank_reg_val    =bank_num * 0x20;
1082. port = gir_addr +bpa;
1083. outp (port, bank_reg_val);          /* output value to bank register */
1084. }
1085.
```

231928-57

82510 XMODEM Implementation (Continued)

```
PAGE 21       MAIN PROGRAM   ftp.c  82510 XMODEM


1086. getsrc ()
1087. /*************************************************/
1088. /**      read GIR and returns the                **/
1089. /**      source Vector                           **/
1090. /**                                              **/
1091. /**      Timer       - 05 Hex                    **/
1092. /**      Tx Machine  - 04 Hex                    **/
1093. /**      CCR         - 03 Hex                    **/
1094. /**      Rx FIFO     - 02 Hex                    **/
1095. /**      Tx FIFO     - 01 Hex                    **/
1096. /**                                              **/
1097. /*************************************************/
1098.
1099. {
1100. int      v,src;
1101.
1102. v=inp (bpa +2);                      /* read GIR */
1103. src = v & 0x0E;                      /* Mask out all bits except for
1104.                                        bits 1,2 and 3 */
1105. src = src/2;
1106. return(src);
1107. }
1108.
1109. process_cmd ()
1110. /*********************************************************/
1111. /****                                                 *****/
1112. /**** PROCESS COMMAND                                 *****/
1113. /****                Processes User commands          *****/
1114. /****                1 - Transmit                     *****/
1115. /****                2 - Receive                      *****/
1116. /****                * - Reset 82510                  *****/
1117. /****                0 - quit                         *****/
1118. /****                r - Reinitialize 82510           *****/
1119. /****                ! - system monitor               *****/
1120. /****                                                 *****/
1121. /****                                                 *****/
1122. /*********************************************************/
1123.
1124.
1125. {
1126. int      r;
1127. int      exflg =false ;
1128. int      excp ;
1129.
1130.      r = getch ();
1131.      switch (r) {
1132.
1133.          case '0' :
1134.                   exflg = true;              /* exit */
1135.                   break ;
```

231928-58

82510 XMODEM Implementation (Continued)

```
PAGE 22       MAIN PROGRAM   ftp.c  82510 XMODEM


1136.             case '1' :
1137.                 if (tx_state == tx_idle)        /* Transmit Command only
1138.                                                    accepted if idle */
1139.                 {
1140.                     CLMS ();
1141.                     CLL (tx_r,tx_c);
1142.                     MV_CURS (tx_r,tx_c);
1143.                     printf ("file :");          /* Get name of file to Tx */
1144.                     scanf ("%s", &tx_file_name);
1145.                     cll (tx_r,tx_c);
1146.                     open_wind (tx_r,tx_c,"transmitting");
1147.                     open_wind (tx_r,tx_c+14,tx_file_name);
1148.                     tx_cmd = active;            /* Activates flag to signal
1149.                                                    Transmit idle state */
1150.                 }
1151.                 else
1152.                 {
1153.                  beep ();
1154.                  prmsg ("transmission in progress");
1155.                 }
1156.         break;
1157.         case '2' :
1158.                 CLMS ();
1159.                 CLL (rx_r,rx_c);
1160.                 MV_CURS (rx_r,rx_c);
1161.                 printf ("file :");              /* Get rx file name */
1162.                 scanf ("%s", &rx_file_name);
1163.                 cll (rx_r,rx_c);
1164.                 open_wind (rx_r,rx_c,"enabled");
1165.                 open_wind (rx_r,rx_c+14,rx_file_name);
1166.                 rx_cmd =active;                 /* Activate flag to signal
1167.                                                    rx state machine */
1168.                 break;
1169.         case '*' :
1170.                 rst510 ();                      /* reset 82510 */
1171.                 open_wind (24,30,"device reset");
1172.                 break;
1173.         case 'r' :
1174.                 rst510 ();
1175.                 init ();                        /* reinitialize 82510 */
1176.                 enbint4 ();
1177.                 beep ();
1178.                 prmsg (" 82510 reinitialized");
1179.                 break;
1180.         case '!' :
1181.              excp = system ("d:\micom");
1182.         default:
1183.             BEEP ();
1184.             prmsg ("incorrect command, reenter");
1185.             break;
1186.     }
1187.     if (exflg == true)                          /* if exit command issued,
1188.                                                    then quit program */
1189.         return (true);
1190.     else  return (false);
1191. }  /* end of command processing */
```

231928-59

82510 XMODEM Implementation (Continued)

```
PAGE 23      MAIN PROGRAM   ftp.c  82510 XMODEM


1192. asmbpkt (pkts_sent,fp)
1193. /*****************************************************/
1194. /**     Reads file to be transmitted and puts       **/
1195. /**     the data into the proper xmodem packet format **/
1196. /*****************************************************/
1197.
1198. int     pkts_sent;                          /* this value is used to
1199.                                                get the next pkt # */
1200. FILE    *fp;
1201. {
1202. int     sum =0;
1203. int     i,blkcnt;
1204. int     st,ft;
1205. char    cpkt,cpkcmp;
1206.
1207.  blkcnt =fread (&txpack.buffer[0],128,1, fp);   /* read 128 bytes */
1208.  if (blkcnt (1)
1209.    {
1210.      if ((st=feof(fp)) )0 && !(ft=ferror(fp)))
1211.       {
1212.         eof = true;                           /* if end of file then
1213.                                                  signal EOF */
1214.         beep ();
1215.         prmsg ("EOF !!!!!!!!!! ");
1216.       }
1217.          else
1218.         if (ft )0)
1219.            {
1220.             beep ();
1221.             prmsg ("READ ERROR !!!!!!!!!!!!");
1222.             tx_state=tx_idle;
1223.            }
1224.    }
1225.
1226.  cpkt =pkts_sent +1;
1227.  txpack.pack_num = cpkt;
1228.  cpkcmp ="txpack.pack_num;
1229.  txpack.pack_cmpl = cpkcmp;                   /* one's complement of
1230.                                                  packet number */
1231.  for (i=0; i (128; i++)
1232.      sum = sum+txpack.buffer[i];
1233.  txpack.chksm = sum % 255;                    /* checksum calculated */
1234.
1235. }
1236.
1237.
1238. cpy2buf ()
1239. /*******************************/
1240. /** copy packet to tx buffer **/
1241. /*******************************/
1242. {
1243. int i;
1244.
1245. txdata [0] =txpack.head;
1246. txdata [1] =txpack.pack_num;
1247. txdata[2] =txpack.pack_cmpl;
1248. for (i=0; i (128; i++)
1249.      txdata [i+3] = txpack.buffer [i];
1250. txdata [131] =txpack.chksm;
1251. }
```

231928-60

82510 XMODEM Implementation (Continued)

```
PAGE 24       MAIN PROGRAM   ftp.c  82510 XMODEM


1252.
1253. check_wait ()
1254. /*****************************************************/
1255. /**** PROCEUDRE CHECK_WAIT                          *****/
1256. /****                                               *****/
1257. /****            checks  Tx Timer, ccr_to_get and   *****/
1258. /****                    get_ccr_req  and returns:  *****/
1259. /****                                               *****/
1260. /****             Time Out - Tx Timer = 0           *****/
1261. /****             rx_ACK   - Ack received           *****/
1262. /****             rx_NAK   - Nak received           *****/
1263. /****             waiting  - tx Timer not expired   *****/
1264. /****                                               *****/
1265. /****                                               *****/
1266. /*****************************************************/
1267. {
1268.
1269.  if ((!tx_time_cnt) && (get_ccr_rq ==active))       /* if tx Timer expired
1270.                                                       and still waiting
1271.                                                       for control char,then
1272.                                                       Time out */
1273.         return (time_out);
1274.  else
1275.     if (get_ccr_rq == inactive)                   /* Ctl-Char rcvd then
1276.                                                       return status */
1277.        {
1278.         switch (ccr_to_get)
1279.         {
1280.          case ACK :
1281.                 return (rx_ACK);
1282.           break;
1283.
1284.          case NAK :
1285.                 return (rx_NAK);
1286.           break ;
1287.
1288.      default:
1289.                 return (rx_gen);                  /* corrupted ctl char */
1290.           break;
1291.         }
1292.        }
1293.     else
1294.         if ((tx_time_cnt )0) && (get_ccr_rq ==active))
1295.             return (waiting);
1296. }
1297.
1298.
```

231928-61

82510 XMODEM Implementation (Continued)

```
PAGE 25       MAIN PROGRAM   ftp.c  82510 XMODEM


1299. abort_tx ()
1300. /*********************************************************/
1301. /****                                                 *****/
1302. /****    Abort transmission, reintialize              *****/
1303. /****                        Transmitter              *****/
1304. /****                        Flags                    *****/
1305. /****                                                 *****/
1306. /*********************************************************/
1307.
1308. {
1309.         eof =false;
1310.         txrflg = mkpkt;
1311.         quit =false;
1312.         key = 0;
1313.         tx_state =tx_idle;
1314.         tx_cmd = inactive;
1315.         send_ccr_req = inactive;
1316.         tx_indx = 0;
1317.         tx_req =inactive;
1318.         ccr_to_tx = 0;
1319.         tx_byte_cnt =0;
1320.         pkts_sent =0;
1321.         tx_time_cnt =0;
1322.         get_ccr_rq =0;
1323.         ccr_to_get  =0;
1324.         set_bank (00);
1325.         outp ((bpa+1),0x27);
1326.         set_bank (001);
1327.         outp((bpa+6),0x0D);
1328.         tx_state =tx_idle;
1329.         prmsg ("transmitter reset");
1330. }
1331.
1332.
1333. wait_rx ()
1334. /*********************************************************/
1335. /****                                                 *****/
1336. /****    WAIT_RX:                                     *****/
1337. /****          checks rx timer, and returns the       *****/
1338. /****          following value :                      *****/
1339. /****               SOH - SOH received                *****/
1340. /****               EOT - EOT received                *****/
1341. /****               time out - rx timer expired       *****/
1342. /****               waiting - waiting for event       *****/
1343. /****                                                 *****/
1344. /*********************************************************/
1345.
1346. {
1347.     if ((ctl_rxd_flg == active) && (rx_time_cnt !=0))
1348.     {
1349.         ctl_rxd_flg =inactive;
1350.         return ( rx_ctl_chr);
1351.     }
1352.     else
1353.         if ( rx_time_cnt == 0)
1354.             return (time_out);
1355.         else
1356.             return (waiting);
1357. }
```

231928-62

82510 XMODEM Implementation (Continued)

```
PAGE 26       MAIN PROGRAM   ftp.c  82510 XMODEM


1358. chkpkt (pknum)
1359. /*******************************************/
1360. /**     verifies the checksum and packet    **/
1361. /**     number of the received packet       **/
1362. /**     returns a status code               **/
1363. /**                                         **/
1364. /**     EOK      - Packet Ok                **/
1365. /**     EPKNUM   - Error in packet number   **/
1366. /**     ECHKSUM  - Error in Check Sum       **/
1367. /**     EPKCMPL  - Erro in packet complement **/
1368. /**                                         **/
1369. /*******************************************/
1370. int pknum;
1371. {
1372. int i;
1373. int ckm;
1374. int sum = 0;
1375. char    cmpl,rxcmpl,cpk,chrckm;
1376.
1377.
1378.     cpk =pknum;
1379.     if (cpk ==rxdata[0])                        /* packet number correct */
1380.      {
1381.       cmpl = rxdata [0];
1382.       rxcmpl =rxdata [1];
1383.       if (rxcmpl == ~cmpl )                     /* packet number complmt */
1384.         {
1385.             for (i=2; i<130; i++)
1386.                 sum = sum +rxdata [i];
1387.             ckm = sum % 255;
1388.             chrckm = ckm;
1389.             pk_chksm =chrckm;
1390.             if (chrckm == rxdata [130])         /* checksum correct */
1391.                 return (eok);
1392.             else
1393.                 return (echksm);
1394.
1395.          }
1396.          else
1397.             return (epkcmp);
1398.      }
1399.          else
1400.         {
1401.          if ((rxdata [0] == cpk -1) && (cpk )1))    /* old packet number
1402.                                                        received */
1403.             return (eold);
1404.          else
1405.             return (epknum);
1406.          }
1407. }
```

231928-63

82510 XMODEM Implementation (Continued)

```
PAGE 27      MAIN PROGRAM   ftp.c  82510 XMODEM


1408. /*************************************************************/
1409. /****        tx_i_dis    PROCEDURE                          *****/
1410. /****   Disables txm and Tx FIFO interrupts                 *****/
1411. /****   from GER register                                   *****/
1412. /*************************************************************/
1413. tx_i_dis ()
1414. {
1415. mskint4 ();                                    /* disable interrupts */
1416. set_bank (00);                                 /* switch to bank zero */
1417. outp ((bpa+1),(inp(bpa+1) & txidb));           /* mask out interrupts in
1418.                                                    GER - TxM and Tx FIFO */
1419. set_bank (01);                                 /* set bank one */
1420. enbint4 ();                                    /* enable interrupts */
1421.
1422.
1423.
1424.
1425.
1426. }/*************************************************************/
1427. /****           tx_i_enb    PROCEDURE                        *****/
1428. /****   enables the TXM and TX FIFO Interrupts              *****/
1429. /****   from GER register                                   *****/
1430. /*************************************************************/
1431.
1432. tx_i_enb ()
1433. {
1434. mskint4 ();                                  /* disable interrupts */
1435. set_bank (00);                               /* switch to bank zero */
1436. outp ((bpa+1),(inp(bpa+1) | txien));         /* enable TXM AND TX FIFO */
1437. set_bank (01);                               /* return to bank one */
1438. enbint4 ();                                  /* enable interrupts */
1439.
1440. }
1441.
1442. sh_pkt_param ()
1443. /*************************************************/
1444. /** Displays the parametrs of the Received     **/
1445. /** packet number, and the expected parameters **/
1446. /*************************************************/
1447. {
1448. ++ bad_pkt_cnt;
1449. prmsg ("sending NAK ");
1450. printf (" error = %5u",pkst);
1451. mv_curs (13,1);
1452. printf ("exptd pkt # = %3u",exp_pkt_num);
1453. mv_curs (14,1);
1454. printf ("rxd pkt # = %3u", rxdata[0]);
1455. mv_curs (13,40);
1456. printf ("expd pkt cmpl = %#X", (~rxdata[0]));
1457. mv_curs (14,40);
1458. printf ("rxd pkt complement = %#X", rxdata[1]);
1459. mv_curs (15,1);
1460. printf ("rxd chksum = %#X", rxdata[130]);
1461. mv_curs (15,40);
1462. printf ("expd chksum = %#X",pk_chksm);
1463. send_ccr_req =active;
1464. ccr_to_tx = NAK;
1465. }
```

231928-64

82510 XMODEM Implementation (Continued)

```
PAGE 28        MAIN PROGRAM    ftp.c  82510 XMODEM


1466. /*************************************/
1467. /** PROCEDURE BUF_CPY                **/
1468. /**      copies packet to ram buffer **/
1469. /*************************************/
1470. buf_cpy (packt_id)
1471. int packt_id;
1472. {
1473. int      i;
1474. int      indx =0;
1475.
1476. indx = (packt_id-1) *128;
1477. if (indx < (32000 - 129))  /* No overwrite of buffer */
1478. {
1479.      for (i=0; i<128; i++)
1480.           rx_f_buf [indx+i] = rxbuf [i];
1481. }
1482. else
1483.      prmsg ("file too big, cannot save in memory");
1484. }
```

231928-65

82510 XMODEM Implementation (Continued)

```
PAGE  1        DEFINITION FILE   ftp.def  82510 XMODEM


  1. #define s1  "82510 FTP x000 6/30/86"          /* sign on message */
  2. #define bpa 0x3f8                             /* Base address */
  3. #define gir_addr    02
  4. #define esci 27                               /* escape char. in hex */
  5. #define bel  07
  6. #define msg_c 17                              /* coordinates of the
  7.                                                  message line */
  8. #define msg_r 2
  9. #define tx_c  35                              /*
 10. #define tx_r  10
 11. #define rx_c  35                                  coordinates
 12. #define rx_r  12
 13. #define so_c  50
 14. #define so_r  24                              */
 15. #define false  0
 16. #define true   1
 17. #define active 1
 18. #define inactive 0
 19. #define ctl_chr 2                             /* control char transmit */
 20. #define pkt 1                                 /* send packet */
 21. #define eok 5555                              /* packet received ok */
 22. #define echksm 5500                           /* checksum error */
 23. #define epkcmp  5501                          /* packet compl incorrect */
 24. #define eold    5502                          /* old pack num received */
 25. #define epknum 5503                           /* invalid packet # rcvd. */
 26.
 27. /*************************/
 28. /*** tx state defintions***/
 29. /*************************/
 30.
 31. #define tx_idle     000
 32. #define wait_NAK    001
 33. #define TO_err_60   002
 34. #define tx_rdy      003
 35. #define tx_packet   004
 36. #define wait_CC     005
 37. #define tx_pk_comp  006
 38. #define to_err      007
 39. #define txen        0x02
 40. #define mkpkt       111                       /* Transmit packet stages */
 41. #define txmtg       112
 42. #define retx        113
 43. #define waiting     114
 44.
 45. /*************************/
 46. /**  rx state definition **/
 47. /*************************/
 48.
 49. #define rx_idle     000
 50. #define rx_rdy      001
 51. #define rx_pkt      002
 52.
 53.
 54. /*************************/
 55.
 56. #define     time_out    90                    /* rx state signal values */
 57. #define     rx_NAK      91
 58. #define     rx_ACK      92
 59. #define     rx_gen      93
 60.
```

231928-66

82510 XMODEM Implementation (Continued)

```
PAGE  2        DEFINITION FILE   ftp.def  82510 XMODEM


 61.
 62. /*********************************/
 63. /**      Protocol Control       ****/
 64. /**      characters             ****/
 65. /*********************************/
 66.
 67.
 68. #define NAK      0x14                    /* Negative Ack */
 69. #define ACK      0x06                    /* Positive Ack */
 70. #define SOH      0x01                    /* Start of Header */
 71. #define EOT      0x04                    /* End of Text */
 72. #define CAN      0x18
 73. #define NUL      0x00
 74.
 75. /*********************************/
 76. /**      interrupt source       ****/
 77. /*********************************/
 78.
 79. #define timer    05                      /* 82510 int. vectors */
 80. #define txm      04
 81. #define ccr      03
 82. #define rxf      02
 83. #define txf      01
 84. #define txien    0x12                    /* unmask TxM and Tx FIFO */
 85. #define txidb    0x2D                    /* mask TxM and Tx FIFO */
 86. #define ccien    0x04                    /* enable CCR interrupts */
 87. #define ccidb    0x33                    /* mask CCR int */
 88. #define blkenb   0x25                    /* enable, block interrupts
                                                 through GER for 82510 */
 89.
 90.
 91. /*********************************/
 92. /**      8259A values *********/
 93. /*********************************/
 94.
 95. #define eoi      0x20                    /* end of interrupt */
 96. #define ip00     0x20                    /* 8259A port 0 */
 97. #define ip01     0x21                    /* 8259A port 1 */
```

231928-67

82510 XMODEM Implementation (Continued)

```
PAGE  1        CRT I/O ROUTINES     cio.c   82510 XMODEM


  1. #include "ftp.def"
  2.
  3. CLR()
  4. /*************************************************************/
  5. /****                                                     *****/
  6. /****     PROCEDURE CLR                                   *****/
  7. /****                                                     *****/
  8. /****                  clears screen                      *****/
  9. /****                                                     *****/
 10. /****                                                     *****/
 11. /****                                                     *****/
 12. /*************************************************************/
 13.
 14. {
 15. int escchr = esci;
 16.
 17.     putch (escchr);
 18.     printf ("[2J");
 19.
 20. }
 21.
 22.
 23. VOFF ()
 24. /*************************************************************/
 25. /****                                                     *****/
 26. /****        PROCEDURE VOFF                               *****/
 27. /****                                                     *****/
 28. /****                 Turns Reverse Video OFF             *****/
 29. /****                                                     *****/
 30. /****                                                     *****/
 31. /****                                                     *****/
 32. /*************************************************************/
 33.
 34. {
 35. int escchr = esci;
 36.
 37.     putch (escchr);
 38.     printf ("[0m");
 39.
 40. }
 41.
 42.
 43. RVON ()
 44. /*************************************************************/
 45. /****                                                     *****/
 46. /****    PROCEDURE RVON                                   *****/
 47. /****                                                     *****/
 48. /****              Reverse Video ON                       *****/
 49. /****                                                     *****/
 50. /****                                                     *****/
 51. /****                                                     *****/
 52. /*************************************************************/
 53.
 54. {
 55. int escchr = esci;
 56.
 57.     putch (escchr);
 58.     printf ("[7m");
 59. }
 60.
```

231928-68

82510 XMODEM Implementation (Continued)

```
PAGE  2        CRT I/O ROUTINES    cio.c   82510 XMODEM

 61. OPEN_WIND (row,col,stg)
 62. int      row;
 63. int      col;
 64. char     stg[];
 65. /*********************************************************/
 66. /****                                                 *****/
 67. /****   PROCEDURE OPEN_WIND                           *****/
 68. /****                                                 *****/
 69. /****             prints a string in reverse video    *****/
 70. /****             at the given location               *****/
 71. /****                                                 *****/
 72. /****                                                 *****/
 73. /*********************************************************/
 74.
 75. {
 76.
 77.     MV_CURS (row, col);
 78.     RVON ();
 79.     printf ("%s",stg);
 80.     VOFF();
 81.
 82. }
 83. BEEP ()
 84. /*********************************************************/
 85. /****                                                 *****/
 86. /****  PROCEDURE BEEP                                 *****/
 87. /****                                                 *****/
 88. /****              produces a beep                    *****/
 89. /****                                                 *****/
 90. /****                                                 *****/
 91. /****                                                 *****/
 92. /*********************************************************/
 93.
 94. {
 95. int belchr = bel;
 96.
 97.     putch (belchr);
 98.
 99. }
100.
101. CLL(row,col)
102. int row;
103. int col;
104. /*********************************************************/
105. /****                                                 *****/
106. /****   PROCEDURE CLL                                 *****/
107. /****                                                 *****/
108. /****              clear line at given coordinate     *****/
109. /****                                                 *****/
110. /****                                                 *****/
111. /****                                                 *****/
112. /*********************************************************/
113.
114. {
115. int escchr = esci;
116.     MV_CURS (row, col);
117.     putch (escchr);
118.     printf ("[K");
119. }
120.
```

231928-69

82510 XMODEM Implementation (Continued)

```
PAGE  3        CRT I/O ROUTINES    cio.c   82510 XMODEM


121.
122.
123. CLMS()
124. /*********************************************************/
125. /****                                                 *****/
126. /*****       PROCEDURE CLMS                           *****/
127. /****                                                 *****/
128. /****               clear message line                *****/
129. /****                                                 *****/
130. /****                                                 *****/
131. /****                                                 *****/
132. /*********************************************************/
133.
134. {
135.     CLL (msg_r,msg_c);
136. }
137.
138. prmsg (msg)
139. char    msg [];
140. /*********************************************************/
141. /****                                                 *****/
142. /*****   PRINTS MESSAGE AT MESSAGE LINE               *****/
143. /****                                                 *****/
144. /****                                                 *****/
145. /****                                                 *****/
146. /****                                                 *****/
147. /****                                                 *****/
148. /*********************************************************/
149.
150. {
151.     clms ();
152.     printf (" %s", msg);
153. }
154.
155. CLLC ()
156. {
157. int escchr = esci;
158.     putch (escchr);
159.     printf ("[K");
160. }
161.
162. MV_CURS (x,y)
163. /*********************************************************/
164. /****                                                 *****/
165. /****  PROCEDURE MV_CURS                              *****/
166. /****                                                 *****/
167. /****               moves cursor to specified         *****/
168. /****               location.                         *****/
169. /****                                                 *****/
170. /*********************************************************/
171.
172. int x;
173. int y;
174. {
175. int escchr = esci;
176.
177.     putch (escchr);
178.     cprintf ("[%u;%uH",x,y);
179. }
```

231928-70

82510 XMODEM Implementation (Continued)

```
PAGE  1      ASM 86 INTERRUPT INIT. ih1.asm 82510 XMODEM


  1. NAME         ftpih
  2.
  3. DGROUP  GROUP   DATA
  4. DATA    SEGMENT WORD PUBLIC 'DATA'
  5.      ASSUME  DS:DGROUP
  6. DATA    ENDS
  7.
  8. EXTRN        isr_510:far
  9.
 10. _PROG   SEGMENT BYTE PUBLIC 'PROG'
 11.      ASSUME  CS:_PROG
 12.
 13. PUBLIC       init_ih
 14. PUBLIC       ih510
 15.
 16. init_ih      PROC    far
 17.      push    BP
 18.      push    DX
 19.      push    AX
 20.      push    DS
 21.      mov     DX, OFFSET ih510
 22.      push    CS
 23.      pop     DS
 24.      mov     AH,25H          ;DOS vector setup call
 25.      mov     AL,0CH          ;COM1  vector
 26.      INT     21H             ;DOS system call
 27.      pop     DS
 28.      pop     AX
 29.      pop     DX
 30.      pop     BP
 31.      ret
 32. init_ih      ENDP
 33.
 34. ih510        PROC    far
 35.      push    BP
 36.      push    AX
 37.      push    BX
 38.      push    CX
 39.      push    DX
 40.      push    SI
 41.      push    DI
 42.      push    DS
 43.      push    ES
 44.      mov     AX, DGROUP
 45.      mov     DS, AX
 46.      call    isr_510
 47.      pop     ES
 48.      pop     DS
 49.      pop     DI
 50.      pop     SI
 51.      pop     DX
 52.      pop     CX
 53.      pop     BX
 54.      pop     AX
 55.      pop     BP
 56.      iret
 57. ih510        ENDP
 58.
 59. _PROG   ENDS
 60. end
```

231928-71

82510 XMODEM Implementation (Continued)

APPLICATION NOTE

AP-310

June 1987

High Performance Driver for 82510

DAN GAVISH and TSVIKA KURTS
SYSTEM VALIDATION

Order Number: 292038-001

1.0 OVERVIEW

The 82510 Asynchronous Serial Controller is a CHMOS UART which provides high integration features to offload the host CPU and to reduce the system cost.

This Ap-Note presents a mechanism for reduction and optimization of interrupt handling during asynchronous communication using the 82510. The mechanism is valuable in applications where handling of interrupts degrades system performance i.e., when high baud rate is used, when multiple channels are handled or whenever real-time constraints exist. This implementation of the mechanism is a software driver that transmits or receives characters at 288000 bits per second.

The driver is based on the burst algorithm which uses the 82510 features (FIFOs, Timers, Control Character Recognition etc.) to reduce CPU overhead. CPU is significantly off-loaded for other tasks — about 75% of the usual load is saved.

The driver can be easily modified to run in conjunction with other 82510 features such as the MCS-51 9-bit Protocol.

This document provides a full description of the driver. The burst algorithm is presented in Section 3, the software module flow-charts and their descriptions are presented in Section 6, and the PL/M software listing is given in Appendix A.

2.0 INTRODUCTION

2.1 CPU Load Consideration

The trend towards multi-tasking systems, combined with higher baud rates and increasing the number of channels per CPU, has led to the need for decreasing the CPU bandwidth consumed by the async communications for each byte transfer. Whenever the CPU is interrupted, a certain amount of CPU time is lost in implementing the context switch. This overhead can be as high as hundreds of microseconds per interrupt, depending on the specific operating system parameters. Thus, in high baud-rate or multi-channel environments, where the interrupt frequency is very high, a substantial portion of the CPU time is taken up by this interrupt overhead. Therefore, systems usually require minimization of the number of interrupt events. In the case of an asynchronous communication channel, reduction of the number of interrupts can be achieved by servicing (i.e., transferring to/from the buffer) as many characters as possible whenever the interrupt routine is activated. This can be done by utilizing FIFOs to hold received or transmitted characters, so that the CPU is interrupted only after a certain number of characters have been received or transmitted. Using a receive FIFO may cause a potential problem: Due to the random rate of character arrival in asynchronous communications, there is a chance that characters will be "trapped" in the Rx FIFO for extended periods of time. In order to avoid such situations, a Rx FIFO time-out mechanism can be implemented using the 82510 timer. The time-out indicates that a certain amount of time has elapsed since the last read operation was performed. It causes the CPU to check the Rx FIFO and read any characters that are present. This process, however, introduces the additional overhead of the timer interrupt. This Ap-Note describes the use of the burst algorithm to avoid the timer interrupt overhead while maintaining the use of the Rx FIFO.

2.2 82510 Features Used In This Implementation

The following new 82510 features were used in this implementation:

2.2.1 FIFOs

The 82510 is equipped with 2 four-byte FIFOs, one for reception and one for transmission. While characters are being received, a Rx FIFO interrupt is generated, when the Rx FIFO occupancy increases above a programmable threshold. While characters are being transmitted, a Tx FIFO interrupt is generated, when the Tx FIFO occupancy drops below a programmable threshold. The two thresholds are software programmable, for maximum optimization to the system requirements.

2.2.2 TIMER

The 82510 is equipped with two on chip timers. Each timer can be used as a baud rate generator or as a general purpose timer. When two independent baud rates are required for transmit and receive, the two timers can be used to generate both baud rates internally. Otherwise, one timer can be used for external purposes. The timer is loaded with its initial value by a software command and it counts down using system clock pulses. When it expires, a maskable interrupt is generated.

2.2.3 CONTROL CHARACTER RECOGNITION

Depending on the application, the software usually checks the received characters to determine whether certain control characters have been received, in which case special processing is performed. This loads the CPU, as every received character should be compared to a list of control characters. With the 82510, the CPU is offloaded from this overhead. Every received character is checked by the 82510, and compared to either a standard set of control characters (ASCII or EBCDIC) or to special user defined control characters. The software does not need to check the received characters, and a special interrupt is provided when a received control character is detected by the 82510. The specific operation mode (standard set, user defined, etc.) is programmable.

2.2.4 INTERRUPT CONTROLLING MECHANISM

The twenty possible interrupt sources of the 82510 are grouped into six blocks: Timer, Tx machine, Rx machine, Rx FIFO, Tx FIFO, or Modem. Interrupt source blocks are prioritized. The interrupt management is performed by the 82510 hardware. The CPU is interrupted by a single 82510 interrupt signal. The interrupt handler is reported on the highest priority pending interrupt block (GIR) and on all the pending interrupt blocks (GSR), as well as on the specific interrupt source. Interrupts are maskable at the block level and source level. Interrupts can be automatically acknowledged (become not pending) when serviced by the software, or manually acknowledged by an explicit command.

3.0 THE BURST ALGORITHM

3.1 Background

The 82510 FIFOs are used to reduce the CPU interrupt load. When a burst of characters is transmitted or received, the CPU is interrupted only once per transmission or reception of up to four characters. FIFO thresholds are programmable; thus, when high system interrupt latency is expected, an optimal threshold may be selected for the desired trade-off between the CPU load, and the acceptable system interrupt latency. The required Rx FIFO threshold is also a function of the receive character rate. When the rate is high, a deep FIFO is required. When the rate is very low (e.g., hundreds of milliseconds between characters), a low threshold is needed, to reduce the maximum character service latency (a character is available to the application program only after it is stored in the receive buffer).

The software mechanism described here tunes the Rx FIFO threshold dynamically when the incoming character rate is variable. The algorithm uses one of the 82510 on-chip timers for time measurement, in order to automatically adapt the threshold to the character reception rate. This is done without loading the CPU with the overhead of serving excessive interrupts generated by the timer mechanism itself.

3.2 Burst Algorithm Description

The 82510 timer is initialized to the time-out value with every Rx FIFO interrupt. The time-out value is the maximum acceptable time between a character's reception and its storage in the receive buffer, but not less than five character-times. Upon reception of the next character, the timer status is examined to determine whether the character rate is high (the timer has not yet expired) or low (the timer has expired).

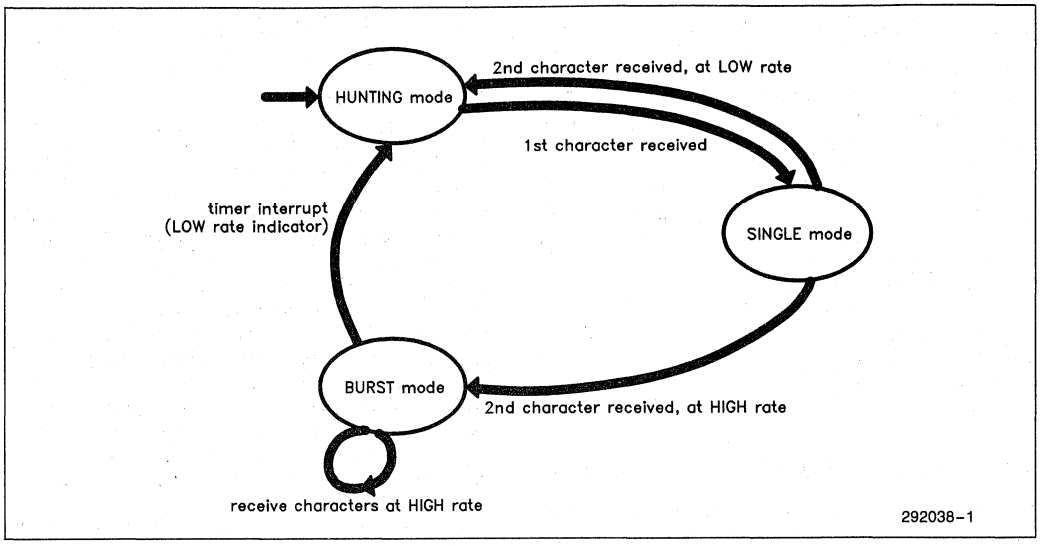


Figure 1. Burst Algorithm State Diagram

The algorithm is best described as a finite state machine that can be in one of three modes: HUNTING mode, SINGLE mode, or BURST mode. In HUNTING mode, after the first character received interrupts the CPU, the mode switches to SINGLE. On receiving a character in SINGLE mode (that is the second character) the timer is examined; if the character rate is very low, the mode is switched back to HUNTING. Otherwise, the rate is high enough to switch to BURST mode. In BURST mode, the Rx FIFO threshold is maximal. The machine remains in BURST mode as long as a burst of characters is being received. When the rate of character reception becomes low, the timer eventually expires generating a timer interrupt which switches the mode back to HUNTING.

Note that while a burst of characters is being received, the CPU is interrupted only once per four received characters. If the characters are received at a very low rate, an interrupt occurs for each received character. The CPU is interrupted by the timer only once, when the burst terminates. See Figure 1 for a state diagram.

For more details about the burst algorithm see paragraph 6.2.

4.0 SOFTWARE MODULE MAP

The driver contains the following software modules:

- MAIN
- BURST ALGORITHM
  - — Burst Algorithm Initialization (*)
  - — Rx FIFO Step (*)
  - — HUNTING mode
  - — SINGLE mode
  - — BURST mode
  - — Timer Step (*)
- INITIALIZATIONS
  - — Wait for Modem Status
- INTERRUPT HANDLER
  - — Rx FIFO Interrupt Service Routine
  - — Tx FIFO Interrupt Service Routine
  - — Status Interrupt Service Routine
  - — Timer Interrupt Service Routine
  - — Modem Interrupt Service Routine

(*) The burst algorithm modules are called by the initialization module and by the interrupt handler modules.

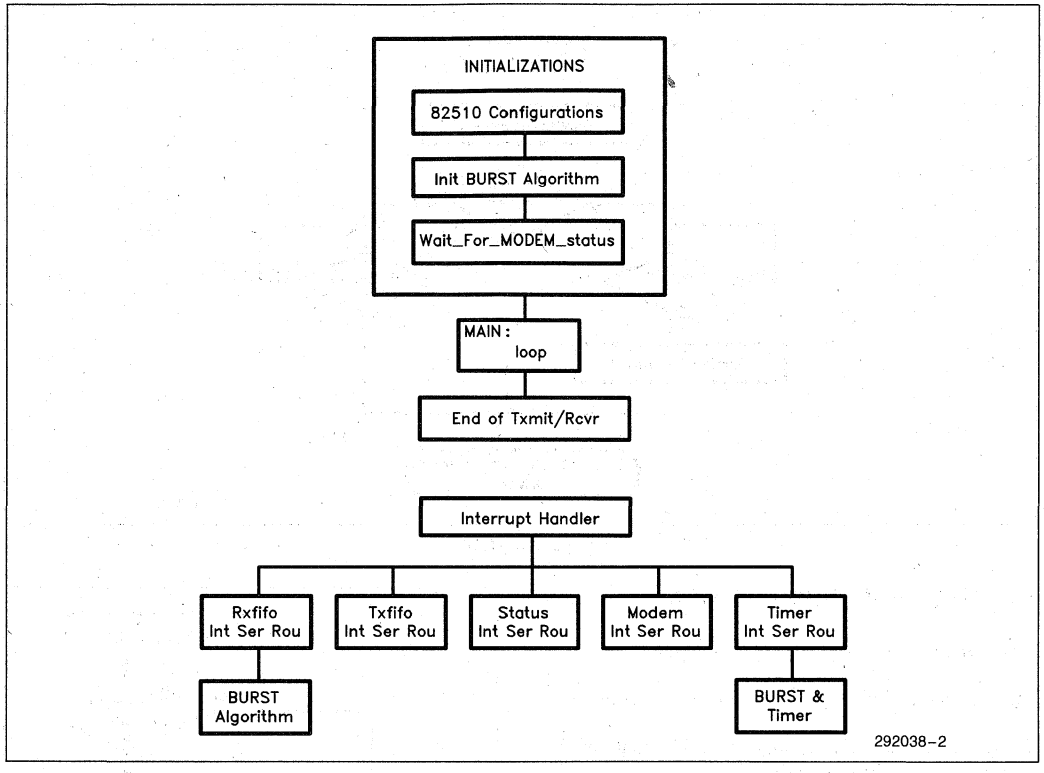


Figure 2. Modules Block Diagram

5.0 HARDWARE VEHICLE DESCRIPTION

The driver was tested at 288000 baud, on an 80186 based system, with an 8 MHz local bus running with 2 wait-states, and an 18.432 MHz 82510 clock. Two stations were involved: one transmitter station and one receiver station. Each station consisted of an iSBC186/51 with a 82510 based SBX board connected to it. See Appendix B for description of the SBX board.

This driver is, nonetheless, suitable for running in a large number of system environments.

6.0 SOFTWARE MODULE DESCRIPTIONS

6.1 MAIN

The MAIN module is a simple example of an application program that uses the driver.

The communication is done between two stations: One station is the transmitter and the other one is the receiver. After interrupts are enabled, the program waits for the Finish_Tx flag or the Finish_Rx flag (for the transmitter or receiver station, respectively) to be set. In the transmitter station, the driver is preloaded with the transmit data. In the receiver station, the received data is displayed after data reception is complete.

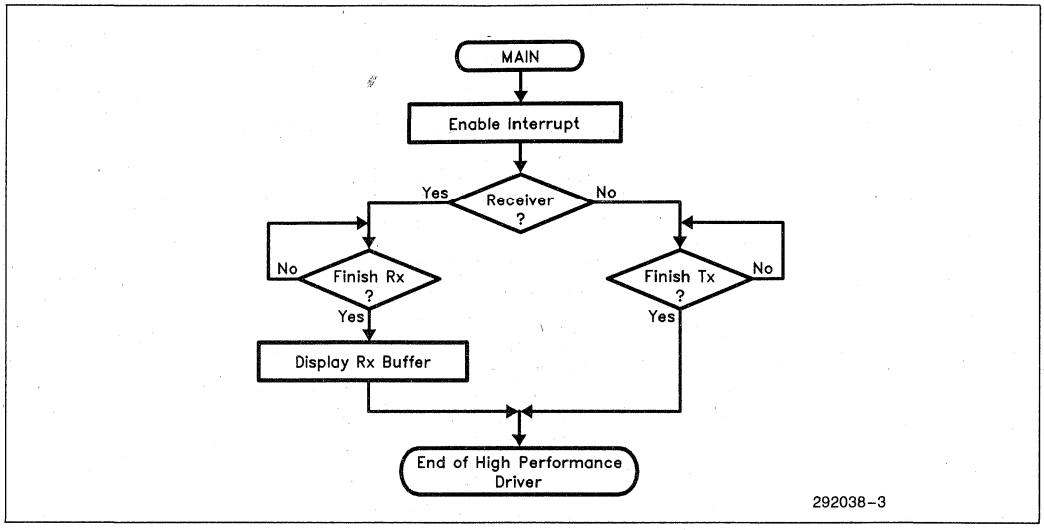


Figure 3. MAIN

6.2 The Burst Algorithm Modules

6.2.1 INITIALIZE THE BURST ALGORITHM

This module is called by the initialization module.

The global variable Burst_algo is used to indicate the current burst algorithm mode.

The burst algorithm is most useful at a baud rate of 9600 or higher. At lower baud rates, where the Rx interrupt rate is very low, the burst algorithm is degenerated (Low_baud is assigned to Burst_algo). At a baud rate of 9600 or more, the burst algorithm mechanism is initialized and starts by disabling the timer interrupt.

The initial state of the burst algorithm is HUNTING mode. In this mode, it is looking for (hunting) the first character. The Rx FIFO threshold is zero, thus the first character received interrupts CPU. This interrupt starts the burst algorithm mechanism.

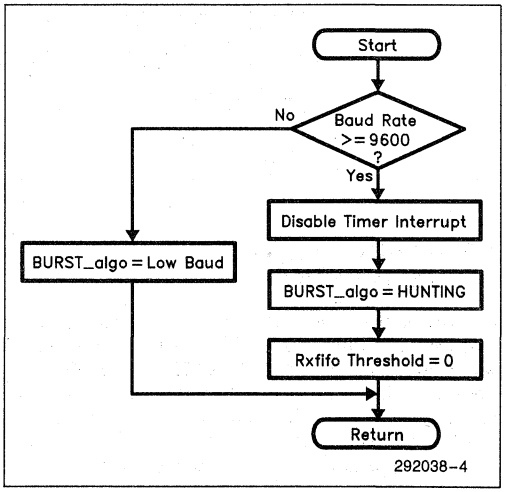


Figure 4. Initialize The Burst Algorithm

6.2.2 BURST ALGORITHM MECHANISM

Modules HUNTING, SINGLE, BURST are called by Rx FIFO interrupt service routine. Module BURST&TIMER is called by timer interrupt service routine.

6.2.2.1 HUNTING Mode

Hunting for the first character received is the first step in the burst algorithm. After the first character is detected, received and handled, it must be determined if reception will be at high or low rate. This is done by starting the timer. HUNTING mode ends by assigning the second step, i.e., SINGLE mode, to Burst_algo.

6.2.2.2 SINGLE Mode

When the second character is received, the burst algorithm is in SINGLE mode. Timer status is read (TMST). If the status indicates that the timer has expired, the receive character rate is low and there is no need to increase the Rx FIFO threshold. The burst algorithm returns to its first state, i.e., HUNTING mode. However, if the timer has not expired, the receive character rate is high, and the Rx FIFO threshold is set to the maximal allowable value. The timer is restarted and the timer interrupt is enabled so that, if it expires before the Rx FIFO exceeds the threshold, a timer interrupt will occur.

SINGLE mode is ended by assigning the third step, BURST mode, to BURST_algo.

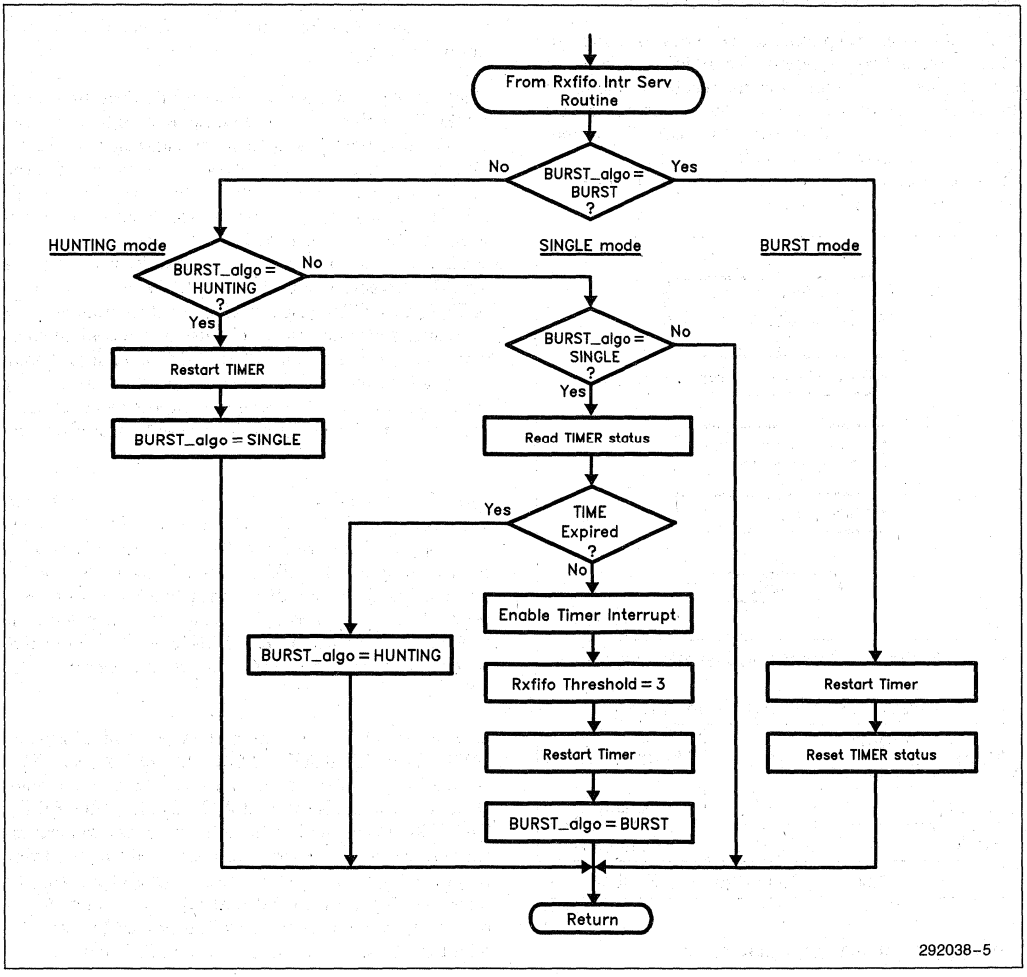


Figure 5. The Burst Algorithm

6.2.2.3 BURST Mode

The algorithm enters BURST mode as soon as the receive character rate is evaluated as high, i.e., when two successive characters are received without a timer expiration. The FIFO is now working at full threshold and the timer is used as a timeout watch dog. BURST mode is the most time-critical path of the algorithm. Therefore, it consumes a minimum amount of real time.

The timer is restarted, in order to restart a new timeout measurement. The timer status is read to trigger automatic reset of the previous status; this is done to avoid the timer interrupt if the timer has expired during the Rx FIFO interrupt service routine execution.

6.2.2.4 Timer Interrupt and Bust Algorithm

If the character reception rate becomes low, then the time between two successive Rx FIFO interrupts increases. Hence, a reduction in the reception rate causes the timeout to expire, and a timer interrupt occurs. This drives the algorithm back to HUNTING mode. The timer interrupt is disabled and the Rx FIFO threshold is configured to zero, to issue an Rx interrupt on the first hunted character.

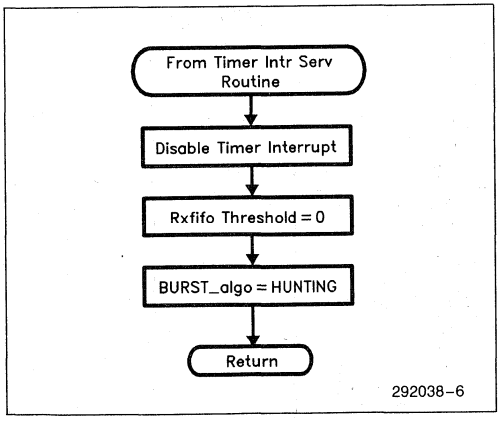


Figure 6. Timer Interrupt and BURST Algorithm

Table 1. BURST Algorithm Modes

Mode	FIFO Threshold	Timer	Timer-Interrupt
Hunting	0	Idle	Disabled
Single	0	Started	Disabled
Burst	Max.	Restarted	Enabled

6.2.3 FLOWCHART DESCRIPTION

The Rx FIFO interrupt handler executes the burst algorithm immediately after the Rx FIFO is emptied (to avoid an overrun error). The module was designed to minimize the CPU overhead inherent in the burst algorithm itself.

BURST mode is assigned the fastest path because it is the most real time sensitive mode.

SINGLE mode has a slightly longer path. However, under a high reception rate, the algorithm passes SINGLE mode once only and then stays in BURST mode until the end of the burst. Under a low reception rate the algorithm passes SINGLE mode many times, but, since the period between two successive Rx interrupts is long, this hardly affects system performance.

6.3 Initializations

This module initializes the driver. It is called at program start-up.

The 82510 is configured for the specific operation mode by the CONFIG_82510 submodule: A Software Reset command is issued, and then the character configuration is selected. In the receiver station ACR0 and ACR1 Registers are loaded with the End-Of-File ASCII character, so that the Control Character Recognition feature of the 82510 can be used to detect the specific file terminator. In the transmitter station, the ASCII characters XOFF and XON are loaded to ACR0 and ACR1, respectively, to detect transmit-off/on requests automatically. The use of the control character recognition feature of the 82510 reduces system overhead, as the software does not need to check every received character. A special interrupt is received when the 82510 hardware detects a received control character.

Interrupt sources are enabled (note that a Tx interrupt will occur immediately). BRGA is loaded to generate the required baud rate (288000 baud in this specific implementation). Rx FIFO depth is set to 4. The Tx and Rx FIFO thresholds are initialized to 0. BRGB is selected to function as a timer, and is loaded with the timeout value (7 ms at 18.432 MHz, in this implementation). The RxC and TxC sources are selected to be BRGA.

The burst algorithm parameters are initialized by INIT_BURST. WAIT_FOR_MODEM_STATUS is called and implements a wait until the modem handshake DSR signal is set. If WAIT_FOR_MODEM_STATUS returns with a timeout error, the modem error is processed. If no error has occurred, the following parameters are initialized: Finish_Rx and Finish_Tx flags, receive and transmit buffer pointers, and the receiver flag. All status registers are cleared by issuing a STATUS CLEAR command to the ICM register.

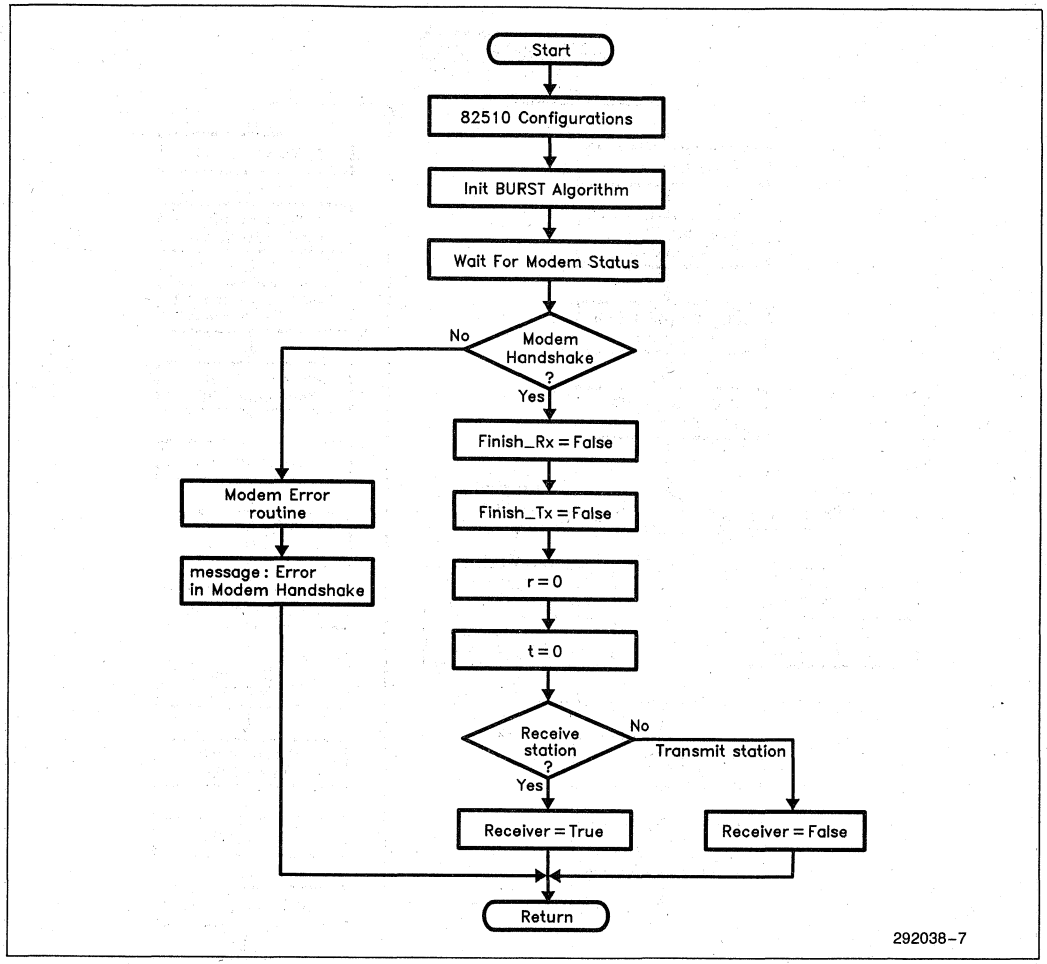


Figure 7. Initializations

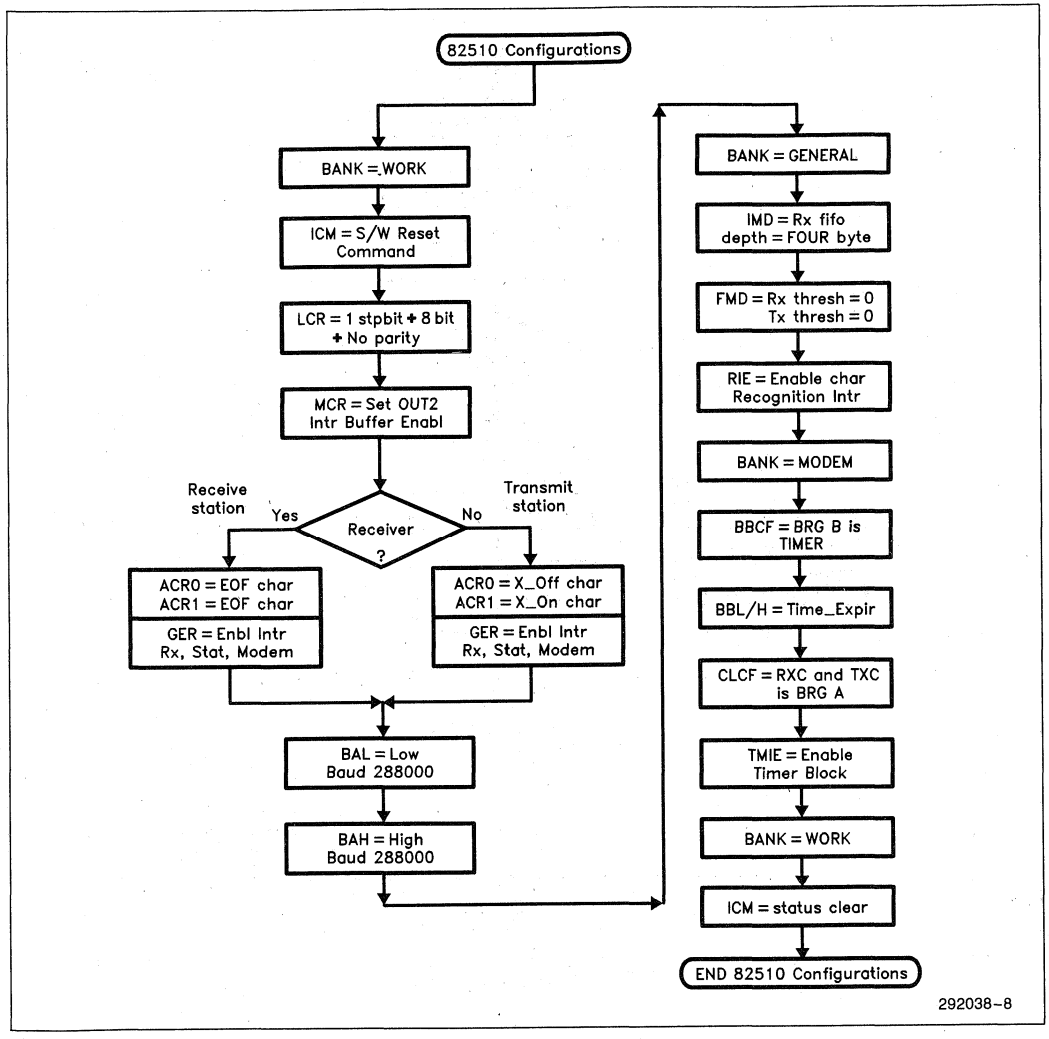


Figure 8. 82510 Configurations

6.3.1 WAIT_FOR_MODEM_STATUS

This module waits, with a timeout, for the DSR modem handshake signal to be set. DSR should be active before any communication starts (it indicates that the modem is active). The returned Modem_Handshake flag indicates normal return (true) or timeout error return (false).

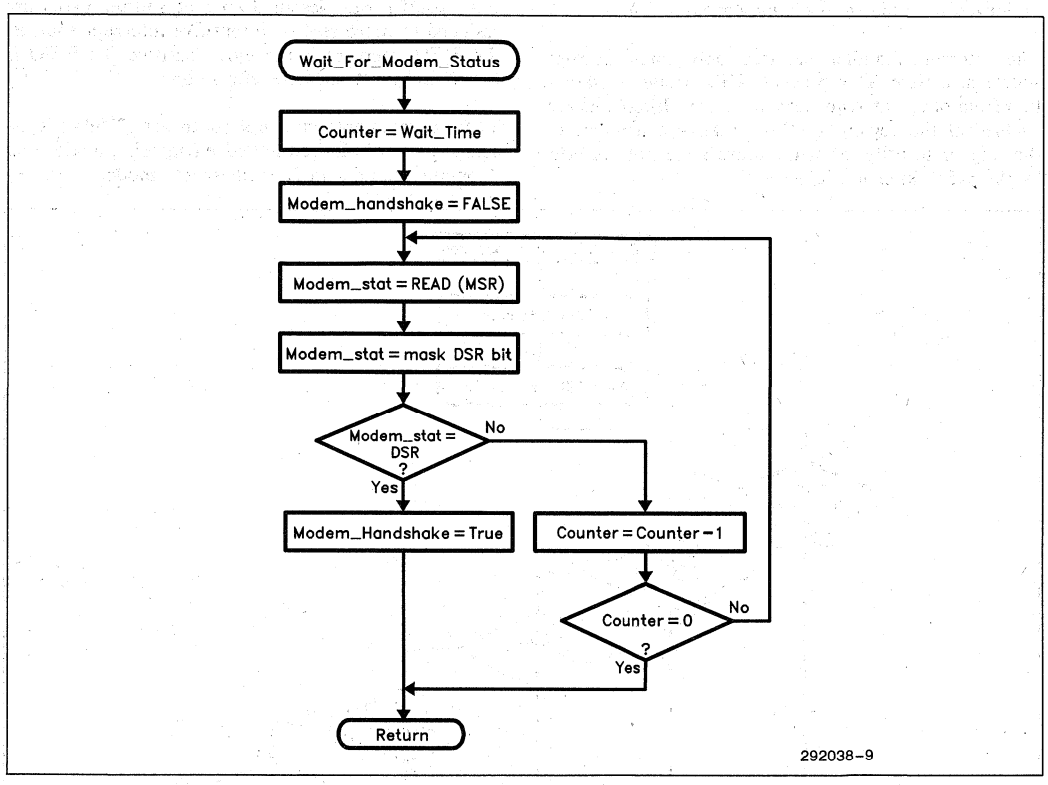


Figure 9. Wait_For_Modem_Status

6.4 Interrupt Handler

The interrupt handler services the 82510 interrupt sources. Since this is a time-critical path, the code is optimized to minimize real time consumption.

The interrupt handler services only one interrupt source at a time. This prevents CPU resource starvation from other interrupt driven devices. Interrupts are enabled at the beginning of the interrupt handler, so that higher priority interrupt sources are not disabled by the 82510 interrupt handler.

6.4.1 INTERRUPT HANDLER STRUCTURE

The interrupt handler identifies the highest priority pending 82510 interrupt, by reading GIR. The interrupt handler was designed so that shorter paths are assigned to more real time sensitive interrupt sources. Rx FIFO interrupt is the most sensitive, Tx FIFO is the second most sensitive, and so on.

The programmable interrupt controller (8259A) is assumed to be configured to "edge triggering mode" and "non-automatic end of the interrupt" mode.

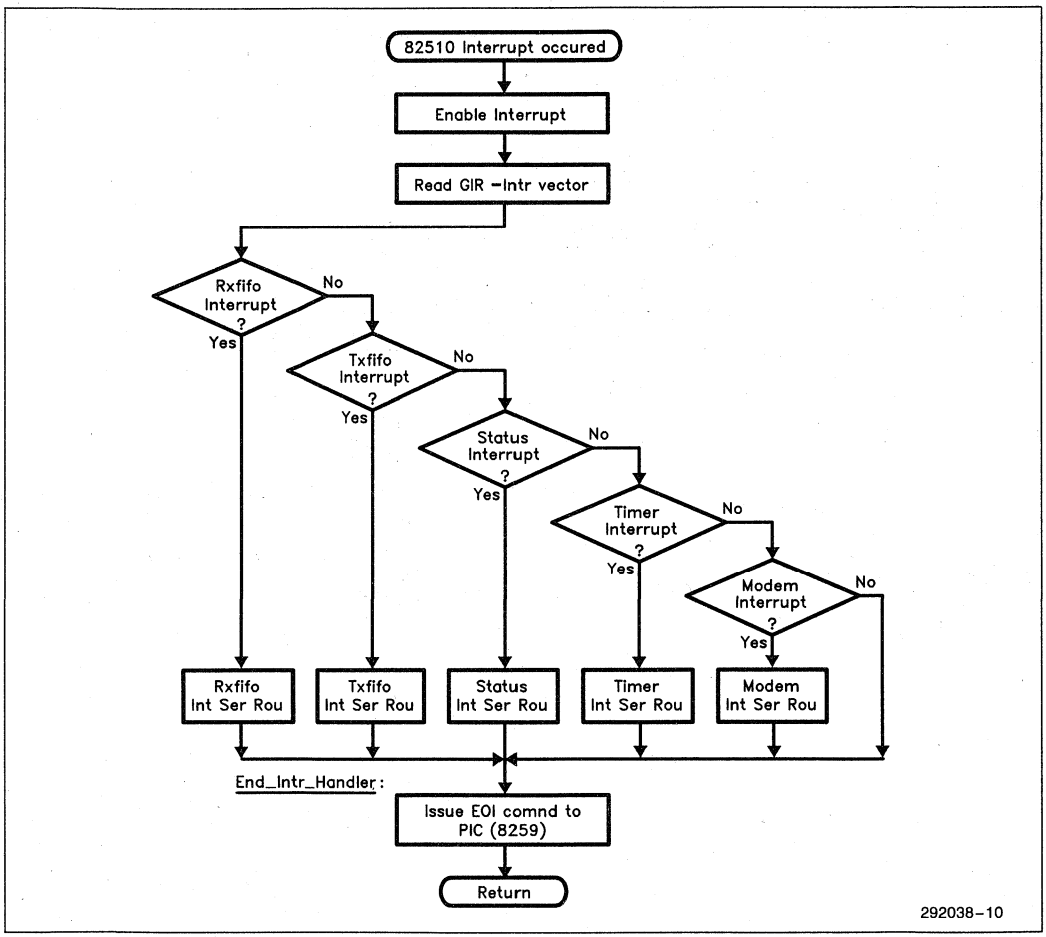


Figure 10. Interrupt Handler

6.4.2 Rx FIFO INTERRUPT SERVICE ROUTINE

The Rx FIFO interrupt service routine first empties the Rx FIFO. The receive data register (RXD) is read, as many times as indicated by the FIFO occupancy register (FLR), and the characters are stored in Rx__Buf.

After emptying the Rx FIFO, the Rx FIFO interrupt service routine executes the burst algorithm (see paragraph 6.2.2). Before leaving the Rx FIFO interrupt service routine, the FIFO occupancy register is rechecked, to empty the Rx FIFO of characters that may have been received during the Rx FIFO interrupt service routine itself. This can happen if the Rx FIFO interrupt service routine has been interrupted by a higher priority interrupt.

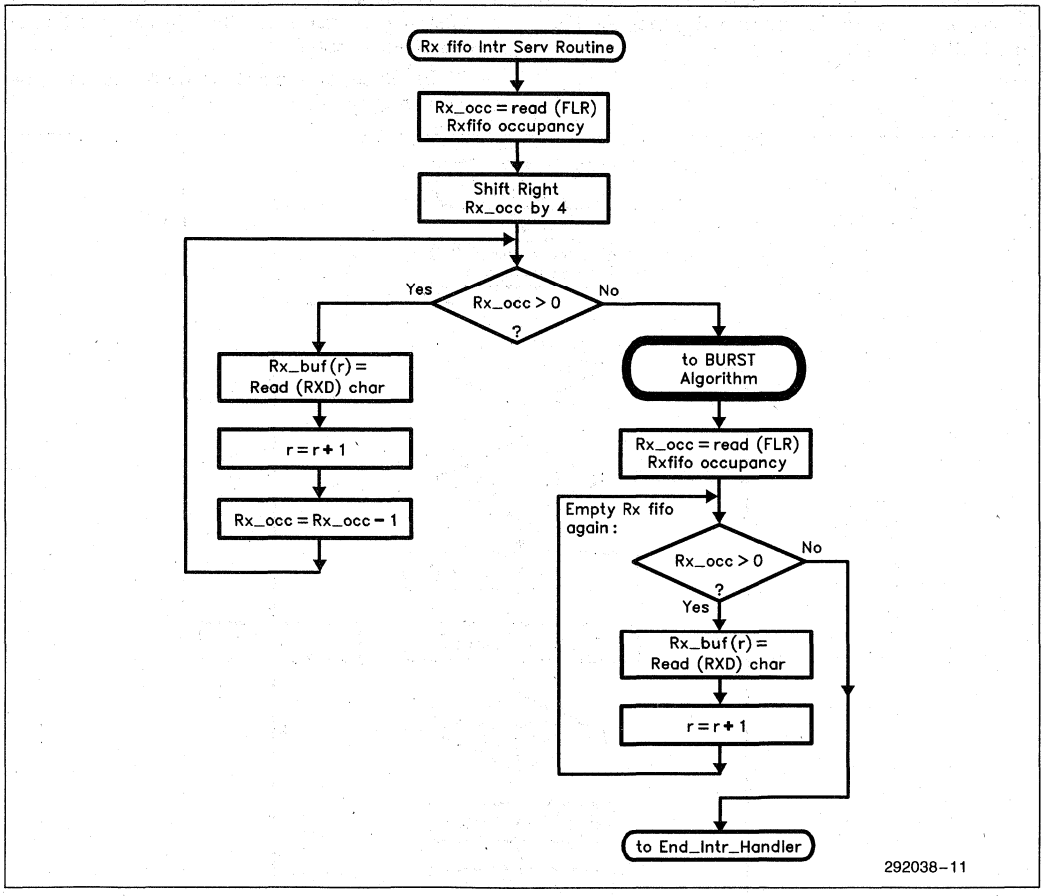


Figure 11. Rx FIFO Interrupt Service Routine

6.4.3 Tx FIFO INTERRUPT SERVICE ROUTINE

The Tx FIFO interrupt service routine fills the Tx FIFO with transmit characters while checking for the End-Of-File terminator. According to the FIFO occupancy register (FLR), the Tx FIFO is loaded (by writing to TXD) until it is full or until the End-Of-File character is detected. The transmitted characters are taken from Tx__Buf. If an End-Of-File character is identified, then the transmission is immediately ended by disabling all 82510 interrupts and setting the Finish__Tx flag.

6.4.4 STATUS INTERRUPT SERVICE ROUTINE

The status interrupt service routine has four objectives:

— To empty the Rx FIFO.

— To stop reception if an End-Of-File character is identified by the control character recognition mechanism (in the receiver station).

— To disable or enable the Tx interrupt if a XOFF or XON character, respectively, is identified by the control character recognition mechanism (in the transmitter station).

— To handle parity, framing, or overrun errors (in the receiver station).

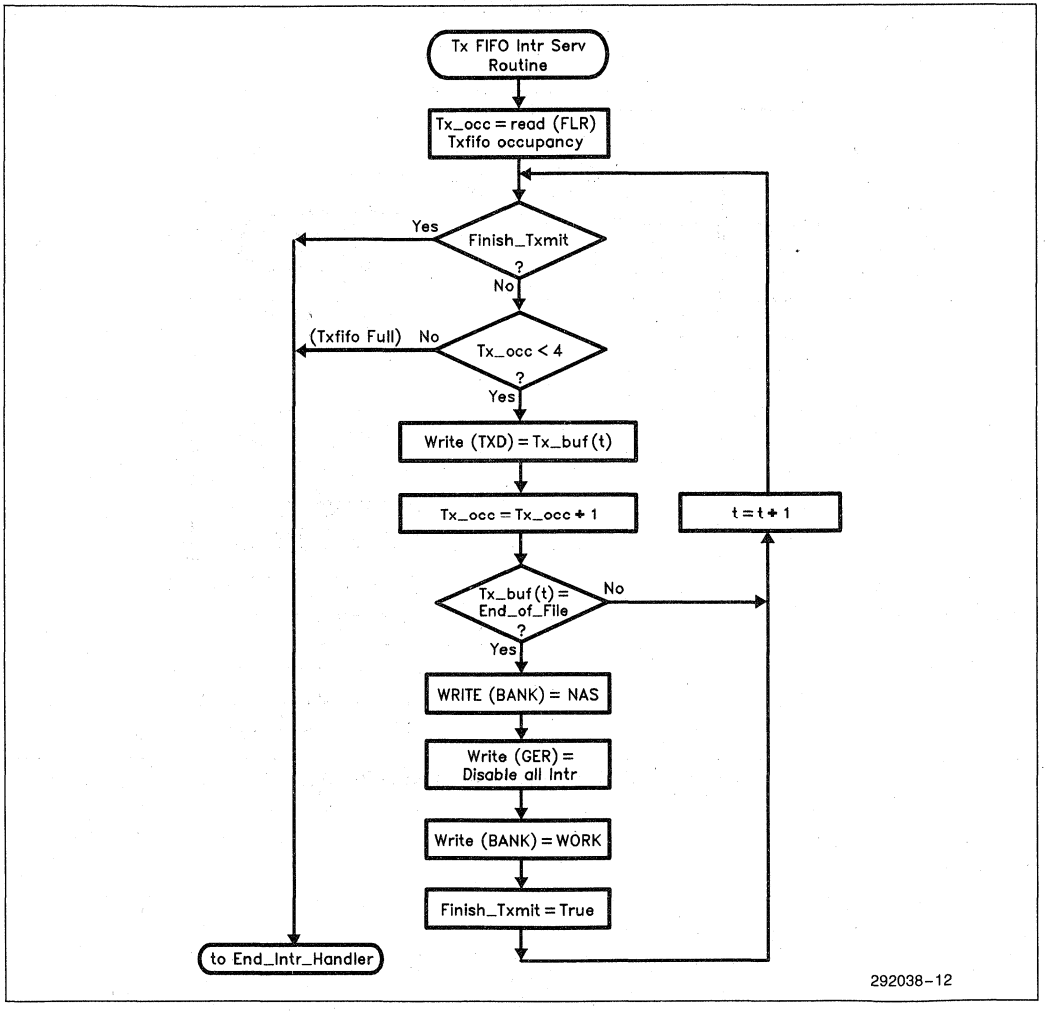


Figure 12. Tx FIFO Intr Service Routine

First the Rx FIFO is emptied. In the receiver station, the RST register is checked to determine whether an End-Of-File terminator has been identified by the 82510, in which case reception is stopped immediately by disabling all interrupt sources and setting the Finish_Rx flag. In the transmitter station, the received characters are checked to identify the received control character. If XOFF is identified, Tx interrupt is disabled. If XON is identified, Tx interrupt is enabled. Note that the software does not need to check for any control character during normal reception; the control characters are identified by the 82510 device.

RST is checked for parity, framing or overrun errors. If one of these errors has occurred, then the error handling routine is executed.

If status interrupt occurs while Burst_algo is assigned to BURST mode, the timer is restarted.

Note that status interrupt is enabled at both stations.

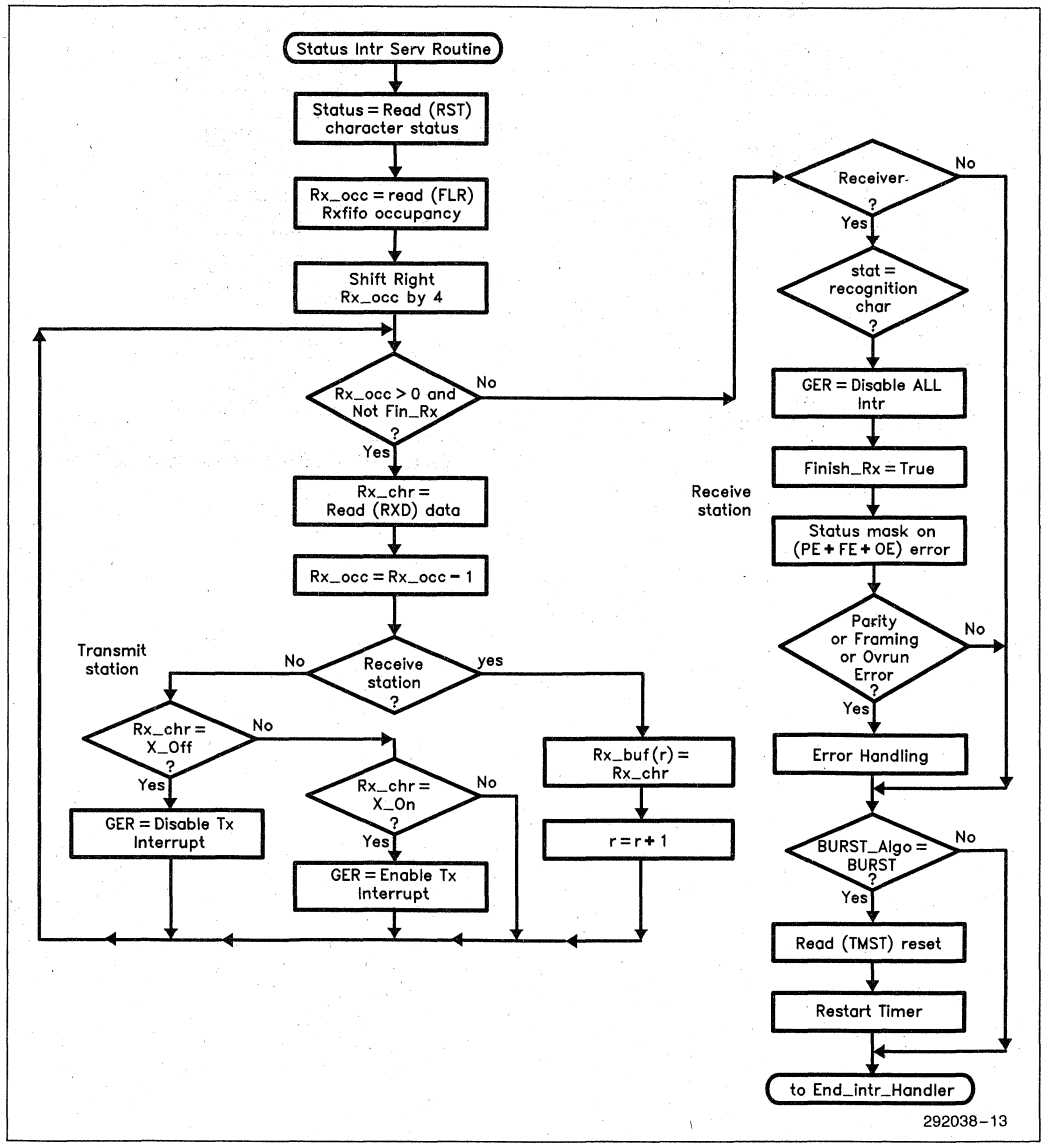


Figure 13. Status Intr. Service Routine

6.4.5 TIMER INTERRUPT SERVICE ROUTINE

A timer interrupt occurs when the receive character rate becomes low. The timer interrupt service routine first empties the Rx FIFO and then switches the burst algorithm to HUNTING mode.

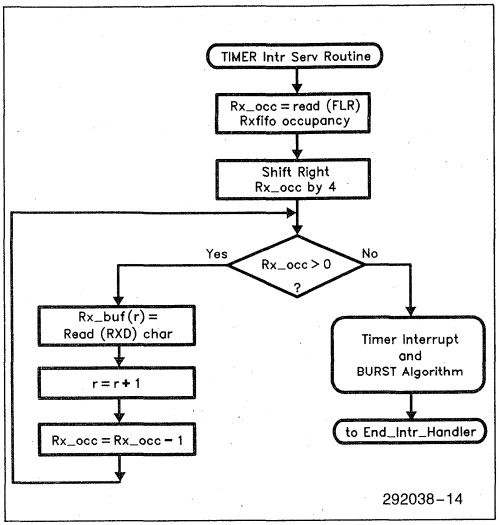


Figure 14. TIMER Intr Service Routine

6.4.6 MODEM INTERRUPT SERVICE ROUTINE

Modem interrupt occurs if one of the modem lines has dropped during transmission or reception. The modem interrupt service routine reads the MSR register to acknowledge the modem interrupt. The modem error routine is then executed.

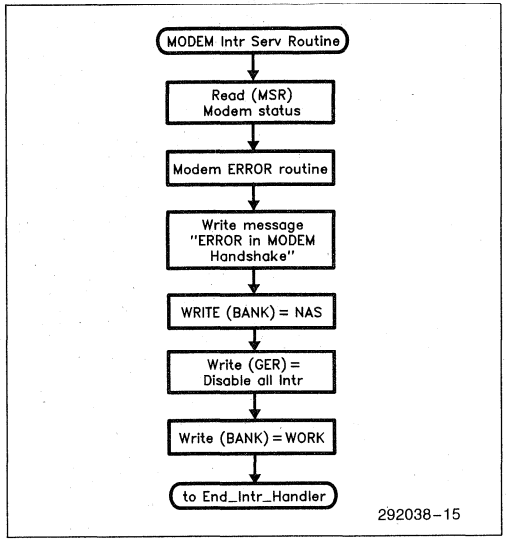


Figure 15. MODEM Intr Service Routine

APPENDIX A
PL/M SOURCE FILE

```
/*************************************************************************
 *                                                                       *
 * 8 2 5 1 0 - H I G H   P E R F O R M A N C E   D r i v e r            *
 *                                                                       *
 * This driver is optimized for Real Time Systems.  It supports          *
 * high system performance. It is based on the "BURST algorithm"         *
 *************************************************************************/


HIGHPERFORMANCE: DO ;

/*************************************************************************
 *                          LITERALS                                     *
 *************************************************************************/

DECLARE LIT             LITERALLY 'LITERALLY';
DECLARE TRUE            LIT '0FFH'          ;
DECLARE FALSE           LIT '00H'           ;
DECLARE BAUD_9600       LIT '003CH'         ;/* Character configurations   */
DECLARE BAUD_19200      LIT '001EH'         ;
DECLARE BAUD_288000     LIT '0002H'         ;
DECLARE DLAB_0          LIT '01111111B'     ;/* Reset DLAB                 */
DECLARE DLAB_1          LIT '10000000B'     ;/* Set   DLAB                 */
DECLARE CR              LIT '0DH'           ;/* Control characters         */
DECLARE LF              LIT '0AH'           ;
DECLARE X_Off           LIT '13H'           ;
DECLARE X_On            LIT '11H'           ;
DECLARE End_Of_File     LIT '1AH'           ;
DECLARE BASE_510        LIT '080H'          ;/* 8 2 5 1 0   registers      */
DECLARE NAS0            LIT '00000000B'     ;
DECLARE WORK1           LIT '00100000B'     ;
DECLARE GEN2            LIT '01000000B'     ;
DECLARE MODM3           LIT '01100000B'     ;
DECLARE TXD             LIT 'BASE_510 + 0'  ;/* BANK 0 - NAS               */
DECLARE RXD             LIT 'BASE_510 + 0'  ;
DECLARE BAL             LIT 'BASE_510 + 0'  ;
DECLARE BAH             LIT 'BASE_510 + 2'  ;
DECLARE GER             LIT 'BASE_510 + 2'  ;
DECLARE GIR             LIT 'BASE_510 + 4'  ;
DECLARE BANK            LIT 'BASE_510 + 4'  ;
DECLARE LCR             LIT 'BASE_510 + 6'  ;
DECLARE MCR             LIT 'BASE_510 + 8'  ;
DECLARE LSR             LIT 'BASE_510 +10'  ;
DECLARE MSR             LIT 'BASE_510 +12'  ;
DECLARE ACR0            LIT 'BASE_510 +14'  ;
DECLARE RXF             LIT 'BASE_510 + 2'  ;/* BANK 1 - WORK              */
DECLARE TXF             LIT 'BASE_510 + 2'  ;
DECLARE TMST            LIT 'BASE_510 + 6'  ;
DECLARE TMCR            LIT 'BASE_510 + 6'  ;
DECLARE FLR             LIT 'BASE_510 + 8'  ;
DECLARE RST             LIT 'BASE_510 +10'  ;
DECLARE RCM             LIT 'BASE_510 +10'  ;
DECLARE TCM             LIT 'BASE_510 +12'  ;
DECLARE GSR             LIT 'BASE_510 +14'  ;
DECLARE ICM             LIT 'BASE_510 +14'  ;
DECLARE FMD             LIT 'BASE_510 + 2'  ;/* BANK 2 - GENERAL CONFIGURE */
DECLARE TMD             LIT 'BASE_510 + 6'  ;
```

292038-16

```
DECLARE IMD             LIT 'BASE_510 + 8'   ;
DECLARE ACR1            LIT 'BASE_510 +10'   ;
DECLARE RIE             LIT 'BASE_510 +12'   ;
DECLARE RMD             LIT 'BASE_510 +14'   ;
DECLARE CLCF            LIT 'BASE_510 + 0'   ;/* BANK 3 - MODEM            */
DECLARE BBL             LIT 'BASE_510 + 0'   ;/* DLAB=1                    */
DECLARE BACF            LIT 'BASE_510 + 2'   ;
DECLARE BBH             LIT 'BASE_510 + 2'   ;/* DLAB=1                    */
DECLARE BBCF            LIT 'BASE_510 + 6'   ;
DECLARE PMD             LIT 'BASE_510 + 8'   ;
DECLARE MIE             LIT 'BASE_510 +10'   ;
DECLARE TMIE            LIT 'BASE_510 +12'   ;
DECLARE OUT2_MCR        LIT '00001000B'      ;/* Specific register bits    */
DECLARE DTR_MCR         LIT '00000001B'      ;
DECLARE DSR_MSR         LIT '00100000B'      ;
DECLARE CLRSTAT_ICM     LIT '00000100B'      ;
DECLARE INTR_510        LIT '21H'            ;
DECLARE PORT_80130M     LIT '0E2H'           ;
DECLARE EN_80130        LIT '0FDH'           ;
DECLARE PORT_EOI        LIT '0E0H'           ;
DECLARE COMM_EOI        LIT '61H'            ;/* End Of Interrupt command  */
DECLARE ENRTX_GER       LIT '00001111B'      ;/* Enable Interrupt bits     */
DECLARE ENTX_GER        LIT '00000010B'      ;
DECLARE ENTXSTAT_GER    LIT '00001110B'      ;
DECLARE ENRX_GER        LIT '00001101B'      ;
DECLARE ENTIMRX_GER     LIT '00101101B'      ;
DECLARE DISTX_GER       LIT '00001101B'      ;
DECLARE DISRX_GER       LIT '00000010B'      ;/* Disable Interrupt bits    */
DECLARE DISRTX_GER      LIT '00000000B'      ;
DECLARE TXTHRESH0_FMD   LIT '00000000B'      ;/* FIFO threshold            */
DECLARE RXTHRESH0_FMD   LIT '00000000B'      ;
DECLARE RXTHRESH3_FMD   LIT '00110000B'      ;
DECLARE MASK_RXOCC      LIT '01110000B'      ;/* Mask on occupancy bits    */
DECLARE MASK_TXOCC      LIT '00000111B'      ;
DECLARE MASK_ACRSTAT    LIT '01000000B'      ;/* Mask on ACR status bits   */
DECLARE CHRLEN_8        LIT '00000011B'      ;/* Async parameters          */
DECLARE STPBIT_1        LIT '00000000B'      ;
DECLARE PARITY_NON      LIT '00000000B'      ;
DECLARE SWRES_CMND      LIT '00010000B'      ;
DECLARE ERRCHR_RST      LIT '00001110B'      ;
DECLARE ACRSTAT_RIE     LIT '01000000B'      ;
DECLARE ACRSTAT_RST     LIT '01000000B'      ;
DECLARE NONI_GIR        LIT '00100001B'      ;/* Interrupt vector          */
DECLARE MODMI_GIR       LIT '00100000B'      ;
DECLARE   TXI_GIR       LIT '00100010B'      ;
DECLARE   RXI_GIR       LIT '00100100B'      ;
DECLARE STATI_GIR       LIT '00100110B'      ;
DECLARE  TIMI_GIR       LIT '00101010B'      ;
DECLARE AUTOACK_IMD     LIT '00001000B'      ;
DECLARE TIMOD_BBCF      LIT '00000000B'      ;/* Timer                     */
DECLARE TIMBI_TMIE      LIT '00000010B'      ;
DECLARE FIFO_IMD        LIT '00000000B'      ;
DECLARE STARTIMB_TMCR   LIT '00100010B'      ;
DECLARE STARTIMB_TMST   LIT '00000010B'      ;
DECLARE RTXCLK_BRGA_CLCF LIT '01010000B'     ;
DECLARE LOW_BAUD        LIT '00H'            ;/* BURST algorithm           */
DECLARE HUNTING_MODE    LIT '01H'            ;
DECLARE  SINGLE_MODE    LIT '02H'            ;
DECLARE   BURST_MODE    LIT '03H'            ;
DECLARE TIME_EXP        LIT '0FFFFH'         ;/* timeout=7mS (at 18.4 Mhz) */
DECLARE WAIT_TIME       LIT '00FFFH'         ;/* WAIT_FOR_MODEM_STATUS     */
```

292038-17

```
/*************************************************************************
 *                          VARIABLES                                    *
 *************************************************************************/

DECLARE TX_PTR POINTER PUBLIC   ;          /* Transmit buffer            */
DECLARE TX_BUF BASED TX_PTR (3000) BYTE ;
DECLARE IX_TX          WORD PUBLIC ;
DECLARE RX_BUF(3000) BYTE PUBLIC;          /* Receive  buffer            */
DECLARE IX_RX          WORD PUBLIC ;
DECLARE INTR_VEC       BYTE PUBLIC ;
DECLARE FIN_TX         BYTE PUBLIC ;       /* Finish Transmission flag   */
DECLARE FIN_RX         BYTE PUBLIC ;       /* Finish Reception    flag   */
DECLARE RX_CHR         BYTE PUBLIC ;
DECLARE TX_CHR         BYTE PUBLIC ;
DECLARE TX_OCC         BYTE PUBLIC ;
DECLARE RX_OCC         BYTE PUBLIC ;
DECLARE STAT           BYTE PUBLIC ;
DECLARE BAUD           WORD PUBLIC ;
DECLARE TEMP           BYTE PUBLIC ;
DECLARE FIN            BYTE PUBLIC ;
DECLARE SELECTION      BYTE PUBLIC ;
DECLARE RECEIVER       BYTE PUBLIC ;       /* Receive station            */
DECLARE BURST_ALGO     BYTE PUBLIC ;       /* BURST algorithm            */
DECLARE MODEM_HANDSHAKE BYTE PUBLIC ;
DECLARE COUNTER        WORD PUBLIC ;
DECLARE RX_ERROR       BYTE PUBLIC ;       /* Error occurred during      */
                                           /* reception                  */


/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/* I/O console utilities                                                 */
$INCLUDE (:F1:TIOHP.PEX)

/* Setup and H/W configurations                                          */
$INCLUDE (:F1:HPUTIL.PEX)

DECLARE MAIN LABEL PUBLIC ;

/*************************************************************************
 * Procedure    INITIALIZATIONS                                          *
 *************************************************************************
 * input:       none                                                     *
 * output:      none                                                     *
 * function:    driver initialization: parameters, 82510                 *
 *              configuration, modem status check.                       *
 * called by:   Main                                                     *
 * calling:     CONFIG_82510, INITIALIZE_BURST, WAIT_FOR_MODEM           *
 *                                                                       *
 * Init the Interrupt mechanism  by  enable Interrupt in GER register    *
 * At the Receive station: Enable Rx FIFO, Status and Modem Interrupts    *
 *                         Disable Timer Interrupt                       *
 * At the Transmit station: Enable Tx FIFO, Status and Modem Interrupts   *
 *                                                                       *
 * flowchart:  figure 7       description:  paragraph 6.3                *
 *************************************************************************/

INITIALIZATIONS: PROCEDURE PUBLIC ;

    DISABLE ;
    CALL SET$INTERRUPT(INTR_510,INTR_HANDLER) ;
                                /* Install THE INTR HANDLER               */

    TX_CHR=00 ;                 /* Clear TX_CHR and RX_CHR                */
    RX_CHR=00 ;
```

292038-18

```
      CALL TEXT ;                 /* TX_PTR is a pointer to the transmitted*/
                                  /* data                                  */
      IX_TX= 0FFFFH ;             /* The index buffer are assigned to -1   */
      IX_RX= 0FFFFH ;
      FIN_TX=FALSE ;              /* Init Finish Transmit and receive flags*/
      FIN_RX=FALSE ;
      RX_BUF(0)=0  ;
      RX_ERROR=FALSE ;            /* Reset the flag                        */

      BAUD=BAUD_288000 ;          /* The Async communication Baud rate is  */
                                  /* the 82510-full scale 288000           */

      CALL CONFIG_82510 ;         /* Configured the 82510:                 */
                                  /* S/W reset, character length, parity,  */
                                  /* stop bit, baud rate and fifo threshol */


/*************************************************************************
 *              INITIALIZE_BURST                                         *
 *************************************************************************
 * input:       none                                                     *
 * output:      Burst_Algo                                               *
 * function:    start Burst algorithm in Hunting mode                    *
 * called by:   INITIALIZATIONS                                          *
 * calling:     none                                                     *
 *                                                                       *
 * flowchart:   figure  4       description:  paragraph 6.2.1            *
 *************************************************************************/

      IF BAUD<=BAUD_9600     THEN BURST_ALGO=HUNTING_MODE ;
                                  /* HUNTING mode:                         */
                                  /* Rx FIFO threshold is 0                */
                                  /* Timer interrupt is disable            */
      ELSE  BURST_ALGO=LOW_BAUD ;

      CALL WAIT_FOR_MODEM_STATUS ;
                                  /* Wait for Modem handshake line "DSR"   */
                                  /* if ACTIVE set      MODEM_HANDSHAKE    */
      TEMP = INPUT(RXD) ;
      TEMP = INPUT(RXD) ;
      TEMP = INPUT(RXD) ;
      TEMP = INPUT(RST) ;


END INITIALIZATIONS ;

/*************************************************************************
 * Procedure    CONFIG_82510                                             *
 *************************************************************************
 * input:       none                                                     *
 * output:      none                                                     *
 * function:    configure the 82510 to a specific operation              *
 *              mode                                                     *
 * called by:   INITIALIZATIONS                                          *
 * calling:     none                                                     *
 *                                                                       *
 * flowchart:   figure 8        description:  paragraph 6.3              *
 *************************************************************************/

CONFIG_82510: PROCEDURE PUBLIC ;

/* Perform Software reset                                                */
OUTPUT(BANK) = WORK1;             /* Move to work bank                   */
OUTPUT(ICM) = SWRES_CMND;         /* S/W reset command                   */
```

292038-19

```
/* BANK  ZERO -  NAS  (The default BANK)                                  */

/* Configured the character by writing to LCR:                           */
/* 1 stop bit, 8 bit lengh, non parity                                   */
OUTPUT(LCR)=(STPBIT_1  + CHRLEN_8 + PARITY_NON) ;
OUTPUT(MCR)=(DTR_MCR OR OUT2_MCR) ;
                              /* Required only in IBM PC environment:    */
                              /* set OUT2 signal to control an external*/
                              /* 3-state buffer that drives the 82510    */
                              /* interrupt signal                        */

IF RECEIVER THEN OUTPUT(ACR0)=End_Of_File ;
                              /* At the Receive station EOF is           */
                              /* recognized to terminate reception       */
ELSE OUTPUT(ACR0)= X_OFF  ;   /* At the Transmit station "X Off" is      */
                              /* recognized to stop transmission         */
                              /* temporary                               */

                              /* Enable 82510 Interrupt by set GER,      */
                              /* done at the end of INITIALIZATIONS      */

                              /* Init the 82510 Interrupt mechanism      */
DISABLE ;
IF RECEIVER THEN OUTPUT(GER)=ENRX_GER ;
                              /* at the  Receive  station                */
ELSE OUTPUT(GER)=ENTXSTAT_GER  ;
                              /* and the Transmit station                */

                              /* Configured baud rate to 288000          */
                              /* by writing to BRG A (BAL and BAH)       */
OUTPUT(LCR)=INPUT(LCR) OR DLAB_1;  /*Set DLAB to allow access to BRG     */
OUTPUT(BAL)=LOW (BAUD_288000) ;
OUTPUT(BAH)=HIGH(BAUD_288000) ;
OUTPUT(LCR)=INPUT(LCR) AND DLAB_0; /* reset DLAB                         */

/* BANK TWO - General configuration                                      */
OUTPUT(BANK)=GEN2 ;

OUTPUT(IMD)=(AUTOACK_IMD OR FIFO_IMD) ;
                              /* Automatic interrupt acknowledge,        */
                              /* Rxfifo depth is four bytes              */

OUTPUT(FMD)=(TXTHRESH0_FMD OR RXTHRESH0_FMD) ;
                              /* Rxfifo threshold is temporally zero     */
                              /* for HUNTING mode (BURST algorithm)      */
                              /* Txfifo threshold is zero for max        */
                              /* interrupt latency                       */

IF RECEIVER THEN OUTPUT(ACR1)=End_Of_File ;
                              /* At the Receive station EOF is           */
                              /* recognized, the same as ACR0            */
ELSE OUTPUT(ACR1)=X_ON ;      /* At the Transmit station "X On"  is      */
                              /* recognized to continue transmission     */

OUTPUT(RIE) = (ACRSTAT_RIE OR INPUT(RIE)) ;
                              /* Enable interrupt on programmed control*/
                              /* character received (ACR0/ACR1)          */

/* BANK THREE - MODEM configuration                                      */
OUTPUT(BANK)=MODM3 ;

OUTPUT(BBCF)=(TIMOD_BBCF) ;   /* BRG B configured to TIMER mode          */
OUTPUT(BANK) = NAS0;          /* Move to nas bank to set DLAB            */
OUTPUT(LCR)=INPUT(LCR) OR DLAB_1 ; /* Set DLAB to allow access to BRG    */
OUTPUT(BANK) = MODM3;         /* MODEM bank                              */
OUTPUT(BBL) = LOW (TIME_EXP); /* Set max timeout (7ms if 18Mhz crystal)*/
```

292038-20

```
OUTPUT(BBH) = HIGH(TIME_EXP);   /* to issue interrupt when time has      */
OUTPUT(BANK) = NAS0;            /* expired. Move to NAS bank again       */
OUTPUT(LCR) =INPUT(LCR) AND DLAB_0 ; /* Reset DLAB                       */
OUTPUT(BANK) = MODM3;           /* Switch to BANK THREE - MODEM          */
OUTPUT(CLCF)=RTXCLK_BRGA_CLCF ; /* The receive and transmit clock source */
                                /* is BRG A                              */

OUTPUT(TMIE)=TIMBI_TMIE ;       /* Enable Timer block interrupt          */
                                /* (stil disabled in Timer bit in GER)   */

/* BANK ONE - general WORK       - The RUNTIME bank                      */
OUTPUT(BANK)=WORK1 ;
OUTPUT(ICM)=CLRSTAT_ICM ;       /* Issues a command to clear all         */
                                /* status registers                      */


/* Remain in   W O R K - THE runtime  bank                               */


END CONFIG_82510 ;

/*************************************************************************
 * Procedure   WAIT_FOR_MODEM_STATUS                                     *
 *************************************************************************
 * input:      none                                                      *
 * output:     Modem_Handshake                                           *
 * function:   waits with a timeout for DSR active,                      *
 *             returns status flag                                       *
 * called by:  INITIALIZATIONS                                           *
 * calling:    none                                                      *
 *                                                                       *
 * flowchart:  figure 9       description:  paragraph 6.3.1              *
 *************************************************************************/

WAIT_FOR_MODEM_STATUS: PROCEDURE PUBLIC ;

MODEM_HANDSHAKE = FALSE ;
COUNTER = WAIT_TIME ;

DO WHILE (NOT MODEM_HANDSHAKE) AND ((COUNTER:=COUNTER-1) > 0 ) ;
   IF (INPUT(MSR) AND DSR_MSR) <> 0   THEN MODEM_HANDSHAKE = TRUE ;
END ;

END WAIT_FOR_MODEM_STATUS ;

/*************************************************************************
 * Procedure   INTERRUPT HANDLER                                         *
 *************************************************************************
 * input:      Tx Buffer                                                 *
 * output:     Rx Buffer, Finish_Tx, Finish_Rx                           *
 * function:   service all 82510 interrupt sources:                      *
 *             Rx Fifo, Tx Fifo, Status, Timer, Modem                    *
 * called by:  82510 hardware interrupt                                  *
 * calling:    Rx_Fifo_Intr, Tx_Fifo_Intr, Status_Intr,                  *
 *             Timer_Intr, Modem_Intr                                    *
 *                                                                       *
 * flowchart:  figure 10   description:  paragraph 6.4, 6.4.1            *
 *************************************************************************/

INTR_HANDLER: PROCEDURE INTERRUPT INTR_510 REENTRANT PUBLIC ;

ENABLE  ;                       /* Enable Interrupts of                  */
                                /* HIGHIER priority devices              */

INTR_VEC=INPUT(GIR);            /* Get the 82510-highest priority        */
                                /* pending interrupt                     */
```

292038-21

```
/*************************************************************************
 *              Rx_FIFO_INTR                                             *
 *************************************************************************
 * input:       none                                                     *
 * output:      Rx_Buffer, Burst_Algo                                    *
 * function:    service Rx Fifo interrupt                                *
 *              receive characters; store in receive buffer              *
 * called by:   INTERRUPT HANDLER                                        *
 * calling:     BURST_ALGO                                               *
 *                                                                       *
 * flowchart:  figure 11      description:  paragraph 6.4.2              *
 *************************************************************************/

IF INTR_VEC=RXI_GIR THEN DO ;

   RX_OCC=INPUT(FLR) ;              /* Rx fifo level occupancy           */
                                    /* Shift the Rx occupancy bit        */
   RX_OCC=SHR(RX_OCC,4) ;           /* to get it's real value            */
                                    /* - OPTIMIZE code -                 */
                                    /* Empty the Rx FIFO and store the   */
                                    /* received character in RX_BUF      */
   RX_BUF(IX_RX:=IX_RX+1)=INPUT(RXD) ;
                                    /* Read the first character immediatly */
                                    /* to save Real Time                 */
   DO WHILE (RX_OCC:=RX_OCC-1) > 0 ;
      RX_BUF(IX_RX:=IX_RX+1)=INPUT(RXD) ;
   END ;



/*************************************************************************
 *              BURST_ALGORITHM                                          *
 *************************************************************************
 * input:       Burst_Algo                                               *
 * output:      Burst_Algo                                               *
 * function:    execute a step in the burst algorithm                    *
 *              after characters are received                            *
 * called by:   Rx_FIFO_INTR                                             *
 * calling:     none                                                     *
 *                                                                       *
 * flowchart: figure 5    description: par. 6.2.2.1 to 6.2.2.3           *
 *************************************************************************/


      /*-------------------------------------------------------------*
       *  B U R S T     M O D E - step 3  (full fifo threshold)      *
       * Reset the Timer status                                      *
       * Restart the Timer                                           *
       *-------------------------------------------------------------*/
      IF BURST_ALGO = BURST_MODE THEN DO ;
         TEMP = INPUT(TMST);
         OUTPUT(TMCR) =STARTIMB_TMCR;
      END;


      /*-------------------------------------------------------------*
       *   H U N T I N G   M O D E - step 1                          *
       * Operate the TIMER                                           *
       * Change to step 2 SINGLE  mode                               *
       *-------------------------------------------------------------*/
      ELSE IF BURST_ALGO = HUNTING_MODE THEN DO ;
         OUTPUT(TMCR)=STARTIMB_TMCR ;
         BURST_ALGO=SINGLE_MODE ;
      END ;
```

292038-22

```
      /*-------------------------------------------------------------------*
       *   S I N G L E     M O D E - step 2                                 *
       * If TIME has expired, means the receive                            *
       * rate is LOW, return to  HUNTING mode                              *
       * If TIME did NOT expire, means the                                 *
       * Receive rate is HIGH, set Rx FIFO threshold, Restart the          *
       * Timer and switch to BURST mode                                    *
       *-------------------------------------------------------------------*/
      ELSE IF BURST_ALGO = SINGLE_MODE THEN DO ;

         IF ((INPUT(TMST) AND STARTIMB_TMST) <>0) THEN
                     BURST_ALGO= HUNTING_MODE ;
         ELSE DO;
            OUTPUT(BANK) = GEN2;/* Switch to BANK TWO - General Config   */
            OUTPUT(FMD)=TXTHRESH0_FMD OR RXTHRESH3_FMD;
            OUTPUT(BANK) =NAS0; /* Switch to BANK ZERO - NAS             */
            OUTPUT(GER) = ENTIMRX_GER;
                                /* Enable TIMER,RX and  MODEM interrupts */
            OUTPUT(BANK)=WORK1; /* Switch to BANK ONE  - WORK            */
            BURST_ALGO = BURST_MODE;
            TEMP = INPUT(TMST); /* Reset timer status                    */
            OUTPUT(TMCR) = STARTIMB_TMCR;
         END;
      END;                      /* End of SINGLE mode                    */

   /* ....End of BURST algorithm.....................................*/

   /*  Another try to empty the Rx fifo                                 */
   /*  before leaving the interrupt handler                             */

      DO WHILE (INPUT(FLR)<>0)  ;
                                /* Empty the Rx FIFO and store the       */
                                /* received character in RX_BUF         */
         RX_BUF(IX_RX:=IX_RX+1)=INPUT(RXD) ;
      END ;

END ;                           /* End of Rx fifo interrupt              */

/*************************************************************************
 *             TxFIFO_INTR                                               *
 *************************************************************************
 * input:      Tx_Buffer                                                 *
 * output:     Finish_tx                                                 *
 * function:   service Tx Fifo interrupt                                 *
 *             transmit characters from transmit buffer (OPTIMIZE code)  *
 * called by:  INTERRUPT HANDLER                                         *
 * calling:    none                                                      *
 *                                                                       *
 * flowchart:  figure 12      description:  paragraph 6.4.3              *
 *************************************************************************/

ELSE IF INTR_VEC=TXI_GIR THEN DO ;
   TX_OCC=INPUT(FLR) AND MASK_TXOCC ;
                                /* Tx fifo level occupancy               */
    /* Fill Tx FIFO, the transmitted characters are taken from TX_buf    */
   DO WHILE (TX_OCC:=TX_OCC+1)<5 ;
      OUTPUT(TXD)=TX_BUF(IX_TX:=IX_TX+1);
      IF TX_BUF(IX_TX)=End_Of_File THEN DO ;
         OUTPUT(BANK)=NAS0 ;    /* Disable Tx interrupt, as the transmit */
         OUTPUT(GER)=DISTX_GER; /* delimiter character  was identified   */
         OUTPUT(BANK)=WORK1 ;   /* Switch to BANK ONE  - WORK            */
         TX_OCC = 5 ;           /* load TX_OCC to terminate external loop*/
         FIN_TX = TRUE ;        /*  Set Finish transmit flag             */
      END ;
   END ;
END ;                           /* End of TXFIFO_INTR                    */
```

292038-23

```
/*************************************************************************
 *              STATUS_INTR                                              *
 *************************************************************************
 * input:       none                                                     *
 * output:      Finish_Rx                                                *
 * function:    service Status interrupt                                 *
 *              Receive  station: EOF terminate the reception            *
 *              Transmit station: X_Off Disable the transmission         *
 *                                X_On  Enable  the transmission         *
 * called by:   INTERRUPT HANDLER                                        *
 * calling:     none                                                     *
 *                                                                       *
 * flowchart:   figure 13       description:  paragraph 6.4.4            *
 *************************************************************************/


ELSE IF INTR_VEC=STATI_GIR THEN DO ;

      STAT=INPUT(RST) ;          /* Get the current RST status          */

      RX_OCC=INPUT(FLR) ;        /* Rx fifo level occupancy             */
      RX_OCC=SHR(RX_OCC,4) ;

      DO WHILE (RX_OCC>0 AND (NOT FIN_RX));
         RX_OCC=RX_OCC-1 ;       /*         First, empty Rx FIFO        */
         RX_CHR=INPUT(RXD) ;

         IF RECEIVER THEN   RX_BUF(IX_RX:=IX_RX+1)=RX_CHR ;

         ELSE  DO ;
            IF RX_CHR = X_OFF THEN DO ;
               OUTPUT(BANK)=NAS0;/* Switch to BANK ZERO - NAS          */
               OUTPUT(GER) = INPUT(GER) AND DISTX_GER ;
                              /* Disable Transmit interrupt             */
               OUTPUT(BANK)=WORK1;/* Switch to BANK ONE  - WORK         */
            END ;
            ELSE IF RX_CHR = X_ON THEN DO ;
               OUTPUT(BANK)= NAS0 ;
               OUTPUT(GER) = INPUT(GER) OR  ENTX_GER ;
                              /* Enable  Transmit interrupt  again      */
               OUTPUT(BANK)= WORK1 ;
            END ;
         END ;
      END ;


   IF RECEIVER THEN DO ;
      IF ((STAT AND ACRSTAT_RST) <> 0) THEN DO ;
         OUTPUT(BANK)= NAS0 ;    /* If End_Of_Line was recognized,      */
         OUTPUT(GER) = DISRTX_GER ;
         OUTPUT(BANK)= WORK1 ;   /* Disable 82510-interrupts and the    */
         FIN_RX      = TRUE ;    /* Reception                           */
      END ;
      ELSE IF ((STAT AND ERRCHR_RST) <> 0) THEN DO ;
         CALL WRITE(@('** ERROR in character Status ',0)) ;
         CALL ERROR_CHAR_HANDLER ;

         IF BURST_ALGO=BURST_MODE THEN DO ;
                              /* In BURST mode do:                      */
            TEMP = INPUT(TMST); /* Reset timer status                   */
            OUTPUT(TMCR) = STARTIMB_TMCR;
         END;                 /* Restart TIMER                          */
      END ;
   END ;

END ;                         /* End of STATUS interrupt               */
```

292038-24

```
/*************************************************************************
 *              TIMER_INTR                                               *
 *************************************************************************
 * input:       none                                                     *
 * output:      Burst_Algo                                               *
 * function:    service Timer interrupt; receive characters              *
 *              and switch Burst_Algo to HUNTING mode                    *
 * called by:   INTERRUPT HANDLER                                        *
 * calling:     BURST &TIMER                                             *
 *                                                                       *
 * flowchart:   figure 14       description:  paragraph 6.4.5            *
 *************************************************************************/


ELSE IF INTR_VEC=TIMI_GIR THEN DO ;

      IF ((RX_OCC:=INPUT(FLR))<>0) THEN DO ;

         RX_OCC=SHR(RX_OCC,4) ; /* Rx fifo level occupancy, shift right  */
                                /* - OPTIMIZE code -                     */
                                /* Empty the Rx FIFO and store the       */
                                /* received character in RX_BUF          */
         RX_BUF(IX_RX:=IX_RX+1)=INPUT(RXD) ;
         DO WHILE (RX_OCC:=RX_OCC-1) > 0 ;
            RX_BUF(IX_RX:=IX_RX+1)=INPUT(RXD) ;
         END ;
                                /* Store the received character in RX_buf*/
      END ;



/*************************************************************************
 *              BURST & TIMER                                            *
 *************************************************************************
 * input:       Burst_Algo                                               *
 * output:      Burst_Algo                                               *
 * function:    execute a step in the burst algorithm                    *
 *              after timer interrupt; switch to HUNTING                 *
 * called by:   TIMER_INTR                                               *
 * calling:     none                                                     *
 *                                                                       *
 * flowchart:   figure  6       description:  paragraph 6.2.2.4          *
 *************************************************************************/

   OUTPUT(BANK) = GEN2;          /* Switch to BANK TWO - General Config   */
   OUTPUT(FMD) = TXTHRESH0_FMD OR RXTHRESH0_FMD;
                                 /* Rxfifo threshold=0, Txfifo threshold=0*/

   OUTPUT(BANK) = NAS0;          /* Switch to BANK ZERO - NAS             */
   OUTPUT(GER) = ENRX_GER;       /* Disable Timer interrupt and           */
   OUTPUT(BANK) = WORK1;         /* Enable RX,STAT,MODEM interrupts       */
   TEMP = INPUT(TMST);           /* Acknowledge TIMER interrupt           */
   BURST_ALGO = HUNTING_MODE ;   /* Back to HUNTING mode                  */
END ;                            /* End of TIMER interrupt                */
```

292038-25

```
/*************************************************************************
 *              MODEM_INTR                                               *
 *************************************************************************
 * input:       none                                                     *
 * output:      none                                                     *
 * function:    service Modem interrupt and handle modem errors.         *
 *              Modem interrupt is occurred if No Modem was setup, or    *
 *              if DSR was dropped in the middle of the communication    *
 * called by:   INTERRUPT HANDLER                                        *
 * calling:     none                                                     *
 *                                                                       *
 * flowchart:   figure 15       description:  paragraph 6.4.6            *
 *************************************************************************/

ELSE IF INTR_VEC=MODMI_GIR THEN DO ;

   STAT=INPUT(MSR) ;              /* Get MODEM status                    */

   CALL ERROR_MODEM_HANDLER ;     /* Handel Modem Errors handshake       */

END ;                             /* End of MODEM interrupt              */


OUTPUT(PORT_EOI)=COMM_EOI ;       /* Write End_Of_Interrupt command to the */
                                  /* PIC (8259A)                         */
END INTR_HANDLER ;


/*************************************************************************
 * Procedure    ERROR_MODEM_HANDLER                                      *
 *************************************************************************/

ERROR_MODEM_HANDLER: PROCEDURE PUBLIC ;

MODEM_HANDSHAKE = FALSE ;         /* Flag indicates that an Error occurred */
                                  /* in Modem                            */

END ERROR_MODEM_HANDLER ;


/*************************************************************************
 * Procedure    ERROR_CHAR_HANDLER                                       *
 *************************************************************************/

ERROR_CHAR_HANDLER: PROCEDURE PUBLIC ;

RX_ERROR      = TRUE  ;           /* Flag indicates that an Error occurred */
                                  /* during Reception                    */

OUTPUT(BANK) = NAS0 ;             /* Switch to BANK ZERO - NAS           */
OUTPUT(GER)  = DISRTX_GER ;       /* Disable all the 82510 Interrupts    */
OUTPUT(BANK) = WORK1 ;            /* Switch to BANK ONE  - WORK          */

END ERROR_CHAR_HANDLER ;
```

292038-26

```
/************************************************************************
 *  Procedure       LOOP                                                *
 *                                                                      *
 * LOOP procedure is executed until Transmission/Reception Finishes     *
 * or until the loop ends.                                              *
 ************************************************************************/

LOOP: PROCEDURE PUBLIC ;
DECLARE N WORD ;
DECLARE NUM WORD ;
DECLARE MAXLOOP BYTE ;
MAXLOOP= 20 ;
NUM=0 ;
DO WHILE ( (NOT FIN_TX) AND (NOT FIN_RX) AND (NUM<MAXLOOP) ) ;
   NUM=NUM+1 ;                   /* Count the LOOP times                */
   CALL WRITELN(@('... Background Program  ...',0));
   ENABLE ;
   CALL TIME(5000) ;             /* Software delay                      */

END ;

IF FIN_TX THEN CALL WRITELN(@('T r a n s m i s s i o n  -  E N D E D ',0));
IF FIN_RX THEN CALL WRITELN(@('R e c e p t i o n   - E N D E D  ',0));

OUTPUT(BANK)=NAS0 ;              /* If Communication is Not ended       */
OUTPUT(GER)=DISRTX_GER ;         /* successfully the Interrupts are     */
OUTPUT(BANK)=WORK1 ;             /* Disabled by MAIN                    */
IF  RECEIVER THEN DO ;           /* Display RX buffer                   */
   IF FIN_RX THEN DO ;
      CALL WRITELN(@('The Received Message:',0)) ;
      CALL DISPTEXT(@RX_BUF) ;
   END ;
   ELSE
      CALL WRITELN(@('** ERROR -THE Reception NOT ended successfully',0)) ;
END ;
ELSE IF (NOT FIN_TX) THEN        /* The Transimt station                */
    CALL WRITELN(@('** ERROR -THE Transmission NOT ended successfully',0));
END LOOP ;

/************************************************************************
 *  Procedure       TEXT                                                *
 ************************************************************************
 * input:       none                                                    *
 * output:      Tx_ptr                                                  *
 * function:    Return a pointr to the Transmit buffer.  Data in the    *
 *              transmit buffer must be trminated by End_Of_File.       *
 * called by:   INITIALIZATIONS                                         *
 * calling:     none                                                    *
 ************************************************************************/

TEXT: PROCEDURE PUBLIC ;

TX_PTR=@('>',
CR,LF,
'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz0123456789',
CR,LF,
'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz0123456789',
CR,LF,
'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz0123456789',
CR,LF,
'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz0123456789',
CR,LF,
'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz0123456789',
CR,LF,End_Of_File,0) ;
                                /* End_Of_File-terminate the Transmission*/
END TEXT ;
```

292038-27

```
/*************************************************************************
 * External    procedures                                                *
 *************************************************************************
 * WRITELN:  I/O console utility - dispaly a string, end with CR         *
 * MENU:     I/O console utility - display a menu, enter the user        *
 *                                 selection                             *
 * DISPTEXT: I/O console utility - display the contents of the           *
 *                                 Receive buffer (Rx_buf)               *
 * INIT_HARDWARE_SETUP: Setup and Hardware configurations of the         *
 *                      specific station                                 *
 *************************************************************************/


/*************************************************************************
 * Procedure   MAIN                                                      *
 *************************************************************************
 * input:      Finish_Rx, Finish_Tx                                      *
 * output:     Receiver flag                                             *
 * function:   get station type (Rx or Tx) from the operator;            *
 *             wait till communication is completed; display;            *
 *               RECEIVER STATION SHOULD BE ACTIVATED FIRST              *
 * called by:  Application                                               *
 * calling:    INITIALIZATIONS, LOOP                                     *
 *                                                                       *
 * flowchart:  figure  3      description:  paragraph 6.1                *
 *************************************************************************/

MAIN:

CALL INIT_HARDWARE_SETUP        ;
                                /* External, Setup and H/W configurations*/
FIN=FALSE ;
DO WHILE NOT(FIN) ;
    SELECTION=0 ;
    CALL WRITELN(@('------------------------------------------- ',0));
    SELECTION=MENU(SELECTION,@('Station: (Quit/Transmitter/Receiver)',0)) ;
                                /* Get operator selection.              */
                                /* Receiver station should be activated */
                                /* prior to the transmitter station     */
    DO CASE SELECTION ;
                FIN=TRUE    ;   /* 0 - Quit of HIGH PERFORMANCE Driver  */
                DO ;            /* 1 - Transmit station                 */
                        RECEIVER=FALSE ;
                        CALL INITIALIZATIONS ;
                        CALL LOOP ;
                END ;
                DO ;            /* 2 - Receive station                  */
                        RECEIVER=TRUE  ;
                        CALL INITIALIZATIONS ;
                        CALL LOOP ;
                END ;
    END ;
END ;

CALL EXIT ;


END HIGHPERFORMANCE ;

/*************************************************************************/
```

292038-28

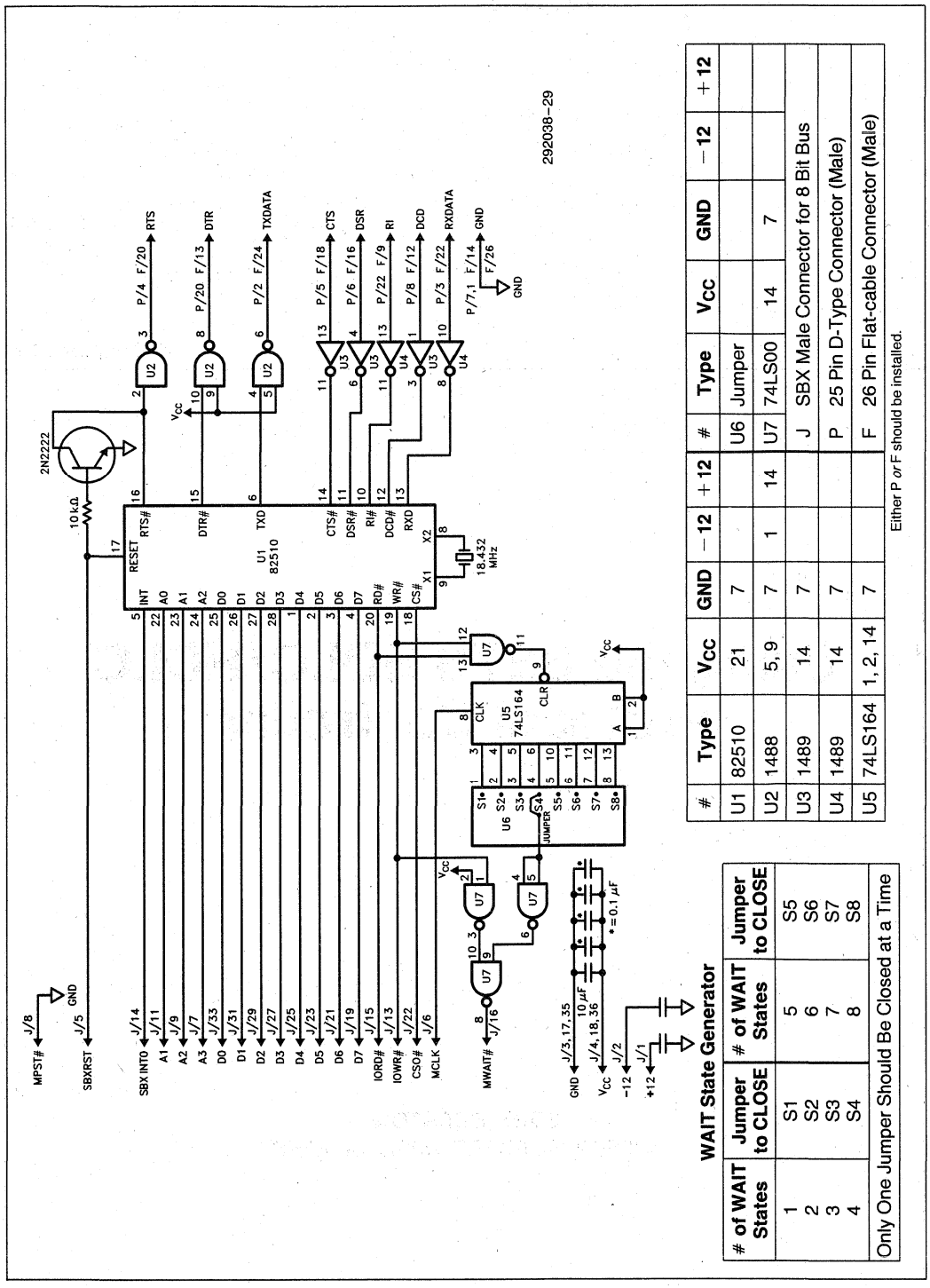
APPENDIX B
82510 BASED SBX SERIAL CHANNEL

This document describes the implementation of an 82510 based SBX board that provides a RS-232 interface to any iSBC board which has an SBX connector. The SBX can be useful for customers that need a fast software development vehicle while the 82510 system hardware is still in the design stage. The customer can also use the SBX for evaluation of the 82510 in a system environment.

In order to minimize the customer's software development costs, the RMX86/286 Terminal Device Driver for the 82510 has also been developed and can be run by the RMX user on his iSBC with the SBX-82510 board described herewith. The RMX86/286 drivers are available from INSITE, along with the source code and the documentation.

BOARD DESCRIPTION (See Figure B-1)

The following 82510 signals are connected directly to the SBX connector (installed on the pin side): DATA, ADDRESS, INTERRUPT, RESET, READ#, WRITE# and CS#. Wait states are generated by a shift register logic (U5, U7), clocked by the MCLK signal of the SBX interface. The number of wait states is selected by installing one of the eight jumpers to select one parallel output of the shift register. The 82510 is clocked by an 18.432 MHz Crystal (using its on-chip oscillator). A discrete transistor is used to pull down the RTS# signal during RESET to set the crystal mode (note that in a larger board, an unused open collector inverter or three-state gate can be used for this purpose). The 82510 is connected to the communication channel through RS-232 line drivers and receivers. Either a 25 pin D-Type connector (P) or a 26 pin Flat-Cable connector (F) is used to connect the board to the RS-232 channel.



#	Type	V_{CC}	GND	−12	+12
U1	82510	21	7		
U2	1488	5, 9	7	1	14
U3	1489	14	7		
U4	1489	14	7		
U5	74LS164	1, 2, 14	7		

#	Type	V_{CC}	GND	−12	+12
U6	Jumper				
U7	74LS00	14	7		

J SBX Male Connector for 8 Bit Bus

P 25 Pin D-Type Connector (Male)

F 26 Pin Flat-cable Connector (Male)

Either P *or* F should be installed.

WAIT State Generator

# of WAIT States	Jumper to CLOSE	# of WAIT States	Jumper to CLOSE
1	S1	5	S5
2	S2	6	S6
3	S3	7	S7
4	S4	8	S8

Only One Jumper Should Be Closed at a Time

Figure B-1. Board Schematics

intel®

APPLICATION NOTE

AP-36

November 1986

Using the 8273 SDLC/HDLC Protocol Controller

JOHN BEASTON
MICROCOMPUTER APPLICATIONS

Order Number: 611001-001

INTRODUCTION

The Intel 8273 is a Data Communications Protocol Controller designed for use in systems utilizing either SDLC or HDLC (Synchronous or High-Level Data Link Control) protocols. In addition to the usual features such as full duplex operation, automatic Frame Check Sequence generation and checking, automatic zero bit insertion and deletion, and TTL compatibility found on other single component SDLC controllers, the 8273 features a frame level command structure, a digital phase locked loop, SDLC loop operation, and diagnostics.

The frame level command structure is made possible by the 8273's unique internal dual processor architecture. A high-speed bit processor handles the serial data manipulations and character recognition. A byte processor implements the frame level commands. These dual processors allow the 8273 to control the necessary byte-by-byte operation of the data channel with a minimum of CPU (Central Processing Unit) intervention. For the user this means the CPU has time to take on additional tasks. The digital phase locked loop (DPLL) provides a means of clock recovery from the received data stream on-chip. This feature, along with the frame level commands, makes SDLC loop operation extremely simple and flexible. Diagnostics in the form of both data and clock loopback are available to simplify board debug and link testing. The 8273 is a dedicated function peripheral in the MCS-80/85 Microcomputer family and as such, it interfaces to the 8080/8085 system with a minimum of external hardware.

This application note explains the 8273 as a component and shows its use in a generalized loop configuration and a typical 8085 system. The 8085 system was used to verify the SDLC operation of the 8273 on an actual IBM SDLC data communications link.

The first section of this application note presents an overview of the SDLC/HDLC protocols. It is fairly tutorial in nature and may be skipped by the more knowledgeable reader. The second section describes the 8273 from a functional standpoint with explanation of the block diagram. The software aspects of the 8273, including command examples, are discussed in the third section. The fourth and fifth sections discuss a loop SDLC configuration and the 8085 system respectively.

SDLC/HDLC OVERVIEW

SDLC is a protocol for managing the flow of information on a data communications link. In other words, SDLC can be thought of as an envelope—addressed, stamped, and containing an s.a.s.e.—in which information is transferred from location to location on a data communications link. (Please note that while SDLC is discussed specifically, all comments also apply to HDLC except where noted.) The link may be either point-to-point or multi-point, with the point-to-point configuration being either switched or nonswitched. The information flow may use either full or half duplex exchanges. With this many configurations supported, it is difficult to find a synchronous data communications application where SDLC would not be appropriate.

Aside from supporting a large number of configurations, SDLC offers the potential of a 2× increase in throughput over the presently most prevalent protocol: Bi-Sync. This performance increase is primarily due to two characteristics of SDLC: full duplex operation and the implied acknowledgement of transferred information. The performance increase due to full duplex operation is fairly obvious since, in SDLC, both stations can communicate simultaneously. Bi-Sync supports only half-duplex (two-way alternate) communication. The increase from implied acknowledgement arises from the fact that a station using SDLC may acknowledge previously received information while transmitting different information. Up to 7 messages may be outstanding before an acknowledgement is required. These messages may be acknowledged as a block rather than singly. In Bi-Sync, acknowledgements are unique messages that may not be included with messages containing information and each information message requires a separate acknowledgement. Thus the line efficiency of SDLC is superior to Bi-Sync. On a higher level, the potential of a 2× increase in performance means lower cost per unit of information transferred. Notice that the increase is not due to higher data link speeds (SDLC is actually speed independent), but simply through better line utilization.

Getting down to the more salient characteristics of SDLC; the basic unit of information on an SDLC link is that of the frame. The frame format is shown in Figure 1. Five fields comprise each frame: flag, address, control, information, and frame check sequence. The flag fields (F) form the boundary of the frame and all

Opening Flag	Address Field (A)	Control Field (C)	Information Field (I)	Frame Check Sequence (FCS)	Closing Flag
01111110	8 Bits	8 Bits	Any Length 0 to N Bits	16 Bits	01111110

Figure 1. SDLC Frame Format

other fields are positionally related to one of the two flags. All frames start with an opening flag and end with a closing flag. Flags are used for frame synchronization. They also may serve as time-fill characters between frames. (There are no intraframe time-fill characters in SDLC as there are in Bi-Sync.) The opening flag serves as a reference point for the address (A) and control (C) fields. The frame check sequence (FCS) is referenced from the closing flag. All flags have the binary configuration 01111110 (7EH).

SDLC is a bit-oriented protocol, that is, the receiving station must be able to recognize a flag (or any other special character) at any time, not just on an 8-bit boundary. This, of course, implies that a frame may be N-bits in length. (The vast majority of applications tend to use frames which are multiples of 8 bits long, however.)

The fact that the flag has a unique binary pattern would seem to limit the contents of the frame since a flag pattern might inadvertently occur within the frame. This would cause the receiver to think the closing flag was received, invalidating the frame. SDLC handles this situation through a technique called zero bit insertion. This techniques specifies that within a frame a binary 0 be inserted by the transmitter after any succession of five contiguous binary 1s. Thus, no pattern of 01111110 is ever transmitted by chance. On the receiving end, after the opening flag is detected, the receiver removes any 0 following 5 consecutive 1s. The inserted and deleted 0s are not counted for error determination.

Before discussing the address field, an explanation of the roles of an SDLC station is in order. SDLC specifies two types of stations: primary and secondary. The primary is the control station for the data link and thus has responsibility of the overall network. There is only one predetermined primary station, all other stations on the link assume the secondary station role. In general, a secondary station speaks only when spoken to. In other words, the primary polls the secondaries for responses. In order to specify a specific secondary, each secondary is assigned a unique 8-bit address. It is this address that is used in the frame's address field.

When the primary transmits a frame to a specific secondary, the address field contains the secondary's address. When responding, the secondary uses its own address in the address field. The primary is never identified. This ensures that the primary knows which of many secondaries is responding since the primary may have many messages outstanding at various secondary stations. In addition to the specific secondary address, an address common to all secondaries may be used for various purposes. (An all 1s address field is usually used for this "All Parties" address.) Even though the primary may use this common address, the secondaries are expected to respond with their unique address. The address field is always the first 8 bits following the opening flag.

The 8 bits following the address field form the control field. The control field embodies the link-level control of SDLC. A detailed explanation of the commands and responses contained in this field is beyond the scope of this application note. Suffice it to say that it is in the control field that the implied acknowledgement is carried out through the use of frame sequence numbers. None of the currently available SDLC single chip controllers utilize the control field. They simply pass it to the processor for analysis. Readers wishing a more detailed explanation of the control field, or of SDLC in general, should consult the IBM documents referenced on the front page overleaf.

In some types of frames, an information field follows the control field. Frames used strictly for link management may or may not contain one. When an information field is used, it is unrestricted in both content and length. This code transparency is made possible because of the zero bit insertion mentioned earlier and the bit-oriented nature of SDLC. Even main memory core dumps may be transmitted because of this capability. This feature is unique to bit-oriented protocols. Like the control field, the information field is not interpreted by the SDLC device; it is merely transferred to and from memory to be operated on and interpreted by the processor.

The final field is the frame check sequence (FCS). The FCS is the 16 bits immediately preceding the closing flag. This 16-bit field is used for error detection through a Cyclic Redundancy Checkword (CRC). The 16-bit transmitted CRC is the complement of the remainder obtained when the A, C, and I fields are "divided" by a generating polynomial. The receiver accumulates the A, C, and I fields and also the FCS into its internal CRC register. At the closing flag, this register contains one particular number for an error-free reception. If this number is not obtained, the frame was received in error and should be discarded. Discarding the frame causes the station to not update its frame sequence numbering. This results in a retransmission after the station sends an acknowledgement from previous frames. [Unlike all other fields, the FCS is transmitted MSB (Most Significant Bit) first. The A, C, and I fields are transmitted LSB (Least Significant Bit) first.] The details of how the FCS is generated and checked is beyond the scope of this application note and since all single component SDLC controllers handle this function automatically, it is usually sufficient to know only that an error has or has not occurred. The IBM documents contain more detailed information for those readers desiring it.

The closing flag terminates the frame. When the closing flag is received, the receiver knows that the preceding 16 bits constitute the FCS and that any bits between the control field and the FCS constitute the information field.

SDLC does not support an interframe time-fill character such as the SYN character in Bi-Sync. If an unusual condition occurs while transmitting, such as data is not available in time from memory or CTS (Clear-to-Send) is lost from the modem, the transmitter aborts the frame by sending an Abort character to notify the receiver to invalidate the frame. The Abort character consists of eight contiguous 1s sent without zero bit insertion. Intraframe time-fill consists of either flags, Abort characters, or any combination of the two.

While the Abort character protects the receiver from transmitted errors, errors introduced by the transmission medium are discovered at the receiver through the FCS check and a check for invalid frames. Invalid frames are those which are not bounded by flags or are too short, that is, less than 32 bits between flags. All invalid frames are ignored by the receiver.

Although SDLC is a synchronous protocol, it provides an optional feature that allows its use on basically asynchronous data links—NRZI (Non-Return-to-Zero-Inverted) coding. NRZI coding specifies that the signal condition does not change for transmitting a binary 1, while a binary 0 causes a change of state. Figure 2 illustrates NRZI coding compared to the normal NRZ. NRZI coding guarantees that an active line will have a transition at least every 5-bit times; long strings of zeroes cause a transition every bit time, while long strings of 1s are broken up by zero bit insertion. Since asynchronous operation requires that the receiver sampling clock be derived from the received data, NRZI encoding plus zero bit insertion make the design of clock recovery circuitry easier.

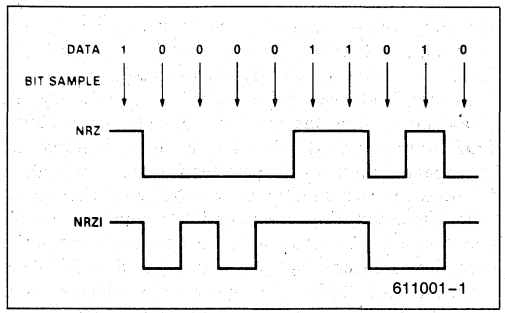


Figure 2. NRZI vs NRZ Encoding

All of the previous discussion has applied to SDLC on either point-to-point or multi-point data networks. SDLC (but not HDLC) also includes specification for a loop configuration. Figure 3 compares these three configurations. IBM uses this loop configuration in its 3650 Retail Store System. It consists of a single loop controller station with one or more down-loop secondary stations. Communications on a loop rely on the secondary stations repeating a received message down loop with a delay of one bit time. The reason for the one bit delay will be evident shortly.

Loop operation defines a new special character: the EOP (End-of-Poll) character which consists of a 0 followed by 7 contiguous, non-zero bit inserted, ones. After the loop controller transmits a message, it idles the line (sends all 1s). The final zero of the closing flag plus the first 7 1s of the idle form an EOP character. While

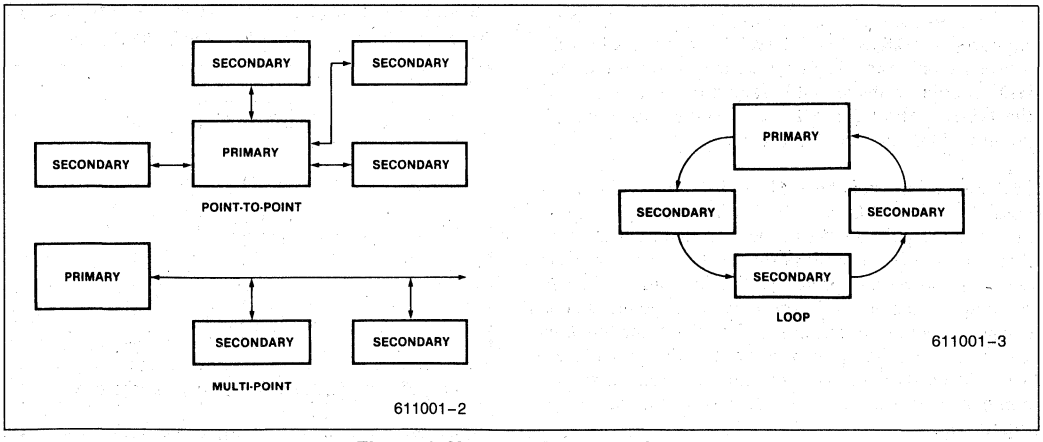


Figure 3. Network Configurations

repeating, the secondaries monitor their incoming line for an EOP character. When an EOP is detected, the secondary checks to see if it has a message to transmit. If it does, it changes the seventh 1 to a 0 (the one bit delay allows time for this) and repeats the modified EOP (now alias flag). After this flag is transmitted, the secondary terminates its repeater function and inserts its message (with multiple preceding flags if necessary). After the closing flag, the secondary resumes its one bit delay repeater function. Notice that the final zero of the secondary's closing flag plus the repeated 1s from the controller form an EOP for the next down-loop secondary, allowing it to insert a message if it desires.

One might wonder if the secondary missed any messages from the controller while it was inserting its own message. It does not. Loop operation is basically half-duplex. The controller waits until it receives an EOP before it transmits its next message. The controller's reception of the EOP signifies that the original message has propagated around the loop followed by any messages inserted by the secondaries. Notice that secondaries cannot communicate with one another directly, all secondary-to-secondary communication takes place by way of the controller.

Loop protocol does not utilize the normal Abort character. Instead, an abort is accomplished by simply transmitting a flag character. Down loop, the receiver sees the abort as a frame which is either too short (if the abort occurred early in the frame) or one with an FCS error. Either results in a discarded frame. For more details on loop operation, please refer to the IBM documents referenced earlier.

Another protocol very similar to SDLC which the 8273 supports is HDLC (High-Level Data Link Control). There are only three basic differences between the two: HDLC offers extended address and control fields, and the HDLC Abort character is 7 contiguous 1s as opposed to SDLC's 8 contiguous 1s.

Extended addressing, beyond the 256 unique addresses possible with SDLC, is provided by using the address field's least significant bit as the extended address modifier. The receiver examines this bit to determine if the octet should be interpreted as the final address octet. As long as the bit is 0, the octet that contains it is considered an extended address. The first time the bit is a 1, the receiver interprets that octet as the final address octet. Thus the address field may be extended to any number of octets. Extended addressing is illustrated in Figure 4a.

A similar technique is used to extend the control field although the extension is limited to only one extra control octet. Figure 4b illustrates control field extension.

Those readers not yet asleep may have noticed the similarity between the SDLC loop EOP character (a 0 followed by 7 1s) and the HDLC Abort (7 1s). This possible incompatibility is neatly handled by the HDLC protocol not specifying a loop configuration.

This completes our brief discussion of the SDLC/HDLC protocols. Now let us turn to the 8273 in particular and discuss its hardware aspects through an explanation of the block diagram and generalized system schematics.

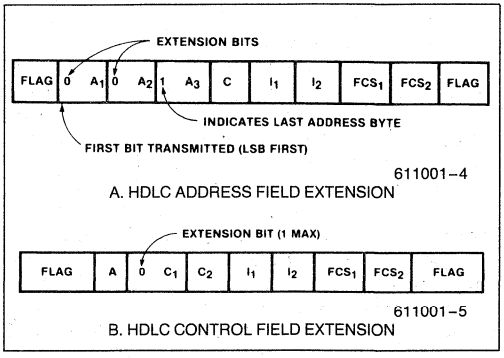


Figure 4

BASIC 8273 OPERATION

It will be helpful for the following discussions to have some idea of the basic operation of the 8273. Each operation, whether it is a frame transmission, reception or port read, etc., is comprised of three phases: the Command, Execution, and Result phases. Figure 5 shows the sequence of these phases. As an illustration of this sequence, let us look at the transmit operation.

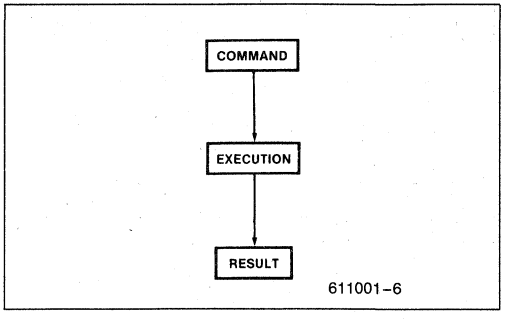


Figure 5. 8273 Operational Phases

When the CPU decides it is time to transmit a frame, the Command phase is entered by the CPU issuing a Transmit Frame command to the 8273. It is not sufficient to just instruct the 8273 to transmit. The frame level command structure sometimes requires more information such as frame length and address and control field content. Once this additional information is sup-

plied, the Command phase is complete and the Execution phase is entered. It is during the Execution phase that the actual operation, in this case a frame transmission, takes place. The 8273 transmits the opening flag, A and C fields, the specified number of I field bytes, inserts the FCS, and closes with the closing flag. Once the closing flag is transmitted, the 8273 leaves the Execution phase and begins the Result phase. During the Result phase the 8273 notifies the CPU of the outcome of the command by supplying interrupt results. In this case, the results would be either that the frame is complete or that some error condition causes the transmission to be aborted. Once the CPU reads all of the results (there is only one for the Transmit Frame command), the Result phase and consequently the operation, is complete. Now that we have a general feeling for the operation of the 8273, let us discuss the 8273 in detail.

HARDWARE ASPECTS OF THE 8273

The 8273 block diagram is shown in Figure 6. It consists of two major interfaces: the CPU module interface and the modem interface. Let's discuss each interface separately.

CPU Interface

The CPU interface consists of four major blocks: Control/Read/Write logic (C/R/W), internal registers, data transfer logic, and data bus buffers.

The CPU module utilizes the C/R/W logic to issue commands to the 8273. Once the 8273 receives a command and executes it, it returns the results (good/bad completion) of the command by way of the C/R/W logic. The C/R/W logic is supported by seven registers which are addressed via the A_0, A_1, $\overline{RD}$, and $\overline{WR}$ signals, in addition to $\overline{CS}$. The A_0 and A_1 signals are generally derived from the two low order bits of the CPU module address bus while $\overline{RD}$ and $\overline{WR}$ are the normal I/O Read and Write signals found on the system control bus. Figure 7 shows the address of each register using the C/R/W logic. The function of each register is defined as follows:

Address Inputs		Control Inputs	
A_1	A_0	$\overline{CS}\bullet\overline{RD}$	$\overline{CS}\bullet\overline{WR}$
0	0	Status	Command
0	1	Result	Parameter
1	0	TxI/R	Test Mode
1	1	RxI/R	—

Figure 7. 8273 Register Selection

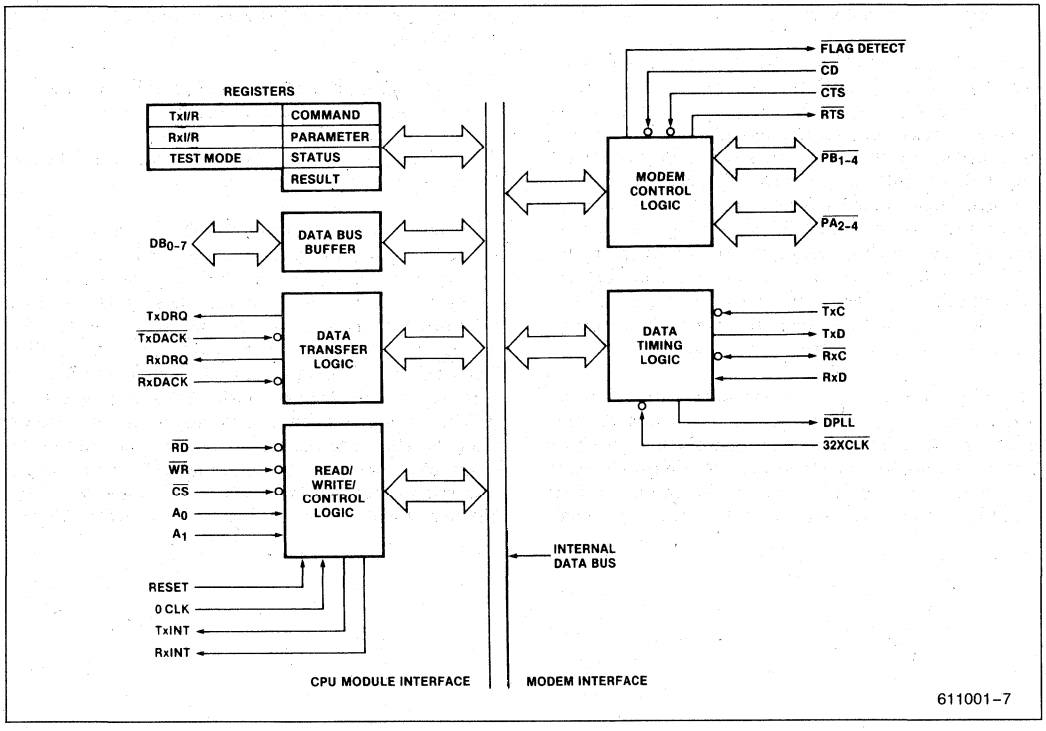


Figure 6. 8273 Block Diagram

Command—8273 operations are initiated by writing the appropriate command byte into this register.

Parameter—Many commands require more information than found in the command itself. This additional information is provided by way of the parameter register.

Immediate Result (Result)—The completion information (results) for commands which execute immediately are provided in this register.

Transmit Interrupt Result (TxI/R)—Results of transmit operations are passed to the CPU in this register.

Receiver Interrupt Result (RxI/R)—Receive operation results are passed to the CPU via this register.

Status—The general status of the 8273 is provided in this register. The Status register supplies the handshaking necessary during various phases of the 8273 operation.

Test Mode—This register provides a software reset function for the 8273.

The commands, parameters, and bit definition of these registers are discussed in the following software section. Notice that there are not specific transmit or receive data registers. This feature is explained in the data transfer logic discussion.

The final elements of the C/R/W logic are the interrupt lines (RxINT and TxINT). These lines notify the CPU module that either the transmitter or the receiver requires service; i.e., results should be read from the appropriate interrupt result register or a data transfer is required. The interrupt request remains active until all the associated interrupt results have been read or the data transfer is performed. Though using the interrupt lines relieves the CPU module of the task of polling the 8273 to check if service is needed, the state of each interrupt line is reflected by a bit in the Status register and non-interrupt driven operation is possible by examining the contents of these bits periodically.

The 8273 supports two independent data interfaces through the data transfer logic; receive data and transmit data. These interfaces are programmable for either DMA or non-DMA data transfers. While the choice of the configuration is up to the system designer, it is based on the intended maximum data rate of the communications channel. Figure 8 illustrates the transfer rate of data bytes that are acquired by the 8273 based on link data rate. Full-duplex data rates above 9600 baud usually require DMA. Slower speeds may or may not require DMA depending on the task load and interrupt response time of the processor.

Figure 9 shows the 8273 in a typical DMA environment. Notice that a separate DMA controller, in this case the Intel 8257, is required. The DMA controller supplies the timing and addresses for the data transfers while the 8273 manages the requesting of transfers and the actual counting of the data block lengths. In this case, elements of the data transfer interface are:

TxDRQ: Transmit DMA Request—Asserted by the 8273, this line requests a DMA transfer from memory to the 8273 for transmit.

TxDACK: Transmit DMA Acknowledge—Returned by the 8257 in response to TxDRQ, this line notifies the 8273 that a request has been granted, and provides access to the transmitter data register.

RxDRQ: Receive DMA Request—Asserted by the 8273, it requests a DMA transfer from the 8273 to memory for a receive operation.

RxDACK: Receive DMA Acknowledge—Returned by the 8257, it notifies the 8273 that a receive DMA cycle has been granted, and provides access to the receiver data register.

RD: Read—Supplied by the 8257 to indicate data is to be read from the 8273 and placed in memory.

WR: Write—Supplied by the 8257 to indicate data is to be written to the 8273 from memory.

To request a DMA transfer the 8273 raises the appropriate DMA request line; let us assume it is a transmitter request (TxDRQ). Once the 8257 obtains control of the system bus by way of its HOLD and HLDA (hold acknowledge) lines, it notifies the 8273 that TxDRQ has been granted by returning $\overline{\text{TxDACK}}$ and $\overline{\text{WR}}$. The $\overline{\text{TxDACK}}$ and $\overline{\text{WR}}$ signals transfer data to the 8273 for a transmit, independent of the 8273 chip select pin ($\overline{\text{CS}}$). A similar sequence of events occurs for receiver requests. This "hard select" of data into the transmitter or out of the receiver alleviates the need for the normal transmit and receive data registers addressed by a combination of address lines, CS, and WR or RD. Competi-

tive devices that do not have this "hard select" feature require the use of an external multiplexer to supply the correct inputs for register selection during DMA. (Do not forget that the SDLC controller sees both the addresses and control signals supplied by the DMA controller during DMA cycles.) Let us look at typical frame transmit and frame receive sequences to better see how the 8273 truly manages the DMA data transfer.

Before a frame can be transmitted, the DMA controller is supplied, by the CPU, the starting address for the desired information field. The 8273 is then commanded to transmit a frame. (Just how this is done is covered later during our software discussion.) After the command, but before transmission begins, the 8273 needs a little more information (parameters). Four parameters are required for the transmit frame command: the address field byte, the control field byte, and two bytes which are the least significant and most significant bytes of the information field byte length. Once all four parameters are loaded, the 8273 makes RTS (Request-to-Send) active and waits for CTS (Clear-to-Send) to go active. Once CTS is active, the 8273 starts the frame transmission. While the 8273 is transmitting the opening flag, address field, and control field; it starts making transmitter DMA requests. These requests continue at character (byte) boundaries until the pre-loaded number of bytes of information field have been transmitted. At this point the requests stop, the FCS and closing flag are transmitted, and the TxINT line is raised, signaling the CPU that the frame transmission is complete. Notice that after the initial command and parameter loading, absolutely no CPU intervention was required (since DMA is used for data transfers) until the entire frame was transmitted. Now let's look at a frame reception.

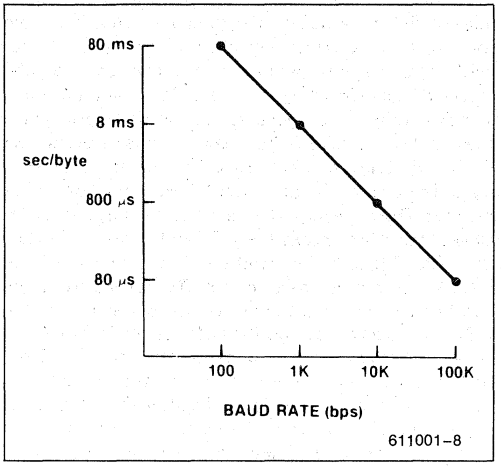


Figure 8. Byte Transfer Rate vs Baud Rate

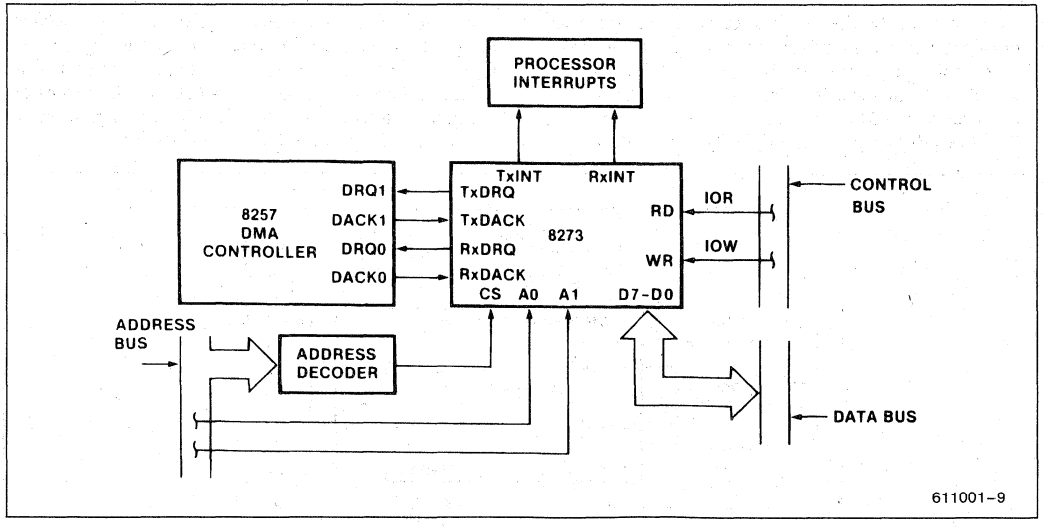


Figure 9. DMA, Interrupt-Driven System

The receiver operation is very similar. Like the initial transmit sequence, the DMA controller is loaded with a starting address for a receiver data buffer and the 8273 is commanded to receive. Unlike the transmitter, there are two different receive commands: General Receive, where all received frames are transferred to memory, and Selective Receive, where only frames having an address field matching one of two preprogrammed 8273 address fields are transferred to memory. Let's assume for now that we want to general receive. After the receive command, two parameters are required before the receiver becomes active: the least significant and most significant bytes of the receiver buffer length. Once these bytes are loaded, the receiver is active and the CPU may return to other tasks. The next frame appearing at the receiver input is transferred to memory using receiver DMA requests. When the closing flag is received, the 8273 checks the FCS and raises its RxINT line. The CPU can then read the results which indicate if the frame was error-free or not. (If the received frame had been longer than the pre-loaded buffer length, the CPU would have been notified of that occurrence earlier with a receiver error interrupt. The command description section contains a complete list of error conditions.) Like the transmit example, after the initial command, the CPU is free for other tasks until a frame is completely received. These examples have illustrated the 8273's management of both the receiver and transmitter DMA channels.

It is possible to use the DMA data transfer interface in a non-DMA interrupt-driven environment. In this case, 4 interrupt levels are used: one each for TxINT and RxINT, and one each for TxDRQ and RxDRQ. This configuration is shown in Figure 10. This configuration offers the advantages that no DMA controller is required and data requests are still separated from result (completion) requests. The disadvantages of the configuration are that 4 interrupt levels are required and that the CPU must actually supply the data transfers. This, of course, reduces the maximum data rate compared to the configuration based strictly on DMA. This system could use an Intel 8259 8-level Priority Interrupt Controller to supply a vectored CALL (subroutine) address based on requests on its inputs. The 8273 transmitter and receiver make data requests by raising the respective DRQ line. The CPU is interrupted by the 8259 and vectored to a data transfer routine. This routine either writes (for transmit) or reads (for receive) the 8273 using the respective TxDACK or RxDACK line. The DACK lines serve as "hard" chip selects into and out of the 8273. $\overline{\text{TxDACK}}$ + $\overline{\text{WR}}$ writes data into the 8273 for transmit. $\overline{\text{RxDACK}}$ + $\overline{\text{RD}}$ reads data from the 8273 for receive.) The CPU is notified of operation completion and results by way of TxINT and RxINT lines. Using the 8273, and the 8259, in this way, provides a very effective, yet simple, interrupt-driven interface.

Figure 11 illustrates a system very similar to that described above. This system utilizes the 8273 in a non-DMA data transfer mode as opposed to the two DMA approaches shown in Figures 9 and 10. In the non-DMA case, data transfer requests are made on the TxINT and RxINT lines. The DRQ lines are not used. Data transfer requests are separated from result requests by a bit in the Status register. Thus, in response to an interrupt, the CPU reads the Status register and branches to either a result or a data transfer routine based on the status of one bit. As before, data transfers are made via using the DACK lines as chip selects to the transmitter and receiver data registers.

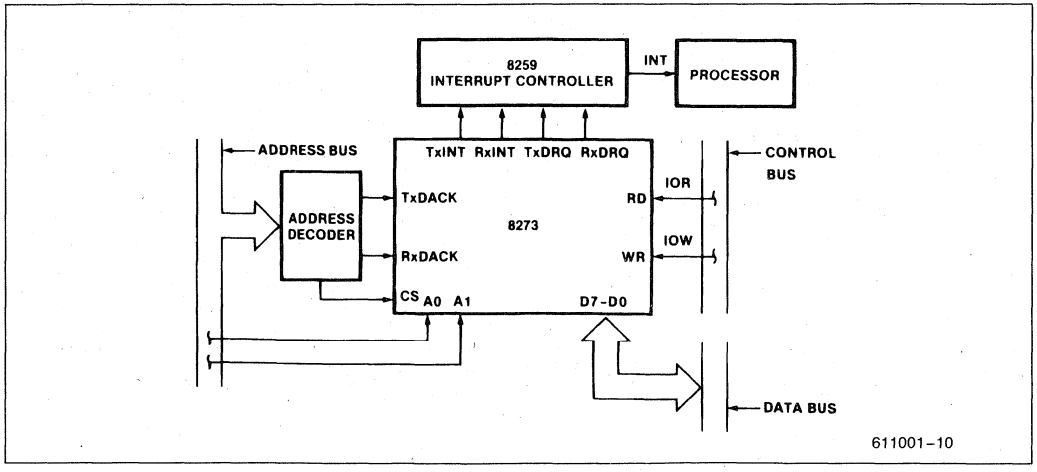


Figure 10. Interrupt-Based DMA System

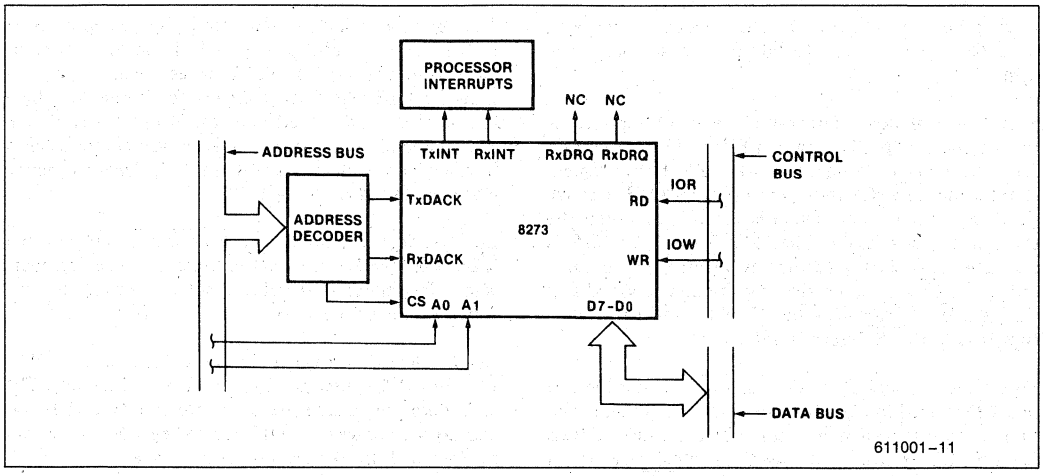


Figure 11. Non-DMA Interrupt-Driven System

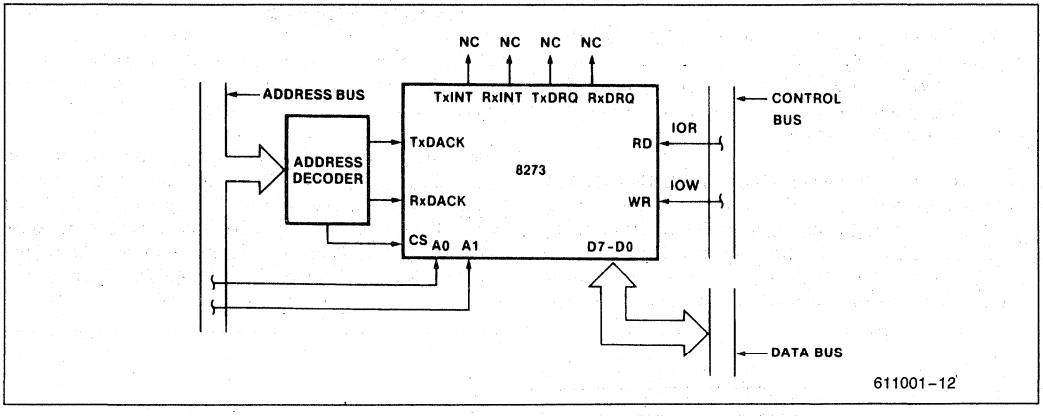


Figure 12. Polled System

Figure 12 illustrates the simplest system of all. This system utilizes polling for all data transfers and results. Since the interrupt pins are reflected in bits in the Status register, the software can read the Status register periodically looking for one of these to be set. If it finds an INT bit set, the appropriate Result Available bit is examined to determine if the "interrupt" is a data transfer or completion result. If a data transfer is called for, the DACK line is used to enter or read the data from the 8273. If the interrupt is a completion result, the appropriate result register is read to determine the good/bad completion of the operation.

The actual selection of either DMA or non-DMA modes is controlled by a command issued during initialization. This command is covered in detail during the software discussion.

The final block of the CPU module interface is the Data Bus Buffer. This block supplies the tri-state, bidirectional data bus interface to allow communication to and from the 8273.

Modem Interface

As the name implies, the modem interface is the modem side of the 8273. It consists of two major blocks: the modem control block and the serial data timing block.

The modem control block provides both dedicated and user-defined modem control functions. All signals supported by this interface are active low so that EIA in-

verting drivers (MC1488) and inverting receivers (MC1489) may be used to interface to standard modems.

Port A is a modem control input port. Its representation on the data bus is shown in Figure 13. Bits D_0 and D_1 have dedicated functions. D_0 reflects the logical state of the $\overline{CTS}$ (Clear-to-Send) pin. [If $\overline{CTS}$ is active (low), D_0 is a 1.] This signal is used to condition the start of a transmission. The 8273 waits until $\overline{CTS}$ is active before it starts transmitting a frame. While transmitting, if $\overline{CTS}$ goes inactive, the frame is aborted and the CPU is interrupted. When the CPU reads the interrupt result, a $\overline{CTS}$ failure is indicated.

D_1 reflects the logical state of the $\overline{CD}$ (Carrier Detect) pin. $\overline{CD}$ is used to condition the start of a frame reception. $\overline{CD}$ must be active in time for a frame's address field. If $\overline{CD}$ is lost (goes inactive) while receiving a frame, an interrupt is generated with a $\overline{CD}$ failure result. $\overline{CD}$ may go inactive between frames.

Bits D_2 thru D_4 reflect the logical state of the $\overline{PA_2}$ thru $\overline{PA_4}$ pins respectively. These inputs are user defined. The 8273 does not interrogate or manipulate these bits. Bits D_5, D_6, and D_7 are not used and each is read as a 1 for a Read Port A command.

Port B is a modem control output port. Its data bus representation is shown in Figure 14. As in Port A, the bit values represent the logical condition of the pins. D_0 and D_5 are dedicated function outputs. D_0 represents the $\overline{RTS}$ (Request-to-Send) pin. $\overline{RTS}$ is normally used to notify the modem that the 8273 wishes to transmit. This function is handled automatically by the 8273. If $\overline{RTS}$ is inactive (pin is high) when the 8273 is commanded to transmit, the 8273 makes it active and then waits for $\overline{CTS}$ before transmitting the frame. One byte time after the end of the frame, the 8273 returns $\overline{RTS}$ to its inactive state. However, if $\overline{RTS}$ was active when a transmit command is issued, the 8273 leaves it active when the frame is complete.

Bit D_5 reflects the state of the $\overline{Flag\ Detect}$ pin. This pin is activated whenever an active receiver sees a flag character. This function is useful to activate a timer for line activity timeout purposes.

Bits D_1 thru D_4 provide four user-defined outputs. Pins $\overline{PB_1}$ thru $\overline{PB_4}$ reflect the logical state of these bits. The 8273 does not interrogate or manipulate these bits. D_6 and D_7 are not used. In addition to being able to output to Port B, Port B may be read using a Read Port B command. All Modem control output pins are forced high on reset. (All commands mentioned in this section are covered in detail later.)

The final block to be covered is the serial data timing block. This block contains two sections: the serial data logic and the digital phase locked loop (DPLL).

Elements of the serial data logic section are the data pins, TxD (transmit data output) and RxD (receive data input), and the respective data clocks, $\overline{TxC}$ and $\overline{RxC}$. The transmit and receive data is synchronized by the $\overline{TxC}$ and $\overline{RxC}$ clocks. Figure 15 shows the timing for these signals. The leading edge (negative transition)

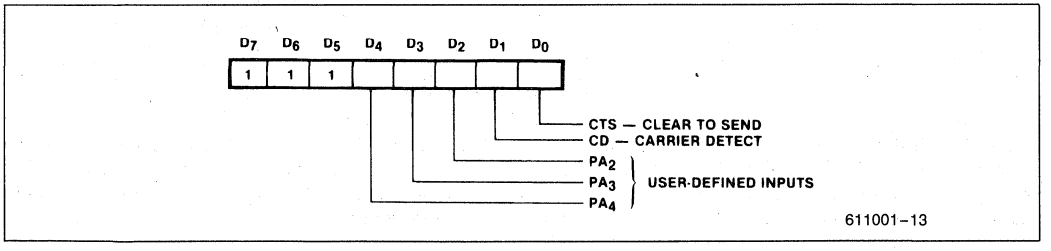


Figure 13. Port A (Input) Bit Definition

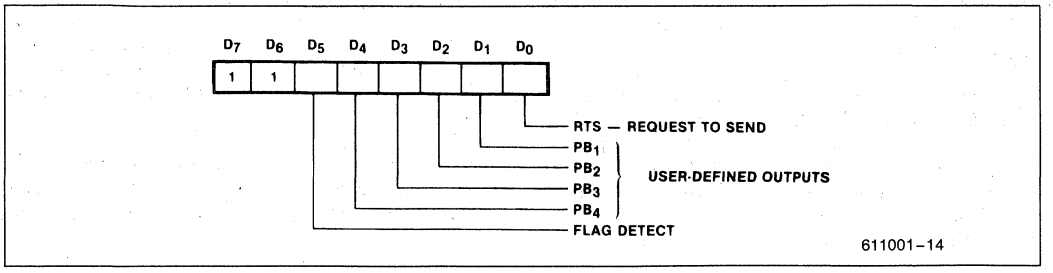


Figure 14. Port B (Output) Bit Definition

of $\overline{TxC}$ generates new transmit data and the trailing edge (positive transition) of $\overline{RxC}$ is used to capture the receive data.

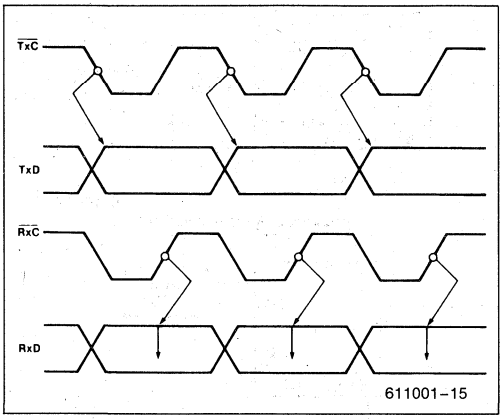


Figure 15. Transmit/Receive Timing

It is possible to reconfigure this section under program control to perform diagnostic functions; both data and clock loopback are available. In data loopback mode, the TxD pin is internally routed to the RxD pin. This allows simple board checkout since the CPU can send an SDLC message to itself. (Note that transmitted data will still appear on the TxD pin.)

When data loopback is utilized, the receiver may be presented incorrect sample timing ($\overline{RxC}$) by the external circuitry. Clock loopback overcomes this problem by allowing the internal routing of $\overline{TxC}$ and $\overline{RxC}$. Thus the same clock used to transmit the data is used to receive it. Examination of Figure 15 shows that this method ensures bit synchronism. The final element of the serial data logic is the Digital Phase Locked Loop.

The DPLL provides a means of clock recovery from the received data stream. This feature allows the 8273 to interface without external synchronizing logic to low cost asynchronous modems (modems which do not supply clocks). It also makes the problem of clock timing in loop configurations trivial.

To use the DPLL, a clock at 32 times the required baud rate must be supplied to the $\overline{32\times\ CLK}$ pin. This clock provides the interval that the DPLL samples the received data. The DPLL uses the 32× clock and the received data to generate a pulse at the $\overline{DPLL}$ output pin. This $\overline{DPLL}$ pulse is positioned at the nominal center of the received data bit cell. Thus the $\overline{DPLL}$ output may be wired to $\overline{RxC}$ and/or $\overline{TxC}$ to supply the data timing. The exact position of the pulse is varied depending on the line noise and bit distortion of the received data. The adjustment of the $\overline{DPLL}$ position is determined according to the rules outlined in Figure 16.

Adjustments to the sample phase of $\overline{DPLL}$ with respect to the received data is made in discrete increments. Referring to Figure 16, following the occurrence of $\overline{DPLL}$

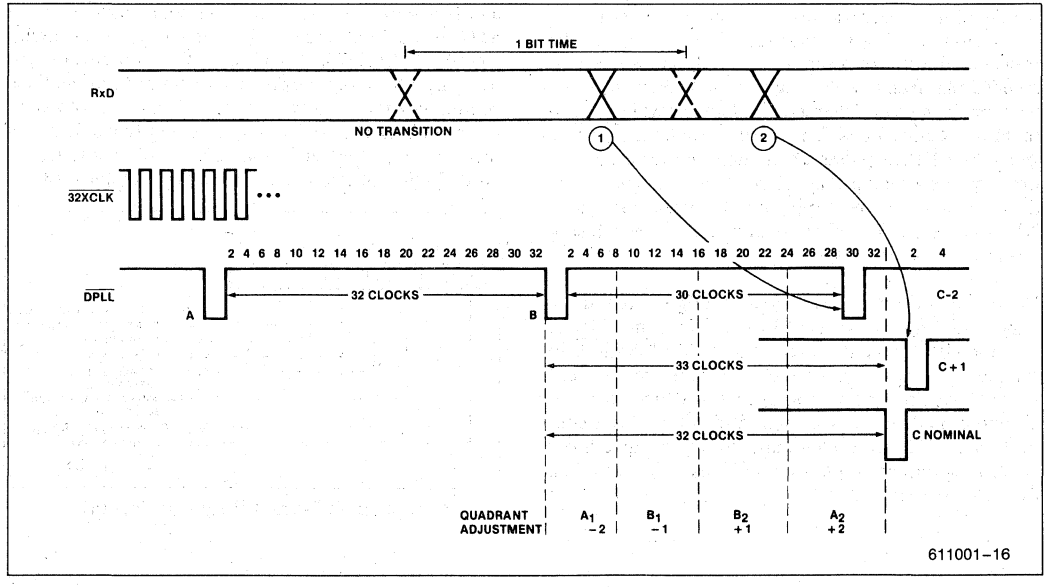


Figure 16. DPLL Phase Adjustments

pulse A, the DPLL counts $\overline{32\times\ CLK}$ pulses and examines the received data for a data edge. Should no edge be detected in 32 pulses, the $\overline{DPLL}$ positions the next $\overline{DPLL}$ pulse (B) at 32 clock pulses from pulse A. Since no new phase information is contained in the data stream, the sample phase is assumed to be at nominal $1\times$ baud rate. Now assume a data edge occurs after $\overline{DPLL}$ pulse B. The distance from B to the next pulse C is influenced according to which quadrant (A_1, B_1, B_2, or A_2) the data edge falls in. (Each quadrant represents 8 $\overline{32\times\ CLK}$ times.) For example, if the edge is detected in quadrant A_1, it is apparent that pulse B was too close to the data edge and the time to the next pulse must be shortened. The adjustment for quadrant A_1 is specified as -2. Thus, the next $\overline{DPLL}$ pulse, pulse C, is positioned 32 $-$ 2 or 30 $\overline{32\times\ CLK}$ pulses following $\overline{DPLL}$ pulse B. This adjustment moves pulse C closer to the nominal bit center of the next received data cell. A data edge occurring in quadrant B_2 would have caused the adjustment to be small, namely 32 + 1 or 33 $\overline{32\times\ CLK}$ pulses. Using this technique, the $\overline{DPLL}$ pulse converges to the nominal bit center within 12 data transitions, worse case—4-bit times adjusting through quadrant A_1 or A_2 and 8-bit times adjusting through B_1 or B_2.

When the receive data stream goes idle after 15 ones, DPLL pulses are generated at 32 pulse intervals of the $32\times$ CLK. This feature allows the DPLL pulses to be used as both transmitter and receiver clocks.

In order to guarantee sufficient transitions of the received data to enable the $\overline{DPLL}$ to lock, NRZI encoding of the data is recommended. This ensures that, within a frame, data transitions occur at least every five bit times—the longest sequence of 1s which may be transmitted with zero bit insertion. It is also recommended that frames following a line idle be transmitted with preframe sync characters which provide a minimum of 12 transitions. This ensures that the $\overline{DPLL}$ is generating $\overline{DPLL}$ pulses at the nominal bit centers in time for the opening flag. (Two 00H characters meet this requirement by supplying 16 transitions with NRZI encoding. The 8273 contains a mode which supplies such a preframe sync.)

Figure 17 illustrates 8273 clock configurations using either synchronous or asynchronous modems. Notice how the $\overline{DPLL}$ output is used for both $\overline{TxC}$ and $\overline{RxC}$ in the asynchronous case. This feature eliminates the need for external clock generation logic where low cost asynchronous modems are used and also allows direct connection of 8273s for the ultimate in low cost data links. The configuration for loop applications is discussed in a following section.

This completes our discussion of the hardware aspects of the 8273. Its software aspects are now discussed.

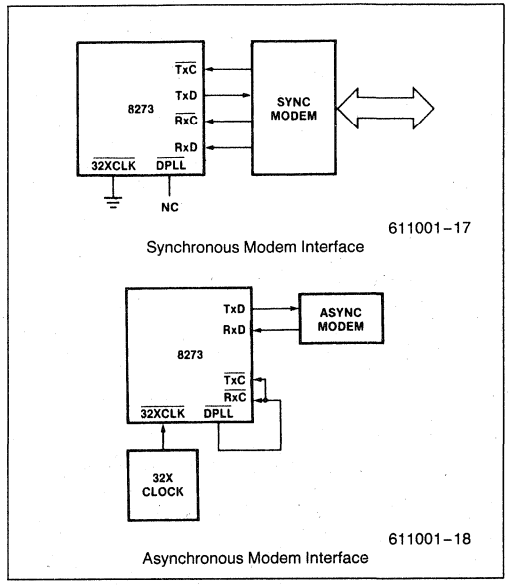


Figure 17. Serial Data Timing Configuration

SOFTWARE ASPECTS OF THE 8273

The software aspects of the 8273 involve the communication of both commands from the CPU to the 8273 and the return of results of those commands from the 8273 to the CPU. Due to the internal processor architecture of the 8273, this CPU-8273 communication is basically a form of interprocessor communication. Such communication usually requires a form of protocol of its own. This protocol is implemented through use of handshaking supplied in the 8273 Status register. The bit definition of this register is shown in Figure 18.

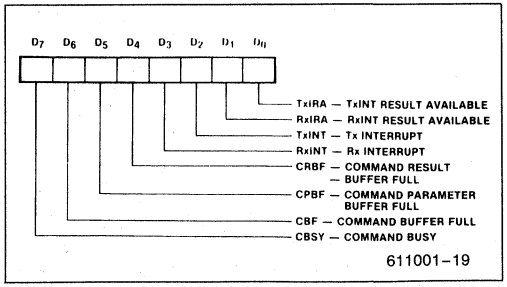


Figure 18. Status Register Format

CBSY: Command Busy—CBSY indicates when the 8273 is in the command phase. CBSY is set when the CPU writes a command into the Command register, starting the Command phase. It is reset when the last parameter is deposited in the Parameter register and accepted by the 8273, completing the Command phase.

CBF: Command Buffer Full—When set, this bit indicates that a byte is present in the Command register. This bit is normally not used.

CPBF: Command Parameter Buffer Full—This bit indicates that the Parameter register contains a parameter. It is set when the CPU deposits a parameter in the Parameter register. It is reset when the 8273 accepts the parameter.

CRBF: Command Result Buffer Full—This bit is set when the 8273 places a result from an immediate type command in the Result register. It is reset when the CPU reads the result from the Result register.

RxINT: Receiver Interrupt—The state of the RxINT pin is reflected by this bit. RxINT is set by the 8273 whenever the receiver needs servicing. RxINT is reset when the CPU reads the results or performs the data transfer.

TxINT: Transmitter Interrupt—This bit is identical to RxINT except action is initiated based on transmitter interrupt sources.

RxIRA: Receiver Interrupt Result Available—RxIRA is set when the 8273 places an interrupt result byte into the RxI/R register. RxIRA is reset when the CPU reads the RxI/R register.

TxIRA: Transmitter Interrupt Result Available—TxIRA is the corresponding Result Available bit for the transmitter. It is set when the 8273 places an interrupt result byte in the TxI/R register and reset when the CPU reads the register.

The significance of each of these bits will be evident shortly. Since the software requirements of each 8273 phase are essentially independent, each phase is covered separately.

Command Phase Software

Recalling the Command phase description in an earlier section, the CPU starts the Command phase by writing a command byte into the 8273 Command register. If further information about the command is required by the 8273, the CPU writes this information into the Parameter register. Figure 19 is a flowchart of the Command phase. Notice that the CBSY and CPBF bits of the Status register are used to handshake the command and parameter bytes. Also note that the chart shows

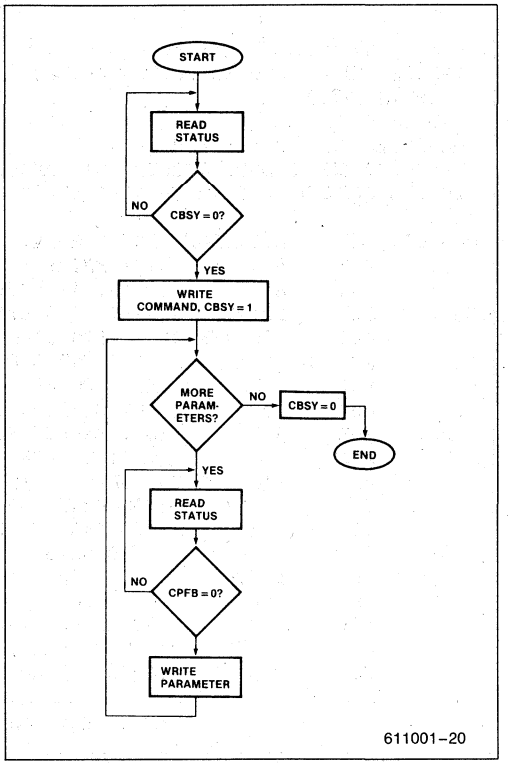


Figure 19. Command Phase Flowchart

that a command may not be issued if the Status register indicates the 8273 is busy (CBSY = 1). If a command is issued while CBSY = 1, the original command is overwritten and lost. (Remember that CBSY signifies the command phase is in progress and not the actual execution of the command.) The flowchart also includes a Parameter buffer full check. The CPU must wait until CPBF = 0 before writing a parameter to the Parameter register. If a parameter is issued while CPBF = 1, the previous parameter is overwritten and lost. An example of command output assembly language software is provided in Figure 20a. This software assumes that a command buffer exists in memory. The buffer is pointed at by the HL register. Figure 20b shows the command buffer structure.

The 8273 is a full duplex device, i.e., both the transmitter and receiver may be executing commands or passing interrupt results at any given time. (Separate Rx and Tx interrupt pins and result registers are provided for this reason.) However, there is only one Command register. Thus, the Command register must be used for only one command sequence at a time and the transmitter and receiver may never be simultaneously in a command phase. A detailed description of the commands and their parameters is presented in a following section.

```
;FUNCTION: COMMAND DISPATCHER
;INPUTS: HL - COMMAND BUFFER ADDRESS
;OUTPUTS: NONE
;CALLS: NONE
;DESTROYS: A,B,H,L,F/F'S
;DESCRIPTION: CMDOUT ISSUES THE COMMAND + PARAMETERS
;IN THE COMMAND BUFFER POINTED AT BY HL
;
CMDOUT: LXI    H,CMDBUF  ;POINT HL AT BUFFER
        MOV    B,M       ;1ST ENTRY IS PAR. COUNT
        INX    H         ;POINT AT COMMAND BYTE
CMD1:   IN     STAT73    ;READ 8273 STATUS
        RLC    ;ROTATE CBSY INTO CARRY
        JC     CMD1      ;WAIT UNTIL CBSY=0
        MOV    A,M       ;MOVE COMMAND BYTE TO A
        OUT    COMM73    ;PUT COMMAND IN COMMAND REG
CMD2:   MOV    A,B       ;GET PARAMETER COUNT
        ANA    A         ;TEST IF ZERO
        RZ     ;IF 0 THEN DONE
        INX    H         ;NOT DONE, SO POINT AT NEXT PAR
        DCR    B         ;DEC PARAMETER COUNT
CMD3:   IN     STAT73    ;READ 8273 STATUS
        ANI    CPBF      ;TEST CPBF BIT
        JNZ    CMD3      ;WAIT UNTIL CPBF IS 0
        MOV    A,M       ;GET PARAMETER FROM BUFFER
        OUT    PARM73    ;OUTPUT PAR TO PARAMETER REG
        JMP    CMD2      ;CHECK IF MORE PARAMETERS
```

Figure 20A. Command Phase Software

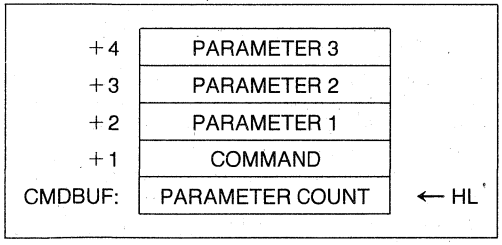


Figure 20B. Command Buffer Format

Execution Phase Software

During the Execution phase, the operation specified by the Command phase is performed. If the system utilizes DMA for data transfers, there is no CPU involvement during this phase, so no software is required. If non-DMA data transfers are used, either interrupts or polling is used to signal a data transfer request.

For interrupt-driven transfers the 8273 raises the appropriate INT pin. When responding to the interrupt,

the CPU must determine whether it is a data transfer request or an interrupt signaling that an operation is complete and results are available. The CPU determines the cause by reading the Status register and interrogating the associated IRA (Interrupt Result Available) bit (TxIRA for TxINT and RxIRA for RxINT). If the IRA = 0, the interrupt is a data transfer request. If the IRA = 1, an operation is complete and the associated Interrupt Result register must be read to determine the completion status (good/bad/etc.). A software interrupt handler implementing the above sequence is presented as part of the Result phase software.

When polling is used to determine when data transfers are required, the polling routine reads the Status register looking for one of the INT bits to be set. When a set INT bit is found, the corresponding IRA bit is examined. Like in the interrupt-driven case, if the IRA = 0, a data transfer is required. If IRA = 1, an operation is complete and the Interrupt Result register needs to be read. Again, example polling software is presented in the next section.

Result Phase Software

During the Result phase the 8273 notifies the CPU of the outcome of a command. The Result phase is initiated by either a successful completion of an operation or an error detected during execution. Some commands such as reading or writing the I/O ports provide immediate results, that is, there is essentially no delay from the issuing of the command and when the result is available. Other commands such as frame transmit, take time to complete so their result is not available immediately. Separate result registers are provided to distinguish these two types of commands and to avoid interrupt handling for simple results.

Immediate results are provided in the Result register. Validity of information in this register is indicated to the CPU by way of the CRBF bit in the Status register. When the CPU completes the Command phase of an immediate command, it polls the Status register waiting until CRBF = 1. When this occurs, the CPU may read the Result register to obtain the immediate result. The Result register provides only the results from immediate commands.

Example software for handling immediate results is shown in Figure 21. The routine returns with the result in the accumulator. The CPU then uses the result as is appropriate.

All non-immediate commands deal with either the transmitter or receiver. Results from these commands are provided in the TxI/R (Transmit Interrupt Result) and RxI/R (Receive Interrupt Result) registers respectively. Results in these registers are conveyed to the CPU by the TxIRA and RxIRA bits of the status register. Results of non-immediate commands consist of one byte result interrupt code indicating the condition for the interrupt and, if required, one or more bytes supplying additional information. The interrupt codes and the meaning of the additional results are covered following the detailed command description.

Non-immediate results are passed to the CPU in response to either interrupts or polling of the Status register. Figure 22 illustrates an interrupt-driven result handler. (Please note that all of the software presented in this application note is not optimized for either speed or code efficiency. They are provided as a guide and to illustrate concepts.) This handler provides for interrupt-driven data transfers as was promised in the last section. Users employing DMA-based transfers do not

```
;FUNCTION: IMDRLT
;INPUTS: NONE
;OUTPUTS: RESULT REGISTER IN A
;CALLS: NONE
;DESTROYS: A, F/F'S
;DESCRIPTION: IMDRLT IS CALLED AFTER A CMDOUT FOR AN
;IMMEDIATE COMMAND TO READ THE RESULT REGISTER
;
IMDRLT: IN    STAT 73   ;READ 8273 STATUS
        ANI   CRBF      ;TEST IF RESULT REG READY
        JZ    IMDRLT    ;WAIT IF CRBF=0
        IN    RESL73    ;READ RESULT REGISTER
        RET   ;RETURN
```

Figure 21. Immediate Result Handler

```
;FUNCTION: RXI - INTERRUPT DRIVEN RESULT/DATA HANDLER
;INPUTS: RCRBUF, RCVPNT
;CALLS: NONE
;OUTPUTS: RCRBUF, RCVPNT
;DESTROYS: NOTHING
;DESCRIPTION: RXI IS ENTERED AT A RECEIVER INTERRUPT.
;THE INTERRUPT IS TESTED FOR DATA TRANSFER (IRA=0)
;OR RESULT (IRA=1).  FOR DATA TRANSFER, THE DATA IS
;PLACED IN A BUFFER AT RCVPNT. RESULTS ARE PLACED IN
;A BUFFER AT RCRBUF.
;A FLAG(RXFLAG) IS SET IF THE INTERRUPT WAS A RESULT.
;(DATA TRANSFER INSTRUCTIONS ARE DENOTED BY (*) AND
;MAYBE ELIMINATED BY USERS USING DMA.
;
RXI:    PUSH    H       ;SAVE HL
        PUSH    PSW     ;SAVE PSW
        PUSH    B       ;SAVE B
        IN      STAT73  ; (*) READ 8273 STATUS
        ANI     RXIRA   ; (*) TEST IRA BIT
        JZ      RXI2    ; (*) IF 0, DATA TRANSFER NEEDED
RXI1:   LHLD    RCRBUF  ;GET RESULT BUFFER POINTER
        IN      STAT73  ;READ 8273 STATUS AGAIN
        ANI     RXINT   ;TEST INT BIT
        JZ      RXI4    ;IF 0, THEN DONE
        IN      STAT73  ;READ 8273 STATUS AGAIN
        ANI     RXIRA   ;TEST IRA AGAIN
        JZ      RXI1    ;LOOP UNTIL RESULT IS READY
        IN      RXIR73  ;READY, READ RXI/R
        MOV     M,A     ;STORE RESULT IN BUFFER
        INX     H       ;BUMP RESULT POINTER
        SHLD    RCRBUF  ;RESTORE BUFFER POINTER
        JMP     RXI1    ;GO BACK TO SEE IF MORE
RXI2:   SHLD    RCVPNT  ; (*) GET DATA BUFFER POINTER
        IN      RCVDAT  ; (*) READ DATA VIA RXDACK
        MOV     M,A     ; (*) STORE DATA IN BUFFER
        INX     H       ; (*) BUMP DATA POINTER
        JMP     RXI3    ; (*) DONE
RXI4:   MVI     A,01H   ;SET RX FLAG TO SHOW COMPLETION
        STA     RXFLAG  ;COMPLETION
RXI3:   POP     B       ;RESTORE BC
        POP     PSW     ;RESTORE PSW
        POP     H       ;RESTORE HL
        EI      ;ENABLE INTERRUPTS
        RET     ;DONE


;FUNCTION: TXI - INTERRUPT DRIVEN RESULT/DATA HANDLER
;INPUTS: TXRBUF, TXPNT, TXFLAG
;OUTPUTS: TXRBUF, TXPNT, TXFLAG
;CALLS: NONE
;DESTROYS: NOTHING
;DESCRIPTION: TXI IS ENTERED AT A TRANSMITTER INTERRUPT.
;THE INTERRUPT IS TESTED BY WAY OF THE IRA BIT TO SEE
;IF A DATA TRANSFER OR RESULT COMPLETION HAS OCCURED.
;FOR DATA TRANSFERS (IRA=0), THE DATA IS OBTAINED FROM
;A BUFFER LOCATION POINTED AT BY TXPNT.  FOR COMPLETION,
;(IRA=1), THE RESULTS ARE READ AND PLACED AT A RESULT
;BUFFER POINTED AT BY TXRBUF, AND THE TXFLAG IS SET
;TO INDICATE TO THE MAIN PROGRAM THAT A OPERATION IS
;COMPLETE. TX OPERATIONS HAVE ONLY ONE RESULT.
;DATA TRANSFER INSTRUCTIONS ARE DENOTED BY (*). THESE
;MAYBE REMOVED BY USERS USING DMA.
;
TXI:    PUSH    H       ;SAVE HL
        PUSH    PSW     ;SAVE PSW
        IN      STAT73  ; (*) READ 8273 STATUS
        ANI     TXIRA   ; (*) TEST TXIRA BIT
        JZ      TXI2    ; (*) IF 0, DATA TRANSFER
        IN      TXIR73  ;1, THEN READ TXIR
        LHLD    TXRBUF  ;GET RESULT BUFFER POINTER
        MOV     M,A     ;STORE RESULT IN BUFFER
        INX     H       ;BUMP RESULT POINTER
        SHLD    TXRBUF  ;RESTORE RESULT POINTER
        MVI     A,01H   ;SET TXFLAG TO SHOW COMPLETION
        STA     TXFLAG  ;SET FLAG
TXI1:   POP     PSW     ;RESTORE PSW
        POP     H       ;RESTORE HL
        EI      ;ENABLE INTERRUPTS
        RET     ;DONE
TXI2:   LHLD    TXPNT   ; (*) GET DATA POINTER
        MOV     A,M     ; (*) GET DATA FROM BUFFER
        OUT     TXDATA  ; (*) OUTPUT TO 8273 VIA TXDACK
        INX     H       ; (*) BUMP DATA POINTER
        SHLD    TXPNT   ; (*) RESTORE POINTER
        JMP     TXI1    ; (*) RETURN AFTER RESTORE
```

611001-61

Figure 22. Interrupt-Driven Result Handlers with Non-DMA Data Transfers

need the lines where the IRA bit is tested for zero. (These lines are denoted by an asterisk in the comments column.) Note that the INT bit is used to determine when all results have been read. All results must be read. Otherwise, the INT bit (and pin) will remain high and further interrupts may be missed. These routines place the results in a result buffer pointed at by RCRBUF and TxRBUF.

A typical result handler for systems utilizing polling is shown in Figure 23. Data transfers are also handled by this routine. This routine utilizes the routines of Figure 22 to handle the results.

At this point, the reader should have a good conceptual feel about how the 8273 operates. It is now time for the particulars of each command to be discussed.

```
;FUNCTION: POLOP
;INPUTS: NONE
;OUTPUTS: C=0 (NO STATUS), =1 (RX COMPLETION),
;    =2 (TX COMPLETION), =3 (BOTH)
;CALLS: TXI, RXI
;DESTROYS: B,C
;DESCRIPTION: POLOP IS CALLED TO POLL THE 8273 FOR
;DATA TRANSFERS AND COMPLETION RESULTS.  THE
;ROUTINES TXI AND RXI ARE USED FOR THE ACTUAL
;TRANSFERS AND BUFFER WORK.  POLOP RETURNS
;THE STATUS OF THEIR ACTION.
;
POLOP:  PUSH    PSW     ;SAVE PSW
        MVI     C,00H   ;CLEAR C
POLOP1: IN      STAT73  ;READ 8273 STATUS
        ANI     INT     ;ARE TXINT OR RXINT SET?
        JZ      PEXIT   ;NO, EXIT
        IN      STAT73  ;READ 8273 STATUS
        ANI     RXINT   ;TEST RX INT
        JNZ     RXIC    ;YES, GO SERVICE RX
        CALL    TXI     ;MUST BE TX, GO SERVICE IT
        LDA     TXFLAG  ;GET TX FLAG
        CPI     01H     ;WAS IT A COMPLETION? (01)
        JNZ     PEXIT   ;NO, SO JUST EXIT
        INR     C       ;YES, UPDATE C
        INR     C
        JMP     POLOP1  ;TRY AGAIN
;
RXIC:   CALL    RXI     ;GO SERVICE RX
        LDA     RXFLAG  ;GET RX FLAG
        CPI     01H     ;WAS IT A COMPLETION? (01)
        JNZ     PEXIT   ;NO, SO JUST EXIT
        INR     C       ;YES, UPDATE C
        JMP     POLOP1  ;TRY AGAIN
;
PEXIT:  POP     PSW     ;RESTORE PSW
        RET     ;RETURN WITH COMP. STATUS IN C
```

611001-62

Figure 23. Polling Result Handler

8273 COMMAND DESCRIPTION

In this section, each command is discussed in detail. In order to shorten the notation, please refer to the command key in Table 1. The 8273 utilizes five different command types: Initialization/Configuration, Receive, Transmit, Reset, and Modem Control.

Table 1. Command Summary Key

B_0, B_1	—LSB and MSB of Receive Buffer Length
R_0, R_1	—LSB and MSB of Received Frame Length
L_0, L_1	—LSB and MSB of Transmit Frame Length
A_1, A_2	—Match Addresses for Selective Receive
RIC	—Receiver Interrupt Result Code
TIC	—Transmitter Interrupt Result Code
A	—Address Field of Received Frame
C	—Control Field of Received Frame

Initialization/Configuration Commands

The Initialization/Configuration commands manipulate registers internal to the 8273 that define the various operating modes. These commands either set or reset specified bits in the registers depending on the type of command. One parameter is required. Set commands perform a logical OR operation of the parameter (mask) and the internal register. This mask contains 1s where register bits are to be set. A "0" in the mask causes no change in the corresponding register bit. Reset commands perform a logical AND operation of the parameter (mask) and the internal register, i.e., the mask is "0" to reset a register bit and a "1" to cause no change. Before presenting the commands, the register bit definitions are discussed.

Operating Mode Register (Figure 24)

D_7–D_6: *Not Used*—These bits must not be manipulated by any command; i.e., D_7–D_6 must be 0 for the Set command and 1 for the Reset command.

D_5: *HDLC Abort*—When this bit is set, the 8273 will interrupt when 7 1s (HDLC Abort) are received by an active receiver. When reset, an SDLC Abort (8 1s) will cause an interrupt.

D_4: *EOP Interrupt*—Reception of an EOP character (0 followed by 7 1s) will cause the 8273 to interrupt the CPU when this bit is set. Loop controller stations use this mode as a signal that a polling frame has completed the loop. No EOP interrupt is generated when this bit is reset.

D_3: *Early Tx Interrupt*—This bit specifies when the transmitter should generate an end of frame interrupt. If this bit is set, an interrupt is generated when the last data character has been passed to the 8273. If the user software issues another transmit command within two byte times, the final flag interrupt does not occur and the new frame is transmitted with only one flag of separation. If this restriction is not met, more than one flag will separate the frames and a frame complete interrupt is generated after the closing flag. If the bit is reset, only the frame complete interrupt occurs. This bit, when set, allows a single flag to separate consecutive frames.

D_2: *Buffered Address and Control*—When set, the address and control fields of received frames are buffered in the 8273 and passed to the CPU as results after a received frame interrupt (they are not transferred to memory with the information field). On transmit, the A and C fields are passed to the 8273 as parameters. This mode simplifies buffer management. When this bit is reset, the A and C fields are passed to and from memory as the first two data transfers.

D_1: *Preframe Sync*—When set, the 8273 prefaces each transmitted frame with two characters before the opening flag. These two characters provide 16 transitions to allow synchronization of the opposing receiver. To guarantee 16 transitions, the two characters are 55H-55H for non-NRZI mode (see Serial I/O Register description) or 00H-00H for NRZI mode. When reset, no preframe characters are transmitted.

D_0: *Flag Stream*—When set, the transmitter will start sending flag characters as soon as it is idle; i.e., immediately if idle when the command is issued or after a transmission if the transmitter is active when this bit is set. When reset, the transmitter starts sending Idle characters on the next character boundary if idle already, or at the end of a transmission if active.

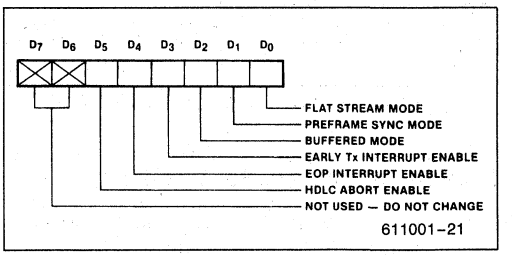


Figure 24. Operating Mode Register

Serial I/O Mode Register (Figure 25)

D_7–D_3: *Not Used*—These bits must be 0 for the Set command and 1 for the Reset command.

D_2: *Data Loopback*—When set, transmitted data (TxD) is internally routed to the receive data circuitry. When reset, TxD and RxD are independent.

D_1: *Clock Loopback*—When set, $\overline{\text{TxC}}$ is internally routed to $\overline{\text{RxC}}$. When reset, the clocks are independent.

D_0: *NRZI (Non-Return to Zero Inverted*—When set, the 8273 assumes the received data is NRZI encoded, and NRZI encodes the transmitted data. When reset, the received and transmitted data are treated as a normal positive logic bit stream.

Data Transfer Mode Register (Figure 26)

D_7–D_1: *Not Used*—These bits must be 0 for the Set command and 1 for the Reset command.

D_0: *Interrupt Data Transfer*—When set, the 8273 will interrupt the CPU when data transfers are required (the corresponding IRA Status register bit will be 0 to signify a data transfer interrupt rather than a Result phase interrupt). When reset, 8273 data transfers are performed through DMA requests on the DRQ pins without interrupting the CPU.

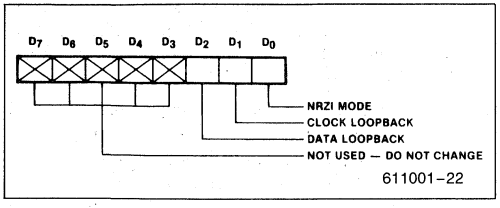


Figure 25. Serial I/O Mode Register

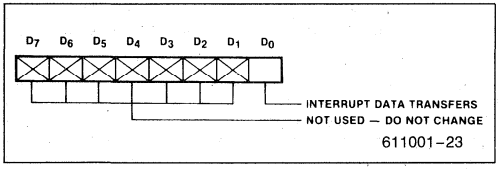


Figure 26. Data Transfer Mode Register

One Bit Delay Register (Figure 27)

D_7: *One Bit Delay*—When set, the 8273 retransmits the received data stream one bit delayed. This mode is entered and exited at a received character boundary. When reset, the transmitted and received data are independent. This mode is utilized for loop operation and is discussed in a later section.

D_6–D_0: *Not Used*—These bit must be 0 for the Set command and 1 for the Reset command.

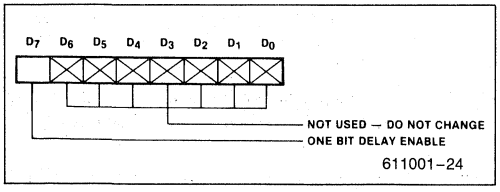


Figure 27. One Bit Delay Mode Register

Figure 28 shows the Set and Reset commands associated with the above registers. The mask which sets or resets the desired bits is treated as a single parameter. These commands do not interrupt nor provide results during the Result phase. After reset, the 8273 defaults to all of these bits reset.

Register	Command	Hex Code	Parameter
One Bit Delay Mode	Set	A4	Set Mask
	Reset	64	Reset Mask
Data Transfer Mode	Set	97	Set Mask
	Reset	57	Reset Mask
Operating Mode	Set	91	Set Mask
	Reset	51	Reset Mask
Serial I/O Mode	Set	A0	Set Mask
	Reset	60	Reset Mask

Figure 28. Initialization/Configuration Command Summary

Receive Commands

The 8273 supports three receive commands plus a receiver disable function.

General Receive

When commanded to General Receive, the 8273 passes all frames either to memory (DMA mode) or to the CPU (non-DMA mode) regardless of the contents of the frame's address field. This command is used for primary and loop controller stations. Two parameters are required: B_0 and B_1. These parameters are the LSB and MSB of the receiver buffer size. Giving the 8273 this extra information alleviates the CPU of the burden of checking for buffer overflow. The 8273 will interrupt the CPU if the received frame attempts to overfill the allotted buffer space.

Selective Receive

In Selective Receive, two additional parameters besides B_0 and B_1 are required: A_1 and A_2. These parameters are two address match bytes. When commanded to Selective Receive, the 8273 passes to memory or the CPU only those frames having an address field matching either A_1 or A_2. This command is usually used for secondary stations with A_1 being the secondary address and A_2 is the "All Parties" address. If only one match byte is needed, A_1 and A_2 should be equal. As in General Receive, the 8273 counts the incoming data bytes and interrupts the CPU if B_0, B_1 is exceeded.

Selective Loop Receive

This command is very similar in operation to Selective Receive except that One Bit Delay mode must be set

and that the loop is captured by placing transmitter in Flag Stream mode automatically after an EOP character is detected following a selectively received frame. The details of using the 8273 in loop configurations is discussed in a later section so please hold questions until then.

The handling of interrupt results is common among the three commands. When a frame is received without error, i.e., the FCS is correct and $\overline{CD}$ (Carrier Detect) was active throughout the frame or no attempt was made to overfill the buffer; the 8273 interrupts the CPU following the closing flag to pass the completion results. These results, in order, are the receiver interrupt result code (RIC), and the byte length of the information field of the received frame (R_0, R_1). If Buffered mode is selected, the address and control fields are passed as two additional results. If Buffered mode is not selected, the address and control fields are passed as the first two data transfers and R_0, R_1 reflect the information field length plus two.

Receive Disable

The receiver may also be disabled using the Receive Disable command. This command terminates any receive operation immediately. No parameters are required and no results are returned.

The details for the Receive command are shown in Figure 29. The interrupt result code key is shown in Figure 30. Some explanation of these result codes is appropriate.

The interrupt result code is the first byte passed to the CPU in the RxI/R register during the Result phase. Bits D_4–D_0 define the cause of the receiver interrupt. Since each result code has specific implications, they are discussed separately below.

Command	Hex Code	Parameters	Results* RxI/R
General Receive	C0	B_0, B_1	RIC, R_0, R_1, A, C
Selective Receive	C1	B_0, B_1, A_1, A_2	RIC, R_0, R_1, A, C
Selective Loop Receive	C2	B_0, B_1, A_1, A_2	RIC, R_0, R_1, A, C
Disable Receiver	C5	None	None

*NOTE:
A and C are passed as results only in buffered mode.

Figure 29. Receiver Command Summary

RIC D_7–D_0	Receiver Interrupt Result Code	Rx Status After INT
* 00000	A_1 Match or General Receive	Active
* 00001	A_2 Match	Active
000 00011	CRC Error	Active
000 00100	Abort Detected	Active
000 00101	Idle Detected	Disabled
000 00110	EOP Detected	Disabled
000 00111	Frame < 32 Bits	Active
000 01000	DMA Overrun	Disabled
000 01001	Memory Buffer Overflow	Disabled
000 01010	Carrier Detect Failure	Disabled
000 01011	Receiver Interrupt Overrun	Disabled
*D_7–D_5	**Partial Byte Received**	
111	All 8 Bits of Last Byte	
000	D_0	
100	D_1–D_0	
010	D_2–D_0	
110	D_3–D_0	
001	D_4–D_0	
101	D_5–D_0	
011	D_6–D_0	

Figure 30. Receiver Interrupt Result Codes (RIC)

The first two result codes result from the error-free reception of a frame. If the frame is received correctly after a General Receive command, the first result is returned. If either Selective Receive command was used (normal or loop), a match with A_1 generates the first result code and a match with A_2 generates the second. In either case, the receiver remains active after the interrupt; however, the internal buffer size counters are not reset. That is, if the receive command indicated 100 bytes were allocated to the receive buffer (B_0, B_1) and an 80-byte frame was received correctly, the maximum next frame size that could be received without recommanding the receiver (resetting B_0 and B_1) is 20 bytes. Thus, it is common practice to recommand the receiver after each frame reception. DMA and/or memory pointers are usually updated at this time. (Note that users who do not wish to take advantage of the 8273's buffer management features may simply use B_0, B_1 = 0FFH for each receive command. Then frames of 65K bytes may be received without buffer overflow errors.)

The third result code is a CRC error. This indicates that a frame was received in the correct format (flags, etc.); however, the received FCS did not check with the internally generated FCS. The frame should be discarded. The receiver remains active. (Do not forget that even though an error condition has been detected, all frame information up until that error has either been transferred to memory or passed to the CPU. This information should be invalidated. This applies to all receiver error conditions.) Note that the FCS, either transmitted or received, is never available to the CPU.

The Abort Detect result occurs whenever the receiver sees either an SDLC (8 1s) or an HDLC (7 1s), depending on the Operating Mode register. However, the intervening Abort character between a closing flag and an Idle does not generate an interrupt. If an Abort character (seen by an active receiver within a frame) is not preceded by a flag and is followed by an idle, an interrupt will be generated for the Abort, followed by an Idle interrupt one character time later. The Idle Detect result occurs whenever 15 consecutive 1s are received. After the Abort Detect interrupt, the receiver remains active. After the Idle Detect interrupt, the receiver is disabled and must be recommanded before further frames may be received.

If the EOP Interrupt bit is set in the Operating Mode register, the EOP Detect result is returned whenever an EOP character is received. The receiver is disabled, so the Idle following the EOP does not generate an Idle Detect interrupt.

The minimum number of bits in a valid frame between the flags is 32. Fewer than 32 bits indicates an error. If Buffered mode is selected, such frames are ignored, i.e., no data transfers or interrupts are generated. In non-Buffered mode, a < 32-bit frame generates an interrupt with the < 32-bit frame result since data transfers may already have disturbed the 8257 or interrupt handler. The receiver remains active.

The DMA Overrun results from the $\overline{\text{DMA}}$ controller being too slow in extracting data from the 8273, i.e., the $\overline{\text{RxDACK}}$ signal is not returned before the next received byte is ready for transfer. The receiver is disabled if this error condition occurs.

The Memory Buffer Overflow result occurs when the number of received bytes exceeds the receiver buffer length supplied by the B_0 and B_1 parameters in the receive command. The receiver is disabled.

The Carrier Detect Failure result occurs when the $\overline{\text{CD}}$ pin goes high (inactive) during reception of a frame. The $\overline{\text{CD}}$ pin is used to qualify reception and must be active by the time the address field starts to be received. If $\overline{\text{CD}}$ is lost during the frame, a $\overline{\text{CD}}$ Failure interrupt is generated and the receiver is disabled. No interrupt is generated if $\overline{\text{CD}}$ goes inactive between frames.

If a condition occurs requiring an interrupt be generated before the CPU has finished reading the previous interrupt results, the second interrupt is generated after the current Result phase is complete (the RxINT pin and status bit go low then high). However, the interrupt result for this second interrupt will be a Receive Interrupt Overrun. The actual cause of the second interrupt is lost. One case where this may occur is at the end of a received frame where the line goes idle. The 8273 generates a received frame interrupt after the closing flag and then 15-bit times later, generates an Idle Detect interrupt. If the interrupt service routine is slow in reading the first interrupt's results, the internal RxI/R register still contains result information when the Idle Detect interrupt occurs. Rather than wiping out the previous results, the 8273 adds a Receive Interrupt Overrun result as an extra result. If the system's interrupt structure is such that the second interrupt is not acknowledged (interrupts are still disabled from the first interrupt), the Receive Interrupt Overrun result is read as an extra result, after those from the first interrupt. If the second interrupt is serviced, the Receive Interrupt Overrun is returned as a single result. (Note that the INT pins supply the necessary transitions to support a Programmable Interrupt Controller such as the Intel 8259. Each interrupt generates a positive-going edge on the appropriate INT pin and the high level is held until the interrupt is completely serviced.) In general, it is possible to have interrupts occurring at one character time intervals. Thus the interrupt handling software must have at least that much response and service time.

The occurrence of Receive Interrupt Overruns is an indication of marginal software design; the system's interrupt response and servicing time is not sufficient for the

data rates being attempted. It is advisable to configure the interrupt handling software to simply read the interrupt results, place them into a buffer, and clear the interrupt as quickly as possible. The software can then examine the buffer for new results at its leisure, and take appropriate action. This can easily be accomplished by using a result buffer flag that indicates when new results are available. The interrupt handler sets the flag and the main program resets it once the results are retrieved.

Both SDLC and HDLC allow frames which are of arbitrary length (>32 bits). The 8273 handles this N-bit reception through the high order bits (D_7–D_5) of the result code. These bits code the number of valid received bits in the last received information field byte. This coding is shown in Figure 30. The high order bits of the received partial byte are indeterminate. [The address, control, and information fields are transmitted least significant bit (A_0) first. The FCS is complemented and transmitted most significant bit first.]

Transmit Commands

The 8273 transmitter is supported by three Transmit commands and three corresponding Abort commands.

Transmit Frame

The Transmit Frame command simply transmits a frame. Four parameters are required when Buffered mode is selected and two when it is not. In either case, the first two parameters are the least and the most significant bytes of the desired frame length (L_0, L_1). In Buffered mode, L_0 and L_1 equal the length in bytes of the desired information field, while in the non-Buffered mode, L_0 and L_1 must be specified at the information field length plus two. (L_0 and L_1 specify the number of data transfers to be performed.) In Buffered mode, the address and control fields are presented to the transmitter as the third and fourth parameters respectively. In non-Buffered mode, the A and C fields must be passed as the first two data transfers.

When the Transmit Frame command is issued, the 8273 makes $\overline{RTS}$ (Request-to-Send) active (pin low) if it was not already. It then waits until $\overline{CTS}$ (Clear-to-Send) goes active (pin low) before starting the frame. If the Preframe Sync bit in the Operating Mode register is set, the transmitter prefaces two characters (16 transitions) before the opening flag. If the Flag Stream bit is set in the Operating Mode register, the frame (including Preframe Sync if selected) is started on a flag boundary. Otherwise the frame starts on a character boundary.

At the end of the frame, the transmitter interrupts the CPU (the interrupt results are discussed shortly) and returns to either Idle or Flag Stream, depending on the Flag Stream bit of the Operating Mode register. If $\overline{RTS}$ was active before the transmit command, the 8273 does not change it. If it was inactive, the 8273 will deactivate it within one character time.

Loop Transmit

Loop Transmit is similar to Frame Transmit (the parameter definition is the same). But since it deals with loop configurations, One Bit Delay mode must be selected.

If the transmitter is not in Flag Stream mode when this command is issued, the transmitter waits until after a received EOP character has been converted to a flag (this is done automatically) before transmitting. (The one bit delay is, of course, suspended during transmit.) If the transmitter is already in Flag Stream mode as a result of a selectively received frame during a Selective Loop Receive command, transmission will begin at the next flag boundary for Buffered mode or at the third flag boundary for non-Buffered mode. This discrepancy is to allow time for enough data transfers to occur to fill up the internal transmit buffer. At the end of a Loop Transmit, the One Bit Delay mode is re-entered and the flag stream mode is reset. More detailed loop operation is covered later.

Transmit Transparent

The Transmit Transparent command enables the 8273 to transmit a block of raw data. This data is without SDLC protocol, i.e., no zero bit insertion, flags, or FCS. Thus it is possible to construct and transmit a Bi-Sync message for front-end processor switching or to construct and transmit an SDLC message with incorrect FCS for diagnostic purposes. Only the L_0 and L_1 parameters are used since there are not fields in this mode. (The 8273 does not support a Receive Transparent command.)

Abort Commands

Each of the above transmit commands has an associated Abort command. The Abort Frame Transmit command causes the transmitter to send eight contiguous ones (no zero bit insertion) immediately and then revert to either idle or flag streaming based on the Flag Stream bit. (The 8 1s as an Abort character is compatible with both SDLC and HDLC.)

For Loop Transmit, the Abort Loop Transmit command causes the transmitter to send one flag and then revert to one bit delay. Loop protocol depends upon FCS errors to detect aborted frames.

The Abort Transmit Transparent simply causes the transmitter to revert to either idles or flags as a function of the Flag Stream mode specified.

The Abort commands require no parameters, however, they do generate an interrupt and return a result when complete.

A summary of the Transmit commands is shown in Figure 31. Figure 32 shows the various transmit interrupt result codes. As in the receiver operation, the transmitter generates interrupts based on either good completion of an operation or an error condition to start the Result phase.

The Early Transmit Interrupt result occurs after the last data transfer to the 8273 if the Early Transmit Interrupt bit is set in the Operating Mode register. If the 8273 is commanded to transmit again within two character times, a single flag will separate the frames. (Buffered mode must be used for a single flag to separate the frames. If non-Buffered mode is selected, three flags will separate the frames.) If this time constraint is not met, another interrupt is generated and multiple flags or idles will separate the frames. The second interrupt is the normal Frame Transmit Complete interrupt. The Frame Transmit Complete result occurs at the closing flag to signify a good completion.

The DMA Underrun result is analogous to the DMA Overrun result in the receiver. Since SDLC does not support intraframe time fill, if the DMA controller or CPU does not supply the data in time, the frame must be aborted. The action taken by the transmitter on this error is automatic. It aborts the frame just as if an Abort command had been issued.

Clear-to-Send Error result is generated if $\overline{CTS}$ goes inactive during a frame transmission. The frame is aborted as above.

The Abort Complete result is self-explanatory. Please note however that no Abort Complete interrupt is generated when an automatic abort occurs. The next command type consists of only one command.

Reset Command

The Reset command provides a software reset function for the 8273. It is a special case and does not utilize the normal command interface. The reset facility is provided in the Test Mode register. The 8273 is reset by simply outputting a 01H followed by a 00H to the Test Mode register. Writing the 01 followed by the 00 mimicks the action required by the hardware reset. Since the 8273 requires time to process the reset internally, at least 10 cycles of the ϕCLK clock must occur between the writing of the 01 and the 00. The action taken is the same as if a hardware reset is performed, namely:

1) The modem control outputs are forced high inactive.

Command	Hex Code	Parameters*	Results Txl/R
Transmit Frame	C8	L_0, L_1, A, C	TIC
Abort	CC	None	TIC
Loop Transmit	CA	L_0, L_1, A, C	TIC
Abort	CE	None	TIC
Transmit Transparent	C0	L_0, L_1	TIC
Abort	CD	None	TIC

*NOTE:
A and C are passed as parameters in buffered mode only.

Figure 31. Transmitter Command Summary

RIC D_7–D_0	Transmitter Interrupt Result Code	Tx Status after INT
000 01100	Early Tx Interrupt	Active
000 01101	Frame Tx Complete	Idle or Flags
000 01110	DMA Underrun	Abort
000 01111	Clear to Send Error	Abort
000 10000	Abort Complete	Idle or Flags

Figure 32. Transmitter Interrupt Result Codes

2) The 8273 Status register is cleared.

3) Any commands in progress cease.

4) The 8273 enters an idle state until the next command is issued.

Modem Control Commands

The modem control ports were discussed earlier in the Hardware section. The commands used to manipulate these ports are shown in Figure 33. The Read Port A and Read Port B commands are immediate. The bit definition for the returned byte is shown in Figures 13 and 14. Do not forget that the returned value represents the logical condition of the pin, i.e., pin active (low) = bit set.

The Set and Reset Port B commands are similar to the Initialization commands in that they use a mask parameter which defines the bits to be changed. Set Port B utilizes a logical OR mask and Reset Port B uses a logical AND mask. Setting a bit makes the pin active (low). Resetting the bit deactivates the pin (high).

To help clarify the numerous timing relationships that occur and their consequences, Figures 34 and 35 are provided as an illustration of several typical sequences. It is suggested that the reader go over these diagrams and re-read the appropriate part of the previous sections if necessary.

HDLC CONSIDERATIONS

The 8273 supports HDLC as well as SDLC. Let's discuss how the 8273 handles the three basic HDLC/SDLC differences: extended addressing, extended control, and the 7 1s Abort character.

Recalling Figure 4a, HDLC supports an address field of indefinite length. The actual amount of extension used is determined by the least significant bit of the characters immediately following the opening flag. If the LSB is 0, more address field bytes follow. If the LSB is 1, this byte is the final address field byte. Software must be used to determine this extension.

If non-Buffered mode is used, the A, C, and I fields are in memory. The software must examine the initial characters to find the extent of the address field. If Buffered mode is used, the characters corresponding to the SDLC A and C fields are transferred to the CPU as interrupt results. Buffered mode assumes the two characters following the opening flag are to be transferred as interrupt results regardless of content or meaning. (The 8273 does not know whether it is being used in an SDLC or an HDLC environment.) In SDLC, these characters are necessarily the A and C field bytes, however in HDLC, their meaning may change depending on the amount of extension used. The software must recognize this and examine the transferred results as possible address field extensions.

Frames may still be selectively received as is needed for secondary stations. The Selective Receive command is still used. This command qualifies a frame reception on the first byte following the opening flag matching either of the A_1 or A_2 match byte parameters. While this does not allow qualification over the complete range of HDLC addresses, it does perform a qualification on the first address byte. The remaining address field bytes, if any, are then examined via software to completely qualify the frame.

Once the extent of the address field is found, the following bytes form the control field. The same LSB test used for the address field is applied to these bytes to determine the control field extension, up to two bytes maximum. The remaining frame bytes in memory represent the information field.

The Abort character difference is handled in the Operating Mode register. If the HDLC Abort Enable bit is set, the reception of seven contiguous ones by an active receiver will generate an Abort Detect interrupt rather than eight ones. (Note that both the HDLC Abort Enable bit and the EOP Interrupt bit must not be set simultaneously.)

Now let's move on to the SDLC loop configuration discussion.

Port	Command	Hex Code	Parameter	Reg Result
A Input	Read	22	None	Port Value
B Output	Read	23	None	Port Value
	Set	A3	Set Mask	None
	Reset	63	Reset Mask	None

Figure 33. Modem Control Command Summary

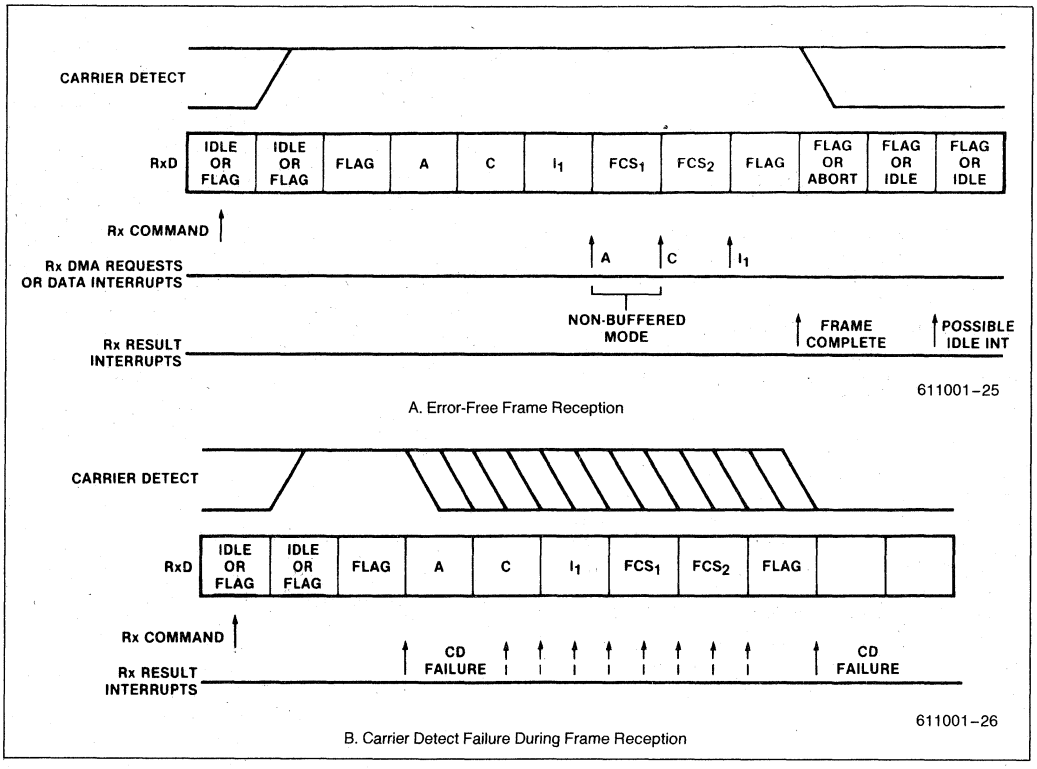


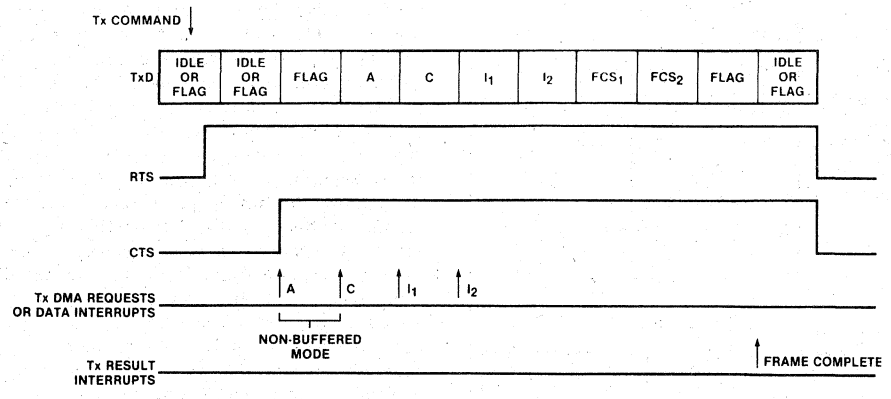
Figure 34. Sample Receiver Timing Diagrams

LOOP CONFIGURATION

Aside from use in the normal data link applications, the 8273 is extremely attractive in loop configuration due to the special frame-level loop commands and the Digital Phase Locked Loop. Toward this end, this section details the hardware and software considerations when using the 8273 in a loop application.

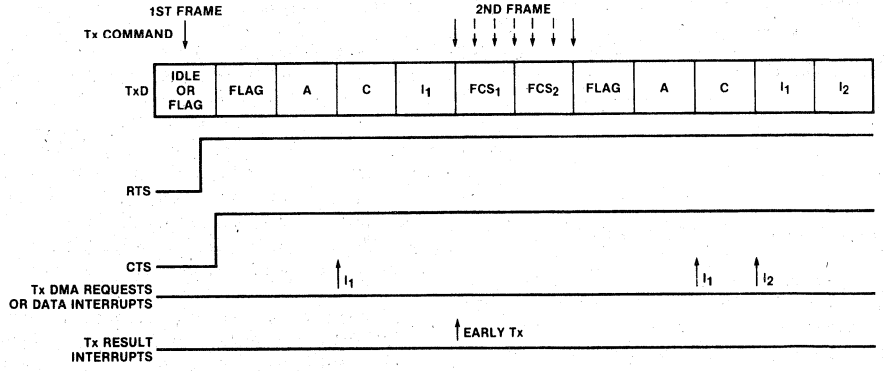
The loop configuration offers a simple, low-cost solution for systems with multiple stations within a small physical location, i.e., retail stores and banks. There are two primary reasons to consider a loop configuration. The interconnect cost is lower for a loop over a multi-point configuration since only one twisted pair or fiber optic cable is used. (The loop configuration does not support the passing of distinct clock signals from station to station.) In addition, loop stations do not need the intelligence of a multi-point station since the loop protocol is simpler. The most difficult aspects of loop station design are clock recovery and implementation of one bit delay (both are handled neatly by the 8273).

Figure 36 illustrates a typical loop configuration with one controller and two down-loop secondaries. Each station must derive its own data timing from the received data stream. Recalling our earlier discussion of the DPLL, notice that $\overline{TxC}$ and $\overline{RxC}$ clocks are provided by the $\overline{DPLL}$ output. The only clock required in the secondaries is a simple, non-synchronized clock at 32 times the desired baud rate. The controller requires both 32× and 1× clocks. (The 1× is usually implemented by dividing the 32× clock with a 5-bit divider. However, there is no synchronism requirement between these clocks so any convenient implementation may be used.)



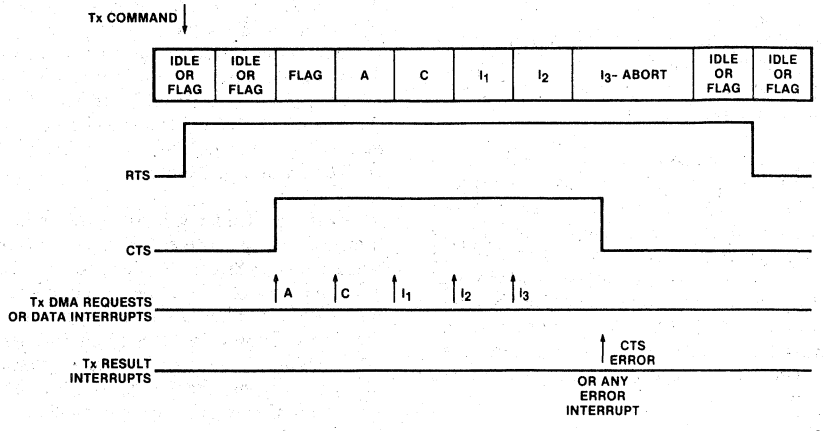
611001-27

A. Error-Free Frame Transmission



611001-28

B. Diagram Showing Tx Command Queing and Early Tx Interrupt
(Single flag between frames) Buffered Mode is Assumed



611001-29

C. CTS Failure (or other error) During Transmission

Figure 35. Sample Transmitter Timing Diagrams

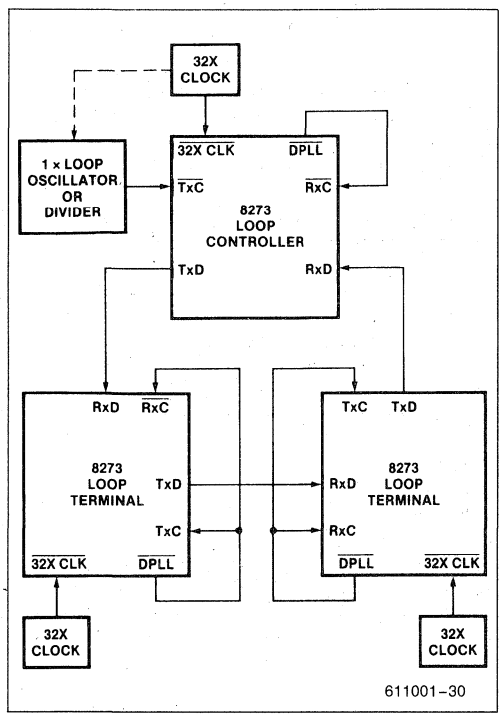


Figure 36. SDLC Loop Application

A quick review of loop protocol is appropriate. All communication on the loop is controlled by the loop controller. When the controller wishes to allow the secondaries to transmit, it sends a polling frame (the control field contains a poll code) followed by an EOP (End-of-Poll) character. The secondaries use the EOP character to capture the loop and insert a response frame as will be discussed shortly.

The secondaries normally operate in the repeater mode, retransmitting received data with one bit time of delay. All received frames are repeated. The secondary uses the one bit time of delay to capture the loop.

When the loop is idle (no frames), the controller transmits continuous flag characters. This keeps transitions on the loop for the sake of down-loop phase locked loops. When the controller has a non-polling frame to transmit, it simply transmits the frame and continues to send flags. The non-polling frame is then repeated around the loop and the controller receives it to signify a complete traversal of the loop. At the particular secondary addressed by the frame, the data is transferred to memory while being repeated. Other secondaries simply repeat it.

If the controller wants to poll the secondaries, it transmits a polling frame followed by all 1s (no zero bit insertion). The final zero of the closing frame plus the first seven 1s form an EOP. While repeating, the secondaries monitor their incoming line for an EOP. When an EOP is received, the secondary checks if it has any response for the controller. If not, it simply continues repeating. If the secondary has a response, it changes the seventh EOP one into a zero (the one bit time of delay allows time for this) and repeats it, forming a flag for the down-loop stations. After this flag is transmitted, the secondary terminates its repeater function and inserts its response frame (with multiple preceding flags if necessary). After the closing flag of the response, the secondary re-enters its repeater function, repeating the up-loop controller 1s. Notice that the final zero of the response's closing flag plus the repeated 1s from the controller form a new EOP for the next down-loop secondary. This new EOP allows the next secondary to insert a response if it desires. This gives each secondary a chance to respond.

Back at the controller, after the polling frame has been transmitted and the continuous 1s started, the controller waits until it receives an EOP. Receiving an EOP signifies to the controller that the original frame has propagated around the loop followed by any responses inserted by the secondaries. At this point, the controller may either send flags to idle the loop or transmit the next frame. Let's assume that the loop is implemented completely with the 8273s and describe the command flows for a typical controller and secondary.

The loop controller is initialized with commands which specify that the NRZI, Preframe Sync, Flag Stream, and EOP Interrupt modes are set. Thus, the controller encodes and decodes all data using NRZI format. Preframe Sync mode specifies that all transmitted frames be prefaced with 16 line transitions. This ensures that the minimum of 12 transitions needed by the DPLL to lock after an all 1s line has occurred by the time the secondary sees a frame's opening flag. Setting the Flag Stream mode starts the transmitter sending flags which idles the loop. And the EOP Interrupt mode specifies that the controller processor will be interrupted whenever the active receiver sees an EOP, indicating the completion of a poll cycle.

When the controller wishes to transmit a non-polling frame, it simply executes a Frame Transmit command. Since the Flag Stream mode is set, no EOP is formed after the closing flag. When a polling frame is to be transmitted, a General Receive command is executed first. This enables the receiver and allows reception of all incoming frames; namely, the original polling frame plus any response frames inserted by the secondaries. After the General Receive command, the frame is transmitted with a Frame Transmit command. When the frame is complete, a transmitter interrupt is gener-

ated. The loop controller processor uses this interrupt to reset Flag Stream mode. This causes the transmitter to start sending all 1s. An EOP is formed by the last flag and the first 7 1s. This completes the loop controller transmit sequence.

At any time following the start of the polling frame transmission the loop controller receiver will start receiving frames. (The exact time difference depends, of course, on the number of down-loop secondaries due to each inserting one bit time of delay.) The first received frame is simply the original polling frame. However, any additional frames are those inserted by the secondaries. The loop controller processor knows all frames have been received when it sees an EOP Interrupt. This interrupt is generated by the 8273 since the EOP Interrupt mode was set during initialization. At this point, the transmitter may be commanded either to enter Flag Stream mode, idling the loop, or to transmit the next frame. A flowchart of this sequence is shown in Figure 37.

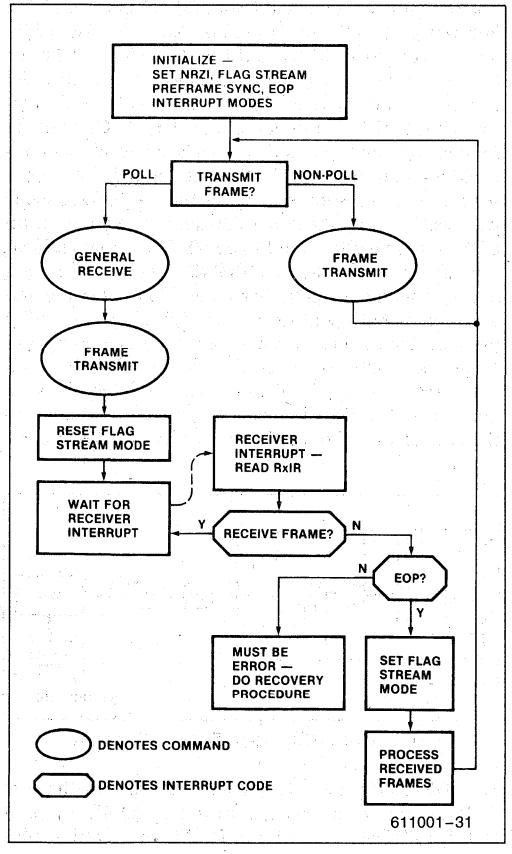


Figure 37. Loop Controller Flowchart

The secondaries are initialized with the NRZI and One Bit Delay modes set. This puts the 8273 into the repeater mode with the transmitter repeating the received data with one bit time of delay. Since a loop station cannot transmit until it sees an EOP character, any transmit command is queued until an EOP is received. Thus whenever the secondary wishes to transmit a response, a Loop Transmit command is issued. The 8273 then waits until it receives an EOP. At this point, the receiver changes the EOP into a flag, repeats it, resets One Bit Delay mode stopping the repeater function, and sets the transmitter into Flag Stream mode. This captures the loop. The transmitter now inserts its message. At the closing flag, Flag Stream mode is reset, and One Bit Delay mode is set, returning the 8273 to repeater function and forming an EOP for the next down-loop station. These actions happen automatically after a Loop Transmit command is issued.

When the secondary wants its receiver enabled, a Selective Loop Receive command is issued. The receiver then looks for a frame having a match in the Address field. Once such a frame is received, repeated, and transferred to memory, the secondary's processor is interrupted with the appropriate Match interrupt result and the 8273 continues with the repeater function until an EOP is received, at which point the loop is captured as above. The processor should use the interrupt to determine if it has a message for the controller. If it does, it simply issues a Loop Transmit command and things progress as above. If the processor has no message, the software must reset the Flag Stream mode bit in the Operating Mode register. This will inhibit the 8273 from capturing the loop at the EOP. (The match frame and the EOP may be separated in time by several frames depending on how many up-loop stations inserted messages of their own.) If the timing is such that the receiver has already captured the loop when the Flag Stream mode bit is reset, the mode is exited on a flag boundary and the frame just appears to have extra closing flags before the EOP. Notice that the 8273 handles the queuing of the transmit commands and the setting and resetting of the mode bits automatically. Figure 38 illustrates the major points of the secondary command sequence.

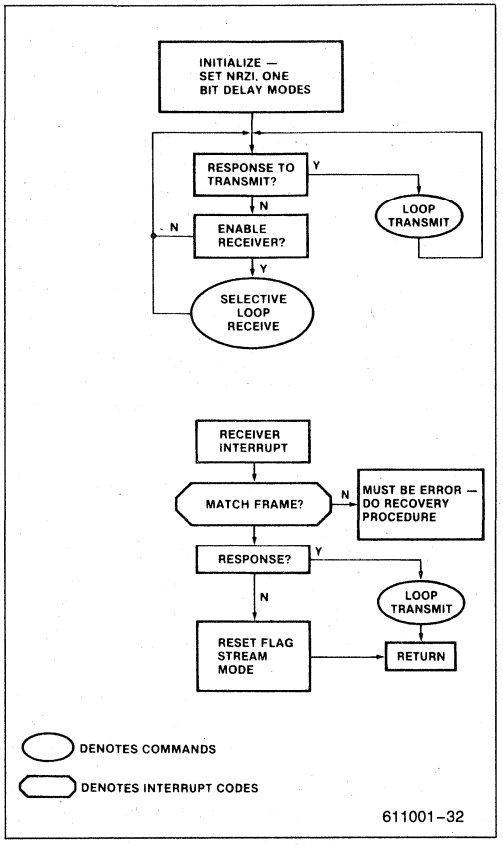


Figure 38. Loop Secondary Flowchart

When an off-line secondary wishes to come on-line, it must do so in a manner which does not disturb data on the loop. Figure 39 shows a typical hardware interface. The line labeled Port could be one of the 8273 Port B outputs and is assumed to be high (1) initially. Thus up-loop data is simply passed down-loop with no delay; however, the receiver may still monitor data on the loop. To come on-line, the secondary is initialized with only the EOP Interrupt mode set. The up-loop data is then monitored until an EOP occurs. At this point, the secondary's CPU is interrupted with an EOP interrupt. This signals the CPU to set One Bit Delay mode in the 8273 and then to set Port low (active). These actions switch the secondary's one bit delay into the loop. Since after the EOP only 1s are traversing the loop, no loop disturbance occurs. The secondary now waits for the next EOP, captures the loop, and inserts a "new on-line" message. This signals the controller that a new secondary exists and must be acknowledged. After the secondary receives its acknowledgement, the normal command flow is used.

It is hopefully evident from the above discussion that the 8273 offers a very simple and easy to implement solution for designing loop stations whether they are controllers or down-loop secondaries.

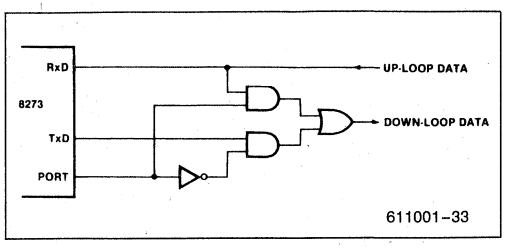


Figure 39. Loop Interface

APPLICATION EXAMPLE

This section describes the hardware and software of the 8273/8085 system used to verify the 8273 implementation of SDLC on an actual IBM SDLC Link. This IBM link was gratefully volunteered by Raytheon Data Systems in Norwood, Mass. and I wish to thank them for their generous cooperation. The IBM system consisted of a 370 Mainframe, a 3705 Communications Processor, and a 3271 Terminal Controller. A Comlink II Modem supplied the modem interface and all communications took place at 4800 baud. In addition to observing correct responses, a Spectron D601B Datascope was used to verify the data exchanges. A block diagram of the system is shown in Figure 40. The actual verification was accomplished by the 8273 system receiving and responding to polls from the 3705. This method was used on both point-to-point and multi-point configurations. No attempt was made to implement any higher protocol software over that of the poll and poll responses since such software would not affect the verification of the 8273 implementation. As testimony to the ease of use of the 8273, the system worked on the first try.

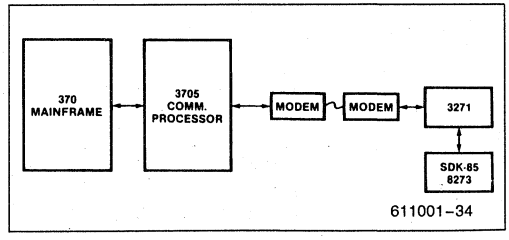


Figure 40. Raytheon Block Diagram

An SDK-85 (System Design Kit) was used as the core 8085 system. This system provides up to 4K bytes of ROM/EPROM, 512 bytes of RAM, 76 I/O pins, plus

two timers as provided in two 8755 Combination EPROM/I/O devices and two 8155 Combination RAM/I/O/Timer devices. In addition, 5 interrupt inputs are supplied on the 8085. The address, data, and control buses are buffered by the 8212 and 8216 latches and bidirectional bus drivers. Although it was not used in this application, an 8279 Display Driver/Keyboard Encoder is included to interface the on-board display and keyboard. A block diagram of the SDK-85 is shown in Figure 41. The 8273 and associated circuitry was constructed on the ample wire-wrap area provided for the user.

The example 8237/8085 system is interrupt-driven and uses DMA for all data transfers supervised by an 8257 DMA Controller. A 2400 baud asynchronous line, implemented with an 8251A USART, provides communication between the software and the user. 8253 Programmable Interval Timer is used to supply the baud rate clocks for the 8251A and 8273. (The 8273 baud rate clocks were used only during initial system debug. In actual operation, the modem supplied these clocks via the RS-232 interface.) Two 2142 1K x 4 RAMs provided 512 bytes of transmitter and 512 bytes of receiver buffer memory. (Command and result buffers, plus miscellaneous variables are stored in the 8155s.) The RS-232 interface utilized MC1488 and MC1489 RS-232 drivers and receivers. The schematic of the system is shown in Figure 42.

One detail to note is the DMA and interrupt structure of the transmit and receive channels. In both cases, the receiver is always given the higher priority (8257 DMA channel 0 has priority over the remaining channels and the 8085 RST 7.5 interrupt input has priority over the RST 6.5 input.) Although the choice is arbitrary, this technique minimizes the chance that received data could be lost due to other processor or DMA commitments.

Also note that only one 8205 Decoder is used for both peripheral and memory Chip Select. This was done to eliminate separate memory and I/O decoders since it was known beforehand that neither address space would be completely filled.

The 4 MHz crystal and 8224 Clock Generator were used only to verify that the 8273 operates correctly at that maximum spec speed. In a normal system, the 3.072 MHz clock from the 8085 would be sufficient. (This fact was verified during initial checkout.)

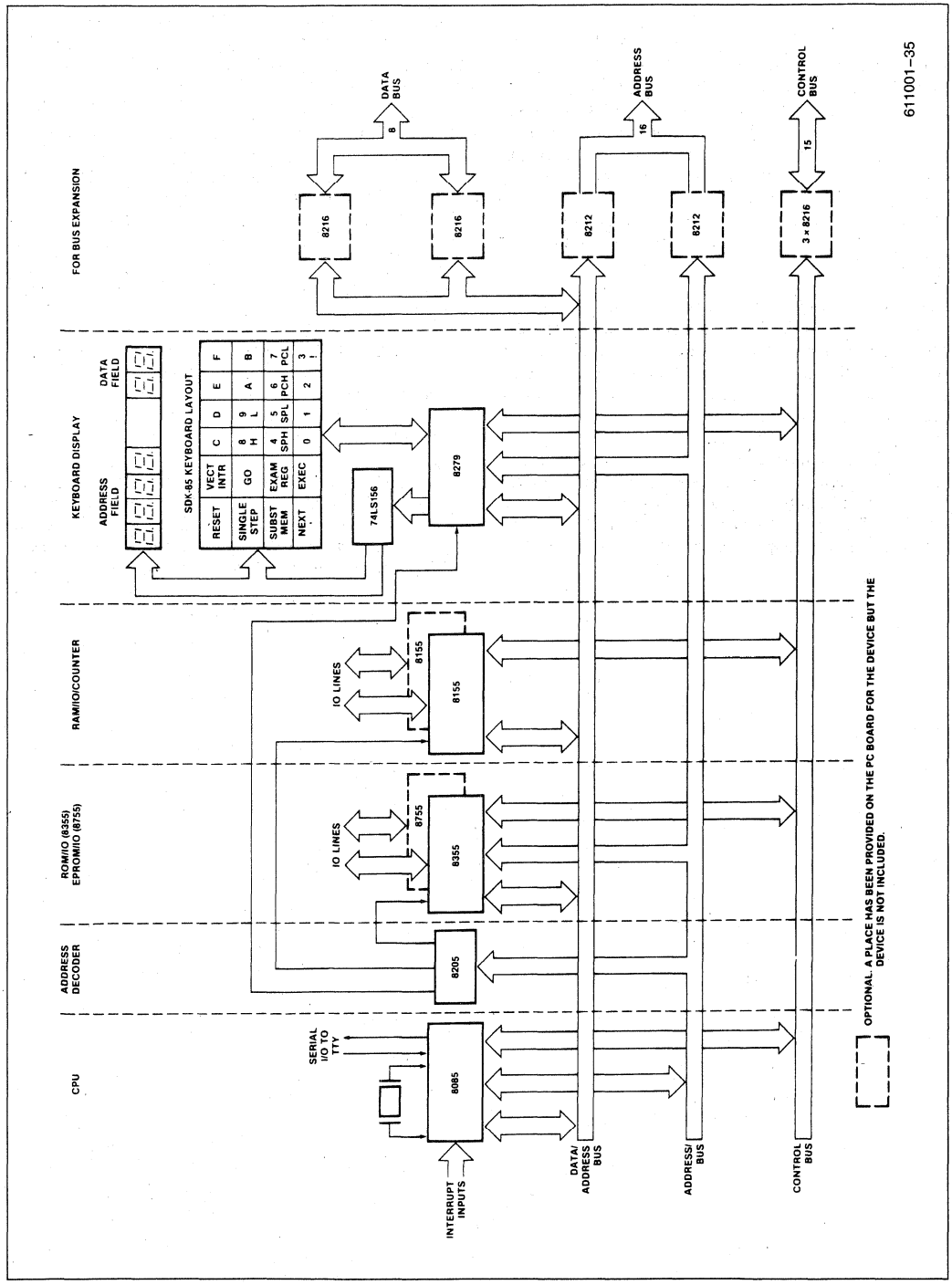


Figure 41. SDK-85 Functional Block Diagram

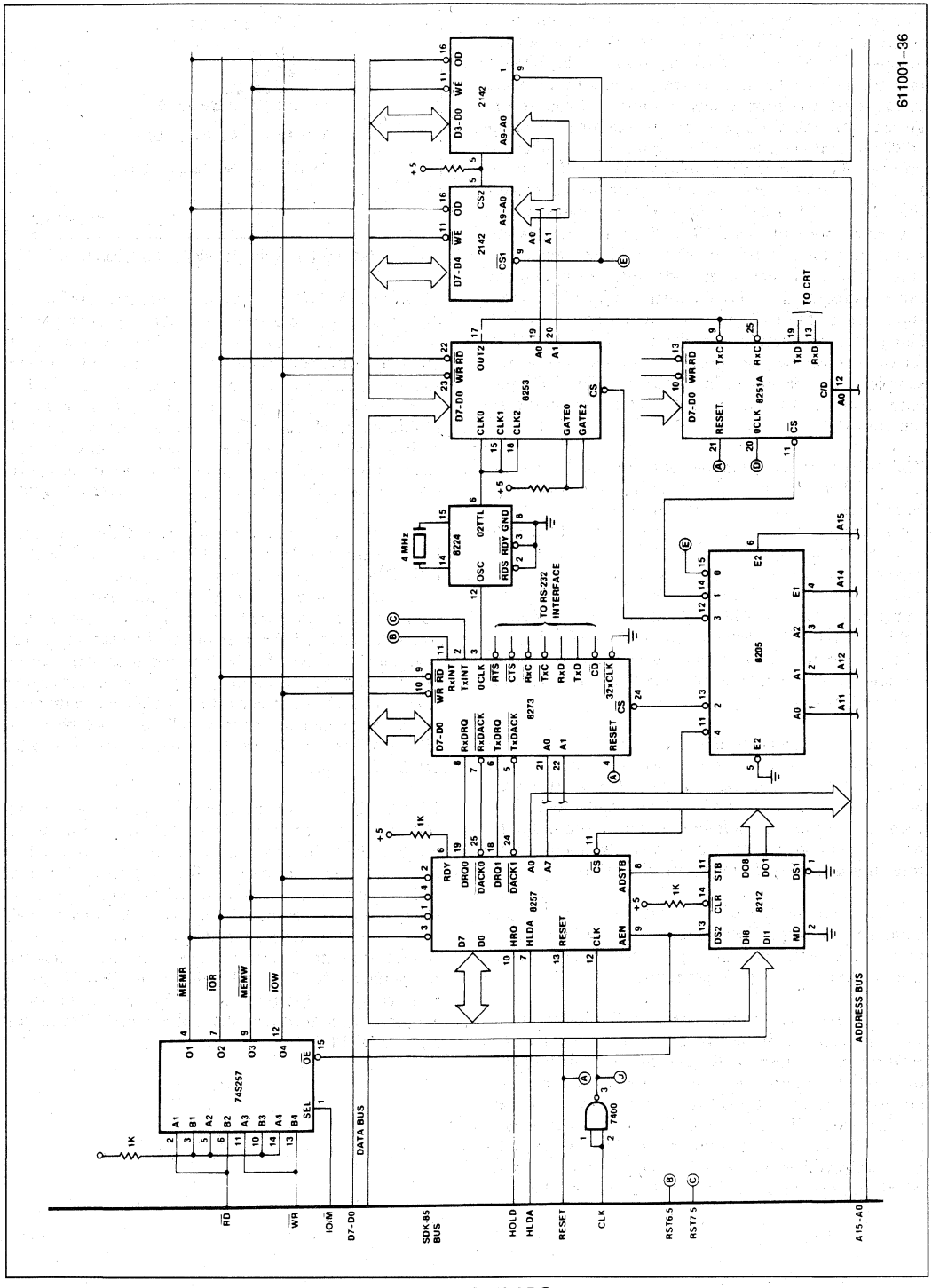


Figure 42. 8273/SDK-85 System

The software consists of the normal monitor program supplied with the SDK-85 and a program to input commands to the 8273 and to display results. The SDK-85 monitor allows the user to read and write on-board RAM, start execution at any memory location, to single-step through a program, and to examine any of the 8085's internal registers. The monitor drives either the on-board keyboard/LED display or a serial TTY interface. This monitor was modified slightly in order to use the 8251A with a 2400 baud CRT as opposed to the 110 baud normally used. The 8273 program implements monitor-like user interface. 8273 commands are entered by a two-character code followed by any parameters required by that command. When 8273 interrupts occur, the source of the interrupt is displayed along with any results associated with it. To gain a flavor of how the user/program interface operates, a sample output is shown in Figure 43. The 8273 program prompt character is a "–" and user inputs are underlined.

The "SO 05" implements the Set Operating Mode command with a parameter of 05H. This sets the Buffer and Flag Stream modes. "SS 01" sets the 8273 in NRZI mode using the Set Serial I/O Mode command. The next command specifies General Receiver with a receiver buffer size of 0100H bytes (B_0 = 00, B_1 = 01). The "TF" command causes the 8273 to transmit a frame containing an address field of C2H and control field of 11H. The information field is 001122. The "TF" command has a special format. The L_0 and L_1 parameters are computed from the number of information field bytes entered.

After the TF command is entered, the 8273 transmits the frame (assuming that the modem protocol is observed). After the closing flag, the 8273 interrupts the 8085. The 8085 reads the interrupt results and places them in a buffer. The software examines this buffer for new results and if new results exist, the source of the interrupt is displayed along with the results.

In this example, the 0DH result indicates a Frame Complete interrupt. There is only one result for a transmitter interrupt, the interrupt's trailing zero results were included to simplify programming.

The next event is a frame reception. The interrupt results are displayed in the order read from the 8273. The E0H indicates a General Receive interrupt with the last byte of the information field received on an 8-bit boundary. The 03 00 (R_0, R_1) results show that there are 3H bytes of information field received. The remaining two results indicate that the received frame had a C2H address field and a 34H control field. The 3 bytes of information field are displayed on the next line.

```
8273 MONITOR V1.2
– SO 05
– SS 01
– GR 00 01
– TF C2 11 00 11 22
–
TxINT – 0D 00 00 00 00
–
RxINT – E0 03 00 C2 34
FF EE DD
–
```

611001-63

Figure 43. Sample 8273 Monitor I/O

Figures 44 through 51 show the flowcharts used for the 8273 program development. The actual program listing is included as Appendix A. Figure 44 is the main status poll loop. After all devices are initialized and a prompt character displayed, a loop is entered at LOOPIT. This loop checks for a change of status in the result buffer or if a keyboard character has been received by the 8251 or if a poll frame has been received. If any of these conditions are met, the program branches to the appropriate routine. Otherwise, the loop is traversed again.

The result buffer is implemented as a 255-byte circular buffer with two pointers: CNADR and LDADR. CNADR is the console pointer. It points to the next result to be displayed. LDADR is the load pointer. It points to the next empty position in the buffer into which the interrupt handler places the next result. The same buffer is used for both transmitter and receiver results. LOOPIT examines these pointers to detect when CNADR is not equal to LDADR indicating that the buffer contains results which have not been displayed. When this occurs, the program branches to the DISPLY routine.

DISPLY determines the source of the undisplayed results by testing the first result. This first result is not necessarily the interrupt result code. If this result is 0CH or greater, the result is from a transmitter interrupt. Otherwise it is from a receiver source. The source of the result code is then displayed on the console along with the next four results from the buffer. If the source was a transmitter interrupt, the routine merely repoints the pointer CNADR and returns to LOOPIT. For a receiver source, the receiver data buffer is displayed in addition to the receiver interrupt results before returning to LOOPIT.

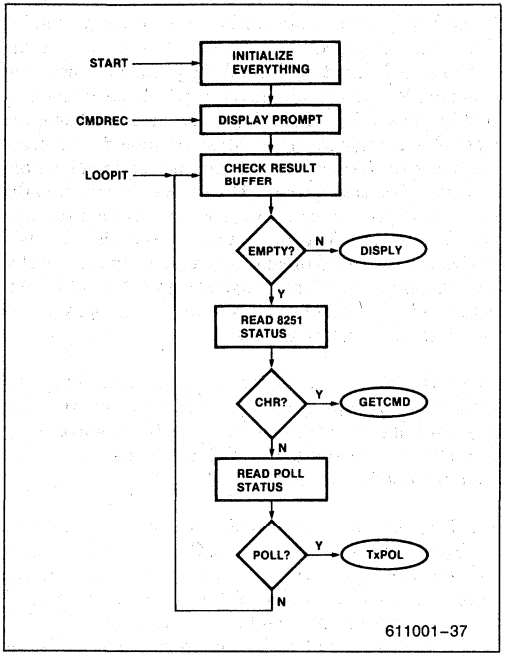


Figure 44. Main Status Poll Loop

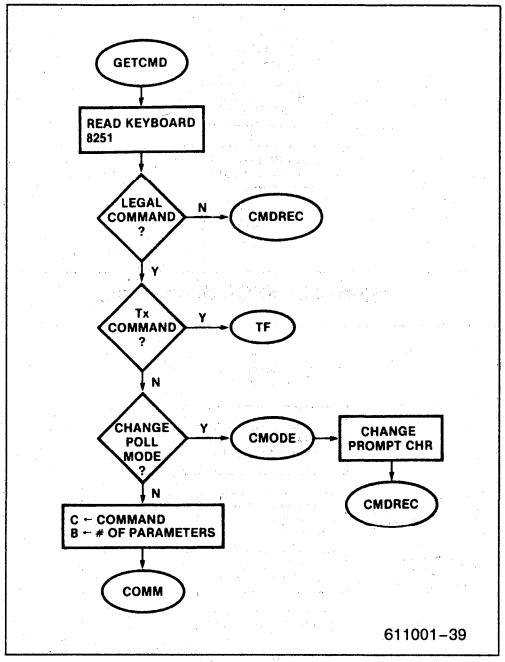


Figure 46. GETCMD Subroutine

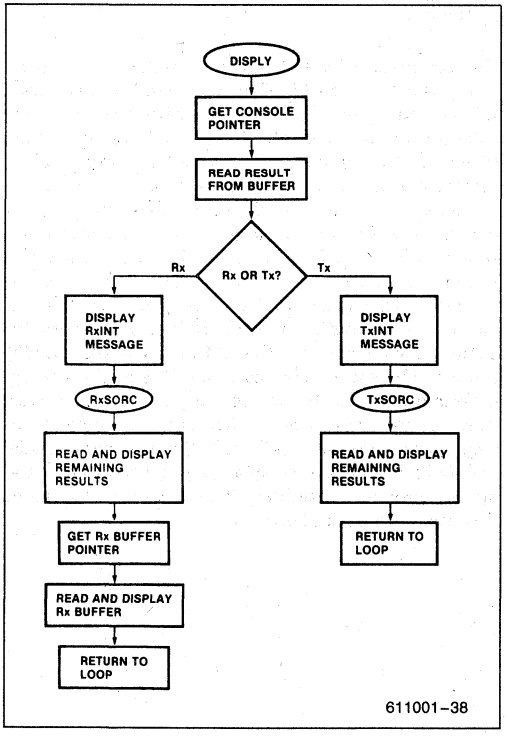


Figure 45. DISPLY Subroutine

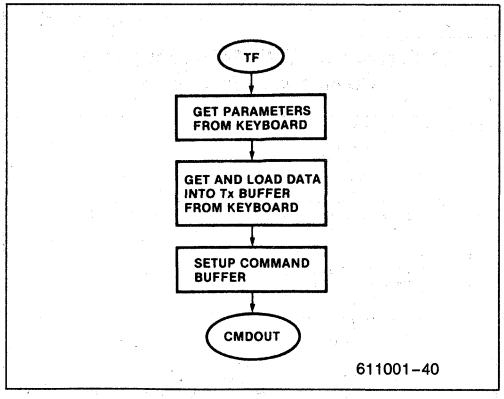


Figure 47. TF Subroutine

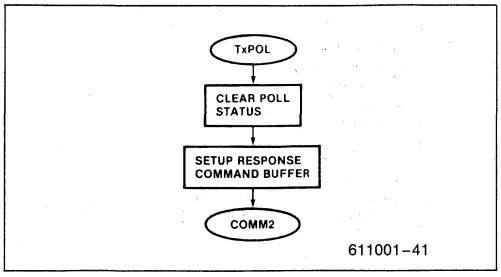


Figure 48. TxPOL Subroutine

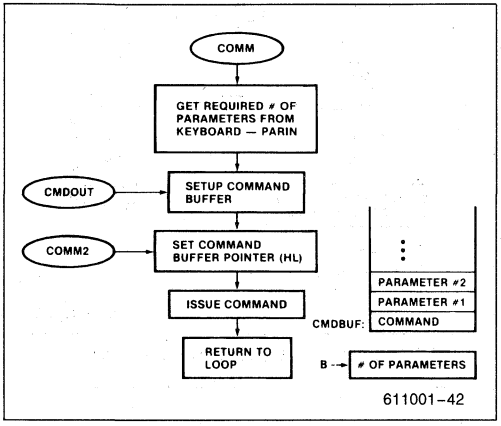


Figure 49. COMM Subroutine with Command Buffer Format

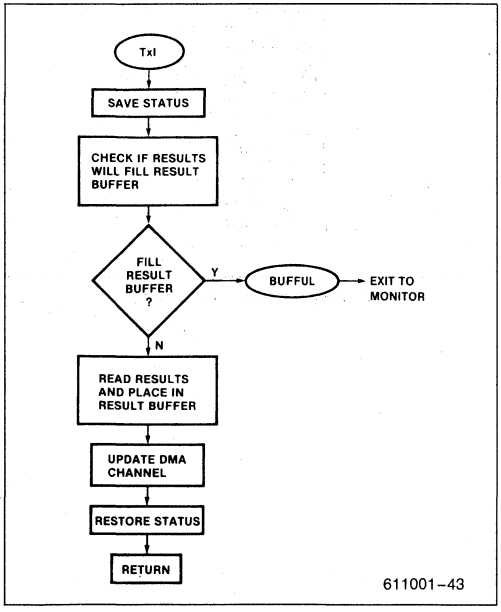


Figure 50. TxI (Transmitter Interrupt) Routine

If the result buffer pointers indicate an empty buffer, the 8251A is polled for a keyboard character. If the 8251 has a character, GETCMD is called. There the character is read and checked if legal. Illegal characters simply cause a reprompt. Legal characters indicate the start of a command input. Most commands are organized as two characters signifying the command action; i.e., GR—General Receive. The software recognizes the two character command code and takes the appropriate action. For non-Transmit type commands, the hex equivalent of the command is placed in the C register and the number of parameters associated with that command is placed in the B register. The program then branches to the COMM routine.

The COMM routine builds the command buffer by reading the required number of parameters from the keyboard and placing them at the buffer pointed at by CMDBUF. The routine at COMM2 then issues this command buffer to the 8273.

If a Transmit type command is specified, the command buffer is set up similarly to the COMM routine; however, since the information field data is entered from the keyboard, an intermediate routine, TF, is called. TF loads the transmit data buffer pointed at by TxBUF. It counts the number of data bytes entered and loads this number into the command buffer as L_0, L_1. The command is then issued to the 8273 by jumping to CMDOUT.

One command does not directly result in a command being issued to the 8273. This command, Z, operates a software flip-flop which selects whether the software will respond automatically to received polling frames. If the Poll-Response mode is selected, the prompt character is changed to a '+'. If a frame is received which contains a prearranged poll control field, the memory location POLIN is made nonzero by the receiver interrupt handler. LOOPIT examines this location and if it is nonzero, causes a branch to the TxPOL routine. The TxPOL routine clears POLIN, sets a pointer to a special command buffer at CMDBUF1, and issues the command by way of the COMM2 entry in the COMM routine. The special command buffer contains the appropriate response frame for the poll frame received. These actions only occur when the Z command has changed the prompt to a '+'. If the prompt is normal '−', polling frames are displayed as normal frames and no response is transmitted. The Poll-Response mode was used during the IBM tests.

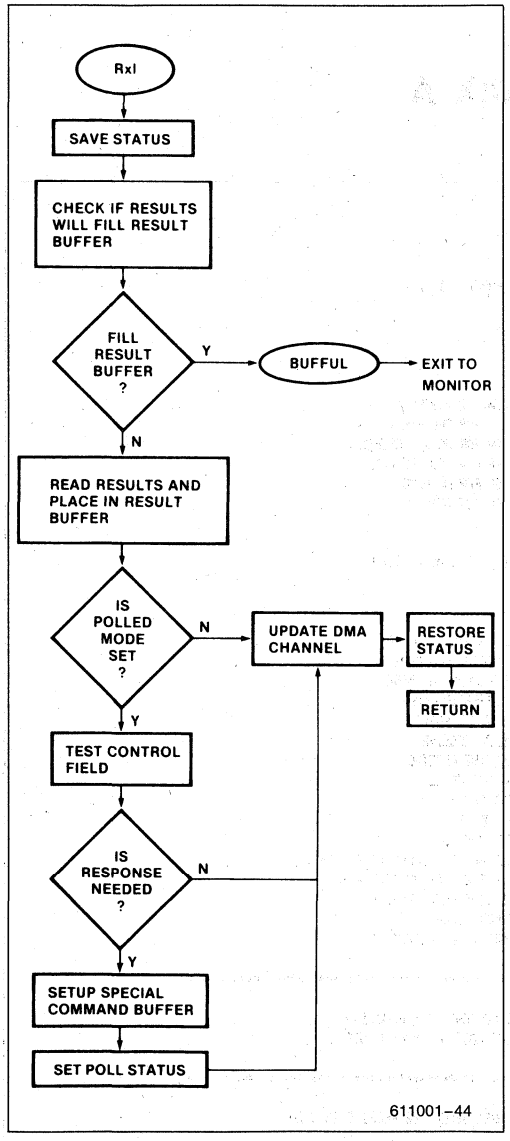


Figure 51. RxI (Receiver Interrupt) Routine

The final two software routines are the transmitter and receiver interrupt handlers. The transmit interrupt handler, TxI, simply saves the registers on the stack and checks if loading the result buffer will fill it. If the result buffer will overfill, the program is exited and control is passed to the SDK-85 monitor. If not, the results are read from the TxI/R register and placed in the result buffer at LDADR. The DMA pointers are then reset, the registers restored, and interrupts enabled. Execution then returns to the pre-interrupt location.

The receiver interrupt handler, RxI, is only slightly more complex. As in TxI, the registers are saved and the possibility of overfilling the result buffer is examined. If the result buffer is not full, the results are read from RxI/R and placed in the buffer. At this point the prompt character is examined to see if the Poll-Response mode is selected. If so, the control field is compared with two possible polling control fields. If there is a match, the special command buffer is loaded and the poll indicator, POLIN, is made nonzero. If no match occurred, no action is taken. Finally, the receiver DMA buffer pointers are reset, the processor status restored, and interrupts are enabled. The RET instruction returns execution to the pre-interrupt location.

This completes the discussion of the 8273/8085 system design.

CONCLUSION

This application note has covered the 8273 in some detail. The simple and low cost loop configuration was explored and an 8273/8085 system was presented as a sample design illustrating the DMA/interrupt-driven interface. It is hoped that the major features of the 8273, namely the frame-level command structure and the Digital Phase Locked Loop, have been shown to be a valuable asset in an SDLC system design.

APPENDIX A

```
ASM80 :F1:RAYT73.SRC

ISIS-II 8080/8085 MACRO ASSEMBLER, X108        MODULE   PAGE    1

 LOC  OBJ        SEQ        SOURCE STATEMENT

                   1 $NOPAGING MOD85  NOCOND
 0000              2 TRUE    EQU     00H             ;00 FOR RAYTHEON
                   3 ;                               ;FF FOR SELF-TEST
 0000              4 TRUE1   EQU     00H             ;00 FOR NORMAL RESPONSE
                   5 ;                               ;FF FOR LOOP RESPONSE
 0000              6 DEM     EQU     00H             ;00 FOR NO DEMO
                   7 ;                               ;FF FOR DEMO
                   8 ;
                   9 ;
                  10 ;GENERAL 8273 MONITOR WITH RAYTHEON POLL MODE ADDED
                  11 ;
                  17 ;
                  18 ;
                  19 ;COMMAND SUPPORTED ARE: RS - RESET SERIAL I/O MODE
                  20 ;                       SS - SET SERIAL I/O MODE
                  21 ;                       RO - RESET OPERATING MODE
                  22 ;                       SO - SET OPERATING MODE
                  23 ;                       RD - RECEIVER DISABLE
                  24 ;                       GR - GENERAL RECEIVE
                  25 ;                       SR - SELECTIVE RECEIVE
                  26 ;                       TF - TRANSMIT FRAME
                  27 ;                       AF - ABORT FRAME
                  28 ;                       SP - SET PORT B
                  29 ;                       RP - RESET PORT B
                  30 ;                       RB - RESET ONE BIT DELAY (PAR = 7F)
                  31 ;                       SB - SET ONE BIT DELAY (PAR = 80)
                  32 ;                       SL - SELECTIVE LOOP RECEIVE
                  33 ;                       TL - TRANSMIT LOOP
                  34 ;                       Z  - CHANGE MODES FLIP/FLOP
                  38 ;
                  39 ;*************************************************************************
                  40 ;
                  41 ;NOTE:  'SET' COMMANDS IMPLEMENT LOGICAL 'OR' FUNCTIONS
                  42 ;       'RESET' COMMANDS IMPLEMENT LOGICAL 'AND' FUNCTIONS
                  43 ;
                  44 ;*************************************************************************
                  45 ;
                  46 ;BUFFERED MODE MUST BE SELECTED WHEN SELECTIVE RECEIVE IS USED.
                  47 ;
                  48 ;COMMAND FORMAT IS: 'COMMAND (2 LTRS)' 'PAR.#1' 'PAR.#2' ETC.
                  49 ;
                  50 ;THE TRANSMIT FRAME COMMAND FORMAT IS: 'TF' 'A' 'C' 'BUFFER CONTENTS'.
                  51 ;       NO LENGTH COUNT IS NEEDED.  BUFFER CONTENTS IS ENDED WITH A CR.
                  52 ;
                  53 ;*************************************************************************
                  54 ;
                  55 ;POLLED MODE:   WHEN POLLED MODE IS SELECTED (DENOTED BY A '+' PROMPT), IF
```

611001-45

```
                        ;             A SNRM-P OR RR(0)-P IS RECEIVED, A RESPONSE FRAME OF NSA-F
                        ;             OR RR(0)-F IS TRANSMITTED.  OTHER COMMANDS OPERATE NORMALLY.
                        ;
                        ;*************************************************************************
                        ;
                        ;8273 EQUATES
                        ;
0090                    STAT73  EQU     90H             ;STATUS REGISTER
0090                    COMM73  EQU     90H             ;COMMAND REGISTER
0091                    PARM73  EQU     91H             ;PARAMETER REGISTER
0091                    RESL73  EQU     91H             ;RESULT REGISTER
0092                    TXIR73  EQU     92H             ;TX INTERRUPT RESULT REGISTER
0093                    RXIR73  EQU     93H             ;RX INTERRUPT RESULT REGISTER
0092                    TEST73  EQU     92H             ;TEST MODE REGISTER
0020                    CPBF    EQU     20H             ;PARAMETER BUFFER FULL BIT
0004                    TXINT   EQU     04H             ;TX INTERRUPT BIT IN STATUS REGISTER
0008                    RXINT   EQU     08H             ;RX INTERRUPT BIT IN STATUS REGISTER
0001                    TXIRA   EQU     01H             ;TX INT RESULT AVAILABLE BIT
0002                    RXIRA   EQU     02H             ;RX INT RESULT AVAILABLE BIT
                        ;
                        ;8253 EQUATES
                        ;
009B                    MODE53  EQU     9BH             ;8253 MODE WORD REGISTER
009C                    CNT053  EQU     9CH             ;COUNTER 0 REGISTER
009D                    CNT153  EQU     9DH             ;COUNTER 1 REGISTER
009E                    CNT253  EQU     9EH             ;COUNTER 2 REGISTER
000C                    COBR    EQU     000CH           ;CONSOLE BAUD RATE (2400)
0036                    MDCNT0  EQU     36H             ;MODE FOR COUNTER 0
00B6                    MDCNT2  EQU     0B6H            ;MODE FOR COUNTER 2
2017                    LKBR1   EQU     2017H           ;8273 BAUD RATE LSB ADR
2018                    LKBR2   EQU     2018H           ;8273 BAUD RATE MSB ADR
                        ;
                        ;BAUD RATE TABLE:       BAUD RATE       LKBR1   LKBR2
                        ;                       *********       *****   *****
                        ;                         9600           2E      00
                        ;                         4800           5C      00
                        ;                         2400           B9      00
                        ;                         1200           72      01
                        ;                          600           E5      02
                        ;                          300           C9      05
                        ;
                        ;
                        ;8257 EQUATES
                        ;
00A8                    MODE57  EQU     0A8H            ;8257 MODE PORT
00A0                    CH0ADR  EQU     0A0H            ;CH0 (RX) ADR REGISTER
00A1                    CH0TC   EQU     0A1H            ;CH0 TERMINAL COUNT REGISTER
00A2                    CH1ADR  EQU     0A2H            ;CH1 (TX) ADR REGISTER
00A3                    CH1TC   EQU     0A3H            ;CH1 TERMINAL COUNT REGISTER
00A8                    STAT57  EQU     0A8H            ;STATUS REGISTER
8200                    RXBUF   EQU     8200H           ;RX BUFFER START ADDRESS
8000                    TXBUF   EQU     8000H           ;TX BUFFER START ADDRESS
0062                    DRDMA   EQU     62H             ;DISABLE RX DMA CHANNEL, TX STILL ON
41FF                    RXTC    EQU     41FFH           ;TERMINAL COUNT AND MODE FOR RX CHANNEL
0063                    ENDMA   EQU     63H             ;ENABLE BOTH TX AND RX CHANNELS-EXT. WR, TX STOP
0061                    DTDMA   EQU     61H             ;DISABLE TX DMA CHANNEL, RX STILL ON
81FF                    TXTC    EQU     81FFH           ;TERMINAL COUNT AND MODE FOR TX CHANNEL
                        ;
```

611001-46

```
                    118 ;8251A EQUATES
                    119 ;
0089                120 CNTL51  EQU     89H             ;CONTROL WORD REGISTER
0089                121 STAT51  EQU     89H             ;STATUS REGISTER
0088                122 TXD51   EQU     88H             ;TX DATA REGISTER
0088                123 RXD51   EQU     88H             ;RX DATA REGISTER
00CE                124 MDE51   EQU     0CEH            ;MODE 16X,2 STOP,NO PARITY
0027                125 CMD51   EQU     27H             ;COMMAND, ENABLE TX&RX
0002                126 RDY     EQU     02H             ;RXRDY BIT
                    127 ;
                    128 ;MONITOR SUBROUTINE EQUATES
                    129 ;
061F                130 GETCH   EQU     061FH           ;GET CHR FROM KEYBOARD, ASCII IN CH
05F8                131 ECHO    EQU     05F8H           ;ECHO CHR TO DISPLAY
075E                132 VALDG   EQU     075EH           ;CHECK IF VALID DIGIT, CARRY SET IF VALID
05BB                133 CNVBN   EQU     05BBH           ;CONVERTS ASCII TO HEX
05EB                134 CRLF    EQU     05EBH           ;DISPLAY CR, HENCE LF TOO
06C7                135 NMOUT   EQU     06C7H           ;CONVERT BYTE TO 2 ASCII CHR AND DISPLAY
                    136 ;
                    137 ;MISC EQUATES
                    138 ;
20C0                139 STKSRT  EQU     20C0H           ;STACK START
0003                140 CNTLC   EQU     03H             ;CNTL-C EQUIVALENT
0008                141 MONTOR  EQU     0008H           ;MONITOR
2000                142 CMDBUF  EQU     2000H           ;START OF COMMAND BUFFER
2020                143 CMDBF1  EQU     2020H           ;POLL MODE SPECIAL TX COMMAND BUFFER
000D                144 CR      EQU     0DH             ;ASCII CR
000A                145 LF      EQU     0AH             ;ASCII LF
20D4                146 RST75   EQU     20D4H           ;RST7.5 JUMP ADDRESS
20CE                147 RST65   EQU     20CEH           ;RST6.5 JUMP ADDRESS
2010                148 LDADR   EQU     2010H           ;RESULT BUFFER LOAD POINTER STORAGE
2013                149 CNADR   EQU     2013H           ;RESULT BUFFER CONSOLE POINTER STORAGE
2800                150 RESBUF  EQU     2800H           ;RESULT BUFFER START - 255 BYTES
0093                151 SNRMP   EQU     93H             ;SNRM-P CONTROL CODE
0011                152 RR0P    EQU     11H             ;RR(0)-P CONTROL CODE
0073                153 NSAF    EQU     73H             ;NSA-F CONTROL CODE
0011                154 RR0F    EQU     11H             ;RR(0)-F CONTROL CODE
2015                155 PRMPT   EQU     2015H           ;PRMPT STORAGE
2016                156 POLIN   EQU     2016H           ;POLL MODE SELECTION INDICATOR
2027                157 DEMODE  EQU     2027H           ;DEMO MODE INDICATOR
                    161 ;
                    162 ;*************************************************************************
                    163 ;
                    164 ;RAM STORAGE DEFINITIONS:
                    165 ;       LOC             DEF
                    166 ;       ---             ---
                    167 ;       2000-200F       COMMAND BUFFER
                    168 ;       2010-2011       RESULT BUFFER LOAD POINTER
                    169 ;       2013-2014       RESULT BUFFER CONSOLE POINTER
                    170 ;       2015            PROMPT CHARACTER STORAGE
                    171 ;       2016            POLL MODE INDICATOR
                    172 ;       2017            BAUD RATE LSB FOR SELF-TEST
                    173 ;       2018            BAUD RATE MSB FOR SELF-TEST
                    177 ;       2019            SPARE
                    179 ;       2020-2026       RESPONSE COMMAND BUFFER FOR POLL MODE
                    180 ;       2800-28FF       RESULT BUFFER
                    181 ;
                    182 ;*************************************************************************
```

611001-47

```
                        183 ;
                        184 ;PROGRAM START
                        185 ;
                        186 ;INITIALIZE 8253, 8257, 8251A, AND RESET 8273.
                        187 ;ALSO SET NORMAL MODE, AND PRINT SIGNON MESSAGE
                        188 ;
0800                    189         ORG     800H
                        190
0800 31C020             191 START:  LXI     SP,STKSRT       ;INITIALIZE SP
0803 3E36               192         MVI     A,MDCNT0        ;8253 MODE SET
0805 D39B               193         OUT     MODE53          ;8253 MODE PORT
0807 3A1720             194         LDA     LKBR1           ;GET 8273 BAUD RATE LSB
080A D39C               195         OUT     CNT053          ;USING COUNTER 0 AS BAUD RATE GEN
080C 3A1820             196         LDA     LKBR2           ;GET 8273 BUAD RATE MSB
080F D39C               197         OUT     CNT053          ;COUNTER 0
0811 CD1A0B             198         CALL    RXDMA           ;INITIALIZE 8257 RX DMA CHANNEL
0814 CD350B             199         CALL    TXDMA           ;INITIALIZE 8257 TX DMA CHANNEL
0817 3E01               200         MVI     A,01H           ;OUTPUT 1 FOLLOWED BY A 0
0819 D392               201         OUT     TEST73          ;TO TEST MODE REGISTER
081B 3E00               202         MVI     A,00H           ;TO RESET THE 8273
081D D392               203         OUT     TEST73
081F 3E2D               204         MVI     A,'-'           ;NORMAL MODE PROMPT CHR
0821 321520             205         STA     PRMPT           ;PUT IN STORAGE
0824 3E00               206         MVI     A,00H           ;TX POLL RESPONSE INDICATOR
0826 321620             207         STA     POLIN           ;0 MEANS NO SPECIAL TX
0829 322720             208         STA     DEMODE          ;CLEAR DEMO MODE
082C 21A30C             212         LXI     H,SIGNON                ;SIGNON MESSAGE ADR
082F CD920C             213         CALL    TYMSG           ;DISPLAY SIGNON
                        214 ;
                        215 ;MONITOR USES JUMPS IN RAM TO DIRECT INTERRUPTS
                        216 ;
0832 21D420             217         LXI     H,RST75         ;RST7.5 JUMP LOCATION USED BY MONITOR
0835 01000C             218         LXI     B,RXI           ;ADDRESS OF RX INT ROUTINE
0838 36C3               219         MVI     M,0C3H          ;LOAD 'JMP' OPCODE
083A 23                 220         INX     H               ;INC POINTER
083B 71                 221         MOV     M,C             ;LOAD RXI LSB
083C 23                 222         INX     H               ;INC POINTER
083D 70                 223         MOV     M,B             ;LOAD RXI MSB
083E 21CE20             224         LXI     H,RST65         ;RST6.5 JUMP LOCATION USED BY MONITOR
0841 01CE0C             225         LXI     B,TXI           ;ADDRESS OF TX INT ROUTINE
0844 36C3               226         MVI     M,0C3H          ;LOAD 'JMP' OPCODE
0846 23                 227         INX     H               ;INC POINTER
0847 71                 228         MOV     M,C             ;LOAD TXI LSB
0848 23                 229         INX     H               ;INC POINTER
0849 70                 230         MOV     M,B             ;LOAD TXI MSB
084A 3E18               231         MVI     A,18H           ;GET SET TO RESET INTERRUPTS
084C 30                 232         SIM                     ;RESET INTERRUPTS
084D FB                 233         EI                      ;ENABLE INTERRUPTS
                        234 ;
                        235 ;INITIALIZE BUFFER POINTER
                        236 ;
                        237 ;
084E 210028             238         LXI     H,RESBUF        ;SET RESULT BUFFER POINTERS
0851 221320             239         SHLD    CNADR           ;RESULT CONSOLE POINTER
0854 221020             240         SHLD    LDADR           ;RESULT LOAD POINTER
                        241 ;
                        242 ;MAIN PROGRAM LOOP - CHECKS FOR CHANGE IN RESULT POINTERS, USART STATUS,
                        243 ;       OR POLL STATUS
```

611001-48

```
                        244 ;
0857 CDEB05             245 CMDREC:  CALL    CRLF            ;DISPLAY CR
085A 3A1520             246          LDA     PRMPT           ;GET CURRENT PROMPT CHR
085D 4F                 247          MOV     C,A             ;MOVE TO C
085E CDF805             248          CALL    ECHO            ;DISPLAY IT
0861 2A1320             249 LOOPIT:  LHLD    CNADR           ;GET CONSOLE POINTER
0864 7D                 250          MOV     A,L             ;SAVE POINTER LSB
0865 2A1020             251          LHLD    LDADR           ;GET LOAD POINTER
0868 BD                 252          CMP     L               ;SAME LSB?
0869 C2390A             253          JNZ     DISPY           ;NO, RESULTS NEED DISPLAYING
086C DB89               259          IN      STAT51          ;YES, CHECK KEYBOARD
086E E602               260          ANI     RDY             ;CHR RECEIVED?
0870 C27D08             261          JNZ     GETCMD          ;MUST BE CHR SO GO GET IT
0873 3A1620             262          LDA     POLIN           ;GET POLL MODE STATUS
0876 A7                 263          ANA     A               ;IS IT 0?
0877 C24C09             264          JNZ     TXPOL           ;NO, THEN POLL OCCURRED
087A C36108             265          JMP     LOOPIT          ;YES, TRY AGAIN
                        266 ;
                        267 ;
                        268 ;COMMAND RECOGNIZER ROUTINE
                        269 ;
                        270 ;
087D CD1F06             271 GETCMD:  CALL    GETCH           ;GET CHR
0880 CDF805             272          CALL    ECHO            ;ECHO IT
0883 79                 273          MOV     A,C             ;SETUP FOR COMPARE
0884 FE52               274          CPI     'R'             ;R?
0886 CAAF08             275          JZ      RDWN            ;GET MORE
0889 FE53               276          CPI     'S'             ;S?
088B CAD708             277          JZ      SDWN            ;GET MORE
088E FE47               278          CPI     'G'             ;G?
0890 CAFF08             279          JZ      GDWN            ;GET MORE
0893 FE54               280          CPI     'T'             ;T?
0895 CA0E09             281          JZ      TDWN            ;GET MORE
0898 FE41               282          CPI     'A'             ;A?
089A CA2209             283          JZ      ADWN            ;GET MORE
089D FE5A               284          CPI     'Z'             ;Z?
089F CA3109             285          JZ      CMODE           ;YES, GO CHANGE MODE
08A2 FE03               290          CPI     CNTLC           ;CNTL-C?
08A4 CA0800             291          JZ      MONTOR          ;EXIT TO MONITOR
08A7 0E3F               292 ILLEG:   MVI     C,'?'           ;PRINT ?
08A9 CDF805             293          CALL    ECHO            ;DISPLAY IT
08AC C35708             294          JMP     CMDREC          ;LOOP FOR COMMAND
                        295
08AF CD1F06             296 RDWN:    CALL    GETCH           ;GET NEXT CHR
08B2 CDF805             297          CALL    ECHO            ;ECHO IT
08B5 79                 298          MOV     A,C             ;SETUP FOR COMPARE
08B6 FE4F               299          CPI     'O'             ;O?
08B8 CA5D09             300          JZ      ROCMD           ;RO COMMAND
08BB FE53               301          CPI     'S'             ;S?
08BD CA6709             302          JZ      RSCMD           ;RS COMMAND
08C0 FE44               303          CPI     'D'             ;D?
08C2 CA7109             304          JZ      RDCMD           ;RD COMMAND
08C5 FE50               305          CPI     'P'             ;P?
08C7 CAD809             306          JZ      RPCMD           ;RP COMMAND
08CA FE52               307          CPI     'R'             ;R?
08CC CA0008             308          JZ      START           ;START OVER
08CF FE42               309          CPI     'B'             ;B?
08D1 CA7B09             310          JZ      RBCMD           ;RB COMMAND
```

611001-49

```
08D4 C3A708      311          JMP     ILLEG          ;ILLEGAL, TRY AGAIN
                 312
08D7 CD1F06      313 SDWN:    CALL    GETCH          ;GET NEXT CHR
08DA CDF805      314          CALL    ECHO           ;ECHO IT
08DD 78          315          MOV     A,B            ;SETUP FOR COMPARE
08DE FE4F        316          CPI     'O'            ;O?
08E0 CAA609      317          JZ      SOCMD          ;SO COMMAND
08E3 FE53        318          CPI     'S'            ;S?
08E5 CAB009      319          JZ      SSCMD          ;SS COMMAND
08E8 FE52        320          CPI     'R'            ;R?
08EA CABA09      321          JZ      SRCMD          ;SR COMMAND
08ED FE50        322          CPI     'P'            ;P?
08EF CAE209      323          JZ      SPCMD          ;SP COMMAND
08F2 FE42        324          CPI     'B'            ;B?
08F4 CA8509      325          JZ      SBCMD          ;SB COMMAND
08F7 FE4C        326          CPI     'L'            ;L?
08F9 CA8F09      327          JZ      SLCMD          ;SL COMMAND
08FC C3A708      328          JMP     ILLEG          ;ILLEGAL, TRY AGAIN
                 329
08FF CD1F06      330 GDWN:    CALL    GETCH          ;GET NEXT CHR
0902 CDF805      331          CALL    ECHO           ;ECHO IT
0905 78          332          MOV     A,B            ;SETUP FOR COMPARE
0906 FE52        333          CPI     'R'            ;R?
0908 CAC409      334          JZ      GRCMD          ;GR COMMAND
090B C3A708      335          JMP     ILLEG          ;ILLEGAL, TRY AGAIN
                 336
090E CD1F06      337 TDWN:    CALL    GETCH          ;GET NEXT CHR
0911 CDF805      338          CALL    ECHO           ;ECHO IT
0914 78          339          MOV     A,B            ;SETUP FOR COMPARE
0915 FE46        340          CPI     'F'            ;F?
0917 CAEC09      341          JZ      TFCMD          ;TF COMMAND
091A FE4C        342          CPI     'L'            ;L?
091C CA9909      343          JZ      TLCMD          ;TL COMMAND
091F C3A708      344          JMP     ILLEG          ;ILLEGAL, TRY AGAIN
                 345
0922 CD1F06      346 ADWN:    CALL    GETCH          ;GET NEXT CHR
0925 CDF805      347          CALL    ECHO           ;ECHO IT
0928 78          348          MOV     A,B            ;SETUP FOR COMPARE
0929 FE46        349          CPI     'F'            ;F?
092B CACE09      350          JZ      AFCMD          ;AF COMMAND
092E C3A708      351          JMP     ILLEG          ;ILLEGAL, TRY AGAIN
                 352 ;
                 353 ;RESET POLL MODE RESPONSE - CHANGE PROMPT CHR AS INDICATOR
                 354 ;
0931 F3          355 CMODE:   DI                     ;DISABLE INTERRUPTS
0932 3A1520      356          LDA     PRMPT          ;GET CURRENT PROMPT
0935 FE2D        357          CPI     '-'            ;NORMAL MODE?
0937 C24309      358          JNZ     SW             ;NO, CHANGE IT
093A 3E2B        359          MVI     A,'+'          ;NEW PROMPT
093C 321520      360          STA     PRMPT          ;STORE NEW PROMPT
093F FB          365          EI                     ;ENABLE INTERRUPTS
0940 C35708      366          JMP     CMDREC         ;RETURN TO LOOP
0943 3E2D        367 SW:      MVI     A,'-'          ;NEW PROMPT CHR
0945 321520      368          STA     PRMPT          ;STORE IT
0948 FB          369          EI                     ;ENABLE INTERRUPTS
0949 C35708      370          JMP     CMDREC         ;RETURN TO LOOP
                 371 ;
                 372 ;
```

611001-50

```
                        373 ;TRANSMIT ANSWER TO POLL SETUP
                        374 ;
094C 3E00               382 TXPOL:  MVI     A,00H           ;CLEAR POLL INDICATOR
094E 321620             384         STA     POLIN           ;INDICATOR ADR
0951 216108             385         LXI     H,LOOPIT        ;SETUP STACK FOR COMMAND OUTPUT
0954 E5                 386         PUSH    H               ;PUT RETURN TO CMDREC ON STACK
0955 0604               387         MVI     B,04H           ;GET # OF PARAMETERS READY
0957 212020             388         LXI     H,CMDBF1        ;POINT TO SPECIAL BUFFER
095A C3FF0A             389         JMP     COMM2           ;JUMP TO COMMAND OUTPUTER
                        390 ;
                        391 ;
                        392 ;
                        393 ;COMMAND IMPLEMENTING ROUTINES
                        394 ;
                        395 ;
                        396 ;RO - RESET OPERATING MODE
                        397 ;
095D 0601               398 ROCMD:  MVI     B,01H           ;# OF PARAMETERS
095F 0E51               399         MVI     C,51H           ;COMMAND
0961 CDE50A             400         CALL    COMM            ;GET PARAMETERS AND ISSUE COMMAND
0964 C35708             401         JMP     CMDREC          ;GET NEXT COMMAND
                        402 ;
                        403 ;RS - RESET SERIAL I/O MODE COMMAND
                        404 ;
0967 0601               405 RSCMD:  MVI     B,01H           ;# OF PARAMETERS
0969 0E60               406         MVI     C,60H           ;COMMAND
096B CDE50A             407         CALL    COMM            ;GET PARAMETERS AND ISSUE COMMAND
096E C35708             408         JMP     CMDREC          ;GET NEXT COMMAND
                        409 ;
                        410 ;RD - RECEIVER DISABLE COMMAND
                        411 ;
0971 0600               412 RDCMD:  MVI     B,00H           ;# OF PARAMETERS
0973 0EC5               413         MVI     C,0C5H          ;COMMAND
0975 CDE50A             414         CALL    COMM            ;ISSUE COMMAND
0978 C35708             415         JMP     CMDREC          ;GET NEXT COMMAND
                        416 ;
                        417 ;RB - RESET ONE BIT DELAY COMMAND
                        418 ;
097B 0601               419 RBCMD:  MVI     B,01H           ;# OF PARAMETERS
097D 0E64               420         MVI     C,64H           ;COMMAND
097F CDE50A             421         CALL    COMM            ;GET PARAMETER AND ISSUE COMMAND
0982 C35708             422         JMP     CMDREC          ;GET NEXT COMMAND
                        423 ;
                        424 ;SB - SET ONE BIT DELAY COMMAND
                        425 ;
0985 0601               426 SBCMD:  MVI     B,01H           ;# OF PARAMETERS
0987 0EA4               427         MVI     C,0A4H          ;COMMAND
0989 CDE50A             428         CALL    COMM            ;GET PARAMETER AND ISSUE COMMAND
098C C35708             429         JMP     CMDREC          ;GET NEXT COMMAND
                        430 ;
                        431 ;SL - SELECTIVE LOOP RECEIVE COMMAND
                        432 ;
098F 0604               433 SLCMD:  MVI     B,04H           ;# OF PARAMETERES
0991 0EC2               434         MVI     C,0C2H          ;COMMAND
0993 CDE50A             435         CALL    COMM            ;GET PARAMETERS AND ISSUE COMMAND
0996 C35708             436         JMP     CMDREC          ;GET NEXT COMMAND
                        437 ;
                        438 ;TL - TRANSMIT LOOP COMMAND
```

611001-51

```
                        439 ;
0999 210020             440 TLCMD:  LXI     H,CMDBUF        ;SET COMMAND BUFFER POINTER
099C 0602               441         MVI     B,02H           ;LOAD PARAMETER COUNTER
099E 36CA               442         MVI     M,0CAH          ;LOAD COMMAND INTO BUFFER
09A0 210220             443         LXI     H,CMDBUF+2      ;POINT AT ADR AND CNTL POSITIONS
09A3 C3F609             444         JMP     TFCMD1          ;FINISH OFF COMMAND IN TF ROUTINE
                        445 ;
                        446 ;SO - SET OPERATING MODE COMMAND
                        447 ;
09A6 0601               448 SOCMD:  MVI     B,01H           ;# OF PARAMETERS
09A8 0E91               449         MVI     C,91H           ;COMMAND
09AA CDE50A             450         CALL    COMM            ;GET PARAMETER AND ISSUE COMMAND
09AD C35708             451         JMP     CMDREC          ;GET NEXT COMMAND
                        452 ;
                        453 ;SS - SET SERIAL I/O COMMAND
                        454 ;
09B0 0601               455 SSCMD:  MVI     B,01H           ;# OF PARAMETERS
09B2 0EA0               456         MVI     C,0A0H          ;COMMAND
09B4 CDE50A             457         CALL    COMM            ;GET PARAMETER AND ISSUE COMMAND
09B7 C35708             458         JMP     CMDREC          ;GET NEXT COMMAND
                        459 ;
                        460 ;SR - SELECTIVE RECEIVE COMMAND
                        461 ;
09BA 0604               462 SRCMD:  MVI     B,04H           ;# OF PARAMETERS
09BC 0EC1               463         MVI     C,0C1H          ;COMMAND
09BE CDE50A             464         CALL    COMM            ;GET PARAMETERS AND ISSUE COMMAND
09C1 C35708             465         JMP     CMDREC          ;GET NEXT COMMAND
                        466 ;
                        467 ;GR - GENERAL RECEIVE COMMAND
                        468 ;
09C4 0602               469 GRCMD:  MVI     B,02H           ;NO PARAMETERS
09C6 0EC0               470         MVI     C,0C0H          ;COMMAND
09C8 CDE50A             471         CALL    COMM            ;ISSUE COMMAND
09CB C35708             472         JMP     CMDREC          ;GET NEXT COMMAND
                        473 ;
                        474 ;AF - ABORT FRAME COMMAND
                        475 ;
09CE 0600               476 AFCMD:  MVI     B,00H           ;NO PARAMETERS
09D0 0ECC               477         MVI     C,0CCH          ;COMMAND
09D2 CDE50A             478         CALL    COMM            ;ISSUE COMMAND
09D5 C35708             479         JMP     CMDREC          ;GET NEXT COMMAND
                        480 ;
                        481 ;RP - RESET PORT COMMAND
                        482 ;
09D8 0601               483 RPCMD:  MVI     B,01H           ;# OF PARAMETERS
09DA 0E63               484         MVI     C,63H           ;COMMAND
09DC CDE50A             485         CALL    COMM            ;GET PARAMETER AND ISSUE COMMAND
09DF C35708             486         JMP     CMDREC          ;GET NEXT COMMAND
                        487 ;
                        488 ;SP - SET PORT COMMAND
                        489 ;
09E2 0601               490 SPCMD:  MVI     B,01H           ;# OF PARAMETERS
09E4 0EA3               491         MVI     C,0A3H          ;COMMAND
09E6 CDE50A             492         CALL    COMM            ;GET PARAMETER AND ISSUE COMMAND
09E9 C35708             493         JMP     CMDREC          ;GET NEX COMMAND
                        494 ;
                        495 ;TF - TRANSMIT FRAME COMMAND
                        496 ;
```

611001-52

```
09EC 210020      497 TFCMD:  LXI    H,CMDBUF       ;SET COMMAND BUFFER POINTER
09EF 0602        498         MVI    B,02H          ;LOAD PARAMETER COUNTER
09F1 36C8        499         MVI    M,0C8H         ;LOAD COMMAND INTO BUFFER
09F3 210220      500         LXI    H,CMDBUF+2     ;POINT AT ADR AND CNTL POSITIONS
09F6 78          501 TFCMD1: MOV    A,B            ;TEST PARAMETER COUNT
09F7 A7          502         ANA    A              ;IS IT 0?
09F8 CA070A      503         JZ     TBUFL          ;YES, LOAD TX DATA BUFFER
09FB CDAD0A      504         CALL   PARIN          ;GET PARAMETER
09FE DAA708      505         JC     ILLEG          ;ILLEGAL CHR RETURNED
0A01 23          506         INX    H              ;INC COMMAND BUFFER POINTER
0A02 05          507         DCR    B              ;DEC PARAMETER COUNTER
0A03 77          508         MOV    M,A            ;LOAD PARAMETER INTO COMMAND BUFFER
0A04 C3F609      509         JMP    TFCMD1         ;GET NEXT PARAMETER
                 510 .
0A07 210080      511 TBUFL:  LXI    H,TXBUF        ;LOAD TX DATA BUFFER POINTER
0A0A 010000      512         LXI    B,0000H        ;CLEAR BC - BYTE COUNTER
0A0D C5          513 TBUFL1: PUSH   B              ;SAVE BYTE COUNTER
0A0E CDAD0A      514         CALL   PARIN          ;GET DATA, ALIAS PARAMETER
0A11 DA1B0A      515         JC     ENDCHK         ;MAYBE END IF ILLEGAL
0A14 77          516         MOV    M,A            ;LOAD DATA BYTE INTO BUFFER
0A15 23          517         INX    H              ;INC BUFFER POINTER
0A16 C1          518         POP    B              ;RESTORE BYTE COUNTER
0A17 03          519         INX    B              ;INC BYTE COUNTER
0A18 C30D0A      520         JMP    TBUFL1         ;GET NEXT DATA
0A1B FE0D        521 ENDCHK: CPI    CR             ;RETURNED ILLEGAL CHR CR?
0A1D CA240A      522         JZ     TBUFFL         ;YES, THEN TX BUFFER FULL
0A20 C1          523         POP    B              ;RESTORE B TO SAVE STACK
0A21 C3A708      524         JMP    ILLEG          ;ILLEGAL CHR
0A24 C1          525 TBUFFL: POP    B              ;RESTORE BYTE COUNTER
0A25 210120      526         LXI    H,CMDBUF+1     ;POINT INTO COMMAND BUFFER
0A28 71          527         MOV    M,C            ;STORE BYTE COUNT LSB
0A29 23          528         INX    H              ;INC POINTER
0A2A 70          529         MOV    M,B            ;STORE BYTE COUNT MSB
0A2B 0604        530         MVI    B,04H          ;LOAD PARAMETER COUNT INTO B
0A2D 21360A      531         LXI    H,TFRET        ;GET RETURN ADR FOR THIS ROUTINE
0A30 C5          532         PUSH   B              ;PUSH ONCE
0A31 E3          533         XTHL                  ;PUT RETURN ON STACK
0A32 C5          534         PUSH   B              ;PUSH IT SO CMDOUT CAN USE IT
0A33 C3FB0A      535         JMP    CMDOUT         ;ISSUE COMMAND
0A36 C35708      536 TFRET:  JMP    CMDREC         ;GET NEXT COMMAND
                 537 ;
                 538 ;
                 539 ;ROUTINE TO DISPLAY RESULT IN RESULT BUFFER WHEN LOAD AND CONSOLE
                 540 ;POINTERS ARE DIFFERENT.
                 541 ;
                 542 ;
0A39 1605        543 DISPY:  MVI    D,05H          ;D IS RESULT COUNTER
0A3B 2A1320      544         LHLD   CNADR          ;GET CONSOLE POINTER
0A3E E5          545         PUSH   H              ;SAVE IT
0A3F 7E          546         MOV    A,M            ;GET RESULT IC
0A40 E61F        547         ANI    1FH            ;LIMIT TO RESULT CODE
0A42 FE0C        548         CPI    0CH            ;TEST IF RX OR TX SOURCE
0A44 DA620A      549         JC     RXSORC         ;CARRY, THEN RX SOURCE
0A47 21C30C      550 TXSORC: LXI    H,TXIMSG       ;TX INT MESSAGE
0A4A CD920C      551         CALL   TYMSG          ;DISPLAY IT
0A4D E1          552 DISPY2: POP    H              ;RESTORE CONSOLE POINTER
0A4E 7E          553 DISPY1: MOV    A,M            ;GET RESULT
0A4F CDC706      554         CALL   NMOUT          ;CONVERT AND DISPLAY
```

611001-53

```
0A52 0E20       555         MVI     C,' '           ;SP CHR
0A54 CDF805     556         CALL    ECHO            ;DISPLAY IT
0A57 2C         557         INR     L               ;INC BUFFER POINTER
0A58 15         558         DCR     D               ;DEC RESULT COUNTER
0A59 C24E0A     559         JNZ     DISPY1          ;NOT DONE
0A5C 221320     560         SHLD    CNADR           ;UPDATE CONSOLE POINTER
0A5F C35708     561         JMP     CMDREC          ;RETURN TO LOOP
                562 ;
                563 ;
                564 ;RECEIVER SOURCE - DISPLAY RESULTS AND RECEVIE BUFFER CONTENTS
                565 ;
                566 ;
0A62 21B80C     567 RXSORC: LXI     H,RXIMSG        ;RX INT MESSAGE ADR
0A65 CD920C     568         CALL    TYMSG           ;DISPLAY MESSAGE
0A68 E1         569         POP     H               ;RESTORE CONSOLE POINTER
0A69 7E         570 RXS1:   MOV     A,M             ;RETRIEVE RESULT FROM BUFFER
0A6A CDC706     571         CALL    NMOUT           ;CONVERT AND DISPLAY IT
0A6D 0E20       572         MVI     C,' '           ;ASCII SP
0A6F CDF805     573         CALL    ECHO            ;DISPLAY IT
0A72 2C         574         INR     L               ;INC CONSOLE POINTER
0A73 15         575         DCR     D               ;DEC RESULT COUNTER
0A74 7A         576         MOV     A,D             ;GET SET TO TEST COUNTER
0A75 FE04       577         CPI     04H             ;IS THE RESULT R0?
0A77 CA820A     578         JZ      R0PT            ;YES, GO SAVE IT
0A7A FE03       579         CPI     03H             ;IS THE RESULT R1?
0A7C CA870A     580         JZ      R1PT            ;YES, GO SAVE IT
0A7F A7         581 RXS2:   ANA     A               ;TEST RESULT COUNTER
0A80 C2690A     582         JNZ     RXS1            ;NOT DONE YET, GET NEXT RESULT
0A83 221320     583         SHLD    CNADR           ;DONE, SO UPDATE CONSOLE POINTER
0A86 CDEB05     584         CALL    CRLF            ;DISPLAY CR
0A89 210082     585         LXI     H,RXBUF         ;POINT AT RX BUFFER
0A8C C1         586         POP     B               ;RETRIEVE RECEIVED COUNT
0A8D 78         587 RXS3:   MOV     A,B             ;IS COUNT 0?
0A8E B1         588         ORA     C               ;
0A8F CA5708     589         JZ      CMDREC          ;YES, GO BACK TO LOOP
0A92 7E         590         MOV     A,M             ;NO, GET CHR
0A93 C5         591         PUSH    B               ;SAVE BC
0A94 CDC706     592         CALL    NMOUT           ;CONVERT AND DISPLAY CHR
0A97 0E20       593         MVI     C,' '           ;ASCII SP
0A99 CDF805     594         CALL    ECHO            ;DISPLAY IT TO SEPARATE DATA
0A9C C1         595         POP     B               ;RESTORE BC
0A9D 0B         596         DCX     B               ;DEC COUNT
0A9E 23         597         INX     H               ;INC POINTER
0A9F C38D0A     598         JMP     RXS3            ;GET NEXT CHR
                599
0AA2 4E         600 R0PT:   MOV     C,M             ;GET R0 FOR RESULT BUFFER
0AA3 C5         601         PUSH    B               ;SAVE IT
0AA4 C37F0A     602         JMP     RXS2            ;RETURN
                603
0AA7 C1         604 R1PT:   POP     B               ;GET R0
0AA8 46         605         MOV     B,M             ;GET R1 FOR RESULT BUFFER
0AA9 C5         606         PUSH    B               ;SAVE IT
0AAA C37F0A     607         JMP     RXS2
                608 ;
                609 ;
                610 ;
                611 ;PARAMETER INPUT - PARAMETER RETURNED IN E REGISTER
                612 ;
```

611001-54

```
                    613 ;
0AAD C5             614 PARIN:  PUSH    B           ;SAVE BC
0AAE 1601           615         MVI     D,01H       ;SET CHR COUNTER
0AB0 CD1F06         616         CALL    GETCH       ;GET CHR
0AB3 CDF805         617         CALL    ECHO        ;ECHO IT
0AB6 79             618         MOV     A,C         ;PUT CHR IN A
0AB7 FE20           619         CPI     ' '         ;SP?
0AB9 C2E00A         620         JNZ     PARIN1      ;NO, ILLEGAL, TRY AGAIN
0ABC CD1F06         621 PARIN3: CALL    GETCH       ;GET CHR OF PARAMETER
0ABF CDF805         622         CALL    ECHO        ;ECHO IT
0AC2 CD5E07         623         CALL    VALDG       ;IS IT A VALID CHR?
0AC5 D2E00A         624         JNC     PARIN1      ;NO, TRY AGAIN
0AC8 CDB805         625         CALL    CNVBN       ;CONVERT IT TO HEX
0ACB 4F             626         MOV     C,A         ;SAVE IT IN C
0ACC 7A             627         MOV     A,D         ;GET CHR COUNTER
0ACD A7             628         ANA     A           ;IS IT 0?
0ACE CADC0A         629         JZ      PARIN2      ;YES,DONE WITH THIS PARAMETER
0AD1 15             630         DCR     D           ;DEC CHR COUNTER
0AD2 AF             631         XRA     A           ;CLEAR CARRY
0AD3 79             632         MOV     A,C         ;RECOVER 1ST CHR
0AD4 17             633         RAL                 ;ROTATE LEFT 4 PLACES
0AD5 17             634         RAL
0AD6 17             635         RAL
0AD7 17             636         RAL
0AD8 5F             637         MOV     E,A         ;SAVE IT IN E
0AD9 C3BC0A         638         JMP     PARIN3      ;GET NEXT CHR
0ADC 79             639 PARIN2: MOV     A,C         ;2ND CHR IN A
0ADD B3             640         ORA     E           ;COMBINE BOTH CHRS
0ADE C1             641         POP     B           ;RESTORE BC
0ADF C9             642         RET                 ;RETURN TO CALLING PROGRAM
0AE0 79             643 PARIN1: MOV     A,C         ;PUT ILLEGAL CHR IN A
0AE1 37             644         STC                 ;SET CARRY AS ILLEGAL STATUS
0AE2 C1             645         POP     B           ;RESTORE BC
0AE3 C9             646         RET                 ;RETURN TO CALLING PROGRAM
                    647 ;
                    648 ;
                    649 ;JUMP HERE IF BUFFER FULL
                    650 ;
0AE4 CF             651 BUFFUL: DB      0CFH        ;EXIT TO MONITOR
                    652 ;
                    653 ;
                    654 ;COMMAND DISPATCHER
                    655 ;
                    656 ;
0AE5 210020         657 COMM:   LXI     H,CMDBUF    ;SET POINTER
0AE8 C5             658         PUSH    B           ;SAVE BC
0AE9 71             659         MOV     M,C         ;LOAD COMMAND INTO BUFFER
0AEA 78             660 COMM1:  MOV     A,B         ;CHECK PARAMETER COUNTER
0AEB A7             661         ANA     A           ;IS IT 0?
0AEC CAFB0A         662         JZ      CMDOUT      ;YES, GO ISSUE COMMAND
0AEF CDAD0A         663         CALL    PARIN       ;GET PARAMETER
0AF2 DAA708         664         JC      ILLEG       ;ILLEGAL CHR RETURNED
0AF5 23             665         INX     H           ;INC BUFFER POINTER
0AF6 05             666         DCR     B           ;DEC PARAMETER COUNTER
0AF7 77             667         MOV     M,A         ;PARAMETER TO BUFFER
0AF8 C3EA0A         668         JMP     COMM1       ;GET NEXT PARAMETER
0AFB 210020         669 CMDOUT: LXI     H,CMDBUF    ;REPOINT POINTER
0AFE C1             670         POP     B           ;RESTORE PARAMETER COUNT
```

611001-55

```
0AFF DB90       671 COMM2:  IN      STAT73      ;READ 8273 STATUS
0B01 07         672         RLC                 ;ROTATE CBSY INTO CARRY
0B02 DAFF0A     673         JC      COMM2       ;WAIT FOR OK
0B05 7E         674         MOV     A,M         ;OK, MOVE COMMAND INTO A
0B06 D390       675         OUT     COMM73      ;OUTPUT COMMAND
0B08 78         676 PAR1:   MOV     A,B         ;GET PARAMETER COUNT
0B09 A7         677         ANA     A           ;IS IT 0?
0B0A C8         678         RZ                  ;YES, DONE, RETURN
0B0B 23         679         INX     H           ;INC COMMAND BUFFER POINTER
0B0C 05         680         DCR     B           ;DEC PARAMETER COUNT
0B0D DB90       681 PAR2:   IN      STAT73      ;READ STATUS
0B0F E620       682         ANI     CPBF        ;IS CPBF BIT SET?
0B11 C20D0B     683         JNZ     PAR2        ;WAIT TIL ITS 0
0B14 7E         684         MOV     A,M         ;OK, GET PARAMETER FROM BUFFER
0B15 D391       685         OUT     PARM73      ;OUTPUT PARAMETER
0B17 C3080B     686         JMP     PAR1        ;GET NEXT PARAMETER
                687 ;
                688 ;
                689 ;INITIALIZE AND ENABLE RX DMA CHANNEL
                690 ;
                691 ;
0B1A 3E62       692 RXDMA:  MVI     A,DRDMA     ;DISABLE RX DMA CHANNEL
0B1C D3A8       693         OUT     MODE57      ;8257 MODE PORT
0B1E 010082     694         LXI     B,RXBUF     ;RX BUFFER START ADDRESS
0B21 79         695         MOV     A,C         ;RX BUFFER LSB
0B22 D3A0       696         OUT     CH0ADR      ;CH0 ADR PORT
0B24 78         697         MOV     A,B         ;RX BUFFER MSB
0B25 D3A0       698         OUT     CH0ADR      ;CH0 ADR PORT
0B27 01FF41     699         LXI     B,RXTC      ;RX CH TEERMINAL COUNT
0B2A 79         700         MOV     A,C         ;RX TERMINAL COUNT LSB
0B2B D3A1       701         OUT     CH0TC       ;CH0 TC PORT
0B2D 78         702         MOV     A,B         ;RX TERMINAL COUNT MSB
0B2E D3A1       703         OUT     CH0TC       ;CH0 TC PORT
0B30 3E63       704         MVI     A,ENDMA     ;ENABLE DMA WORD
0B32 D3A8       705         OUT     MODE57      ;8257 MODE PORT
0B34 C9         706         RET                 ;RETURN
                707 ;
                708 ;
                709 ;INITIALIZE AND ENABLE TX DMA CHANNEL
                710 ;
                711 ;
0B35 3E61       712 TXDMA:  MVI     A,DTDMA     ;DISABLE TX DMA CHANNEL
0B37 D3A8       713         OUT     MODE57      ;8257 MODE PORT
0B39 010080     714         LXI     B,TXBUF     ;TX BUFFER START ADDRESS
0B3C 79         715         MOV     A,C         ;TX BUFFER LSB
0B3D D3A2       716         OUT     CH1ADR      ;CH1 ADR PORT
0B3F 78         717         MOV     A,B         ;TX BUFFER MSB
0B40 D3A2       718         OUT     CH1ADR      ;CH1 ADR PORT
0B42 01FF81     719 TXDMA1: LXI     B,TXTC      ;TX CH TERMINAL COUNT
0B45 79         720         MOV     A,C         ;TX TERMINAL COUNT LSB
0B46 D3A3       721         OUT     CH1TC       ;CH1 TC PORT
0B48 78         722         MOV     A,B         ;TX TERMINAL COUNT MSB
0B49 D3A3       723         OUT     CH1TC       ;CH1 TC PORT
0B4B 3E63       724         MVI     A,ENDMA     ;ENABLE DMA WORD
0B4D D3A8       725         OUT     MODE57      ;8257 MODE PORT
0B4F C9         726         RET                 ;RETURN
                727 ;
                728 ;
```

611001-56

```
                    729 ;INERRUPT PROCESSING SECTION
                    730 ;
0C00                731         ORG     0C00H
                    732 ;
                    733 ;
                    734 ;RECEIVER INTERRUPT - RST 7.5 (LOC 3CH)
                    735 ;
0C00 E5             736 RXI:    PUSH    H               ;SAVE HL
0C01 F5             737         PUSH    PSW             ;SAVE PSW
0C02 C5             738         PUSH    B               ;SAVE BC
0C03 D5             739         PUSH    D               ;SAVE DE
0C04 3E62           740         MVI     A,DRDMA         ;DISABLE RX DMA
0C06 D3A8           741         OUT     MODE57          ;8257 MODE PORT
0C08 3E18           742         MVI     A,18H           ;RESET RST7.5 F/F
0C0A 30             743         SIM
0C0B 1604           744         MVI     D,04H           ;D IS RESULT COUNTER
0C0D 2A1020         745         LHLD    LDADR           ;GET LOAD POINTER
0C10 E5             746         PUSH    H               ;SAVE IT
0C11 E5             747         PUSH    H               ;SAVE IT AGAIN
0C12 45             748         MOV     B,L             ;SAVE LSB
0C13 2A1320         749         LHLD    CNADR           ;GET CONSOLE POINTER
0C16 04             750 RXI1:   INR     B               ;BUMP LOAD POINTER LSB
0C17 78             751         MOV     A,B             ;GET SET TO TEST
0C18 BD             752         CMP     L               ;LOAD=CONSOLE?
0C19 CAE40A         753         JZ      BUFFUL          ;YES, BUFFER FULL
0C1C 15             754         DCR     D               ;DEC COUNTER
0C1D C2160C         755         JNZ     RXI1            ;NOT DONE, TRY AGAIN
0C20 1605           756         MVI     D,05H           ;RESET COUNTER
0C22 E1             757         POP     H               ;RESTORE LOAD POINTER
0C23 DB90           758 RXI2:   IN      STAT73          ;READ STATUS
0C25 E608           759         ANI     RXINT           ;TEST RX INT BIT
0C27 CA390C         760         JZ      RXI3            ;DONE, GO FINISH UP
0C2A DB90           761         IN      STAT73          ;READ STATUS AGAIN
0C2C E602           762         ANI     RXIRA           ;IS RESULT READY?
0C2E CA230C         763         JZ      RXI2            ;NO, TEST AGAIN
0C31 DB93           764         IN      RXIR73          ;YES, READ RESULT
0C33 77             765         MOV     M,A             ;STORE IN BUFFER
0C34 2C             766         INR     L               ;INC BUFFER POINTER
0C35 15             767         DCR     D               ;DEC COUNTER
0C36 C3230C         768         JMP     RXI2            ;GET MORE RESULTS
0C39 7A             769 RXI3:   MOV     A,D             ;GET SET TO TEST
0C3A A7             770         ANA     A               ;ALL RESULTS?
0C3B CA450C         771         JZ      RXI4            ;YES, SO FINISH UP
0C3E 3600           772         MVI     M,00H           ;NO, LOAD 0 TIL DONE
0C40 2C             773         INR     L               ;BUMP POINTER
0C41 15             774         DCR     D               ;DEC COUNTER
0C42 C3390C         775         JMP     RXI3            ;GO AGAIN
0C45 221020         776 RXI4:   SHLD    LDADR           ;UPDATE LOAD POINTER
0C48 3A1520         777         LDA     PRMPT           ;GET MODE INDICATOR
0C4B FE2D           778         CPI     '-'             ;NORMAL MODE?
0C4D CA850C         779         JZ      RXI6            ;YES, CLEAN UP BEFORE RETURN
                    780 ;
                    781 ;       POLL MODE SO CHECK CONTROL BYTE
                    782 ;       IF CONTROL IS A POLL, SET UP SPECIAL TX COMMAND BUFFER
                    783 ;       AND RETURN WITH POLL INDICATOR NOT 0
                    784 ;
0C50 E1             785         POP     H               ;GET PREVIOUS LOAD ADR POINTER
0C51 7E             786         MOV     A,M             ;GET IC BYTE FROM BUFFER
```

611001-57

```
0C52 E61E       787          ANI    1EH          ;LOOK AT GOOD FRAME BITS
0C54 C2890C     788          JNZ    RXI5         ;IF NOT 0, INTERRUPT WASN'T FROM A GOOD FRAME
0C57 2C         789          INR    L            ;BYPASS R0 AND R1 IN BUFFER
0C58 2C         790          INR    L
0C59 2C         791          INR    L
0C5A 56         792          MOV    D,M          ;GET ADR BYTE AND SAVE IT IN D
0C5B 2C         793          INR    L
0C5C 7E         794          MOV    A,M          ;GET CNTL BYTE FROM BUFFER
0C5D FE93       795          CPI    SNRMP        ;WAS IT SNRM-P?
0C5F CA6C0C     796          JZ     T1           ;YES, GO SET RESPONSE
0C62 FE11       797          CPI    RR0P         ;WAS IT RR(0)-P?
0C64 C2890C     798          JNZ    RXI5         ;YES, GO SET RESPONSE, OTHERWISE RETURN
0C67 1E11       799          MVI    E,RR0F       ;RR(0)-P SO SET RESPONSE TO RR(0)-F
0C69 C36E0C     800          JMP    TXRET        ;GO FINISH LOADING SPECIAL BUFFER
0C6C 1E73       801 T1:      MVI    E,NSAF       ;SNRM-P SO SET RESPONSE TO NSA-F
0C6E 212020     802 TXRET:   LXI    H,CMDBF1     ;SPECIAL BUFFER ADR
0C71 36C8       806          MVI    M,0C8H       ;LOAD TX FRAME COMMAND
0C73 23         808          INX    H            ;INC POINTER
0C74 3600       809          MVI    M,00H        ;L0=0
0C76 23         810          INX    H            ;INC POINTER
0C77 3600       811          MVI    M,00H        ;L1=0
0C79 23         812          INX    H            ;INC POINTER
0C7A 72         813          MOV    M,D          ;LOAD RCVD ADR BYTE
0C7B 23         814          INX    H            ;INC POINTER
0C7C 73         815          MOV    M,E          ;LOAD RESPONSE CNTL BYTE
0C7D 3E01       816          MVI    A,01H        ;SET POLL INDICATOR NOT 0
0C7F 321620     817          STA    POLIN        ;LOAD POLL INDICATOR
0C82 C3890C     818          JMP    RXI5         ;RETURN
                819
0C85 E1         820 RXI6:    POP    H            ;CLEAN UP STACK IF NORMAL MODE
0C86 C3890C     821          JMP    RXI5         ;RETURN
                822
0C89 CD1A0B     823 RXI5:    CALL   RXDMA        ;RESET DMA CHANNEL
0C8C D1         824          POP    D            ;RESTORE REGISTERS
0C8D C1         825          POP    B
0C8E F1         826          POP    PSW
0C8F E1         827          POP    H
0C90 FB         828          EI                  ;ENABLE INTERRUPTS
0C91 C9         829          RET                 ;RETURN
                830 ;
                831 ;
                832 ;MESSAGE TYPER - ASSUMES MESSAGE STARTS AT HL
                833 ;
                834 ;
0C92 C5         835 TYMSG:   PUSH   B            ;SAVE BC
0C93 7E         836 TYMSG2:  MOV    A,M          ;GET ASCII CHR
0C94 23         837          INX    H            ;INC POINTER
0C95 FEFF       838          CPI    0FFH         ;STOP?
0C97 CAA10C     839          JZ     TYMSG1       ;YES, GET SET FOR EXIT
0C9A 4F         840          MOV    C,A          ;SET UP FOR DISPLAY
0C9B CDF805     841          CALL   ECHO         ;DISPLAY CHR
0C9E C3930C     842          JMP    TYMSG2       ;GET NEXT CHR
0CA1 C1         843 TYMSG1:  POP    B            ;RESTORE BC
0CA2 C9         844          RET                 ;RETURN
                845 ;
                846 ;
                847 ;SIGNON MESSAGE
                848 ;
```

611001-58

```
0CA3 0D          849 SIGNON: DB      CR,'8273 MONITOR  V1.1',CR,0FFH
0CA4 38323733
0CA8 204D4F4E
0CAC 49544F52
0CB0 20205631
0CB4 2E31
0CB6 0D
0CB7 FF
                 850 ;
                 851 ;
                 852 ;
                 853 ;RECEIVER INTERRUPT MESSAGES
                 854 ;
                 855 ;
0CB8 0D          856 RXIMSG: DB      CR,'RX INT - ',0FFH
0CB9 52582049
0CBD 4E54202D
0CC1 20
0CC2 FF
                 857 ;
                 858 ;TRANSMITTER INTERRUPT MESSAGES
                 859 ;
0CC3 0D          860 TXIMSG: DB      CR,'TX INT - ',0FFH
0CC4 54582049
0CC8 4E54202D
0CCC 20
0CCD FF
                 861 ;
                 862 ;
                 863 ;TRANSMITTER INTERRUPT ROUTINE
                 864 ;
0CCE E5          865 TXI:    PUSH    H               ;SAVE HL
0CCF F5          866         PUSH    PSW             ;SAVE PSW
0CD0 C5          867         PUSH    B               ;SAVE BC
0CD1 D5          868         PUSH    D               ;SAVE DE
0CD2 3E61        869         MVI     A,DTDMA         ;DISABLE TX DMA
0CD4 D3A8        870         OUT     MODE57          ;8257 MODE PORT
0CD6 1604        871         MVI     D,04H           ;SET COUNTER
0CD8 2A1020      872         LHLD    LDADR           ;GET LOAD POINTER
0CDB E5          873         PUSH    H               ;SAVE IT
0CDC 45          874         MOV     B,L             ;SAVE LSB IN B
0CDD 2A1320      875         LHLD    CNADR           ;GET CONSOLE POINTER
0CE0 04          876 TXI1:   INR     B               ;INC POINTER
0CE1 78          877         MOV     A,B             ;GET SET TO TEST
0CE2 BD          878         CMP     L               ;LOAD=CONSOLE?
0CE3 CAE40A      879         JZ      BUFFUL          ;YES, BUFFER FULL
0CE6 15          880         DCR     D               ;NO, TEST NEXT LOCATION
0CE7 C2E00C      881         JNZ     TXI1            ;TRY AGAIN
0CEA E1          882         POP     H               ;RESTORE LOAD POINTER
0CEB DB92        883         IN      TXIR73          ;READ RESULT
0CED 77          884         MOV     M,A             ;STORE IN BUFFER
0CEE 2C          885         INR     L               ;INR POINTER
0CEF 3600        886         MVI     M,00H           ;EXTRA RESULT SPOTS 0
0CF1 2C          887         INR     L
0CF2 3600        888         MVI     M,00H
0CF4 2C          889         INR     L
0CF5 3600        890         MVI     M,00H
0CF7 2C          891         INR     L
```

611001-59

```
0CF8 3600       892         MVI     M,00H
0CFA 2C         893         INR     L
0CFB 221020     894         SHLD    LDADR           ;UPDATE LOAD POINTER
0CFE CD350B     899         CALL    TXDMA           ;RESET DMA CHANNEL
0D01 D1         900         POP     D               ;RESTORE DE
0D02 C1         901         POP     B               ;RESTORE BC
0D03 F1         902         POP     PSW             ;RESTORE PSW
0D04 E1         903         POP     H               ;RESTORE HL
0D05 FB         904         EI                      ;ENABLE INTERRUPTS
0D06 C9         905         RET                     ;RETURN
                906 ;
                907 ;
                952 ;
                953 ;
                954         END

PUBLIC SYMBOLS


EXTERNAL SYMBOLS


USER SYMBOLS
ADWN   A 0922   AFCMD  A 09CE   BUFFUL A 0AE4   CH0ADR A 00A0   CH0TC  A 00A1   CH1ADR A 00A2   CH1TC  A 00A3
CMD51  A 0027   CMDBF1 A 2020   CMDBUF A 2000   CMDOUT A 0AFB   CMDREC A 0857   CMODE  A 0921   CNADR  A 2013
CNT053 A 009C   CNT153 A 009D   CNT253 A 009E   CNTL51 A 0089   CNTLC  A 0003   CNVBN  A 05BB   COBR   A 000C
COMM   A 0AE5   COMM1  A 0AEA   COMM2  A 0AFF   COMM73 A 0090   CPBF   A 0020   CR     A 0000   CRLF   A 05EB
DEM    A 0000   DEMODE A 2027   DISPY  A 0A39   DISPY1 A 0A4E   DISPY2 A 0A4D   DRDMA  A 0062   DTDMA  A 0061
ECHO   A 05F8   ENDCHK A 0A1B   ENDMA  A 0063   GDWN   A 08FF   GETCH  A 061F   GETCMD A 087D   GRCMD  A 09C4
ILLEG  A 08A7   LDADR  A 2010   LF     A 000A   LKBR1  A 2017   LKBR2  A 2018   LOOPIT A 0861   MDCNT0 A 0036
MDCNT2 A 00B6   MDE51  A 00CE   MODE53 A 009B   MODE57 A 00A8   MONTOR A 0008   NMOUT  A 06C7   NSAF   A 0073
PAR1   A 0B08   PAR2   A 0B0D   PARIN  A 0AAD   PARIN1 A 0AE0   PARIN2 A 0ADC   PARIN3 A 0ABC   PARM73 A 0091
POLIN  A 2016   PRMPT  A 2015   R0PT   A 0AA2   R1PT   A 0AA7   RBCMD  A 097B   RDCMD  A 0971   RDWN   A 08AF
RDY    A 0002   RESBUF A 2800   RESL73 A 0091   ROCMD  A 095D   RPCMD  A 09D8   RR0F   A 0011   RR0P   A 0011
RSCMD  A 0967   RST65  A 20CE   RST75  A 20D4   RXBUF  A 8200   RXD51  A 0088   RXDMA  A 0B1A   RXI    A 0C00
RXI1   A 0C16   RXI2   A 0C23   RXI3   A 0C39   RXI4   A 0C45   RXI5   A 0C89   RXI6   A 0C85   RXIMSG A 0CB8
RXINT  A 0008   RXIR73 A 0093   RXIRA  A 0002   RXS1   A 0A69   RXS2   A 0A7F   RXS3   A 0A8D   RXSORC A 0A62
RXTC   A 41FF   SBCMD  A 0985   SDWN   A 08D7   SIGNON A 0CA2   SLCMD  A 098F   SNRMP  A 0093   SOCMD  A 09A6
SPCMD  A 09E2   SRCMD  A 09BA   SSCMD  A 09B0   START  A 0800   STAT51 A 0089   STAT57 A 00A8   STAT73 A 0090
STKSRT A 20C0   SW     A 0943   T1     A 0C6C   TBUFFL A 0A24   TBUFL  A 0A07   TBUFL1 A 0A8D   TDWN   A 090E
TEST73 A 0092   TFCMD  A 09EC   TFCMD1 A 09F6   TFRET  A 0A36   TLCMD  A 0999   TRUE   A 0000   TRUE1  A 0000
TXBUF  A 8000   TXD51  A 0088   TXDMA  A 0B35   TXDMA1 A 0B42   TXI    A 0CCE   TXI1   A 0CE0   TXIMSG A 0CC3
TXINT  A 0004   TXIR73 A 0092   TXIRA  A 0001   TXPOL  A 094C   TXRET  A 0C6E   TXSORC A 0A47   TXTC   A 81FF
TYMSG  A 0C92   TYMSG1 A 0CA1   TYMSG2 A 0C93   WALDG  A 075E

ASSEMBLY COMPLETE.  NO ERRORS
```

611001-60

intel®

APPLICATION NOTE

AP-134

October 1986

Asynchronous Communication with the 8274 Multiple-Protocol Serial Controller

Order Number: 210311-002

INTRODUCTION

The 8274 Multiprotocol serial controller (MPSC) is a sophisticated dual-channel communications controller that interfaces microprocessor systems to high-speed serial data links (at speeds to 880K bits per second) using synchronous or asynchronous protocols. The 8274 interfaces easily to most common microprocessors (e.g., 8048, 8051, 8085, 8086, and 8088), to DMA controllers such as the 8237 and 8257, and to the 8089 I/O processor. Both MPSC communication channels are completely independent and can operate in a full-duplex communication mode (simultaneous data transmission and reception).

Communication Functions

The 8274 performs many communications-oriented functions, including:

— Converting data bytes from a microprocessor system into a serial bit stream for transmission over the data link to a receiving system.

— Receiving serial bit streams and reconverting the data into parallel data bytes that can easily be processed by the microprocessor system.

— Performing error checking during data transfers. Error checking functions include computing/transmitting error codes (such as parity bits or CRC bytes) and using these codes to check the validity of received data.

— Operating independently of the system processor in a manner designed to reduce the system overhead involved in data transfers.

System Interface

The MPSC system interface is extremely flexible, supporting the following data transfer modes:

1. Polled Mode. The system processor periodically reads (polls) an 8274 status register to determine when a character has been received, when a character is needed for transmission, and when transmission errors are detected.
2. Interrupt Mode. The MPSC interrupts the system processor when a character has been received, when a character is needed for transmission, and when transmission errors are detected.
3. DMA Mode. The MPSC automatically requests data transfers from system memory for both transmit and receive functions by means of two DMA request signals per serial channel. These DMA request signals may be directly interfaced to an 8237 or 8257 DMA controller or to an 8089 I/O processor.
4. WAIT Mode. The MPSC ready signal is used to synchronize processor data transfers by forcing the processor to enter wait states until the 8274 is ready for another data byte. This feature enables the 8274 to interface directly to an 8086 or 8088 processor by means of string I/O instructions for very high-speed data links.

Scope

This application note describes the use of the 8274 in asynchronous communication modes. Asynchronous communication is typically used to transfer data to/from video display terminals, modems, printers, and other low-to-medium-speed peripheral devices. Use of the 8274 in both interrupt-driven and polled system environments is described. Use of the DMA and WAIT modes are not described since these modes are employed mainly in synchronous communication systems where extremely high data rates are common. Programming examples are written in PL/M-86 (Appendix B and Appendix C). PL/M-86 is executed by the iAPX-86 and iAPX-88 processor families. In addition, PL/M-86 is very similar to PL/M-80 (executed by the MCS-80 and MCS-85 processor families). In addition, Appendix D describes a simple application example using an SDK-86 in an iAPX-86/88 environment.

SERIAL-ASYNCHRONOUS DATA LINKS

A serial asynchronous interface is a method of data transmission in which the receiving and transmitting systems need not be synchronized. Instead of transmitting clocking information with the data, locally generated clocks (16, 32 or 64 times as fast as the data transmission rate) are used by the transmitting and receiving systems. When a character of information is sent by the transmitting system, the character data is framed (preceded and followed) by special START and STOP bits. This framing information permits the receiving system to temporarily synchronize with the data transmission. (Refer to Figure 1 during the following discussion of asynchronous data transmission.)

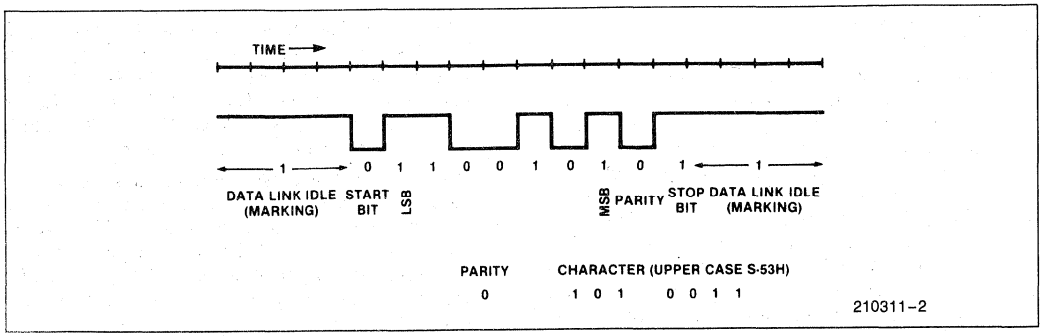


Figure 1. Transmission of a 7-Bit ASCII Character with Even Parity

Normally the data link is in an idle or marking state, continuously transmitting a "mark" (binary 1). When a character is to be sent, the character data bits are immediately preceded by a "space" (binary 0 START bit). The mark-to-space transition informs the receiving system that a character of information will immediately follow the start bit. Figure 1 illustrates the transmission of a 7-bit ASCII character (upper case S) with even parity. Note that the character is transmitted immediately following the start bit. Data bits within the character are transmitted from least-significant to most-significant. The parity bit is transmitted immediately following the character data bits and the STOP framing bit (binary 1) signifies the end of the character.

Asynchronous interfaces are often used with human interface devices such as CRT/keyboard units where the time between data transmissions is extremely variable.

Characters

In asynchronous mode, characters may vary in length from five to eight bits. The character length depends on the coding method used. For example, five-bit characters are used when transmitting Baudot Code, seven-bit characters are required for ASCII data, and eight-bit characters are needed for EBCDIC and binary data. To transmit messages composed of multiple characters, each character is framed and transmitted separately (Figure 2).

This framing method ensures that the receiving system can easily synchronize with the start and stop bits of each character, preventing receiver synchronization errors. In addition, this synchronization method makes both transmitting and receiving systems insensitive to possible time delays between character transmissions.

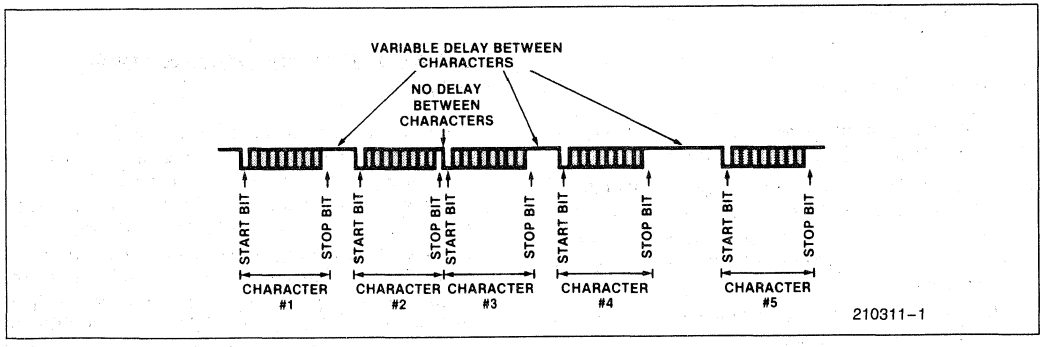


Figure 2. Multiple Character Transmission

Framing

Character framing is accomplished by the START and STOP bits described previously. When the START bit transition (mark-to-space) is detected, the receiving system assumes that a character of data will follow. In order to test this assumption (and isolate noise pulses on the data link), the receiving system waits one-half bit time and samples the data link again. If the link has returned to the marking state, noise is assumed, and the receiver waits for another START bit transition.

When a valid START bit is detected, the receiver samples the data link for each bit of the following character. Character data bits and the parity bit (if required) are sampled at their nominal centers until all required characters are received. Immediately following the data bits, the receiver samples the data link for the STOP bit, indicating the end of the character. Most systems permit specification of 1, 1½, or 2 stop bits.

Timing

The transmitter and receiver in an asynchronous data link arrangement are clocked independently. Normally, each clock is generated locally and the clocks are not synchronized. In fact, each clock may be a slightly different frequency. (In practice, the frequency difference should not exceed a few percent. If the transmitter and receiver clock rates vary substantially, errors will occur because data bits may be incorrectly identified as START or STOP framing bits.) These clocks are designed to operate at 16, 32, or 64 times the communications data rate. These clock speeds allow the receiving device to correctly sample the incoming bit stream.

Serial-interface data rates are measured in bits/second. The term "baud" is used to specify the number of times per second that the transmitted signal level can change states. In general, the baud is not equal to the bit rate. Only when the transmitted signal has two states (electrical levels) is the baud rate equal to the bit rate. Most point-to-point serial data links use RS-232-C, RS-422, or RS-423 electrical interfaces. These specifications call for two electrical signal levels (the baud is equal to the bit rate). Modem interfaces, however, may often have differing bit and baud rates.

While there are generally no limitations on the data transmission rates used in an aysnchronous data link, a limited set of rates has been standardized to promote equipment interconnection. These rates vary from 75 bits per second to 38,400 bits per second. Table 1 illustrates typical asynchronous data rates and the associated clock frequencies required for the transmitter and receiver circuits.

Table 1. Communication Data Rates and Associated Transmitter/Receiver Clock Rates

Data Rate (Bits/Second)	Clock Rate (kHz)		
	X16	X32	X64
75	1.2	2.4	4.8
150	2.4	4.8	9.6
300	4.8	9.6	19.2
600	9.6	19.2	38.4
1200	19.2	38.4	76.8
2400	38.4	76.8	153.6
4800	76.8	153.6	307.2
9600	153.6	307.2	614.2
19200	307.2	614.4	—
38400	614.4	—	—

Parity

In order to detect transmission errors, a parity bit may be added to the character data as it is transferred over the data link. The parity bit is set or cleared to make the total number of "one" bits in the character even (even parity) or odd (odd parity). For example, the letter "A" is represented by the seven-bit ASCII code 1000001 (41H). The transmitted data code (with parity) for this character contains eight bits; 01000001 (41H) for even parity and 11000001 (OC1H) for odd parity. Note that a single bit error changes the parity of the received character and is therefore easily detected. The 8274 supports both odd and even parity checking as well as a parity disable mode to support binary data transfers.

Communication Modes

Serial data transmission between two devices can occur in one of three modes. In the simplex transmission mode, a data link can transmit data in one direction only. In the half-duplex mode, the data link can transmit data in both directions, but not simultaneously. In the full-duplex mode (the most common), the data link can transmit data in both directions simultaneously. The 8274 directly supports the full-duplex mode and will interface to simplex and half-duplex communication data links with appropriate software controls.

BREAK Condition

Asynchronous data links often include a special sequence known as a break condition. A break condition is initiated when the transmitting device forces the data link to a spacing state (binary 0) for an extended length of time (typically 150 milliseconds). Many terminals contain keys to initiate a break sequence. Under software control, the 8274 can initiate a break sequence when transmitting data and detect a break sequence when receiving data.

MPSC SYSTEM INTERFACE

Hardware Environment

The 8274 MPSC interfaces to the system processor over an 8-bit data bus. Each serial I/O channel responds to two I/O or memory addresses as shown in Table 2. In addition, the MPSC supports non-vectored and vectored interrupts.

The 8274 may be configured for memory-mapped or I/O-mapped operation.

The 8274-processor hardware interface can be configured in a flexible manner, depending on the operating mode selected—polled, interrupt-driven, DMA, or WAIT. Figure 3 illustrates typical MPSC configurations for use with an 8088 microprocessor in the polled and interrupt-driven modes.

All serial-to-parallel conversion, parallel-to-serial conversion, and parity checking required during asynchronous serial I/O operation is automatically performed by the MPSC.

Operational Interface

Command, parameter, and status information is stored in 21 registers within the MPSC (8 writable registers and 2 readable registers for each channel, plus the interrupt vector register). These registers are all accessed by means of the command/status ports for each channel. An internal pointer register selects which of the command or status registers will be written or read during a command/status access of an MPSC channel. Figure 4 diagrams the command/status register architecture for each serial channel. In the following discussion, the writable registers will be referred to as WR0 through WR7 and the readable registers will be referred to as RR0 through RR2.

Table 2. 8274 Addressing

$\overline{CS}$	A_1	A_0	Read Operation	Write Operation
0	0	0	Ch. A Data Read	Ch. A Data Write
0	1	0	Ch. A Status Read	Ch. A Command/Parameter
0	0	1	Ch. B Data Read	Ch. B Data Write
0	1	1	Ch. B Status Read	Ch. B Command/Parameter
1	X	X	High Impedance	High Impedance

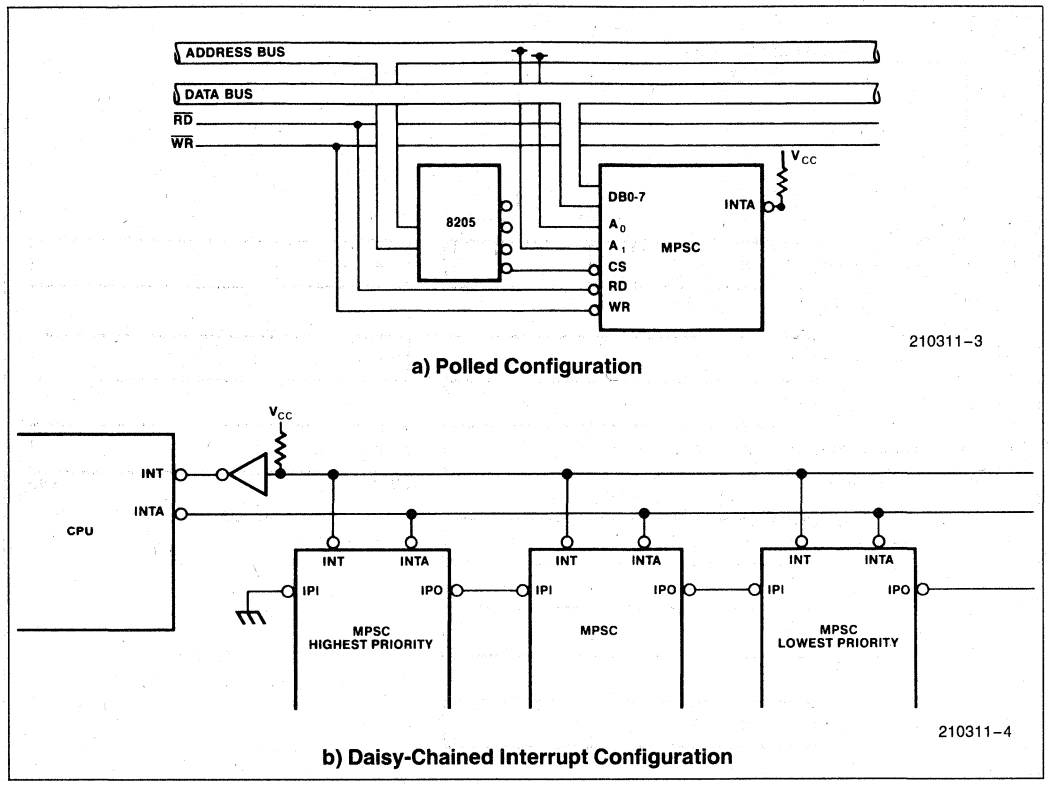


Figure 3. 8274 Hardware Interface for Polled and Interrupt-Driven Environments

The least-significant three bits of WR0 are automatically loaded into the pointer register every time WR0 is written. After reset, WR0 is set to zero so that the first write to a command register causes the data to be loaded into WR0 (thereby setting the pointer register). After WR0 is written, the following read or write accesses the register selected by the pointer. The pointer is reset after the read or write operation is completed. In this manner, reading or writing an arbitrary MPSC channel register requires two I/O accesses. The first access is always a write command. This write command is used to set the pointer register. The second access is either a read or a write command; the pointer register (previously set) will ensure that the correct internal register is read or written. After this second access, the pointer register is automatically reset. Note that writing WR0 and reading RR0 does not require presetting of the pointer register.

During initialization and normal MPSC operation, various registers are read and/or written by the system processor. These actions are discussed in detail in the following paragraphs. Note that WR6 and WR7 are not used in the asynchronous communication modes.

RESET

When the 8274 $\overline{\text{RESET}}$ line is activated, both MPSC channels enter the idle state. The serial output lines are forced to the marking state (high) and the modem interface signals ($\overline{\text{RTS}}$, $\overline{\text{DTR}}$) are forced high. In addition, the pointer register is set to zero.

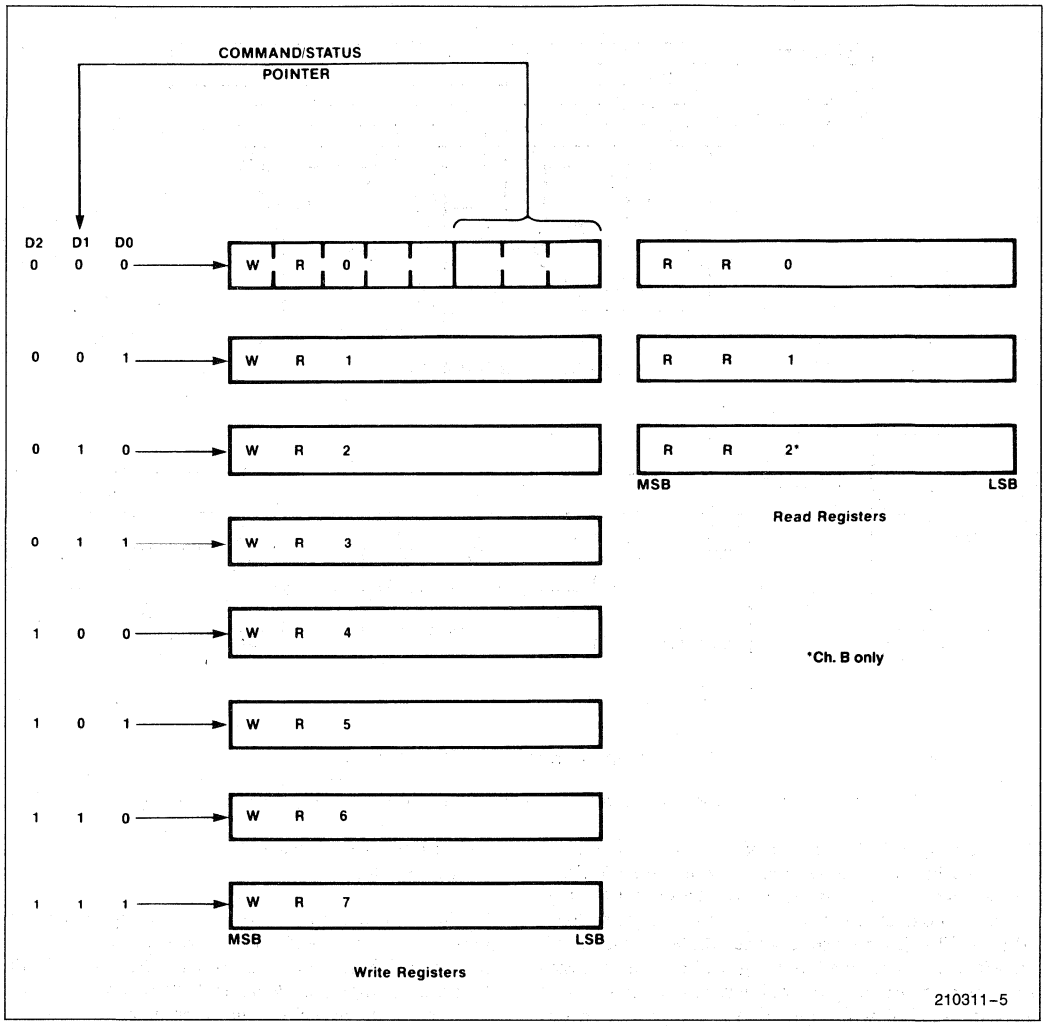


Figure 4. Command/Status Register Architecture (Each Serial Channel)

External/Status Latches

The MPSC continuously monitors the state of four external/status conditions:

1. CTS—clear-to-send input pin.
2. CD—carrier-detect input pin.
3. SYNDET—sync-detect input pin. This pin may be used as a general-purpose input in the asynchronous communication mode.
4. BREAK—a break condition (series of space bits on the receiver input pin).

A change of state in any of these monitored conditions will cause the associated status bit in RR0 (Appendix A) to be latched (and optionally cause an interrupt).

Error Reporting

Three error conditions may be encountered during data reception in the asynchronous mode:

1. Parity. If parity bits are computed and transmitted with each character and the MPSC is set to check parity (bit 0 in WR4 is set), a parity error will occur whenever the number of "1" bits within the character (including the parity bit) does not match the odd/even setting of the parity check flag (bit 1 in WR4).

2. Framing. A framing error will occur if a stop bit is not detected immediately following the parity bit (if parity checking is enabled) or immediately following the most-significant data bit (if parity checking is not enabled).
3. Overrun. If an input character has been assembled but the receiver buffers are full (because the previously received characters have not been read by the system processor), an overrun error will occur. When an overrun error occurs, the input character that has just been received will overwrite the immediately preceding character.

Transmitter/Receiver Initialization

In order to operate in the asynchronous mode, each MPSC channel must be initialized with the following information:

1. Clock Rate. This parameter is specified by bits 6 and 7 of WR4. The clock rate may be set to 16, 32, or 64 times the data-link bit rate. (See Appendix A for WR4 details.)
2. Number of Stop Bits. This parameter is specified by bits 2 and 3 of WR4. The number of stop bits may be set to 1, 1½, or 2. (See Appendix A for WR4 details.)
3. Parity Selection. Parity may be set for odd, even, or no parity by bits 0 and 1 of WR4. (See Appendix A for WR4 details.)
4. Receiver Character Length. This parameter sets the length of received characters to 5, 6, 7, or 8 bits. This parameter is specified by bits 6 and 7 of WR3. (See Appendix A for WR3 details.)
5. Receiver Enable. The serial-channel receiver operation may be enabled or disabled by setting or clearing bit 0 of WR3. (See Appendix A for WR3 details.)
6. Transmitter Character Length. This parameter sets the length of transmitted characters to 5, 6, 7, or 8 bits. This parameter is specified by bits 5 and 6 of WR5. (See Appendix A for WR5 details.) Characters of less than 5 bits in length may be transmitted by setting the transmitted length to five bits (set bits 5 and 6 of WR5 to 1).

   The MPSC then determines the actual number of bits to be transmitted from the character data byte. The bits to be transmitted must be right justified in the data byte, the next three bits must be set to 0 and all remaining bits must be set to 1. The following table illustrates the data formats for transmission of 1 to 5 bits of data:

D7	D6	D5	D4	D3	D2	D1	D0	Number of Bits Transmitted (Character Length)
1	1	1	1	0	0	0	c	1
1	1	1	0	0	0	c	c	2
1	1	0	0	0	c	c	c	3
1	0	0	0	c	c	c	c	4
0	0	0	c	c	c	c	c	5

7. Transmitter Enable. The serial channel transmitter operation may be enabled or disabled by setting or clearing bit 3 of WR5. (See Appendix A for WR5 details.)

For data transmissions via a modem or RS-232-C interface, the following information must also be specified:

1. Request-to-Send/Data-Terminal-Ready. Must be set to indicate status of data terminal equipment. Request-to-send is controlled by bit 1 of WR5 and data terminal ready is controlled by bit 7. (See Appendix A for WR5 details.)
2. Auto Enable. May be set to allow the MPSC to automatically enable the channel transmitter when the clear-to-send signal is active and to automatically enable the receiver when the carrier-detect signal is active. Auto Enable is controleld by bit 5 of WR3. (See Appendix A for WR3 details.)

During initialization, it is desirable to guarantee that the external/status latches reflect the latest interface information. Since up to two state changes are internally stored by the MPSC, at least two Reset External/Status Interrupt commands must be issued. This procedure is most easily accomplished by simply issuing this reset command whenever the pointer register is set during initialization.

An MPSC initialization procedure (MPSCRXINIT) for asynchronous communication is listed in Appendix B. Figure 5 illustrates typical MPSC initialization parameters for use with this procedure.

```
call MPSC$RX$INIT(41,  1,1,0,1,  3,1,1,  3,1,1,0,1);

        initializes the 8274 at address 41 as follows:

X16 clock rate              Enable transmitter
1 stop bit                    and receiver
Odd parity                  Auto enable set
8-bit characters            DTR and RTS set
  (Tx and Rx)               Break transmission disabled
```

Figure 5. Sample 8274 Initialization Procedure for Polled Operation

Polled Operation

In the polled mode, the processor must monitor the MPSC status by testing the appropriate bits in the read register. Data available, status, and error conditions are represented in RR0 and RR1 for channels A and B. An example of MPSC-polled transmitter/receiver routines are given in Appendix B. The following routines are detailed:

1. MPSC$POLL$RCV$CHARACTER—This procedure receives a character from the serial data link. The routine waits until the character-available flag in RR0 has been set. When this flag indicates that a character is available, RR1 is checked for errors (overrun, parity, or framing). If an error is detected, the character in the MPSC receive buffer must be read and discarded and the error routine (RECEIVE$ERROR) is called. If no receive errors have been detected, the character is input from the 8274 data port and returned to the calling program.

   MPSC$POLL$RCV$CHARACTER requires three parameters—the address of the 8274 channel data port (data$port), the address of the 8274 channel command port (cmd$port), and the address of a byte variable in which to store the received character (character$ptr).

2. MPSC$POLL$TRAN$CHARACTER—This procedure transmits a character to the serial data link. The routine waits until the transmitter-buffer-empty flag has been set in RR0 before writing the character to the 8274.

   MPSC$POLL$TRAN$CHARACTER requires three parameters—the address of the 8274 channel data port (data$port), the address of the 8274 channel command port (cmd$port), and the character of data that is to be transmitted (character).

3. RECEIVE$ERROR—This procedure processes receiver errors. First, an Error Reset command is written to the affected channel. All additional error processing is dependent on the specific application. For example, the receiving device may immediately request retransmission of the character or wait until a message has been completed.

   RECEIVE$ERROR requires two parameters—the address of the affected 8274 command port (cmd$port) and the error status (status) from 8274 register RR1.

Interrupt-Driven Operation

In an interrupt-driven environment, all receiver operations are reported to the system processor by means of interrupts. Once a character has been received and assembled, the MPSC interrupts the system processor. The system processor must then read the character from the MPSC data buffer and clear the current interrupt. During transmission, the system processor starts serial I/O by writing the first character of a message to the MPSC. The MPSC interrupts the system processor whenever the next character is required (i.e., when the transmitter buffer is empty) and the processor responds by writing the next character of the message to the MPSC data port for the appropriate channel.

By using interrupt-driven I/O, the MPSC proceeds independently of the system processor, signalling the processor only when characters are required for transmission, when characters are received from the data link, or when errors occur. In this manner, the system processor may continue execution of other tasks while serial I/O is performed concurrently.

Interrupt Configurations

The 8274 is designed to interface to 8085- and 8086-type processors in much the same manner as the 8259A is designed. When operating in the 8085 mode, the 8274 causes a "call" to a prespecified, interrupt-service routine location. In the 8086 mode, the 8274 presents the processor with a one-byte interrupt-type number. This interrupt-type number is used to "vector" through the 8086 interrupt service table. In either case, the interrupt service address or interrupt-type number is specified during MPSC initialization.

To shorten interrupt latency, the 8274 can be programmed to modify the prespecified interrupt vector so that no software overhead is required to determine the cause of an interrupt. When this "status affects vector" mode is enabled, the following eight interrupts are differentiated automatically by the 8274 hardware:

1. Channel B Transmitter Buffer Empty.
2. Channel B External/Status Transition.
3. Channel B Character Available.
4. Channel B Receive Error.
5. Channel A Transmitter Buffer Empty.
6. Channel A External/Status Transition.
7. Channel A Character Available.
8. Channel A Receive Error.

Interrupt Sources/Priorities

The 8274 has three interrupt sources for each channel:

1. Receiver (RxA, RxB). An interrupt is initiated when a character is available in the receiver buffer or when a receiver error (parity, framing, or overrun) is detected.
2. Transmitter (TxA, TxB). An interrupt is initiated when the transmitter buffer is empty and the 8274 is ready to accept another character for transmission.

3. External/Status (ExTA, ExTB). An interrupt is initiated when one of the external/status conditions (CDE, CTS, SYNDET, BREAK) changes state.

The 8274 supports two interrupt priority orderings (selectable during MPSC initialization) as detailed in Appendix A, WR2, CH-A.

Interrupt Initialization

In addition to the initialization parameters required for polled operation, the following parameters must be supplied to the 8274 to specify interrupt operation:

1. Transmit Interrupt Enable. Transmitter-buffer-empty interrupts are separately enabled by bit 1 of WR1. (See Appendix A for WR1 details.)
2. Receive Interrupt Enable. Receiver interrupts are separately enabled in one of three modes: a) interrupt on first received character only and on receive errors (used for message-oriented transmission systems), b) interrupt on all received characters and on receive errors, but do not interrupt on parity errors, and c) interrupt on all received characters and on receive errors (including parity errors). The ability to separately disable parity interrupts can be extremely useful when transmitting messages. Since the parity error bit in RR1 is latched, it will not be reset until an error reset operation is performed. Therefore, the parity error bit will be set if any parity errors were detected in a multi-character message. If this mode is used, the serial I/O software must poll the parity error bit at the completion of a message and issue an error reset if appropriate. The receiver interrupt mode is controlled by bits 3 and 4 of WR1. (See Appendix A for WR1 details.)
3. External/Status Interrupts. External/Status interrupts can be separately enabled by bit 0 of WR1. (See Appendix A for WR1 details.)
4. Interrupt Vector. An eight-bit interrupt-service routine location (8085) or interrupt type (8086) is specified through WR2 of channel B. (See Appendix A for WR2 details.) Table 3 lists interrupt vector addresses generated by the 8274 in the "status affects vector" mode.
5. "Status Affects Vector" Mode. The 8274 will automatically modify the interrupt vector if bit 3 of WR1 is set. (See Appendix A for WR1 details.)
6. System Configuration. Specifies the 8274 data transfer mode. Three configuration modes are available: a) interrupt-driven operation for both channels, b) DMA operation for both channels, and c) DMA operation for channel A, interrupt-driven operation for channel B. The system configuration is specified by means of bits 0 and 1 of WR2 (channel A). (See Appendix A for WR2 details.)
7. Interrupt Priorities. The 8274 permits software specification of receive/transmit priorities by means of bit 2 of WR2 (channel A). (See Appendix A for WR2 details.)
8. Interrupt Mode. Specifies whether the MPSC is to operate in a non-vectored mode (for use with an external interrupt controller), in an 8086-vectored mode, or in an 8085-vectored mode. This parameter is specified through bits 3 and 4 of WR2 (channel A). (See Appendix A for WR2 details.)

An MPSC interrupt initialization procedure (MPSCINTINIT) is listed in Appendix C.

Table 3. MPSC-Generated Interrupt Vectors in "Status Affects Vector" Mode

V7	V6	V5	V4	V3	V2	V1	V0	V7	V6	V5	V4	V3	V2	V1	V0	Original Vector (Specified during Initialization)
8086 Interrupt Type								8085 Interrupt Location								Interrupt Condition
V7	V6	V5	V4	V3	0	0	0	V7	V6	V5	0	0	0	V1	V0	Channel B Transmitter Buffer Empty
V7	V6	V5	V4	V3	0	0	1	V7	V6	V5	0	0	1	V1	V0	Channel B External/Status Change
V7	V6	V5	V4	V3	0	1	0	V7	V6	V5	0	1	0	V1	V0	Channel B Receiver Character Available
V7	V6	V5	V4	V3	0	1	1	V7	V6	V5	0	1	1	V1	V0	Channel B Receive Error
V7	V6	V5	V4	V3	1	0	0	V7	V6	V5	1	0	0	V1	V0	Channel A Transmitter Buffer Empty
V7	V6	V5	V4	V3	1	0	1	V7	V6	V5	1	0	1	V1	V0	Channel A External/Status Change
V7	V6	V5	V4	V3	1	1	0	V7	V6	V5	1	1	0	V1	V0	Channel A Receiver Character Available
V7	V6	V5	V4	V3	1	1	1	V7	V6	V5	1	1	1	V1	V0	Channel A Receive Error

Interrupt Service Routines

Appendix C lists four interrupt service procedures, a buffer transmission procedure, and a buffer reception procedure that illustrate the use of the 8274 in interrupt-driven environments. Use of these procedures assumes that the 8086/8088 interrupt vector is set to 20H and that channel B is used with the "status affects vector" mode enabled.

1. TRANSMIT$BUFFER—This procedure begins serial transmission of a data buffer. Two parameters are required—a pointer to the buffer (buf$ptr) and the length of the buffer (buf$length). The procedure first sets the global buffer pointer, buffer length, and initial index for the transmitter-interrupt service routine and initiates transmission by writing the first character of the buffer to the 8274. The procedure then enters a wait loop until the I/O completion status is set by the transmit-interrupt service routine (MPSC$TRANSMIT$CHARACTER$INT).
2. RECEIVE$BUFFER—This procedure inputs a line (terminated by a line feed) from a serial I/O port. Two parameters are required—a pointer to the input buffer (buf$ptr) and a pointer to the buffer length variable (buf$length$ptr). The buffer length will be set by this procedure when the complete line has been input. The procedure first sets the global buffer pointer and initial index for the receiver interrupt service routine. RECEIVE$BUFFER then enters a wait loop until the I/O completion status is set by the receive interrupt routine (MPSC$RECEIVE-$CHARACTER$INT).
3. MPSC$TRANSMIT$CHARACTER$INT—This procedure is executed when the MPSC Tx-buffer-empty interrupt is acknowledged. If the current transmit buffer index is less than the buffer length, the next character in the buffer is written to the MPSC data port and the buffer pointer is updated. Otherwise, the transmission complete status is posted.
4. MPSC$RECEIVE$CHARACTER$INT—This procedure is executed when a character has been assembled by the MPSC and the MPSC has issued a character-available interrupt. If no input buffer has been set up by RECEIVE$BUFFER, the character is ignored. If a buffer has been set up, but it is full, a receive overrun error is posted. Otherwise, the received character is read from the MPSC data port and the buffer index is updated. Finally, if the received character is a line feed, the reception complete status is posted.
5. RECEIVE$ERROR$INT—This procedure is executed when a receive error is detected. First, the error conditions are read from RR1 and the character currently in the MPSC receive buffer is read and discarded. Next, an Error Reset command is written to the affected channel. All additional error processing is application dependent.
6. EXTERNAL$STATUS$CHANGE$INT—This procedure is executed when an external status condition change is detected. The status conditions are read from RR0 and a Reset External/Status Interrupt command is issued. Further error processing is application dependent.

DATA LINK INTERFACE

Serial Data Interface

Each serial I/O channel within the 8274 MPSC interfaces to two data link lines—one line for transmitting data and one for receiving data. During transmission, characters are converted from parallel data format (as supplied by the system processor or DMA device) into a serial bit stream (with START and STOP bits) and clocked out on the TxD pin. During reception, a serial bit stream is input on the RxD pin, framing bits are stripped out of the data stream, and the resulting character is converted to parallel data format and passed to the system processor or DMA device.

Data Clocking

As discussed previously, the frequency of data transmission/reception on the data link is controlled by the MPSC clock in conjunction with the programmed clock divider (in register WR4). The 8274 is designed to permit all four serial interface lines (TxD and RxD for each channel) to operate at different data rates. Four clock input pins (TxC and RxC for each channel) are available for this function. Note that the clock rate divider spcified in WR4 is used for both RxC and TxC on the appropriate channel; clock rate dividers for each channel are independent.

Modem Control

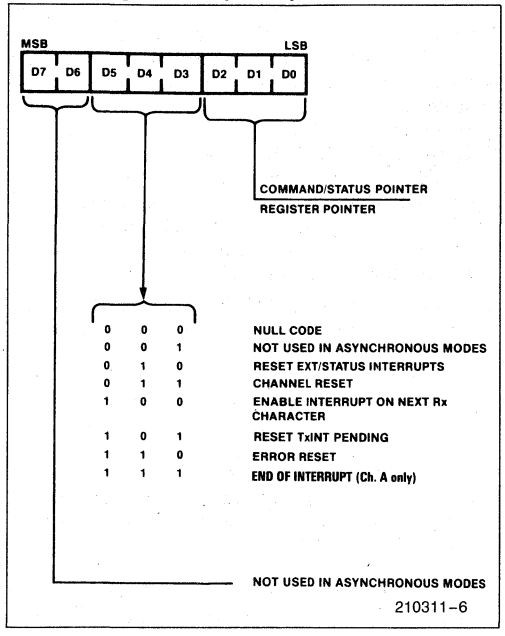
The following four modem interface signals may be connected to the 8274:

1. Data Terminal Ready (DTR). This interface signal (output by the 8274) is software controlled through bit 7 of WR5. When active, DTR indicates that the data terminal/computer equipment is active and ready to interact with the data communications channel. In addition, this signal prepares the modem for connection to the communication channel and maintains connections previously established (e.g., manual call origination).
2. Request To Send (RTS). This interface signal (output by the 8274) is software controlled through bit 1 of WR5. When active, RTS indicates that the data terminal/computer equipment is ready to transmit data. When the RTS bit is reset in asynchronous mode, the signal does not go high until the transmitter is empty.
3. Clear To Send (CTS). This interface signal (input to the 8274) is supplied by the modem in response to an active RTS signal. CTS indicates that the data terminal/computer equipment is permitted to transmit data. The state of CTS is available to the programmer as bit 5 of RR0. In addition, if the auto enable control is set (bit 5 of WR3), the 8274 will not transmit data bytes until CTS has been activated. If CTS becomes inactive during transmission of a character, the current character transmission is completed before the transmitter is disabled.
4. Carrier Detect (CD). This interface signal (input to the 8274) is supplied by the modem to indicate that a data carrier signal has been detected and that a valid data signal is present on the RxD line. The state of CD is available to the programmer as bit 3 of RR0. In addition, if the auto enable control is set (bit 5 of WR3), the 8274 will not enable the serial receiver until CD has been activated. If the CD signal becomes inactive during reception of a character, the receiver is disabled, and the partially received character is lost.

In addition to the above modem interface signals, the 8274 SYNDET input pin for channel A may be used as a general-purpose input in the asynchronous communication mode. The status of this signal is available to the programmer as bit 4 of status register RR0.

APPENDIX A
COMMAND/STATUS DETAILS FOR ASYNCHRONOUS COMMUNICATION

Write Register 0 (WR0):



D2,D1,D0 Command/Status Register Pointer bits determine which write-register the next byte is to be written into, or which read-register the next byte is to be read from. After reset, the first byte written into either channel goes into WR0. Following a read or write to any register (except WR0) the pointer will point to WR0.

D5,D4,D3 Command bits determine which of the basic seven commands are to be performed.

Command 0 Null—has no effect.

Command 1 Note used in asynchronous modes.

Command 2 Reset External/Status Interrupts—resets the latched status bits of RR0 and reenables them, allowing interrupts to occur again.

Command 3 Channel Reset—resets the Latched Status bits of RR0, the interrupt prioritization logic and all control registers for the channel. Four extra system clock cycles should be allowed for MPSC reset time before any additional commands or controls are written into the channel.

Command 4 Enable Interrupt on Next Receive Character—if the Interrupt-on-First-Receive Character mode is selected, this command reactivates that mode after each complete message is received to prepare the MPSC for the next message.

Command 5 Reset Transmitter Interrupt Pending—if the Transmit Interrupt mode is selected, the MPSC automatically interrupts data when the transmit buffer becomes empty. When there are no more characters to be sent, issuing this command prevents further transmitter interrupts until the next character has been completely sent.

Command 6 Error Reset—error latches, Parity and Overrun errors in RR1 are reset.

Command 7 End of Interrupt—resets the interrupt-in-service latch of the highest-priority internal device under service.

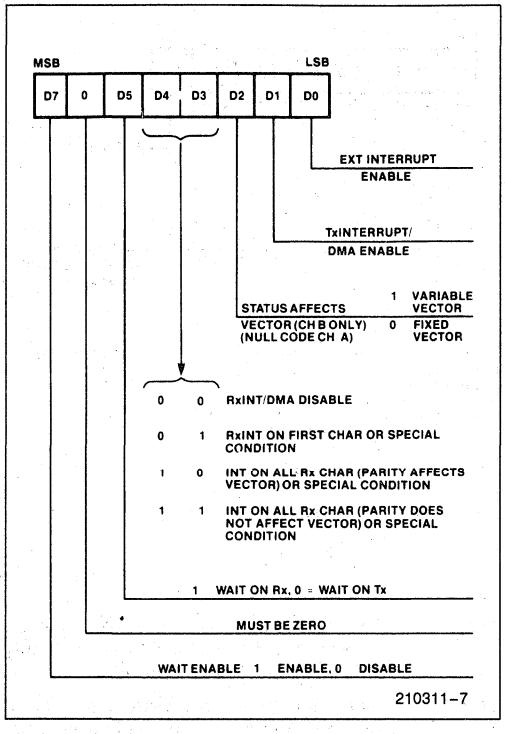
Write Register 1 (WR1):

D0 External/Status Interrupt Enable—allows interrupt to occur as the result of transitions on the $\overline{CD}$, $\overline{CTS}$ or $\overline{SYNDET}$ inputs. Also allows interrupts as the result of a Break/Abort detection and termination, or at the beginning of CRC, or sync character transmission when the Transmit Underrun/EOM latch becomes set.

D1 Transmitter Interrupt/DMA Enable—allows the MPSC to interrupt or request a DMA transfer when the transmitter buffer becomes empty.

D2 Status Affects Vector—(WR1, D2 active in channel B only.) If this bit is not set, then the fixed vector, programmed in WR2, is returned from an interrupt acknowledge sequence. If the bit is set, then the vector returned from an interrupt acknowledge is variable as shown in the Interrupt Vector Table.

Write Register 1 (WR1):



D4,D3 Receive Interrupt Mode.

0 0 Receive Interrupts/DMA Disabled.

0 1 Receive Interrupt on First Character Only or Special Condition.

1 0 Interrupt on All Receive Characters of Special Condition (Parity Error is a Special Receive Condition).

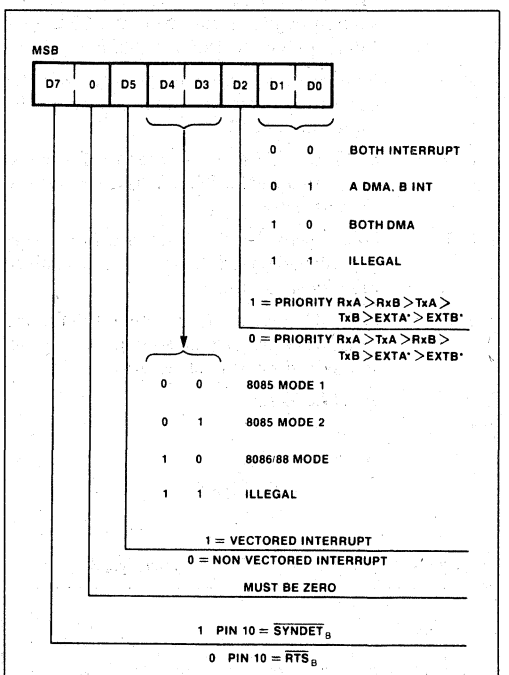
1 1 Interrupt on All Receive Characters or Special Condition (Parity Error is not a Special Receive Condition).

D5 Wait on Receive/$\overline{\text{Transmit}}$—when the following conditions are met, the RDY pin is activated, otherwise it is held in the High-Z state. (Conditions: Interrupt Enabled Mode, Wait Enabled, $\overline{CS}$ = 0, A0 = 0/1, and A1 = 0). The RDY pin is pulled low when the transmitter buffer is full or the receiver buffer is empty and it is driven High when the transmitter buffer is empty or the receiver buffer is full. The RDY_A and RDY_B may be wired *or* connected since only one signal is active at any one time while the other is in the High Z state.

D6 Must be Zero.

D7 Wait Enable—enables the wait function.

Write Register 2 (WR2): Channel A



NOTE: 210311-8

*External Status Interrupt—only if EXT Interrupt Enable (WR1; D0) is set.

D1,D0 System Configuration—These specify the data transfer from MPSC channels to the CPU, either interrupt or DMA based.

0 0 Channel A and Channel B both use interrupts.

0 1 Channel A uses DMA, Channel B uses interrupt.

1 0 Channel A and Channel B both use DMA.

1 1 Illegal Code.

D2 Priority—this bit specifies the relative priorities of the internal MPSC interrupt/DMA sources.

0 (Highest) RxA, TxA, RxA, RxB, TxBExTA, ExTB (Lowest).

1 (Highest) RxA, RxB, TxA, TxB, ExTA, ExTB (Lowest).

D5,D4,D3 Interrupt Code—specifies the behavior of the MPSC when it receives an interrupt acknowledge sequence from the CPU. (See Interrupt Vector Mode Table.)

0 X X Non-vectored interrupts—intended for use with an external interrupt controller such as the 8259A.

1 0 0 8085 Vector Mode 1—intended for use as the primary MPSC in a daisy-chained priority structure.

1 0 1 8085 Vector Mode 2—intended for use as any secondary MPSC in a daisy-chained priority structure.

1 1 0 8086/88 Vector Mode—intended for use as either a primary or secondary in a daisy-chained priority structure.

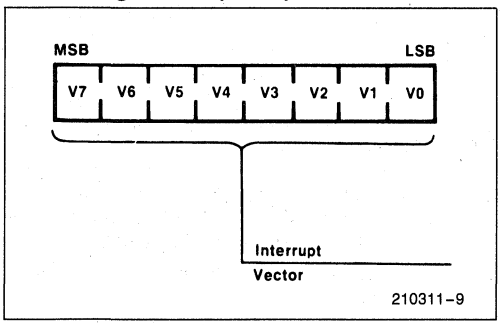
D6 Must be Zero.

D7

0 Pin 10 = $\overline{RTS}_B$.

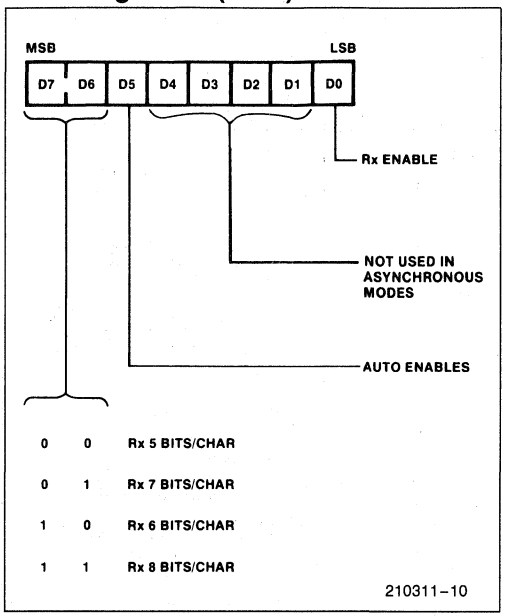
1 Pin 10 = $\overline{SYNDET}_B$.

Write Register 2 (WR2): Channel B



D7–D0 Interrupt vector—this register contains the value of the interrupt vector placed on the data bus during acknowledge sequences.

Write Register 3 (WR3):



D0 Receiver Enable—a one enables the receiver to begin. This bit should be set only after the receiver has been initialized.

D5 Auto Enables—a one written to this bit causes $\overline{CD}$ to be an automatic enable signal for the receiver and $\overline{CTS}$ to be an automatic enable signal for the transmitter. A zero written to this bit limits the effect of $\overline{CD}$ and $\overline{CTS}$ signals to setting/resetting their corresponding bits in the status register (RR0).

D7,D6 Receiver Character length.

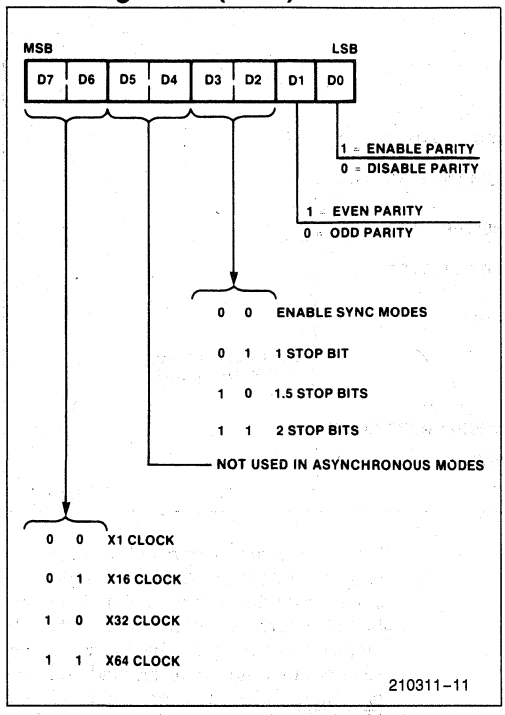
0 0 Receive 5 Data bits/character.

0 1 Receive 7 Data bits/character.

1 0 Receive 6 Data bits/character.

1 1 Receive 8 Data bits/character.

Write Register 4 (WR4):



D0 Parity—a one in this bit causes a parity bit to be added to the programmed number of data bits per character for both the transmitted and received character. If the MPSC is programmed to receive 8 bits per character, the parity bit is not transferred to the microprocessor. With other receiver character lengths, the parity bit is transferred to the microprocessor.

D1 Even/Odd Parity—if parity is enabled, a one in this bit causes the MPSC to transmit and expect even parity, and zero causes it to send and expect odd parity.

D3,D2 Stop Bits.

0 0 Selects synchronous modes.

0 1 Async mode, 1 stop bit/character.

1 0 Async mode, 1½ stop bits/character.

1 1 Async mode, 2 stop bits/character.

D7,D6 Clock mode—selects the clock/data rate multiplier for both the receiver and the transmitter. If the 1x mode is selected, bit synchronization must be done externally.

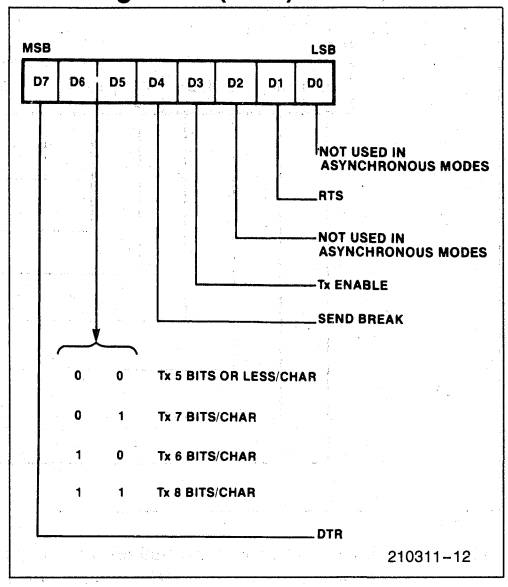
0 0 Clock rate = Data rate × 1.

0 1 Clock rate = Data rate × 16.

1 0 Clock rate = Data rate × 32.

1 1 Clock rate = Data rate × 64.

Write Register 5 (WR5):



D1 Request to Send—a one in this bit forces the $\overline{RTS}$ pin active (low) and zero in this bit forces the $\overline{RTS}$ pin inactive (high). When the RTS bit is reset in asynchronous mode, the signal does not go inactive until the transmitter is empty.

D3 Transmitter Enable—a zero in this bit forces a marking state on the transmitter output. If this bit is set to zero during data or sync character transmission, the marking state is entered after the character has been sent. If this bit is set to zero during transmission of a CRC character, sync or flag bits are substituted for the remainder of the CRC bits.

D4 Send Break—a one in this bit forces the transmit data low. A zero in this bit allows normal transmitter operation.

D6,D5 Transmit Character length.

0 0 Transmit 5 or less bits/character.

0 1 Transmit 7 bits/character.

1 0 Transmit 6 bits/character.

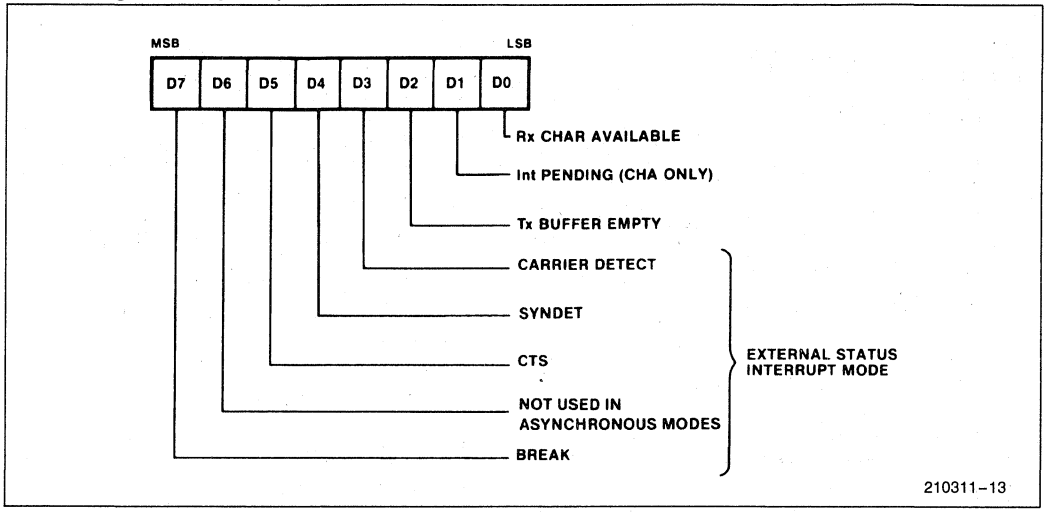
1 1 Transmit 8 bits/character.

Bits to be sent must be right justified, least-significant bit first, e.g.:

D7 D6 D5 D4 D3 D2 D1 D0

0 0 B5 B4 B3 B2 B1 B0

Read Register 0 (RR0):



D0 Receive Character Available—this bit is set when the receive FIFO contains data and is reset when the FIFO is empty.

D1 Interrupt Pending—This Interrupt-Pending bit is reset when an E01 command is issued and there is no other interrupt request pending at that time. In vector mode, this bit is set at the falling edge of the second INTA in an INTA cycle for an internal interrupt request. In non-vector mode, this bit is set at the falling edge of RD input after pointer 2 is specified. This bit is always zero in Channel B.

D2 Transmit Buffer Empty—This bit is set whenever the transmit buffer is empty except when CRC characters are being sent in a synchronous mode. This bit is reset when the transmit buffer is loaded. This bit is set after an MPSC reset.

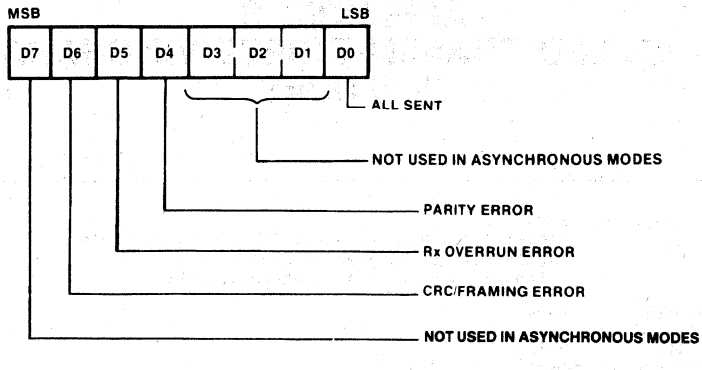
D3 Carrier Detect—This bit contains the state of the $\overline{CD}$ pin at the time of the last change of any of the External/Status bits (CD, CTS, Sync/Hunt, Break/Abort, or Tx Underrun/EOM). Any change of state of the $\overline{CD}$ pin causes the CD bit to be latched and causes an External/Status interrupt. This bit indicates current state of the $\overline{CD}$ pin immediately following a Reset External/Status Interrupt command.

D4 SYNDET—In asynchronous modes, the operation of this bit is similar to the CD status bit, except that it shows the state of the $\overline{SYNDET}$ input. Any High-to-Low transition on the $\overline{SYNDET}$ pin sets this bit, and causes an External/Status interrupt (if enabled). The Reset External/Status Interrupt command is issued to clear the interrupt. A Low-to-High transition clears this bit and sets the External/Status interrupt. When the External/Status interrupt is set by the change in state of any other input or condition, this bit shows the inverted state of the $\overline{SYNDET}$ pin at time of the change. This bit must be read immediately following a Reset External/Status Interrupt command to read the current state of the $\overline{SYNDET}$ input.

D5 Clear to Send—this bit contains the inverted state of the $\overline{CTS}$ pin at the time of the last change of any of the External/Status bits (CD, CTS, Sync/Hunt, Break/Abort, or Tx Underrun/EOM). Any change of state of the $\overline{CTS}$ pin causes the CTS bit to be latched and causes an External/Status interrupt. This bit indicates the inverse of the current state of the $\overline{CTS}$ pin immediately following a Reset External/Status Interrupt command.

D7 Break—in the Asynchronous Receive mode, this bit is set when a Break sequence (null character plus framing error) is detected in the data stream. The External/Status interrupt, if enabled, is set when break is detected. The interrupt service routine must issue the Reset External/Status Interrupt command (WR0, Command 2) to the break detection logic so the Break sequence termination can be recognized.

Read Register 1 (RR1):



The Break bit is reset when the termination of the Break sequence is detected in the incoming data stream. The termination of the Break sequence also causes the External/Status interrupt to be set. The Reset External/Status Interrupt command must be issued to enable the break detection logic to look for the next Break sequence. A single, extraneous null character is present in the receiver after the termination of a break; it should be read and discarded.

D0 All sent—this bit is set when all characters have been sent. It is reset when characters are in the transmitter. In synchronous modes, this bit is always set.

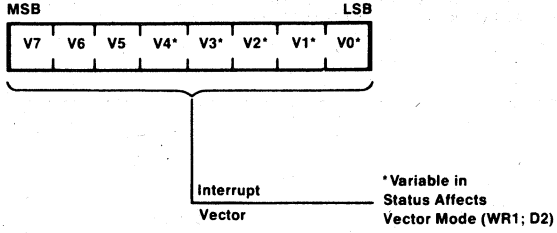
D4 Parity Error—if parity is enabled, this bit is set for received characters whose parity does not match the programmed sense (Even/Odd). This bit is latched. Once an error occurs, it remains set until the Error Reset command is written.

D5 Receive Overrun Error—this bit indicates that the receive FIFO has been overloaded by the receiver. The last character in the FIFO is overwrittenand flagged with this error. Once the overwritten character is read, this error condition is latched until reset by the Error Reset command. If the MPSC is in the "status affects vector" mode, the overrun causes a Special Receive Error Vector.

D6 Framing Error—in async modes, a one in this bit indicates a receive framing error. It can be reset by issuing an Error Reset command.

RR2 Channel B

Read Register 2 (RR2):



D7–D0 Interrupt vector—contains the interrupt vector programmed into WR2. If the "status affects vector" mode is selected, it contains the modified vector. (See WR2.) RR2 contains the modified vector for the highest priority interrupt pending. If no interrupts are pending, the variable bits in the vector are set to one. May be read from Channel B only.

APPENDIX B
MPSC-POLLED TRANSMIT/RECEIVE CHARACTER ROUTINES

```
MPSC$RX$INIT: procedure (cmd$port,
                         clock$rate,stop$bits,parity$type,parity$enable,
                         rx$char$length,rx$enable,auto$enable,
                         tx$char$length,tx$enable,dtr,brk,rts);

  declare cmd$port        byte,
          clock$rate      byte,
          stop$bits       byte,
          parity$type     byte,
          parity$enable   byte,
          rx$char$length  byte,
          rx$enable       byte,
          auto$enable     byte,
          tx$char$length  byte,
          tx$enable       byte,
          dtr             byte,
          brk             byte,
          rts             byte;


  output(cmd$port)=30H;          /* channel reset */

  output(cmd$port)=14H;          /* point to WR4 */
  /* set clock rate, stop bits, and parity information */
  output(cmd$port)=shl(clock$rate,6) or shl(stop$bits,2) or shl(parity$type,1)
                   or parity$enable;

  output(cmd$port)=13H;          /* point to WR3 */
  /* set up receiver parameters */
  output(cmd$port)=shl(rx$char$length,6) or rx$enable or shl(auto$enable,5);

  output(cmd$port)=15H;          /* point to WR5 */
  /* set up transmitter parameters */
  output(cmd$port)=shl(tx$char$length,5) or shl(tx$enable,3) or shl(dtr,7)
                   or shl(brk,4) or shl(rts,1);

  end MPSC$RX$INIT;
```

210311-16

```
MPSC$POLL$RCV$CHARACTER: procedure(data$port,cmd$port,character$ptr) byte;

  declare data$port     byte,
          cmd$port      byte,
          character$ptr pointer,
          character     based character$ptr byte,
          status        byte;

  declare char$avail    literally '1',
          rcv$error     literally '70H';

  /* wait for input character ready */
  while (input(cmd$port) and char$avail) <> 0 do; end;

  /* check for errors in received character */
  output(cmd$port)=1;                          /* point to RR1 */
  if (status:=input(cmd$port) and rcv$error)
    then do;
      character=input(data$port);              /* read character to clear MPSC */
      call RECEIVE$ERROR(cmd$port,status);     /* clear receiver errors */
      return 0;                                /* error return - no character avail */
      end;
    else do;
      character=input(data$port);
      return 0FFH;                              /* good return - character avail */
      end;

end MPSC$POLL$RCV$CHARACTER;


MPSC$POLL$TRAN$CHARACTER: procedure(data$port,cmd$port,character);

  declare data$port      byte,
          cmd$port       byte,
          character      byte;

  declare tx$buffer$empty literally '4';

  /* wait for transmitter buffer empty */
  while not (input(cmd$port) and tx$buffer$empty) do; end;

  /* output character */
  output(data$port)=character;

end MPSC$POLL$TRAN$CHARACTER;


RECEIVE$ERROR: procedure(cmd$port,status);

  declare cmd$port      byte,
          status        byte;

  output(cmd$port)=30H;              /* error reset */

  /* *** other application dependent
         error processing should be placed here   *** */

end RECEIVE$ERROR;
```

210311-17

```
TRANSMIT$BUFFER: procedure(buf$ptr,buf$length)

  declare
    buf$ptr       pointer,
    buf$length    byte;

  /* set up transmit buffer pointer and buffer length in global variables for
     interrupt service */
  tx$buffer$ptr=buf$ptr;
  transmit$length=buf$length;

  transmit$status=not$complete;            /* setup status for not complete */
  output(data$port)=transmit$buffer(0);    /* transmit first character */
  transmit$index=1;                        /* first character transmitted */

  /* wait until transmission complete or error detected */
  while transmit$status = not$complete do; end;
  if transmit$status <> complete
    then return false;
    else return true;

end TRANSMIT$BUFFER;



RECEIVE$BUFFER: procedure (buf$ptr,buf$length$ptr);

  declare
    buf$ptr        pointer,
    buf$length$ptr pointer,
    buf$length     based buf$length$ptr byte;

  /* set up receive buffer pointer in global variable for interrupt service */
  rx$buffer$ptr=buf$ptr;
  receive$index=0;

  receive$status=not$complete;          /* set status to not complete */
  /* wait until buffer received */
  while receive$status = not$complete do; end;
  buf$length=receive$length;
  if receive$status = complete
    then return true;
    else return false;

end RECEIVE$BUFFER;
```

210311-18

APPENDIX C
INTERRUPT-DRIVEN TRANSMIT/RECEIVE SOFTWARE

```
declare
  /* global variables for buffer manipulation */

  rx$buffer$ptr          pointer,           /* pointer to receive buffer */
  receive$buffer based rx$buffer$ptr(128) byte,
  receive$status         byte initial(0),   /* indicates receive buffer status */
  receive$index          byte,              /* current index into receive buffer */
  receive$length         byte,              /* length of final receive buffer */

  tx$buffer$ptr          pointer,           /* pointer to transmit buffer */
  transmit$buffer based tx$buffer$ptr(128) byte,
  transmit$status        byte initial(0),   /* indicates transmit buffer status */
  transmit$index         byte,              /* current index into transmit buffer */
  transmit$length        byte,              /* length of buffer to be transmitted */

  cmd$port               literally '43H',
  data$port              literally '41H',
  a$cmd$port             literally '42H',
  b$cmd$port             literally '43H',
  line$feed              literally '0AH',
  not$complete           literally '0',
  complete               literally '0FFH',
  overrun                literally '1',

  channel$reset          literally '18H',
  error$reset            literally '30H',
  reset$ext$status       literally '10H';
```

210311-20

```
MPSC$INT$INIT: procedure (clock$rate,stop$bits,parity$type,parity$enable,
                          rx$char$length,rx$enable,auto$enable,
                          tx$char$length,tx$enable,dtr,brk,rts,
                          ext$en,tx$en,rx$en,stat$affects$vector,
                          config,priority,vector$int$mode,int$vector);

  declare
        clock$rate        byte,         /* 2-bit code for clock rate divisor */
        stop$bits         byte,         /* 2-bit code for number of stop bits */
        parity$type       byte,         /* 1-bit parity type */
        parity$enable     byte,         /* 1-bit parity enable */
        rx$char$length    byte,         /* 2-bit receive character length */
        rx$enable         byte,         /* 1-bit receiver enable */
        auto$enable       byte,         /* 1-bit auto enable flag */
        tx$char$length    byte,         /* 2-bit transmit character length */
        tx$enable         byte,         /* 1-bit transmitter enable */
        dtr               byte,         /* 1-bit status of DTR pin */
        brk               byte,         /* 1-bit data link break enable */
        rts               byte,         /* 1-bit status of RTS pin */
        ext$en            byte,         /* 1-bit external/status enable */
        tx$en             byte,         /* 1-bit Tx interrupt enable */
        rx$en             byte,         /* 2-bit Rx interrupt enable/mode */
        stat$aff$vector   byte,         /* 1-bit status affects vector flag */
        config            byte,         /* 2-bit system config - int/DMA */
        priority          byte,         /* 1-bit priority flag */
        vector$int$mode   byte,         /* 3-bit interrupt mode code */
        int$vector        byte;         /* 8-bit interrupt type code */


  output(b$cmd$port)=channel$reset;   /* channel reset */

  output(b$cmd$port)=14H;             /* point to WR4 */
  /* set clock rate, stop bits, and parity information */
  output(b$cmd$port)=shl(clock$rate,6) or shl(stop$bits,2) or shl(parity$type,1)
                    or parity$enable;

  output(b$cmd$port)=13H;             /* point to WR3 */
  /* set up receiver parameters */
  output(b$cmd$port)=shl(rx$char$length,6) or rx$enable or shl(auto$enable,5);

  output(b$cmd$port)=15H;          /* point to WR5 */
  /* set up transmitter parameters */
  output(b$cmd$port)=shl(tx$char$length,5) or shl(tx$enable,3) or shl(dtr,7)
                  or shl(brk,4) or shl(rts,1);

  output(b$cmd$port)=12H;               /* point to WR2 */
  /* set up interrupt vector */
  output(b$cmd$port)=int$vector;

  output(a$cmd$port)=12H;               /* point to WR2, channel A */
  /* set up interrupt modes */
  output(a$cmd$port)=shl(vector$int$mode,3) or shl(priority,2) or config;

  output(b$cmd$port)=11H;               /* point to WR1 */
  /* set up interrupt enables */
  output(b$cmd$port)=shl(rx$en,3) or shl(stat$aff$vector,2) or shl(tx$en,1)
                    or ext$en;

  end MPSC$INT$INIT;
```

210311-21

```
MPSC$RECEIVE$CHARACTER$INT: procedure interrupt 22H;

  /* ignore input if no open buffer */
  if receive$status <> not$complete then return;

  /* check for receive buffer overrun */
  if receive$index = 128
    then receive$status=overrun;
    else do;
      /* read character from MPSC and place in buffer - note that the
         parity of the character must be masked off during this step if
         the character is less than 8 bits (e.g., ASCII)  */
      receive$buffer(receive$index),character=input(data$port) and 7FH;
      receive$index=receive$index+1;      /* update receive buffer index */

      /* check for line feed to end line */
      if character = line$feed
        then do; receive$length=receive$index; receive$status=complete; end;
    end;

end MPSC$RECEIVE$CHARACTER$INT;


MPSC$TRANSMIT$CHARACTER$INT: procedure interrupt 20H;

  /* check for more characters to transfer */
  if transmit$index < transmit$length
    then do;
      /* write next character from buffer to MPSC */
      output(data$port)=transmit$buffer(transmit$index);
      transmit$index=transmit$index+1;     /* update transmit buffer index */
    end;
    else transmit$status=complete;

end MPSC$TRANSMIT$CHARACTER$INT;


RECEIVE$ERROR$INT: procedure interrupt 23H;

  declare
    temp                  byte;    /* temporary character storage */

  output(cmd$port)=1;              /* point to RR1 */
  receive$status=input(cmd$port);
  temp=input(data$port);           /* discard character */
  output(cmd$port)=error$reset;    /* send error reset */

  /* *** other application dependent
         error processing should be placed here   *** */

end RECEIVE$ERROR$INT;


EXTERNAL$STATUS$CHANGE$INT: procedure interrupt 21H;

  transmit$status=input(cmd$port)      /* input status change information */
  output(cmd$port)=reset$ext$status;

  /* *** other application dependent
         error processing should be placed here   *** */

end EXTERNAL$STATUS$CHANGE$INT;
```

210311-19

APPENDIX D
APPLICATION EXAMPLE USING SDK-86

This application example shows the 8274 in a simple iAPX-86/88 system. The 8274 controls two separate asynchronous channels using its internal interrupt controller to request all data transfers. The 8274 driver software is described which transmits and receives data buffers provided by the CPU. Also, status registers are maintained in system memory to allow the CPU to monitor progress of the buffers and error conditions.

THE HARDWARE INTERFACE

Nothing could be easier than the hardware design of an interrupt-driven 8274 system. Simply connect the data bus lines, a few bus control lines, supply a timing clock for baud rate and, voila, it's done! For this example, the ubiquitous SDK-86 is used as the host CPU system. The 8274 interface is constructed on the wire-wrap area provided. While discussing the hardware interface, please refer to Diagram 1.

Placing the 8274 on the lower 8 bits of the 8086 data bus allows byte-wide data transfers at even I/O addresses. For simplicity, the 8274's $\overline{CS}$ input is generated by combining the M/IO select line with address line A7 via a 7432. This places the 8274 address range in multiple spots within the 8086 I/O address space. (While fine for this example, a more complete address decoding is recommended for actual prototype systems.) The 8086's A1 and A2 address lines are connected to the A0 and A1 8274 register select inputs respectively. Although other port assignments are possible because of the overlapping address spaces, the following I/O port assignments are used in this example:

Port Function	I/O Address
Data channel A	0000H
Command/status A	0002H
Data channel B	0004H
Command/status B	0006H

To connect the 8274's interrupt controller into the system an inverter and pull-up resistor are needed to convert the 8274's active-low, interrupt-request output, $\overline{INT}$, into the correct polarity for the 8086's INTR interrupt input. The 8274 recognizes interrupt-acknowledge bus cycles by connecting the $\overline{INTA}$ (INTerrupt Acknowledge) lines of the 8274 and 8086 together.

The 8274 ReaD and WRite lines directly connect to the respective 8086 lines. The RESET line requires an inverter. The system clock for the 8274 is provided by the PCLK (peripheral clock) output of the 8284A clock generator.

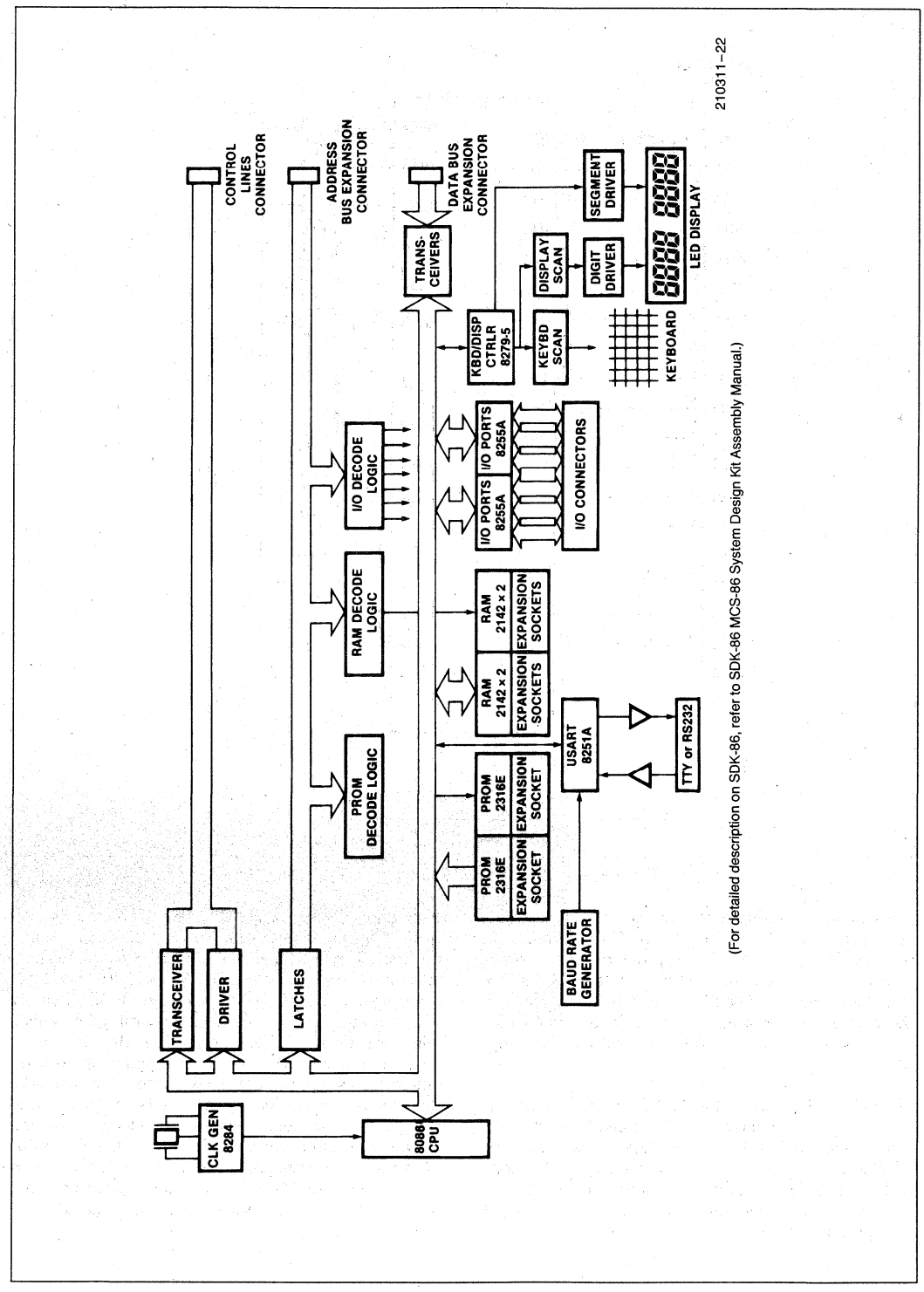
On the 8274's serial side, traditional 1488 and 1489 RS-232 drivers and receivers are used for the serial interface. The onboard baud rate generator supplies the channel baud rate timing. In this example, both sides of both channels operate at the same baud rate although this certainly is not a requirement. (On the SDK-86, the baud rate selection is hard-wired thru jumpers. A more flexible approach would be to incorporate an 8253 Programmable Interval Timer to allow software-configurable baud rate selection.)

That's all there is to it. This hardware interface is completely general-purpose and supports all of the 8274 features except the DMA data transfer mode which requires an external DMA controller. Now let's look at the software interface.

SOFTWARE INTERFACE

In this example, it is assumed that the 8086 has better things to do rather than continuously run a serial channel. Presenting the software as a group of callable procedures lets the designer include them in the main body of another program. The interrupt-driven data transfers give the effect that the serial channels are handled in the background while the main program is executing in the foreground. There are five basic procedures: a serial channel initialization routine and buffer handling routines for the transmit and receive data buffers of each channel. Appendix D-1 shows the entire software listing. Listing line numbers are referenced as each major routing is discussed.

The channel initialization routine (INITIAL 8274), starting with line #203, simply sets each channel into a particular operating mode by loading the command registers of the 8274. In normal operation, once these registers are loaded, they are rarely changed. (Although this example assumes a simple asynchronous operating mode, the concept is easily extended for the byte- and bit-synchronous modes.)



210311-22

(For detailed description on SDK-86, refer to SDK-86 MCS-86 System Design Kit Assembly Manual.)

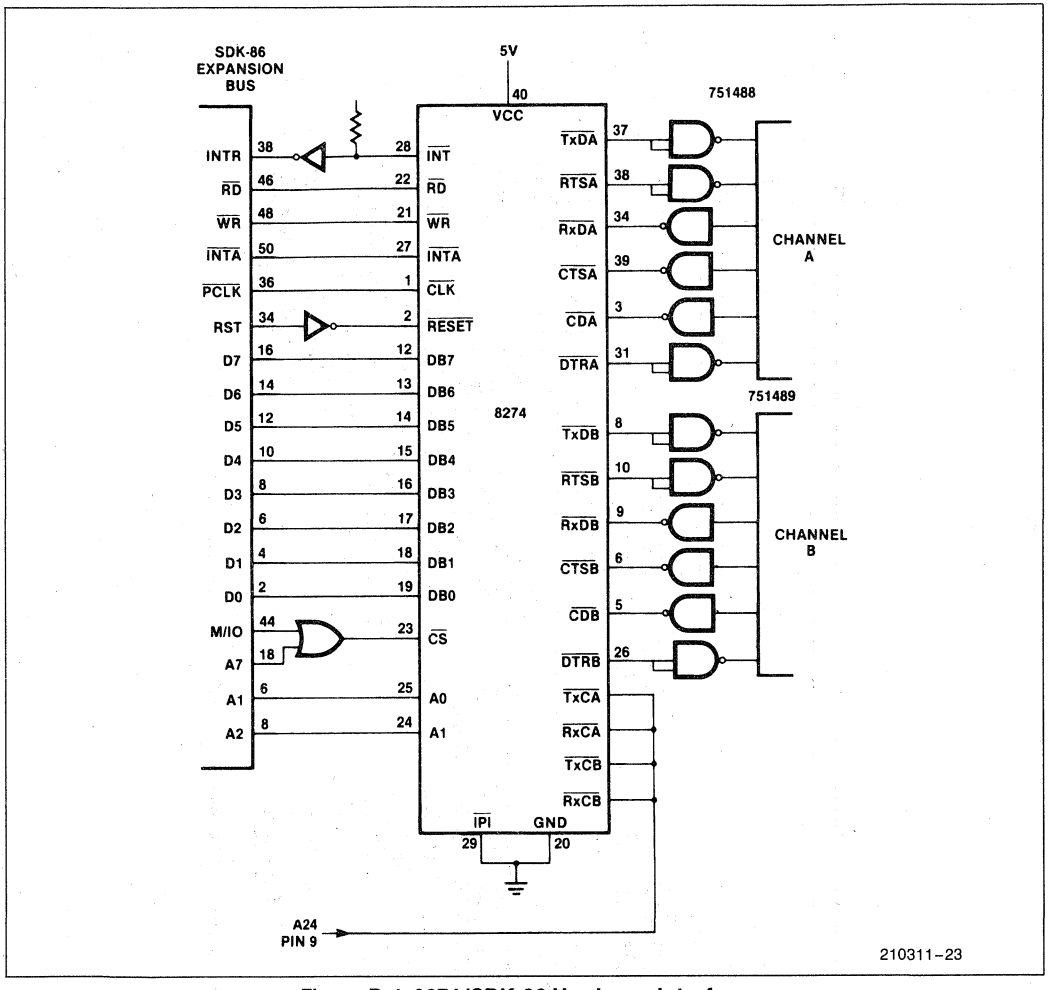


Figure D-1. 8274/SDK-86 Hardware Interface

The channel operating modes are contained in two tables starting with line #163. As the 8274 has only one command register per channel, the remaining seven registers are loaded indirectly through the WR0 (Write Register 0) register. The first byte of each table entry is the register pointer value which is loaded into WR0 and the second byte is the value for that particular register.

The indicated modes set the 8274 for asynchronous operation with data characters 8 bits long, no parity, and 2 stop bits. An X16 baud rate clock is assumed. Also selected is the "interrupt on all RX character" mode with a variable interrupt vector compatible with the 8086/8088. The transmitters are enabled and all model control lines are put in their active state.

In addition to initializing the 8274, this routine also sets up the appropriate interrupt vectors. The 8086 assumes the first 1K bytes of memory contain up to 256 separate interrupt vectors. On the SDK-86 the initial 2K bytes of memory is RAM and therefore must be initialized with the appropriate vectors. (In a prototype system, this initial memory is probably ROM, thus the vector set-up is not needed.) The 8274 supplies up to eight different interrupt vectors. These vectors are developed from internal conditions such as data requests, status changes, or error conditions for each channel. The initialization routine arbitrarily assumes that the initial 8274 vector corresponds to 8086 vector location 80H (memory location 200H). This choice is arbitrary since the 8274 initial vector location is programmable.

Finally, the initialization routine sets up the status and flag in RAM. The meaning and use of these locations are discussed later.

Following the initialization routine are those for the transmit commands (starting with line #268). These commands assume that the host CPU has initialized the publicly declared variables for the transmit buffer pointer, TX__POINTER__CHx, and the buffer length, TX__LENGTH__CHx. The transmit command routines simply clear the transmitter empty flag, TX EMPTY CHx, and load the first character of the buffer into the transmitter. It is necessary to load the first character in this manner since transmitter interrupts are generated only when the 8274's transmit data buffer becomes empty. It is the act of becoming empty which generates the interrupt not simply the buffer being empty, thus the transmitter needs one character to start.

The host CPU can monitor the transmitter empty flag, TX__EMPTY__CHx, in order to determine when transmission of the buffer is complete. Obviously, the CPU should only call the command routine after first checking that the empty flag is set.

After returning to the main program, all transmitter data transfers are handled via the transmitter-interrupt service routines starting at lines #360 and #443. These routines start by issuing an End-Of-Interrupt command to the 8274. (This command resets the internal-interrupt controller logic of the 8274 for this particular vector and opens the logic for other internal interrupt requests. The routines next check the length count. If the buffer is completely transmitted, the transmitter empty flag, TX__EMPTY__CHx, is set and a command is issued to the 8274 to reset its interrupt line. Assuming that the buffer is not completely transmitted, the next character is output to the transmitter. In either case, an interrupt return is executed to return to the main CPU program.

The receiver commands start at line #314. Like the transmit commands, it is assumed that the CPU has initialized the receive-buffer-pointer public variable, RX__POINTER__CHx. This variable points to the first location in an empty receive buffer. The command routines clear the receiver ready flag, RX__READY__CHx, and then set the receiver enable bit in the 8274 WR3 register. With the receiver now enabled, any received characters are placed in the receive buffer using interrupt-driven data transfers.

The received data service routines, starting at lines #402 and #485, simply place the received character in the buffer after first issuing the EOI command. The character is then compared to an ASCII CR. An ASCII CR causes the routine to set the receiver ready flag, RX__READY__CHx, and to disable the receiver. The CPU can interrogate this flag to determine when the buffer contains a new line of data. The receive buffer pointer, RX__POINTER__CHx, points to the last received character and the receive counter, RX__COUNTER__CHx, contains the length.

That completes our discussion of the command routines and their associated interrupt service routines. Although not used by the commands, two additional service routines are included for completeness. These routines handle the error and status-change interrupt vectors.

The error service routines, starting at lines #427 and #510, are vectored to if a special receive condition is detected by the 8274. These special receive conditions include parity, receiver overrun, and framing errors. When this vector is generated, the error condition is indicated in RR1 (Read Register 1). The error service routine issues an EOI command, reads RR1 and places it in the ERROR__MSG__CHx variable, and then issues a reset error command to the 8274. The CPU can monitor the error message location to detect error conditions. The designer, of course, can supply his own error service routine.

Similarly, the status-change routines (starting lines #386 and #469) are initiated by a change in the modem-control status lines CTS/, CD/, or SYNDET/. (Note that WR2 bit 0 controls whether the 8274 generates interrupts based upon changes in these lines. Our WR2 parameter is such that the 8274 is programmed to ignore changes for these inputs.) The service routines simply read RR0, place its contents in the STATUS__MSG__CHx variable and then issue a reset external status command. Read Register 0 contains the state of the modem inputs at the point of the last change.

Well, that's it. This application example has presented useful, albeit very simple, routines showing how the 8274 might be used to transmit and receive buffers using an asynchronous serial format. Extensions for byte- or bit-synchronous formats would require no hardware changes due to the highly programmable nature of the 8274's serial formats.

8274 APPLICATION BRIEF PROGRAM

```
ISIS-II MCS-86 MACRO ASSEMBLER V2.1 ASSEMBLY OF MODULE ASYNCB
OBJECT MODULE PLACED IN :F1:ASYNCB.OBJ
ASSEMBLER INVOKED BY:  ASM86 :F1:ASYNCB.SRC


LOC  OBJ                  LINE     SOURCE

                             1     ;*****************************************************************
                             2     ;*                                                               *
                             3     ;*              8274 APPLICATION BRIEF PROGRAM                   *
                             4     ;*                                                               *
                             5     ;*                                                               *
                             6     ;*                                                               *
                             7     ;* THE 8274 IS INITIALIZED FOR SIMPLE ASYNCHRONOUS SERIAL        *
                             8     ;* FORMAT AND VECTORED INTERRUPT-DRIVEN DATA TRANSFERS.          *
                             9     ;* THE INITIALIZATION ROUTINE ALSO LOADS THE 8086'S INTERRUPT    *
                            10     ;* VECTOR TABLE FROM THE CODE SEGMENT INTO LOW RAM ON THE        *
                            11     ;* SDK-86.  THE TRANSMITTER AND RECEIVER ARE LEFT ENABLED.       *
                            12     ;*                                                               *
                            13     ;* FOR TRANSMIT, THE CPU PASSES IN MEMORY THE POINTER OF A       *
                            14     ;* BUFFER TO TRANSMIT AND THE BYTE LENGTH OF THE BUFFER.         *
                            15     ;* THE DATA TRANSFER PROCEED USING INTERRUPT-DRIVEN TRANSFERS.   *
                            16     ;* A STATUS BIT IN MEMORY IS SET WHEN IF BUFFERS IS EMPTY.       *
                            17     ;*                                                               *
                            18     ;* FOR RECEIVE, THE CPU PASSES THE POINTER OF A BUFFER TO FILL.  *
                            19     ;* THE BUFFER IS FILLED UNTIL A 'CR_CHR' CHARACTER IS RECEIVED.  *
                            20     ;* A STATUS BIT IS SET AND THE CPU MAY READ THE RX POINTER TO    *
                            21     ;* DETERMINE THE LOCATION OF THE LAST CHARACTER.                 *
                            22     ;*                                                               *
                            23     ;* ALL ROUTINES ARE ASSUMED TO EXIST IN THE SAME CODE SEGMENT.   *
                            24     ;* CALL'S TO THE SERVICE ROUTINES ARE ASSUMED TO BE "SHORT" OR    *
                            25     ;* INTRASEGMENT (ONLY THE RETURN ADDRESS IP IS ON THE STACK).    *
                            26     ;*                                                               *
                            27     ;*                                                               *
                            28     ;*                                                               *
                            29     ;*                                                               *
                            30     ;*****************************************************************
```

210311-24

```
MCS-86 MACRO ASSEMBLER   ASYNCB

LOC  OBJ                 LINE    SOURCE

                           31
                           32            NAME    ASYNCB  ;MODULE NAME
                           33
                           34    ;PUBLIC DECLARATIONS FOR COMMAND ROUTINES
                           35
                           36            PUBLIC  INITIAL_8274    ;INITIALIZATION ROUTINE
                           37            PUBLIC  TX_COMMAND_CHB  ;TX BUFFER COMMAND CHANNEL B
                           38            PUBLIC  TX_COMMAND_CHA  ;TX BUFFER COMMAND CHANNEL A
                           39            PUBLIC  RX_COMMAND_CHB  ;RX BUFFER COMMAND CHANNEL B
                           40            PUBLIC  RX_COMMAND_CHA  ;RX BUFFER COMMAND CHANNEL A
                           41
                           42    ;PUBLIC DECLARATIONS FOR STATUS VARIABLES
                           43
                           44            PUBLIC  RX_READY_CHB    ;RX READY FLAG CHB
                           45            PUBLIC  RX_READY_CHA    ;RX READY FLAG CHA
                           46            PUBLIC  TX_EMPTY_CHB    ;TX EMPTY FLAG CHB
                           47            PUBLIC  TX_EMPTY_CHA    ;TX EMPTY FLAG CHA
                           48            PUBLIC  RX_COUNT_CHB    ;RX BUFFER COUNTER CHB
                           49            PUBLIC  RX_COUNT_CHA    ;RX BUFFER COUNTER CHA
                           50            PUBLIC  ERROR_MSG_CHB   ;ERROR FLAG CHB
                           51            PUBLIC  ERROR_MSG_CHA   ;ERROR FLAG CHA
                           52            PUBLIC  STATUS_MSG_CHB  ;STATUS FLAG CHB
                           53            PUBLIC  STATUS_MSG_CHA  ;STATUS FLAG CHA
                           54
                           55    ;PUBLIC DECLARATIONS FOR VARIABLES PASSED TO THE TRANSMIT
                           56    ;AND RECEIVE COMMANDS.
                           57
                           58            PUBLIC  TX_POINTER_CHB  ;TX BUFFER POINTER FOR CHB
                           59            PUBLIC  TX_LENGTH_CHB   ;TX LENGTH OF BUFFER FOR CHB
                           60            PUBLIC  TX_POINTER_CHA  ;TX BUFFER POINTER FOR CHA
                           61            PUBLIC  TX_LENGTH_CHA   ;TX LENGTH OF BUFFER FOR CHA
                           62            PUBLIC  RX_POINTER_CHB  ;RX BUFFER POINTER FOR CHB
                           63            PUBLIC  RX_POINTER_CHA  ;RX BUFFER POINTER FOR CHA
                           64
                           65    ;I/O PORT ASSIGNMENTS
                           66
                           67    ;CHANNEL A PORT ASSIGNMENTS
                           68
0000                       69    DATA_PORT_CHA           EQU     0                       ;DATA I/O PORT
0002                       70    COMMAND_PORT_CHA        EQU     2                       ;COMMAND PORT
0002                       71    STATUS_PORT_CHA         EQU     COMMAND_PORT_CHA        ;STATUS PORT
                           72
                           73    ;CHANNEL B PORT ASSIGNMENTS
                           74
0004                       75    DATA_PORT_CHB           EQU     4                       ;DATA I/O PORT
0006                       76    COMMAND_PORT_CHB        EQU     6                       ;COMMAND PORT
0006                       77    STATUS_PORT_CHB         EQU     COMMAND_PORT_CHB        ;STATUS PORT
                           78
                           79    ;MISC  SYSTEM EQUATES
                           80
0000                       81    CR_CHR          EQU     00H     ;ASCII CR CHARACTER CODE
0200                       82    INT_TABLE_BASE  EQU     200H    ;INT  VECTOR BASE ADDRESS
0500                       83    CODE_START      EQU     500H    ;START LOCATION FOR CODE
                           84
                           85 +1 $EJECT
                           86
                           87    ;RAM ASSSIGNMENTS FOR DATA SEGMENT
                           88
----                       89            DATA    SEGMENT
                           90
```

210311-25

```
MCS-86 MACRO ASSEMBLER    ASYNCB


LOC  OBJ                  LINE     SOURCE

                            91     ;VECTOR INTERRUPT TABLE - ASSUME INITIAL 8274 INTERRUPT
                            92     ;VECTOR IS NUMBER 80 (@200H)   FOR EACH VECTOR, THE TABLE
                            93     ;CONTAINS START LOCATION AND CODE SEGMENT REGISTER VALUE
                            94     ;THE TABLE IS LOADED FROM PROM
                            95
0200                        96             ORG     INT_TABLE_BASE
                            97
0200 0000                   98     TX_VECTOR_CHB   DW      0       ;TX INTERRUPT VECTOR FOR CHB
0202 0000                   99     TX_CS_CHB       DW      0
                           100
0204 0000                  101     STS_VECTOR_CHB  DW      0       ;STATUS INTERRUPT VECTOR FOR CHB
0206 0000                  102     STS_CS_CHB      DW      0
                           103
0208 0000                  104     RX_VECTOR_CHB   DW      0       ;RX INTERRUPT VECTOR FOR CHB
020A 0000                  105     RX_CS_CHB       DW      0
                           106
020C 0000                  107     ERR_VECTOR_CHB  DW      0       ;ERROR INTERRUPT VECTOR FOR CHB
020E 0000                  108     ERR_CS_CHB      DW      0
                           109
0210 0000                  110     TX_VECTOR_CHA   DW      0       ;TX INTERRUPT VECTOR FOR CHA
0212 0000                  111     TX_CS_CHA       DW      0
                           112
0214 0000                  113     STS_VECTOR_CHA  DW      0       ;STATUS INTERRUPT VECTOR FOR CHA
0216 0000                  114     STS_CS_CHA      DW      0
                           115
0218 0000                  116     RX_VECTOR_CHA   DW      0       ;RX INTERRUPT VECTOR FOR CHA
021A 0000                  117     RX_CS_CHA       DW      0
                           118
021C 0000                  119     ERR_VECTOR_CHA  DW      0       ;ERROR INTERRUPT VECTOR FOR CHA
021E 0000                  120     ERR_CS_CHA      DW      0
                           121
                           122     ;MISC RAM LOCATIONS FOR CHANNEL STATUS AND POINTERS
                           123
                           124     ;CHANNEL B POINTERS AND STATUS
                           125
0220 0000                  126     TX_POINTER_CHB  DW      0       ;TX BUFFER POINTER FOR CHB
0222 0000                  127     TX_LENGTH_CHB   DW      0       ;TX BUFFER LENGTH FOR CHB
0224 0000                  128     RX_POINTER_CHB  DW      0       ;RX BUFFER POINTER FOR CHB
0226 0000                  129     RX_COUNT_CHB    DW      0       ;RX LENGTH COUNTER FOR CHB
0228 00                    130     TX_EMPTY_CHB    DB      0       ;TX DONE FLAG
0229 00                    131     RX_READY_CHB    DB      0       ;READY FLAG (1 IF CR_CHR RECEIVED, ELSE 0)
022A 00                    132     STATUS_MSG_CHB  DB      0       ;STATUS CHANGE MESSAGE
022B 00                    133     ERROR_MSG_CHB   DB      0       ;ERROR STATUS LOCATION (0 IF NO ERROR)
                           134
                           135     ;CHANNEL A POINTERS AND STATUS
                           136
022C 0000                  137     TX_POINTER_CHA  DW      0       ;TX BUFFER POINTER FOR CHA
022E 0000                  138     TX_LENGTH_CHA   DW      0       ;TX BUFFER LENGTH FOR CHA
0230 0000                  139     RX_POINTER_CHA  DW      0       ;RX BUFFER POINTER FOR CHA
0232 0000                  140     RX_COUNT_CHA    DW      0       ;RX LENGTH COUNTER FOR CHA
0234 00                    141     TX_EMPTY_CHA    DB      0       ;TX DONE FLAG
0235 00                    142     RX_READY_CHA    DB      0       ;READY FLAG (1 IF CR_CHR RECEIVED, ELSE 0)
0236 00                    143     STATUS_MSG_CHA  DB      0       ;STATUS CHANGE MESSAGE
0237 00                    144     ERROR_MSG_CHA   DB      0       ;ERROR STATUS LOCATION (0 IF NO ERROR)
                           145
----                       146             DATA    ENDS
                           147
                           148 +1  $EJECT
```

210311-26

```
MCS-86 MACRO ASSEMBLER   ASYNCB

LOC  OBJ                 LINE    SOURCE

                         149
----                     150             ABC     SEGMENT
                         151             ASSUME  CS:ABC,DS:DATA,SS:DATA
0500                     152             ORG     CODE_START
                         153
                         154     ;*******************************************************
                         155     ;*                                                     *
                         156     ;*          PARAMETERS FOR CHANNEL INITIALIZATION      *
                         157     ;*                                                     *
                         158     ;*******************************************************
                         159
                         160     ;CHANNEL B PARAMETERS
                         161
                         162             ;WR1 - INTERRUPT ON ALL RX CHR, VARIABLE INT VECTOR, TX INT ENABLE
0500 01                  162     CMDSTRB DB      1,16H
0501 16
                         164             ;WR2 - INTERRUPT VECTOR
0502 02                  165             DB      2,(INT_TABLE_BASE/4)
0503 80
                         166             ;WR3 - RX 8 BITS/CHR, RX DISABLE
0504 03                  167             DB      3,0C0H
0505 C0
                         168             ;WR4 - X16 CLOCK, 2 STOP BITS, NO PARITY
0506 04                  169             DB      4,4CH
0507 4C
                         170             ;WR5 - DTR ACTIVE, TX 8 BITS/CHR, TX ENABLE, RTS ACTIVE
0508 05                  171             DB      5,0EAH
0509 EA
                         172             ;WR6 AND WR7 NOT REQUIRED FOR ASYNC
050A 00                  173             DB      0,0
050B 00
                         174
                         175     ;CHANNEL A PARAMETERS
                         176
                         177             ;WR1 - INTERRUPT ON ALL RX CHR, TX INT ENABLE
050C 01                  178     CMDSTRA DB      1,12H
050D 12
                         179             ;WR2 - VECTORED INTERRUPT FOR 8086
050E 02                  180             DB      2,30H
050F 30
                         181             ;WR3 - RX 8 BITS/CHR, RX DISABLE
0510 03                  182             DB      3,0C0H
0511 C0
                         183             ;WR4 - X16 CLOCK, 2 STOP BITS, NO PARITY
0512 04                  184             DB      4,4CH
0513 4C
                         185             ;WR5 - DTR ACTIVE, TX 8 BITS/CHR, TX ENABLE, RTS ACTIVE
0514 05                  186             DB      5,0EAH
0515 EA
                         187             ;WR6 AND WR7 NOT REQUIRED FOR ASYNC
0516 00                  188             DB      0,0
0517 00
                         189
                         190 +1  $EJECT
```

210311-27

```
MCS-86 MACRO ASSEMBLER   ASYNCB


LOC  OBJ              LINE    SOURCE

                       191
                       192     ;START OF COMMAND ROUTINES
                       193
                       194     ;*************************************************************
                       195     ;*                                                           *
                       196     ;*      INITIALIZATION COMMAND FOR THE 8274 - THE 8274       *
                       197     ;*      IS SETUP ACCORDING TO THE PARAMETERS STORED IN       *
                       198     ;*      PROM ABOVE STARTING AT CMSTRB FOR CHANNEL B AND      *
                       199     ;*      CMSTRA FOR CHANNEL A                                 *
                       200     ;*                                                           *
                       201     ;*************************************************************
                       202
0518                   203     INITIAL_8274:
                       204             ;COPY INTERRUPT VECTOR IP AND CS VALUES FROM PROM TO RAM
0518 C70600020806      205             MOV     TX_VECTOR_CHB, OFFSET XMTINB    ;TX DATA VECTOR CHB
051E 8C0E0202          206             MOV     TX_CS_CHB, CS
0522 C7060402506       207             MOV     STS_VECTOR_CHB, OFFSET STAINB   ;STATUS VECTOR CHB
0528 8C0E0602          208             MOV     STS_CS_CHB, CS
052C C7060802A906      209             MOV     RX_VECTOR_CHB, OFFSET RCVINB    ;RX DATA VECTOR CHB
0532 8C0E0A02          210             MOV     RX_CS_CHB, CS
0536 C7060C027706      211             MOV     ERR_VECTOR_CHB, OFFSET ERRINB   ;ERROR VECTOR CHB
053C 8C0E0A02          212             MOV     RX_CS_CHB, CS
0540 C70610028C06      213             MOV     TX_VECTOR_CHA, OFFSET XMTINA    ;TX DATA VECTOR CHA
0546 8C0E1202          214             MOV     TX_CS_CHA, CS
054A C7061402B906      215             MOV     STS_VECTOR_CHA, OFFSET STAINA   ;STATUS VECTOR CHA
0550 8C0E1602          216             MOV     STS_CS_CHA, CS
0554 C7061802CD06      217             MOV     RX_VECTOR_CHA, OFFSET RCVINA    ;RX DATA VECTOR CHA
055A 8C0E1A02          218             MOV     RX_CS_CHA, CS
055E C7061C02FB06      219             MOV     ERR_VECTOR_CHA, OFFSET ERRINA   ;ERROR VECTOR CHA
0564 8C0E1E02          220             MOV     ERR_CS_CHA, CS
                       221
                       222     ;COPY SETUP TABLE PARAMETERS INTO 8274
                       223
0568 BF0005            224             MOV     DI, OFFSET CMDSTRB      ;INITIALIZE CHB
056B BA0600            225             MOV     DX, COMMAND_PORT_CHB
056E E82E00            226             CALL    SETUP                   ;COPY CHB PARAMETERS
0571 BF0C05            227             MOV     DI, OFFSET CMDSTRA      ;INITIALIZE CHA
0574 BA0200            228             MOV     DX, COMMAND_PORT_CHA
0577 E82500            229             CALL    SETUP                   ;COPY CHA PARAMETERS
                       230
                       231     ;INITIALIZE STATUS BYTES AND FLAGS
                       232
057A B80000            233             MOV     AX, 0
057D A22B02            234             MOV     ERROR_MSG_CHB, AL       ;CLEAR ERROR FLAG CHB
0580 A23702            235             MOV     ERROR_MSG_CHA, AL       ;CLEAR ERROR FLAG CHA
0583 A22A02            236             MOV     STATUS_MSG_CHB, AL      ;CLEAR STATUS FLAG CHB
0586 A23602            237             MOV     STATUS_MSG_CHA, AL      ;CLEAR STATUS FLAG CHA
0589 A32602            238             MOV     RX_COUNT_CHB, AX        ;CLEAR RX COUNTER CHB
058C A33202            239             MOV     RX_COUNT_CHA, AX        ;CLEAR RX COUNTER CHA
058F B001              240             MOV     AL, 1
0591 A22902            241             MOV     RX_READY_CHB, AL        ;SET RX DONE FLAG CHB
0594 A23502            242             MOV     RX_READY_CHA, AL        ;SET RX DONE FLAG CHA
0597 A22802            243             MOV     TX_EMPTY_CHB, AL        ;SET TX DONE FLAG CHB
059A A23402            244             MOV     TX_EMPTY_CHA, AL        ;SET TX DONE FLAG CHA
059D FB                245             STI                             ;ENABLE INTERRUPTS
059E C3                246             RET                             ;RETURN - DONE WITH SETUP
                       247
059F 8A05              248     SETUP:  MOV     AL, [DI]                ;PARAMETER COPYING ROUTINE
05A1 3C00              249             CMP     AL, 0
05A3 7404              250             JE      DONE
```

210311-28

```
LOC  OBJ              LINE    SOURCE

05A5 EE                251             OUT     DX, AL                  ;OUTPUT PARAMETER
05A6 47                252             INC     DI                      ;POINT AT NEXT PARAMETER
05A7 EBF6              253             JMP     SETUP                   ;GO LOAD IT
05A9 C3                254     DONE:   RET                             ;DONE - SO RETURN
                       255
                       256 +1  $EJECT
                       257
                       258     ;*************************************************************
                       259     ;*                                                           *
                       260     ;*    TX CHANNEL B COMMAND ROUTINE - ROUTINE IS CALLED TO    *
                       261     ;*    TRANSMIT A BUFFER.  THE BUFFER STARTING ADDRESS,       *
                       262     ;*    TX_POINTER_CHB, AND THE BUFFER LENGTH, TX_LENGTH_CHB,  *
                       263     ;*    MUST BE INITIALIZED BY THE CALLING PROGRAM             *
                       264     ;*    BOTH ITEMS ARE WORD VARIABLES.                         *
                       265     ;*                                                           *
                       266     ;*************************************************************
                       267
05AA                   268     TX_COMMAND_CHB:
05AA 50                269             PUSH    AX              ;SAVE REGISTERS
05AB 57                270             PUSH    DI
05AC 52                271             PUSH    DX
05AD C606230200        272             MOV     TX_EMPTY_CHB, 0 ;CLEAR EMPTY FLAG
05B2 BA0400            273             MOV     DX, DATA_PORT_CHB       ;SETUP PORT POINTER
05B5 8B3E2002          274             MOV     DI, TX_POINTER_CHB      ;GET TX BUFFER POINTER CHB
05B9 8A05              275             MOV     AL, [DI]        ;GET FIRST CHARACTER TO TX
05BB EE                276             OUT     DX, AL          ;OUTPUT IT TO 8274 TO GET IT STARTED
05BC 5A                277             POP     DX
05BD 5F                278             POP     DI
05BE 58                279             POP     AX
05BF C3                280             RET                     ;RETURN
                       281
                       282     ;*************************************************************
                       283     ;*                                                           *
                       284     ;*    TX CHANNEL A COMMAND ROUTINE - ROUTINE IS CALLED TO    *
                       285     ;*    TRANSMIT A BUFFER.  THE BUFFER STARTING ADDRESS,       *
                       286     ;*    TX_POINTER_CHA, AND THE BUFFER LENGTH, TX_LENGTH_CHA,  *
                       287     ;*    MUST BE INITIALIZED BY THE CALLING PROGRAM.            *
                       288     ;*    BOTH ITEMS ARE WORD VARIABLES.                         *
                       289     ;*                                                           *
                       290     ;*************************************************************
                       291
05C0                   292     TX_COMMAND_CHA:
05C0 50                293             PUSH    AX              ;SAVE REGISTERS
05C1 57                294             PUSH    DI
05C2 52                295             PUSH    DX
05C3 C606340200        296             MOV     TX_EMPTY_CHA, 0 ;CLEAR EMPTY FLAG
05C8 BA0000            297             MOV     DX, DATA_PORT_CHA       ;SETUP PORT POINTER
05CB 8B3E2C02          298             MOV     DI, TX_POINTER_CHA      ;GET TX BUFFER POINTER CHA
05CF 8A05              299             MOV     AL, [DI]        ;GET FIRST CHARACTER TO TX
05D1 EE                300             OUT     DX, AL          ;OUTPUT IT TO 8274 TO GET IT STARTED
05D2 5A                301             POP     DX
05D3 5F                302             POP     DI
05D4 58                303             POP     AX
05D5 C3                304             RET                     ;RETURN
                       305
                       306     ;*************************************************************
                       307     ;*                                                           *
                       308     ;*    RX COMMAND FOR CHANNEL B - THE CALLING ROUTINE MUST    *
                       309     ;*    INITIALIZE RX_POINTER_CHB TO POINT AT THE RECEIVE      *
                       310     ;*    BUFFER BEFORE CALLING THIS ROUTINE                     *
```

210311-29

```
MCS-86 MACRO ASSEMBLER    ASYNCB


LOC  OBJ                  LINE     SOURCE

                          311      *                                                     *
                          312      *******************************************************
                          313
05D6                      314      RX_COMMAND_CHB
05D6 50                   315              PUSH    AX              ;SAVE REGISTERS
05D7 52                   316              PUSH    DX
05D8 C606290200           317              MOV     RX_READY_CHB, 0 ;CLEAR RX READY FLAG
05DD C70629020000         318              MOV     RX_COUNT_CHB, 0 ;CLEAR RX COUNTER
05E3 BA0600               319              MOV     DX, COMMAND_PORT_CHB    ;POINT AT COMMAND PORT
05E6 B003                 320              MOV     AL, 3           ;SET UP FOR WR3
05E8 EE                   321              OUT     DX, AL
05E9 B0C1                 322              MOV     AL, 0C1H        ;WR3 - 8 BITS/CHR, ENABLE RX
05EB EE                   323              OUT     DX, AL
05EC 5A                   324              POP     DX
05ED 58                   325              POP     AX
05EE C3                   326              RET                     ;RETURN
                          327
                          328      ;*************************************************************
                          329      ;*                                                           *
                          330      ;*    RX COMMAND FOR CHANNEL A - THE CALLING ROUTINE MUST    *
                          331      ;*    INITIALIZE RX_POINTER_CHA TO POINT AT THE RECEIVE      *
                          332      ;*    BUFFER BEFORE CALLING THIS ROUTINE                     *
                          333      ;*                                                           *
                          334      ;*************************************************************
                          335
05EF                      336      RX_COMMAND_CHA
05EF 50                   337              PUSH    AX              ;SAVE REGISTERS
05F0 52                   338              PUSH    DX
05F1 C606350200           339              MOV     RX_READY_CHA, 0 ;CLEAR RX READY FLAG
05F6 C706320200000        340              MOV     RX_COUNT_CHA, 0 ;CLEAR RX COUNTER
05FC BA0200               341              MOV     DX, COMMAND_PORT_CHA    ;POINT AT COMMAND PORT
05FF B003                 342              MOV     AL, 3           ;SET UP FOR WR3
0601 EE                   343              OUT     DX, AL
0602 B0C1                 344              MOV     AL, 0C1H        ;WR3 - 8 BITS/CHR, ENABLE RX
0604 EE                   345              OUT     DX, AL
0605 5A                   346              POP     DX
0606 58                   347              POP     AX
0607 C3                   348              RET                     ;RETURN
                          349
                          350 +1   $EJECT
                          351
                          352      ;*************************************************************
                          353      ;*                                                           *
                          354      ;*          START OF INTERRUPT SERVICE ROUTINES              *
                          355      ;*                                                           *
                          356      ;*************************************************************
                          357
                          358      ;CHANNEL B TRANSMIT DATA SERVICE ROUTINE
                          359
0608 52                   360      XMTINB: PUSH    DX              ;SAVE REGISTERS
0609 57                   361              PUSH    DI
060A 50                   362              PUSH    AX
060B E80201               363              CALL    EOI             ;SEND EOI COMMAND TO 8274
060E FF062002             364              INC     TX_POINTER_CHB  ;POINT TO NEXT CHARACTER
0612 FF0E2202             365              DEC     TX_LENGTH_CHB   ;DEC LENGTH COUNTER
0616 740E                 366              JE      XIB             ;TEST IF DONE
0618 BA0400               367              MOV     DX, DATA_PORT_CHB       ;NOT DONE - GET NEXT CHARACTER
061B 8B3E2002             368              MOV     DI, TX_POINTER_CHB
061F 8A05                 369              MOV     AL, [DI]        ;PUT CHARACTER IN AL
0621 EE                   370              OUT     DX, AL          ;OUTPUT IT TO 8274
```

210311-30

```
MCS-86 MACRO ASSEMBLER    ASYNCB

LOC  OBJ                  LINE    SOURCE

0622 58                   371             POP     AX              ;RESTORE REGISTERS
0623 5F                   372             POP     DI
0624 5A                   373             POP     DX
0625 CF                   374             IRET                    ;RETURN TO FOREGROUND
0626 BA0600               375     XIB:    MOV     DX, COMMAND_PORT_CHB    ;ALL CHARACTERS HAVE BEEN SEND
0629 B028                 376             MOV     AL, 28H         ;RESET TRANSMITTER INTERRUPT PENDING
062B EE                   377             OUT     DX, AL
062C C606280201           378             MOV     TX_EMPTY_CHB, 1 ;DONE - SO SET TX EMPTY FLAG CHB
0631 58                   379             POP     AX              ;RESTORE REGISTERS
0632 5F                   380             POP     DI
0633 5A                   381             POP     DX
0634 CF                   382             IRET                    ;RETURN TO FOREGROUND
                          383
                          384     ;CHANNEL B STATUS CHANGE SERVICE ROUTINE
                          385
0635 52                   386     STAINB: PUSH    DX              ;SAVE REGISTERS
0636 57                   387             PUSH    DI
0637 50                   388             PUSH    AX
0638 E8D500               389             CALL    EOI             ;SEND EOI COMMAND TO 8274
063B BA0600               390             MOV     DX, COMMAND_PORT_CHB
063E EC                   391             IN      AL, DX          ;READ RR0
063F A22A02               392             MOV     STATUS_MSG_CHB, AL      ;PUT RR0 IN STATUS MESSAGE
0642 B010                 393             MOV     AL, 10H         ;SEND RESET STATUS INT COMMAND TO 8274
0644 EE                   394             OUT     DX, AL
0645 58                   395             POP     AX              ;RESTORE REGISTERS
0646 5F                   396             POP     DI
0647 5A                   397             POP     DX
0648 CF                   398             IRET
                          399
                          400     ;CHANNEL B RECEIVED DATA SERVICE ROUTINE
                          401
0649 52                   402     RCVINB: PUSH    DX              ;SAVE REGISTERS
064A 57                   403             PUSH    DI
064B 50                   404             PUSH    AX
064C E8C100               405             CALL    EOI             ;SEND EOI COMMAND TO 8274
064F 8B3E2402             406             MOV     DI, RX_POINTER_CHB      ;GET RX CHB BUFFER POINTER
0653 BA0400               407             MOV     DX, DATA_PORT_CHB
0656 EC                   408             IN      AL, DX          ;READ CHARACTER
0657 8805                 409             MOV     [DI], AL        ;STORE IN BUFFER
0659 FF062402             410             INC     RX_POINTER_CHB  ;BUMP THE BUFFER POINTER
065D FF062602             411             INC     RX_COUNT_CHB    ;BUMP THE COUNTER
0661 3C0D                 412             CMP     AL, CR_CHR      ;TEST IF LAST CHARACTER TO BE RECEIVED?
0663 750E                 413             JNE     RIB
0665 C606290201           414             MOV     RX_READY_CHB, 1 ;YES, SET READY FLAG
066A BA0600               415             MOV     DX, COMMAND_PORT_CHB    ;POINT AT COMMAND PORT
066D B003                 416             MOV     AL, 3           ;POINT AT WR3
066F EE                   417             OUT     DX, AL
0670 B0C0                 418             MOV     AL, 0C0H        ;DISABLE RX
0672 EE                   419             OUT     DX, AL
0673 58                   420     RIB:    POP     AX              ;EITHER WAY, RESTORE REGISTERS
0674 5F                   421             POP     DI
0675 5A                   422             POP     DX
0676 CF                   423             IRET                    ;RETURN TO FOREGROUND
                          424
                          425     ;CHANNEL B ERROR SERVICE ROUTINE
                          426
0677 52                   427     ERRINB: PUSH    DX              ;SAVE REGISTERS
0678 50                   428             PUSH    AX
0679 E89400               429             CALL    EOI             ;SEND EOI COMMAND TO 8274
067C BA0600               430             MOV     DX, COMMAND_PORT_CHB
```

210311-31

```
MCS-86 MACRO ASSEMBLER   ASYNCB


LOC  OBJ              LINE    SOURCE

067F B001              431            MOV     AL, 1           ;POINT AT RR1
0681 EE                432            OUT     DX, AL
0682 EC                433            IN      AL, DX          ;READ RR1
0683 A22B02            434            MOV     ERROR_MSG_CHB, AL       ;SAVE IT IN ERROR FLAG
0686 B030              435            MOV     AL, 30H         ;SEND RESET ERROR COMMAND TO 8274
0688 EE                436            OUT     DX, AL
0689 58                437            POP     AX              ;RESTORE REGISTERS
068A 5A                438            POP     DX
068B CF                439            IRET                    ;RETURN TO FOREGROUND
                       440
                       441    ;CHANNEL A TRANSMIT DATA SERVICE ROUTINE
                       442
068C 52                443    XMTINA: PUSH    DX              ;SAVE REGISTERS
068D 57                444            PUSH    DI
068E 50                445            PUSH    AX
068F E87E00            446            CALL    EOI             ;SEND EOI COMMAND TO 8274
0692 FF062C02          447            INC     TX_POINTER_CHA  ;POINT TO NEXT CHARACTER
0696 FF0E2E02          448            DEC     TX_LENGTH_CHA   ;DEC LENGTH COUNTER
069A 740E              449            JE      XIA             ;TEST IF DONE
069C BA0000            450            MOV     DX, DATA_PORT_CHA       ;NOT DONE - GET NEXT CHARACTER
069F 8B3E2C02          451            MOV     DI, TX_POINTER_CHA
06A3 8A05              452            MOV     AL, [DI]        ;PUT CHARACTER IN AL
06A5 EE                453            OUT     DX, AL          ;OUTPUT IT TO 8274
06A6 58                454            POP     AX              ;RESTORE REGISTERS
06A7 5F                455            POP     DI
06A8 5A                456            POP     DX
06A9 CF                457            IRET                    ;RETURN TO FOREGROUND
06AA BA0200            458    XIA     MOV     DX, COMMAND_PORT_CHA    ;ALL CHARACTERS HAVE BEEN SEND
06AD B028              459            MOV     AL, 28H         ;RESET TRANSMITTER INTERRUPT PENDING
06AF EE                460            OUT     DX, AL
06B0 C606340201        461            MOV     TX_EMPTY_CHA, 1 ;DONE - SO SET TX EMPTY FLAG CHB
06B5 58                462            POP     AX              ;RESTORE REGISTERS
06B6 5F                463            POP     DI
06B7 5A                464            POP     DX
06B8 CF                465            IRET                    ;RETURN TO FOREGROUND
                       466
                       467    ;CHANNEL A STATUS CHANGE SERVICE ROUTINE
                       468
06B9 52                469    STAINA  PUSH    DX              ;SAVE REGISTERS
06BA 57                470            PUSH    DI
06BB 50                471            PUSH    AX
06BC E85100            472            CALL    EOI             ;SEND EOI COMMAND TO 8274
06BF BA0200            473            MOV     DX, COMMAND_PORT_CHA
06C2 EC                474            IN      AL, DX          ;READ RR0
06C3 A23602            475            MOV     STATUS_MSG_CHA, AL      ;PUT RR0 IN STATUS MESSAGE
06C6 B010              476            MOV     AL, 10H         ;SEND RESET STATUS INT COMMAND TO 8274
06C8 EE                477            OUT     DX, AL
06C9 58                478            POP     AX              ;RESTORE REGISTERS
06CA 5F                479            POP     DI
06CB 5A                480            POP     DX
06CC CF                481            IRET
                       482
                       483    ;CHANNEL A RECEIVED DATA SERVICE ROUTINE
                       484
06CD 52                485    RCVINA  PUSH    DX              ;SAVE REGISTERS
06CE 57                486            PUSH    DI
06CF 50                487            PUSH    AX
06D0 E83D00            488            CALL    EOI             ;SEND EOI COMMAND TO 8274
06D3 8B3E3002          489            MOV     DI, RX_POINTER_CHA      ;GET RX CHA BUFFER POINTER
06D7 BA0000            490            MOV     DX, DATA_PORT_CHA
```

210311-32

```
MCS-86 MACRO ASSEMBLER    ASYNCB


LOC  OBJ                  LINE     SOURCE

06DA EC                    491             IN      AL, DX          ;READ CHARACTER
06DB 8805                  492             MOV     [DI], AL        ;STORE IN BUFFER
06DD FF063802              493             INC     RX_POINTER_CHA  ;BUMP THE BUFFER POINTER
06E1 FF063202              494             INC     RX_COUNT_CHA    ;BUMP THE COUNTER
06E5 3C0D                  495             CMP     AL, CR_CHR      ;TEST IF LAST CHARACTER TO BE RECEIVED?
06E7 750E                  496             JNE     RIA
06E9 C606350201            497             MOV     RX_READY_CHA, 1 ;YES, SET READY FLAG
06EE BA0200                498             MOV     DX, COMMAND_PORT_CHA    ;POINT AT COMMAND PORT
06F1 B003                  499             MOV     AL, 3           ;POINT AT WR3
06F3 EE                    500             OUT     DX, AL
06F4 B0C0                  501             MOV     AL, 0C0H        ;DISABLE RX
06F6 EE                    502             OUT     DX, AL
06F7 58                    503     RIA.    POP     AX              ;EITHER WAY, RESTORE REGISTERS
06F8 5F                    504             POP     DI
06F9 5A                    505             POP     DX
06FA CF                    506             IRET                    ;RETURN TO FOREGROUND
                           507
                           508     ;CHANNEL A ERROR SERVICE ROUTINE
                           509
06FB 52                    510     ERRINA. PUSH    DX              ;SAVE REGISTERS
06FC 50                    511             PUSH    AX
06FD E81000                512             CALL    EOI             ;SEND EOI COMMAND TO 8274
0700 BA0200                513             MOV     DX, COMMAND_PORT_CHA
0703 B001                  514             MOV     AL, 1           ;POINT AT RR1
0705 EE                    515             OUT     DX, AL
0706 EC                    516             IN      AL, DX          ;READ RR1
0707 A23702                517             MOV     ERROR_MSG_CHA, AL       ;SAVE IT IN ERROR FLAG
070A B030                  518             MOV     AL, 30H         ;SEND RESET ERROR COMMAND TO 8274
070C EE                    519             OUT     DX, AL
070D 58                    520             POP     AX              ;RESTORE REGISTERS
070E 5A                    521             POP     DX
070F CF                    522             IRET                    ;RETURN TO FOREGROUND
                           523
                           524     ;END-OF-INTERRUPT ROUTINE - SENDS EOI COMMAND TO 8274
                           525     ;  THIS COMMAND MUST ALWAYS TO ISSUED ON CHANNEL A
                           526
0710 50                    527     EOI:    PUSH    AX              ;SAVE REGISTERS
0711 52                    528             PUSH    DX
0712 BA0200                529             MOV     DX, COMMAND_PORT_CHA    ;ALWAYS FOR CHANNEL A !!!
0715 B038                  530             MOV     AL, 38H
0717 EE                    531             OUT     DX, AL
0718 5A                    532             POP     DX
0719 58                    533             POP     AX
071A C3                    534             RET
                           535
                           536     ;END OF CODE ROUTINE
                           537
----                       538             ABC     ENDS
                           539             END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

210311-33

REFERENCES

1. 8274 Multiprotocol Serial Controller (MPSC) Data Sheet, Intel Corporation, California, 1980.
2. Basics of Data Communication, Electronics Book Series, McGraw-Hill, New York, 1976.
3. Telecommunications and the Computer, J. Martin, Prentice-Hall, New Jersey, 1976.
4. Technical Aspects of Data Communications, J. McNamara, DEC Press, Massachusetts, 1977.
5. Miscellaneous Data Communications Standards—EIA RS-232-C, EIA RS-422, EIA RS-423, EIA Standard Sales, Washington, D.C.

intel®

APPLICATION NOTE

AP-145

November 1986

Synchronous Communication with the 8274 Multiple Protocol Serial Controller

SIKANDAR NAQVI
APPLICATION ENGINEER

Order Number: 210403-001

INTRODUCTION

The INTEL 8274 is a Multi-Protocol Serial Controller, capable of handling both asynchronous and synchronous communication protocols. Its programmable features allow it to be configured in various operating modes, providing opimization to given data communication application.

This application note describes the features of the MPSC in Synchronous Communication applications only. It is strongly recommended that the reader read the 8274 Data Sheet and Application Note AP134 "Asynchronous Communication with the 8274 Multi-Protocol Serial Controller" before reading this Application Note. This Application note assumes that the reader is familiar with the basic structure of the MPSC, in terms of pin description, Read/Write registers and asynchronous communication with the 8274. Appendix A contains the software listings of the Application Example and Appendix B shows the MPSC Read/Write Registers for quick reference.

The first section of this application note presents an overview of the various synchronous protocols. The second section discusses the block diagram description of the MPSC. This is followed by the description of MPSC interrupt structure and mode of operation in the third and fourth sections. The fifth section describes a hardware/software example, using the INTEL single board computer iSBC88/45 as the hardware vehicle. The sixth section consists of some specialized applications of the MPSC. Finally, in section seven, some useful programming hints are summarized.

SYNCHRONOUS PROTOCOL OVERVIEW

This section presents an overview of various synchronous protocols. The contents of this section are fairly tutorial and may be skipped by the more knowledgeable reader.

Bit Oriented Protocols Overview

Bit oriented protocols have been defined to manage the flow of information on data communication links. One of the most widely known protocols is the one defined by the International Standards Organization: HDLC (High Level Data Link Control). The American Standards Association's protocol, ADCCP is similar to HDLC. CCITT Recommendation X.25 layer 2 is also an acceptable version of HDLC. Finally, IBM's SDLC (Synchronous Data Link Control) is also a subset of the HDLC.

In this section, we will concentrate most of our discussion on HDLC. Figure 1 shows a basic HDLC frame format.

A frame consists of five basic fields: Flag, Address, Control, Data and Error Detection. A frame is bounded by flags—opening and closing flags. An address field is 8 bits wide, extendable to 2 or more bytes. The control field is also 8 bits wide, extendable to two bytes. The data field or information field may be any number of bits. The data field may or may not be on an 8-bit boundary. A powerful error detection code called Frame Check Sequence contains the calculated CRC (Cycle Redundancy Code) for all the bits between the flags.

ZERO BIT INSERTION

The flag has a unique binary bit pattern: 7E HEX. To eliminate the possibility of the data field containing a 7E HEX pattern, a bit stuffing technique called Zero Bit Insertion is used. This technique specifies that during transmission, a binary 0 be inserted by the transmitter after any succession of five contiguous binary 1's. This will ensure that no pattern of 0 1 1 1 1 1 1 0 is ever transmitted between flags. On the receiving side, after receiving the flag, the receiver hardware automatically deletes any 0 following five consecutive 1's. The 8274 performs zero bit insertion and deletion automatically in the SDLC/HDLC mode. The zero-bit stuffing ensures periodic transitions in the data stream. These transitions are necessary for a phase lock circuit, which may be used at the receiver end to generate a receive clock which is in phase to the received data. The inserted and deleted 0's are not included in the CRC checking. The *address* field is used to address a given secondary station. The *control* field contains the link-level control information which includes implied acknowledgement, supervisory commands and responses, etc. A more detailed discussion of higher level protocol functions is beyond the scope of this application note. Interested readers may refer to the references at the end of this application note.

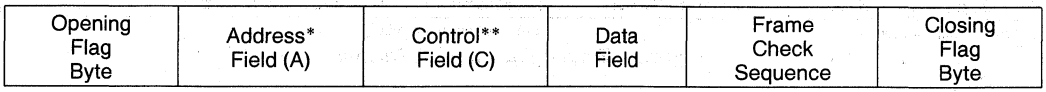


Figure 1. HDLC/SDLC Frame Format

*Extendable to 2 or More Bytes.
**Extendable to 2 Bytes.

The *data field* may be of any length and content in HDLC. Note that SDLC specifies that data field be a multiple of bytes only. In data communications, it is generally desirable to transmit data which may be of any content. This requires that data field should not contain characters which are defined to assist the transmission protocol (like opening flag 7EH in HDLC/SDLC communications). This property is referred to as "data transparency". In HDLC/SDLC, this code transparency is made possible by Zero Bit Insertion discussed earlier and the bit oriented nature of the protocol.

The last field is the FCS (Frame Check Sequence). The FCS uses the error detecting techniques called Cyclic Redundancy Check. In SDLC/HDLC, the CCITT-CRC must be used.

NON-RETURN TO ZERO INVERTED (NRZI)

NRZI is a method of clock and data encoding that is well suited to the HDLC protocol. It allows HDLC protocols to be used with low cost asynchronous modems. NRZI coding is done at the transmitter to enable clock recovery from the data at the receiver terminal by using standard digital phase locked loop techniques. NRZI coding specifies that the signal condition does not change for transmitting a 1, while a 0 causes a change of state. NRZI coding ensures that an active data line will have transition at least every 5-bit times (recall Zero Bit Insertion), while contiguous 0's will cause a change of state. Thus, ZBI and NRZI encoding makes it possible for a phase lock circuit at the receiver end to derive a receive clock (from received data) which is synchronized to the received data and at the same time ensure data transparency.

Byte Synchronous Communication

As the name implies, Byte Synchronous Communication is a synchronous communication protocol which means that the transmitting station is synchronized to the receiving station through the recognition of a special sync character or characters. Two examples of Byte Synchronous protocol are the IBM Bisync and Monosync. Bisync has two starting sync characters per message while monosync has only one sync character. For the sake of brevity, we will only discuss Bisync here. All the discussion is valid for Monosync also. Any exceptions will be noted. Figure 2 shows a typical Bisync message format.

The Bisync protocol is defined for half duplex communication between two or more stations over point to point or multipoint communication lines. Special characters control link access, transmission of data and termination of transmission operations for the system. A detailed discussion of these special control characters (SYN, ENQ, STX, ITB, ETB, ETX, DLE, SOH, ACK0, ACK1, WACK, NAK and EOT, etc) is beyond the scope of this Application Note. Readers interested in more detailed discussion are directed to the references listed at the end of this Application Note.

As shown in Figure 2, each message is preceded by two sync characters. Since the sync characters are defined at the beginning of the message only, the transmitter must insert fill characters (sync) in order to maintain synchronization with the receiver when no data is being transmitted.

TRANSPARENT TRANSMISSION

Bisync protocol requires special control characters to maintain the communication link over the line. If the data is EBCDIC encoded, then transparency is ensured by the fact that the field will not contain any of the bisync control characters. However, if data does not conform to standard character encoding techniques, transparency in bisync is achieved by inserting a special character DLE (Data Link Escape) before and after a string of characters which are to be transmitted transparently. This ensures that any data characters which match any of the special characters are not confused for special characters. An example of a transparent block is shown in Figure 3.

In a transparent mode, it is required that the CRC (BCC) is not performed on special characters. Later on, we will show how the 8274 can be used to achieve transparent transmission in Bisync mode.

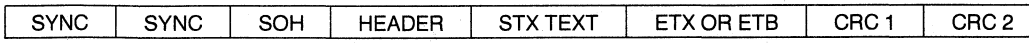
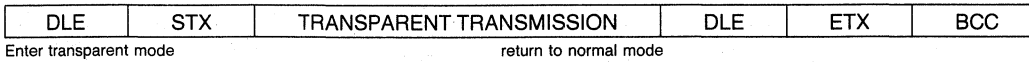


Figure 2. Bisync Message Format



Enter transparent mode return to normal mode

Figure 3. Bisync Transparent Format

BLOCK DIAGRAM

This section discusses the block diagram view of the 8274. The CPU interface and serial interface is discussed separately. This will be followed by a hardware example in the fifth section, which will show how to interface the 8274 with the Intel CPU 8088. The 8274 block diagram is shown in Figure 4.

CPU Interface

The CPU interface to the system interface logic block utilizes the A0, A1, $\overline{CS}$, $\overline{RD}$ and $\overline{WR}$ inputs to communicate with the internal registers of the 8274. Figure 5 shows the address of the internal registers. The DMA interface is achieved by utilizing DMA request lines for each channel: $TxDRQ_A$, $TxDRQ_B$, $RxDRQ_A$, $RxDRQ_B$. Note that $TxDRQ_B$ and $RxDRQ_B$ become $\overline{IPO}$ and $\overline{IPI}$ respectively in non-DMA mode. $\overline{IPI}$ is the Interrupt Priority Input and $\overline{IPO}$ is the Interrupt Priority Output. These two pins can be used for connecting multiple MPSCs in a daisy chain. If the Wait Mode is programmed, then $TxRDQ_A$ and $RxDRQ_A$ pins become RDY_B and RDY_A pins. These pins can be wire-OR'ed and are usually hooked up to the CPU RDY line to synchronize the CPU for block transfers. The $\overline{INT}$ pin is activated whenever the MPSC requires CPU attention. The $\overline{INTA}$ may be used to utilize the powerful vectored mode feature of the 8274. Detailed discussion on these subjects will be done later in this Application Note. The $\overline{RESET}$ pin may be used for hardware reset while the clock is required to click the internal logic on the MPSC.

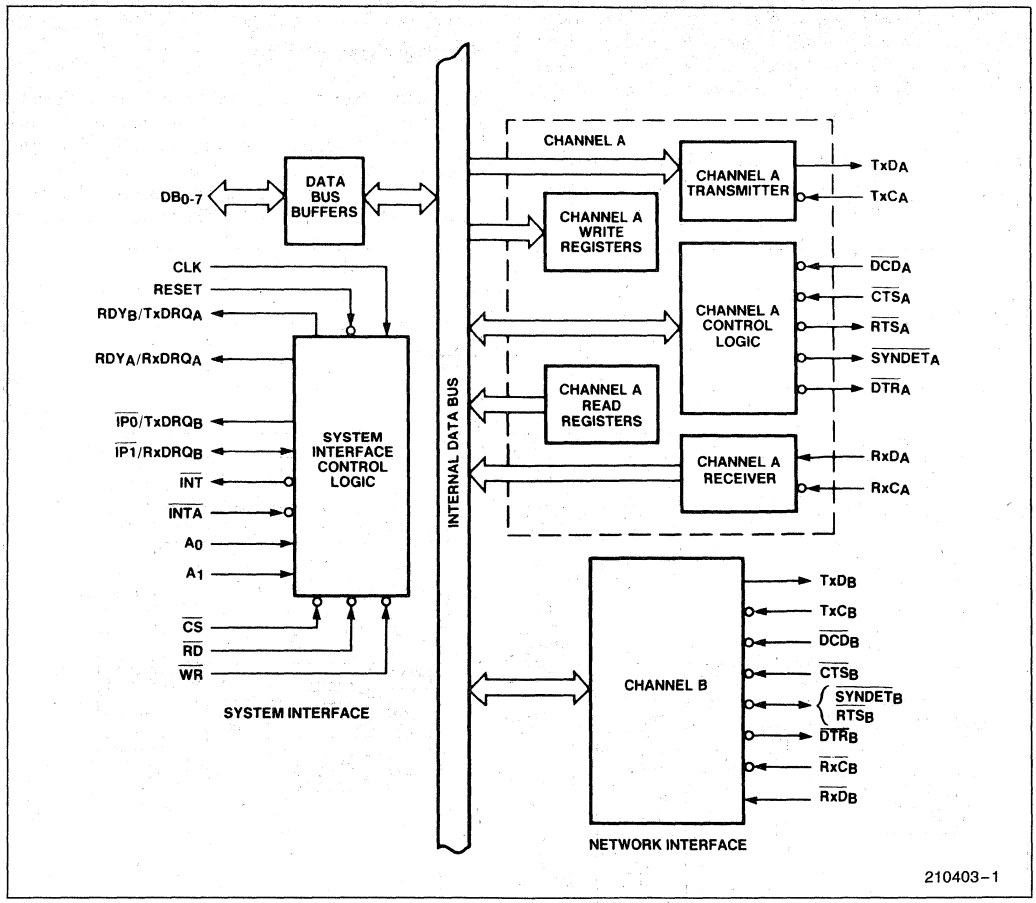


Figure 4. 8274 Block Diagram

$\overline{CS}$	A1	A0	Read Operation	Write Operation
0	0	0	CHA DATA READ	CHA DATA WRITE
0	1	0	CHA STATUS REGISTER (RR0,RR1)	CHA COMMAND/PARAMETER (WR0–WR7)
0	0	1	CHB DATA READ	CHB DATA WRITE
0	1	1	CHB STATUS REGISTER (RR0,RR1,RR2)	CHB COMMAND/PARAMETER (WR0–WR7)
1	X	X	HIGH Z	HIGH Z

Figure 5. Bus Interface

Serial Interface

On the serial side, there are two completely independent channels: Channel A and Channel B. Each channel consists of a transmitter block, receiver block and a set of read/write registers which are used to initialize the device. In addition, a control logic block provides the modem interface pins. Channel B serial interface logic is a mirror image of Channel A serial interface logic, except for one exception: there is only one pin for RTS_B and $SYNDET_B$.

A a given time, this pin is either RTS_B or $SYNDET_B$. This mode is programmable through one of the internal registers on the MPSC.

Transmit and Receive Data Path

Figure 6 shows a block diagram for transmit and receive data path. Without describing each block on the diagram, a brief discussion of the block diagram will be presented here.

TRANSMIT DATA PATH

The transmit data is transferred to the twenty-bit serial shift register. The twenty bits are needed to store two bytes of sync characters in bisync mode. The last three bits of the shift register are used to indicate to the internal control logic that the current data byte has been shifted out of the shift register. The transmit data in the

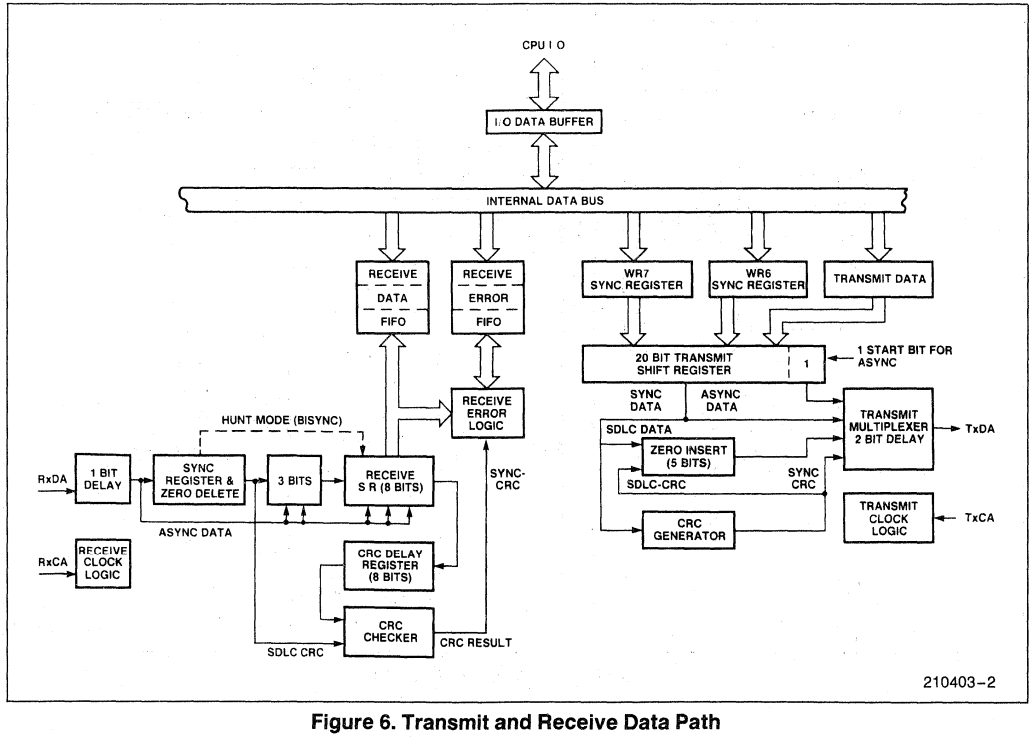


Figure 6. Transmit and Receive Data Path

transmit shift register is shifted out through a two bit delay onto the TxData line. This two bit delay is used to synchronize the internal shift clock with the external transmit clock. The data in the shift register is also presented to zero bit insertion logic which inserts a zero after sensing five contiguous ones in the data stream. In parallel to all this activity, the CRC-generator is computing CRC on the transmitted data and appends the frame with CRC bytes at the end of the data transmission.

RECEIVE DATA PATH

The received data is passed through a one bit delay before it is presented for flag/sync comparison. In bi-sync mode, after the synchronization is achieved, the incoming data bypasses the sync register and enters directly into the three bit buffer on its way to receive shift register. In SDLC mode, the incoming data always passes through the sync register where the data pattern is continuously monitored for contiguous ones for the

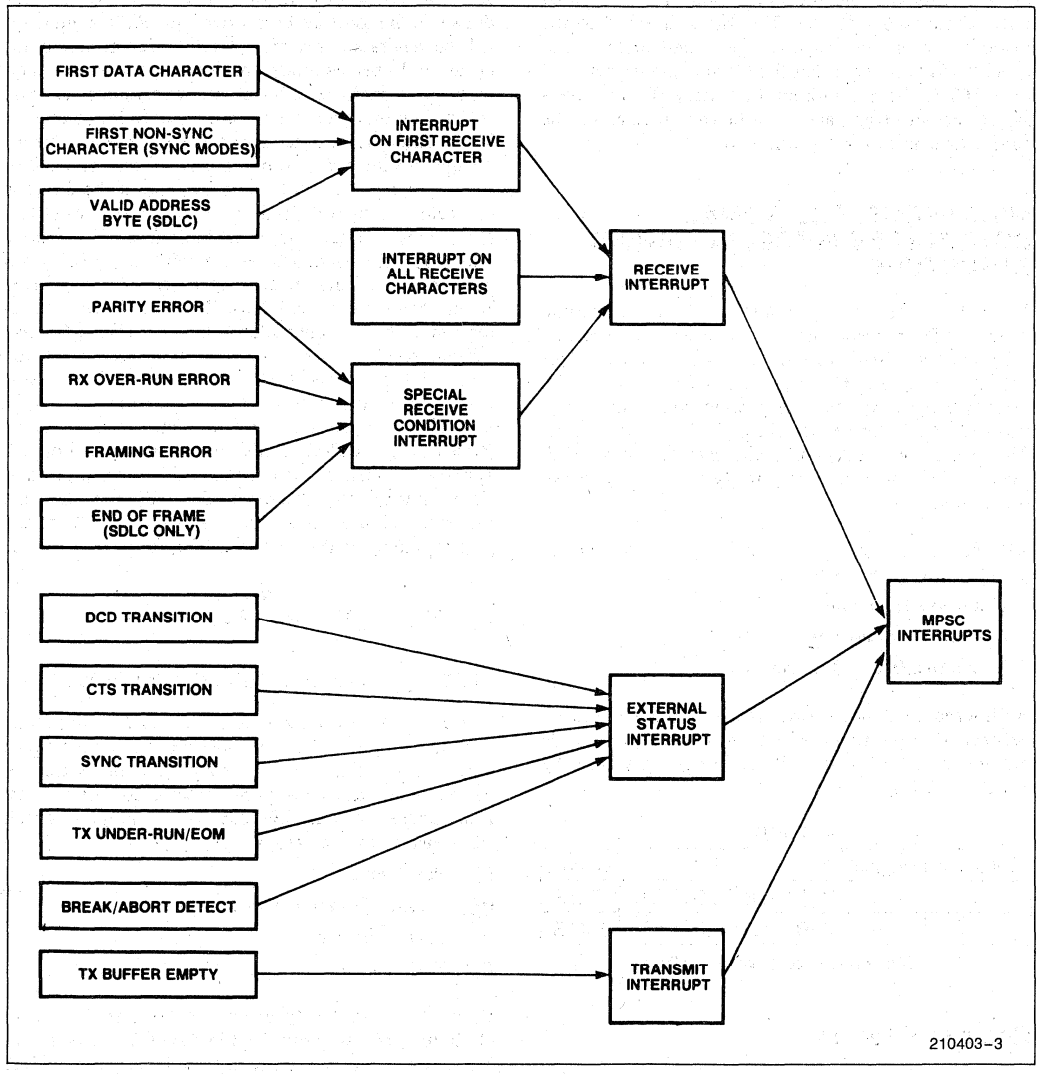


Figure 7. MPSC Interrupt Structure

zero deletion logic. The data then enters the three bit buffer and the receive shift register. From the receive shift register, the data is transferred to the three byte deep FIFO. The data is transferred to the top of the FIFO at the chip clock rate (not the receiver clock). It takes three chip clock/periods to transfer data from the serial shift register to the top of the FIFO. The three bit deep Receive Error FIFO shifts any error condition which may have occurred during a frame reception. While all this is happening, the CRC checker is checking the CRC on the incoming data. The computed CRC is checked with the CRC bytes attached to the incoming frame and an error generated under a no-check condition. Note that the bisync data is presented to the CRC checker with an 8-bit delay. This is necessary to achieve transparency in bisync mode as will be shown later in this Application Note.

MULTI-PROTOCOL SERIAL CONTROLLER (MPSC) INTERRUPT STRUCTURE

The MPSC offers a very powerful interrupt structure, which helps in responding to an interrupt condition very quickly. There are multiple sources of interrupts within the MPSC. However, the MPSC resolves the priority between various interrupting sources and interrupts the CPU for service through the interrupt line. This section presents a comprehensive discussion of all the 8247 interrupts and the priority resolution between these interrupts.

All the sources of interrupts on the 8274 can be grouped into three distinct categories. (See Figure 7.)

1. Receive Interrupts
2. Transmit Interrupts
3. External/Status Interrupts.

An internal interrupt priority structure sets the priority between the interrupts. There are two programmable options available on the MPSC. The priority is set by WR2A, D2 (Figure 8).

PRIORITY

WR2A:D2	Highest					Lowest
0	RxA	TxA	RxB	TxB	EXTA	EXTB
1	RxA	RxB	TxA	TxB	EXTA	EXTB

Figure 8. Interrupt Priority

Receive Interrupt

All receive interrupts may be categorized into two distinct groups: Receive Interrupt on Receive Character and Special Receive Condition Interrupts.

RECEIVE INTERRUPT ON RECEIVE CHARACTER

A receive interrupt is generated when a character is received by the MPSC. However, as will be discussed later, this is a programmable feature on the MPSC. A Rx character available interrupt is generated by the MPSC after the receive character has been assembled by the MPSC. It may be noted that in DMA transfer mode too, a receive interrupt on the first receive character should be programmed. In SDLC mode, if address search mode has been programmed, this interrupt will be generated only after a valid address match has occurred. In bisync mode, this interrupt is generated on receipt of a character after at least two valid sync characters. In monosync mode, a character followed after at least a single valid sync character will generate this interrupt. An interrupt on first receive character signifies the beginning of a valid frame. An end of the frame is characterized by an "End of Frame" Interrupt (RR1: D7).* This bit (RR1:D7) is set in SDLC/HDLC mode only and signifies that a valid ending flag (7EH) has been received. This bit gets reset either by an "Error Reset" command (WR0: D5D4D3 = 110) or upon reception of the first character of the next frame. In multiframe reception, on receiving the interrupt at the "End of Frame" the CPU may issue an Error Reset command which will reset the interrupt. In DMA mode, the interrupt on first receive character is accompanied by a RxDRQ (Receiver DMA request) on the appropriate channel. At the end of the frame, an End of Frame interrupt is generated. The CPU may use this interrupt to jump into a routine which may redefine the receive buffer for the next incoming frame.

*NOTE:
RR1:D7 is bit D7 in Read Register 1.

SPECIAL RECEIVE CONDITION INTERRUPTS

So far, we have assumed that the reception is error free. But this is not 'typical' in most real life applications. Any error condition during a frame reception generates yet another interrupt—special receive condition interrupt. There are four different error conditions which can generate this interrupt.

(i) Parity error
(ii) Receive Overrun error
(iii) Framing error
(iv) End of Frame

(i) Parity error: Parity error is encountered in asynchronous (start-stop bits) and in bisync/monosync protocols. Both odd or even parity can be programmed. A parity error in a received byte will generate a special receive condition interrupt and sets bit 4 in RR1.

(ii) Receive Overrun error: If the CPU or the DMA controller (in DMA mode) fails to read a received character within three byte times after the received character interrupt (or DMA request) was generated, the receiver buffer will overflow and this will generate a special receive condition interrupt and sets bit 5 in RR1.

(iii) Framing error: In asynchronous mode, a framing error will generate a special receive interrupt and set bit D6 in RR1. This bit is not latched and is updated on the next received character.

(iv) End of frame: This interrupt is encountered in SDLC/HDLC mode only. When the MPSC receives the closing flag, it generates the special receive condition interrupt and sets bit D7 in RR1.

All the special receive condition interrupts may be reset by issuing an Error Reset Command.

CRC Error: In SDLC/HDLC and synchronous modes, a CRC error is indicated by bit D6 in RR1. When used to check CRC error, this bit is normally set until a correct CRC match is obtained which resets this bit. After receiving a frame, the CPU must read this bit (RR1:D6) to determine if a valid CRC check had occurred. It may be noted that a CRC error does not generate an interrupt.

It may also be pointed out that in SDLC/HDLC mode, receive DMA requests are disabled by a special receive condition and can only be re-enabled by issuing an Error Reset Command.

Transmit Interrupt

A transmit buffer empty generates a transmit interrupt. This has been discussed earlier under "Transmit in Interrupt Mode" and it would be sufficient to note here that a transmit buffer empty interrupt is generated only when the transmit buffer gets empty—assuming it had a data character loaded into it earlier. This is why on starting a frame transmission, the first data character is loaded by the CPU without a transmit empty interrupt (or DMA request in DMA mode). After this character is loaded into the serial shift register, the buffer becomes empty, and an interrupt (or DMA request) is generated. This interrupt is reset by a "Reset Tx Interrupt/DMA Pending" command (WR0: D5 D4 D3 = 101).

External/Status Interrupt

Continuing our discussion on transmit interrupt, if the transmit buffer is empty and the transmit serial shift register also becomes empty (due to the data character shifted out of the MPSC), a transmit under-run interrupt will be generated. This interrupt may be reset by "Reset External/Status Interrupt" command (WR0: D5 D4 D3 = 101).

The External Status Interrupt can be caused by five different conditions:

(i) CD Transition

(ii) CTS Transition

(iii) Sync/Hunt Transition

(iv) Tx under-run/EOM condition

(v) Break/Abort Detection.

CD, CTS TRANSITION

Any transition on these inputs on the serial interface will generate an External/Status interrupt and set the corresponding bits in status register RR0. This interrupt will also be generated in DMA as well as in Wait Mode. In order to find out the state of the $\overline{CTS}$ or $\overline{CD}$ pins *before* the transition had occurred, RR0 must be read before issuing a Reset External/Status Command through WR0. A read of RR0 after the Reset External/Status Command will give the condition of $\overline{CTS}$ or $\overline{CD}$ pins *after* the transition had occurred. Note that bit D5 in RR0 gives the complement of the state of $\overline{CTS}$ pin while D3 in RR0 reflects the actual state of the $\overline{CD}$ pin.

SYNC HUNT TRANSITION

Any transition of the $\overline{SYNDET}$ input generates an interrupt. However, sync input has different functions in different modes and we shall discuss them individually.

SDLC Mode

In SDLC mode, the $\overline{SYNDET}$ pin is an output. Status register RR1, D4 contains the state of the $\overline{SYNDET}$ pin. The Enter Hunt Mode initially sets this bit in R0. An opening flag in a received SDLC frame resets this bit and generates an external status interrupt. Every time the receiver is enabled or the Enter Hunt Code Command is issued, an external status interrupt will be generated on receiving a valid flag followed by a valid address/data character. This interrupt may be reset by the "Reset External/Status Interrupt" command.

External SYNC Mode

The MPSC can be programmed into External Sync Mode by setting WR4, D5 D4 = 11. The $\overline{SYNDET}$ pin is an input in this case and must be held high until an external character synchronization is established. However, the External Sync mode is enabled by the Enter Hunt Mode control bit (WR3: D4). A high at the $\overline{SYNDET}$ pin holds the Sync/Hunt bit (RR0,D4) in the reset state. When external synchronization is established, $\overline{SYNDET}$ must be driven low on second rising

edge of RxC after the rising edge of RxC on which the last bit of sync character was received. This high to low transition sets the Sync/Hunt bit and generates an external/status interrupt, which must be reset by the Reset External/Status command. If the $\overline{\text{SYNDET}}$ input goes high again, another External Status Interrupt is generated, which may be cleared by Reset External/Status command.

Mono-Sync/Bisync Mode

$\overline{\text{SYNDET}}$ pin acts as an output in this case. The Enter Hunt Mode sets the Sync/Hunt bit in R0. Sync/Hunt bit is reset when the MPSC achieves character synchronization. This high to low transition will generate an external status interrupt. The $\overline{\text{SYNDET}}$ pin goes active every time a sync pattern is detected in the data stream. Once again, the external status interrupt may be reset by the Reset External/Status command.

Tx UNDER-RUN/END OF MESSAGE (EOM)

The transmitter logic includes a transmit buffer and a transmit serial shift register. The CPU loads the character into the transmit buffer which is transferred into the transmit shift register to be shifted out of the MPSC. If the transmit buffer gets empty, a transmit buffer empty interrupt is generated (as discussed earlier). However, if the transmit buffer gets empty *and* the serial shift register gets empty, a transmit under-run condition will be created. This generates an External Status Interrupt and the interrupt can be cleared by the Reset External Status command. The status register RR0, D6 bit is set when the transmitter under-runs. This bit plays an important role in controlling a transmit operation, as will be discussed later in this application note.

BREAK/ABORT DETECTION

In asynchronous mode, bit D7 in RR0 is set when a break condition is detected on the receive data line. This also generates an External/Status interrupt which may be reset by issuing a Reset External/Status Interrupt command to the MPSC. Bit D7 in RR0 is reset when the break condition is terminated on the receive data line and this causes another External/Status interrupt to ge generated. Again, a Reset External/Status Interrupt command will reset this interrupt and will enable the break detection logic to look for the next break sequence.

In SDLC Receive Mode, an Abort sequence (seven or more 1's) detection on the receive data line will generate an External/Status interrupt and set RR0,D7. A Reset External/Status command will clear this interrupt. However, a termination of the Abort sequence will generate another interrupt and set RR0,D7 again. Once again, it may be cleared by issuing Reset External/Status Command.

This concludes our discussion on External Status Interrupts.

Interrupt Priority Resolution

The internal interrupt priority between various interrupt sources is resolved by an internal priority logic circuit, according to the priority set in WR2A. We will now discuss the interrupt timings during the priority resolution. Figures 9 and 10 show the timing diagrams for vectored and non-vectored modes.

VECTORED MODE

We shall assume that the MPSC accepted an internal request for an interrupt by activating the internal INT signal. This leads to generating an external interrupt signal on the $\overline{\text{INT}}$ pin. The CPU responds with an interrupt acknowledge (INTA) sequence. The leading edge of the first $\overline{\text{INTA}}$ pulse sets an internal interrupt acknowledge signal (we will call it Internal INTA). Internal INTA is reset by the high going edge of the third $\overline{\text{INTA}}$ pulse. The MPSC will not accept any internal requests for an interrupt during the period when Internal $\overline{\text{INTA}}$ is active (high). The MPSC resolves the priority during various existing internal interrupt requests during the Interrupt Request Priority Resolve Time, which is defined as the time between the leading edge of the first INTA and the leading edge of the second INTA from the CPU. Once the internal priorities have been resolved, an internal Interrupt-in-service Latch is set. The external $\overline{\text{INT}}$ is also deactivated when the Interrupt-in-Service Latch is set.

The lower priority interrupt requests are not accepted internally until an EOI (WR0: D5 D4 D3 = 111 Ch. A only) command is issued by the CPU. The EOI command enables the lower priority interrupts. However, a higher priority interrupt request will still be accepted (except during the period when internal $\overline{\text{INTA}}$ is active) even though the Internal-in-Service Latch is set.

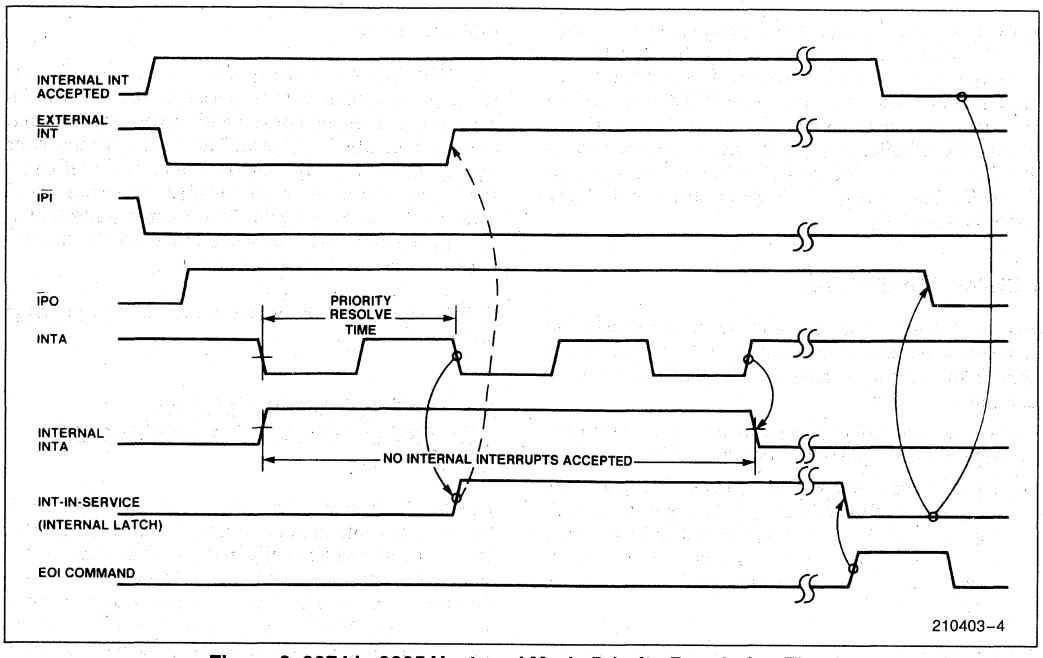


Figure 9. 8274 in 8085 Vectored Mode Priority Resolution Time

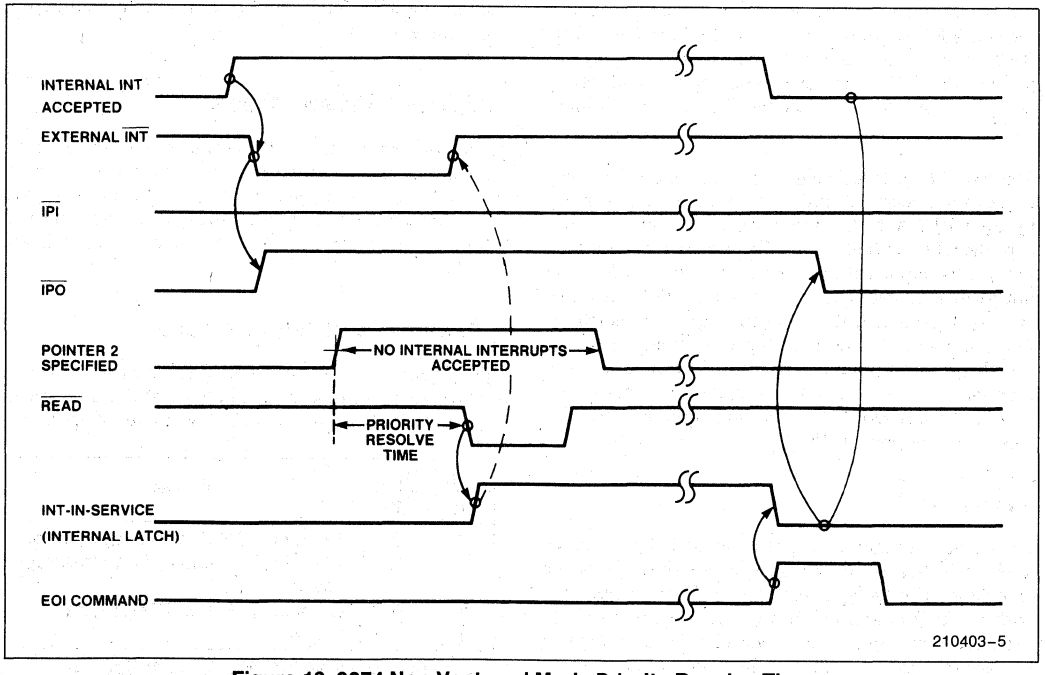


Figure 10. 8274 Non Vectored Mode Priority Resolve Time

This higher priority request will generate another external $\overline{INT}$ and will have to be handled by the CPU according to how the CPU is set up. If the CPU is set up to respond to this interrupt, a new INTA cycle will be repeated as discussed earlier. It may also be noted that a transmitter buffer empty and receive character available interrupts are cleared by loading a character into the MPSC and by reading the character received by the MPSC respectively.

NON-VECTORED MODE

Figure 10 shows the timing of interrupt sequence in non-vectored mode. The explanation of non-vectored is similar to the vector mode, except for the following exceptions.

— No internal priority requests are accepted during the time when pointer 2 for Channel B is specified.

— The interrupt request priority resolution time is the time between the leading edge of pointer 2 and leading edge of RD active. It may be pointed out that in non-vectored mode, it is assumed that the status affects vector mode is used to expedite interrupt response.

On getting an interrupt in non-vectored mode, the CPU must read status register RR2 to find out the cause of the interrupt. In order to do so, first a pointer to status register RR2 is specified and then the status read from RR2. It may be noted here that after specifying the pointer, the CPU must read status register RR2 otherwise, no new interrupt requests will be accepted internally.

Just like the vectored mode, no lower internal priority requests are accepted until an EOI command is issued by the CPU. A higher priority request can still interrupt the CPU (except during the priority request inhibit time). It is important to note here that if the CPU does not perform a read operation after specifying the pointer 2 for Channel B, the interrupt request accepted before the pointer 2 was activated will remain valid and no other request (high or low priority) will be accepted internally. In order to complete a correct priority resolution, it is advised that a read operation be done after specifying the pointer 2B.

IPI and IPO

So far, we have ignored the IPI and IPO signals shown in Figures 9 and 10. We may recall that IPI is the Interrupt-Priority-Input to the MPSC. In conjunction with the IPO (Interrupt Priority Output), it is used to daisy chain multiple MPSCs. MPSC daisy chaining will be discussed in detail later in this application note.

EOI Command

The EOI command as explained earlier, enables the lower priority interrupts by resetting the internal In-Service-Latch, which consequently resets the IPO output to a low state. See Figures 9 and 10 for details. Note that before issuing any EOI command, the internal interrupting source must be satisfied otherwise, same source will interrupt again. The Internal Interrupt is the signal which gets reset when the internal interrupting source is satisfied (see Figure 9).

This concludes our discussion on the MPSC Interrupt Structure.

MULTI-PROTOCOL SERIAL CONTROLLER (MPSC) MODES OF OPERATION

The MPSC provides two fully independent channels that may be configured in various modes of operations. Each channel can be configured into full duplex mode and may operate in a mode or protocol different from the other channel. This feature will be very efficient in an application which requires two data link channels operating in different protocols and possibly at different data rates. This section presents a detailed discussion on all the 8274 modes and shows how to configure it into these modes.

Interrupt Driven Mode

In the interrupt mode, all the transmitter and receiver operations are reported to the processor through interrupts. Interrupts are generated by the MPSC whenever it requires service. In the following discussion, we will discuss how to transmit and receive in interrupt driven mode.

TRANSMIT IN INTERRUPT MODE

The MPSC can be configured into interrupt mode by appropriately setting the bits in WR2 A (Write Register 2, Channel A). Figure 11 shows the modes of operation.

WR2A		Mode
D1	D0	
0	0	CH A and CH B in Interrupt Mode
0	1	CH A in DMA and CH B in Interrupt Mode
1	0	CH A and CH B in DMA Mode
1	1	Illegal

Figure 11. MPSC Mode Selection for Channel A and Channel B

We will limit our discussion to SDLC transmit and receive only. However, exceptions for other synchronous protocols will be pointed out. To initiate a frame transmission, the first data character must be loaded from the CPU, in all cases. (DMA Mode too, as you will notice later in this application note). Note that in SDLC mode, this first data character may be the address of the station addressed by the MPSC. The transmit buffer consists of a transmit buffer and a serial shift register. When the character is transferred from the buffer into the serial shift regiser, an interrupt due to transmit buffer empty is generated. The CPU has one byte time to service this interrupt and load another character into the transmitter buffer. The MPSC will generate an interrupt due to transmit buffer underrun condition if the CPU does not service the Transmit Buffer Empty Interrupt within one byte time.

This process will continue until the CPU is out of any more data characters to be sent. At this point, the CPU does not respond to the interrupt with a character but simply issues a Reset Tx INT/DMA pending command (WR0: D5 D4 D3 = 101). The MPSC will ultimately underrun, which simply means that both the transmit buffer and transmit shift registers are empty. At this point, flag character (7EH) or CRC byte is loaded into the transmit shift register. This sets the transmit underrun bit in RR0 and generates "Transmit Underrun/EOM" interrupt (RR0: D6 = 1).

You will recall that an SDLC frame has two CRC bytes after the data field. 8274 generates the CRC on all the data that is loaded from the CPU. During initialization, there is a choice of selecting a CRC-16 or CCITT-CRC (WR5: D2). In SDLC/HDLC operation, CCITT-CRC must be selected. We will now see how the CRC gets inserted at the end of the data field. Here we have a choice of having the CRC attached to the data field or sending the frame without the CRC bytes. During transmission, a "Reset Tx Underrun/EOM Latch" command (WR0: D7 D6 = 11) will ensure that at the end of the frame when the transmitter underruns, CRC bytes will be automatically inserted at the end of the data field. If the "Reset Tx Underrun/EOM Latch" command was not issued during the transmission of data characters, no CRC would be inserted and the MPSC will transmit flags (7EH) instead.

However, in case of CRC transmission, the CRC transmission sets the Tx Underrun/EOM bit and generates a Transmitter Underrun/EOM Interrupt as discussed earlier. This will have to be reset in the next frame to ensure CRC insertion in the next frame. It is recommended that Tx Underrun/EOM latch be reset very early in the transmission mode, preferably after loading the first character. It may be noted here that Tx Underrun EOM latch cannot be reset if there is no data in the transmit buffer. This means that at least one character has to be loaded into the MPSC before a "Reset Transmit Underrun/EOM Latch" command will be accepted by the MPSC.

When the transmitter is underrun, an interrupt is generated. This interrupt is generated at the beginning of the CRC transmission, thus giving the user enough time (minimum 22 transmit clock cycles) to issue an Abort command (WR0: D5 D4 D3 = 0 0 1) in case if the transmitted data had an error. The Abort Command will ensure that the MPSC transmits at least eight 1's but less than fourteen 1's before the line reverts to continuous flags. The receiver will scratch this frame because of bad CRC.

However, assuming the transmission was good (no Abort Command issued), after the CRC bytes have been transmitted, closing flag (7EH) is loaded into the transmit buffer. When the flag (7EH) byte is transferred to the serial shift register, a transmit buffer empty interrupt is generated. If another frame has to be transmitted, a new data character has to be loaded into the transmit buffer and the complete transmit sequence repeated. If no more frames are to be transmitted, a "Reset Transmit INT/DMA Pending" command (WR0: D5 D4 D3 = 101) will reset the transmit buffer empty interrupt.

For character oriented protocols (Bisync, Monosync), the same discussion is valid, except that during transmit underrun condition and transmit underrun/EOM bit in set state, instead of flags, filler sync characters are transmitted.

CRC Generation

The transmit CRC enable bit (WR5: D0) must be set before loading any data into the MPSC. The CRC generator must be reset to all 1's at the beginning of each frame before CRC computation has begun. The CRC computation starts on the first data character loaded from the CPU and continues until the last data character. The CRC generated is inverted before it is sent on the Tx Data line.

Transmit Termination

A successful transmission can be terminated by issuing a "Reset Transmit Interrupt/DMA Pending" command, as discussed earlier. However, the transmitter may be disabled any time during the transmission and the results will be as shown in Figure 12.

RECEIVE IN INTERRUPT MODE

The receiver has to be initialized into the appropriate receive mode (see sample program later in this application note). The receiver must be programmed into Hunt Mode (WR3: D4) before it is enabled (WR3: D0). The receiver will remain in the Hunt Mode until a flag (or sync character) is received. While in the SDLC/Bisync/Monosync mode, the receiver does not enter the Hunt Mode unless the Hunt bit (WR3, D4) is set again or the receiver is enabled again.

SDLC Address byte is stored in WR6. A global address (FFH) has been hardwired on the MPSC. In address search mode (WR3: D2 = 1), any frame with address matching with the address in WR6 will be received by the MPSC. Frames with global address (FFH) will also be received, irrespective of the condition of address search mode bit (WR3: D2). In general receive mode (WR3: D2 = 0), all frames will be received.

Transmitter Disabled during	Result
1. Data Transmission	Tx Data will send idle characters* which will be zero inserted.
2. CRC Transmission	16 bit transmission, corresponding to 16 bits of CRC will be completed. However, flag bits will be substituted in the CRC field.
3. Immediately after issuing ABORT command.	Abort will still be transmitted—output will be in the mark state.

Figure 12. Transmitter Disabled During Transmission

*NOTE:
Idle characters are defined as a string of 15 or more contiguous ones.

Since the MPSC only recognizes single byte address field, extended address recognition will have to be done by the CPU on the data passed on by the MPSC. If the first address byte is checked by the MPSC, and the CPU determines that the second address byte does not have the correct address field, it must set the Hunt Mode (WR3: D2 = 1) and the MPSC will start searching for a new address byte preceded by a flag.

Programmable Interrupts

The receiver may be programmed into any one of the four modes. See Figure 13 for details.

WR1, CHA		Rx Interrupt Mode
D4	D3	
0	0	Rx INT/DMA disable
0	1	Rx INT on first character
1	0	INT on all Rx characters (Parity affects vector)
1	1	INT on all Rx characters (Parity does not affect vector)

Figure 13. Receiver Interrupt Modes

All receiver interrupts can be disabled by WR1: D4 D3 = 00. Receiver interrupt on first character is normally used to start a DMA transfer or a block transfer sequence using WAIT to synchronize the data transfer to received or transmitted data.

External Status Interrupts

Any change in $\overline{CD}$ input or Abort detection in the received data, will generate an interrupt if External Status Interrupt was enabled (WR1: D0).

Special Receive Conditions

The receiver buffer is quadruply buffered. If the CPU fails to respond to "receive character" available interrupt within a period of three byte times (received bytes), the receiver buffer will overflow and generate an interrupt. Finally, at the end of the received frame, an interrupt will be generated when a valid ending flag has been detected.

Receive Character Length

The receive character length (6, 7 or 8 bits/character) may be changed during reception. However, to ensure that the change is effective on the next received character, this must be done fast enough such that the bits specified for the next character have not been assembled.

CRC Checking

The opening flag in the frame resets the receive CRC generator and any field between the opening and closing flag is checked for the CRC. In case of a CRC error, the CRC/Framing Error bit in status register 1 is set (RR1: D6 = 1). Receiver CRC may be disabled/enabled by WR3,D3. The CRC bytes on the received frame are passed on to the CPU just like data, and may be discarded by the CPU.

Receive Terminator

An end of frame is indicated by End of Frame interrupt. The CPU may issue an "Error Reset" command to reset this interrupt.

DMA (Direct Memory Access) Mode

The 8274 can be interfaced directly to the Intel DMA Controllers 8237A, 8257A and Intel I/O Processor 8089. The 8274 can be programmed into DMA mode by setting appropriate bits in WR2A. See Figure 11 for details.

TRANSMIT IN DMA MODE

After initializing the 8274 into the DMA mode, the first character must be loaded from the CPU to start the DMA cycle. When the first data character (may be the address byte in SDLC) is transferred from the transmit buffer to the transmit serial shift register, the transmit buffer gets empty and a transmit DMA request (TxDRQ) is generated for the channel. Just like the interrupt mode, to ensure that the CRC bytes are included in the frame, the transmit under-run/EOM latch must be reset. This should preferably be done after loading the first character from the CPU. The DMA will progress without any CPU intervention. When the DMA controller reaches the terminal count, it will not respond to the DMA request, thus letting the MPSC under-run. This will ensure CRC transmission. However, the under-run condition will generate an interrupt due to the Tx under-run/EOM bit getting set (RR0: D6). The CPU should issue a "Reset TxInt/DRQ pending" command to reset TxDRQ and issue a "Reset External Status" command to reset Tx Under-run/EOM interrupt. Following the CRC transmission, flag (7EH) will be loaded into the transmit buffer. This will also generate the TxDRQ since the transmit buffer is empty following the transmission of the CRC bytes. The CPU may issue a "Reset TxINT/DRQ pending" command to reset the TxDRQ. "Reset TxINT/DRQ pending" command must be issued before setting up the transmit DMA channel on the DMA Controller, otherwise the MPSC will start the DMA transfer immediately after the DMA channel is set up.

RECEIVE IN DMA MODE

The receiver must be programmed in RxINT on first receive character mode (WR1: D4 D3 = 0 1). Upon receiving the first character, which may be the address byte in SDLC, the MPSC generates an interrupt and also generates a Rx DMA Request (Rx DRQ) for the appropriate channel. The CPU has three byte times to service this interrupt (enable the DMA controller, etc.) before the receiver buffer will overflow. It is advisable to initialize the DMA controller before receiving the first character. In case of high bit rates, the CPU will have to service the interrupt very fast in order to avoid receiver over-run.

Once the DMA is enabled, the received data is transferred to the memory under DMA control. Any received error conditions or external status change condition will generate an interrupt as in the interrupt driven mode. The End of Frame is indicated by the End of Frame interrupt which is generated on reception of the closing flag of the SDLC frame. This End of Frame condition also disables the Receive DMA request. The End of Frame interrupt may be reset by issuing an "Error Reset" command to the MPSC. The "Error Reset" command also re-enables the Receive DMA request. It may be noted that the End of Frame condition sets bit D7 in RR1. This bit gets reset by "Error Reset" command. However, End of Frame bit (RR1: D7) can also be reset by the flag of the next incoming frame. For proper operation, Error Reset Command should be issued "after" the End of Frame Bit (RR1: D7) is set. In a more general case, "Error Reset" command should be issued after End of Frame, Receive over-run or Receive parity bit are set in RR1.

Wait Mode

The wait mode is normally used for block transfer by synchronizing the data transfer through the Ready output from the MPSC, which may be connected to the Ready input of the CPU. The mode can be programmed by WR 1, D7 D5 and may be programmed separately and independently on CH A and CH B. The Wait Mode will be operative if the following conditions are satisfied.

(i) Interrupts are enabled.

(ii) Wait Mode is enabled (WR1: D7)

(iii) CS = 0, A1 = 0

The RDY output becomes active when the transmitter buffer is full or receiver buffer is empty. This way the RDY output from the MPSC can be used to extend the CPU read and write cycle by inserting WAIT states. RDY_A or RDY_B are in high impedance state when the corresponding channel is not selected. This makes it possible to connect RDY_A and RDY_B outputs in wired OR configuration. Caution must be exercised here in using the RDY outputs of the MPSC or else the CPU may hang up for indefinite period. For example, let us assume that transmitter buffer is full and $\overline{RDY_A}$ is active, forcing the CPU into a wait state. If the $\overline{CTS}$ goes inactive during this period, the RDY_A will remain active for indefinite period and CPU will continue to insert wait states.

Vectored/Non-Vectored Mode

The MPSC is capable of providing an interrupt vector in response to the interrupt acknowledge sequence from the CPU. WR2, CH B contains this vector and the vector can be read in status register RR2. WR2, CH A (bit D5) can program the MPSC in vectored or non-vectored mode. See Figure 14 for details.

In both cases, WR2 may still have the vector stored in it. However, in vectored mode, the MPSC will put the vector on the data bus in response to the INTA (Interrupt Acknowledge) sequence as shown in Figure 15. In non-vectored mode, the MPSC will not respond to the INTA sequence. However, the CPU can read the vector by polling Status Register RR2. WR2A, D4 and D3 can be programmed to respond to 8085 or 8086 INTA sequence. It may be noted here that $\overline{IPI}$ (Interrupt Priority In) pin on the MPSC must be active for the vector to appear on the data bus.

WR2A, D5	Interrupt Mode
0	Non-vectored Interrupt
1	Vectored Interrupt

Figure 14. Vectored Interrupt

STATUS AFFECT VECTOR

The Vector stored in WR2B can be modified by the source of the interrupt. This can be done by setting the Status Affect Vector bit (WR1: D2). This powerful feature of the MPSC provides fast interrupt response time, by eliminating the need of writing a routine to read the status of the MPSC. Three bits of the vector are modified in eight different ways as shown on Figure 16. Bits V4, V3, V2 are modified in 8085 based system and bits V2, V1, V0 are modified in 8086/88 based system.

In non-vectored mode, the status affect vector mode can still be used and the vector read by the CPU. Status register RR2B (Read Register 2 in Channel B) will contain this modified vector.

WR2A D5	D4	D3	IPI	Mode	1st INTA	2nd INTA	3rd INTA
0	X	X	X	Non-Vectored	HI-Z	HI-Z	HI-Z
1	0	0	0	8085-1	1100 1101	V7 V6 V5 V4 V3 V2 V1 V0	0000 0000
1	0	0	1	8085-1	1100 1101	HI-Z	HI-Z
1	0	1	0	8085-2	HI-Z	V7 V6 V5 V4 V3 V2 V1 V0	0000 0000
1	0	1	1	8085-2	HI-Z	HI-Z	HI-Z
1	1	0	0	8086	HI-Z	V7 V6 V5 V4 V3 V2 V1 V0	—
1	1	0	1	8086	HI-Z	HI-Z	—

Figure 15. MPSC Vectored Interrupts

(8085 (8086)	V4 V2	V3 V1	V2 V0	Channel	Interrupt Source
	0	0	0	B	Tx Buffer Empty
	0	0	1		EXT/STAT Change
	0	1	0		RX CHAR Available
	0	1	1		Special Rx Condition
	1	0	0	A	Tx Buffer Empty
	1	0	1		EXT/STAT Change
	1	1	0		RX CHAR Available
	1	1	1		Special Rx Condition

Rx Special Condition: Parity Error, Framing Error, Rx Over-run Error, EOF (SDLC).
EXT/STAT Change: Change in Modem Control Pin Status: CTS, DCD, SYNC, EOM, Break/Abort Detection.

Figure 16. Status Affect Vector Mode

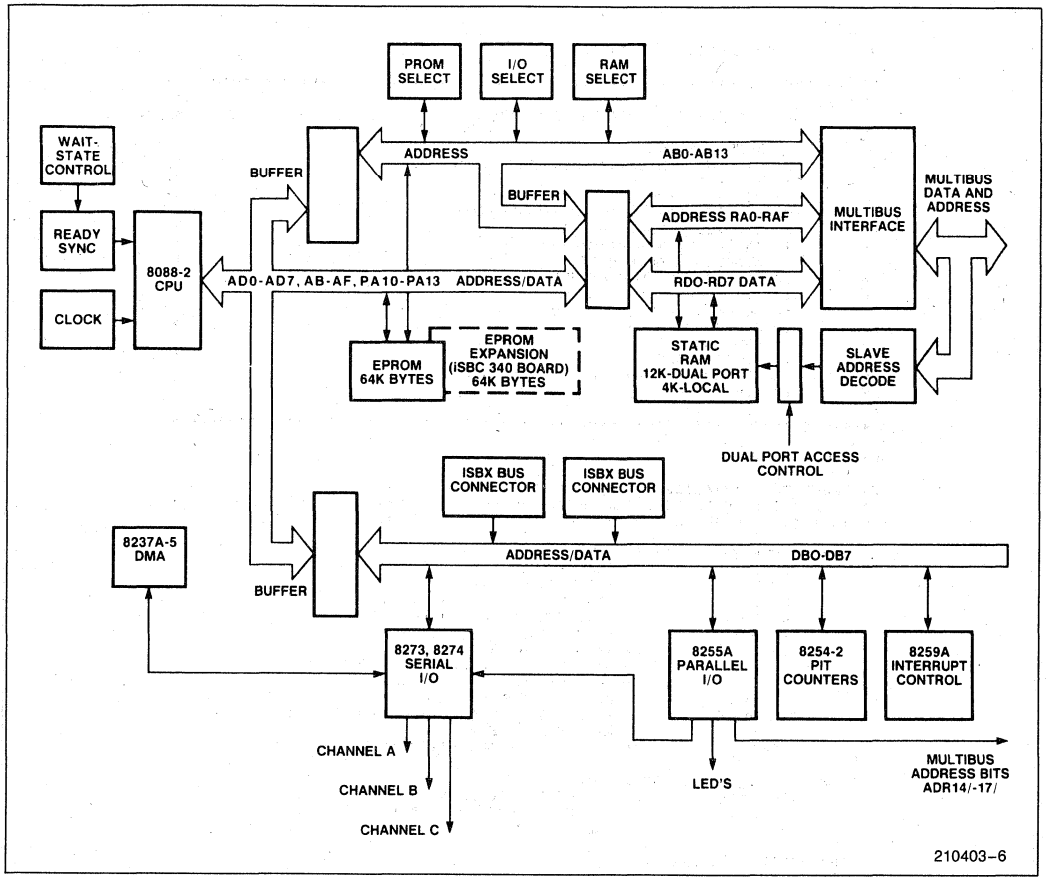


Figure 17. Functional Block Diagram—iSBC® 88/45

APPLICATION EXAMPLE

This section describes the hardware and software of an 8274/8088 system. The hardware vehicle used is the INTEL Single Board Computer iSBC 88/45—Advanced Communication Controller. The software which exercises the 8274 is written in PLM 86. This example will demonstrate how 8274 can be configured into the SDLC mode and transfer data through DMA control. The hardware example will help the reader configure his hardware and the software examples will help in developing an application software. Most software examples closely approximate real data link controller software in the SDLC communication and may be used with very little modification.

iSBC® 88/45

A brief description of the iSBC 88/45 board will be presented here. For more detailed information on the board and the schematics, refer to Hardware Manual for the iSBC 88/45, Advanced Communication Controller. iSBC 88/45 is an intelligent slave/multimaster communication board based on the 8088 processor, the 8274 and the 8273 SDLC/HDLC controller. Figure 17 shows the functional block diagram of the board. The iSBC 88/45 has the following features.

- 8 MHz processor
- 16K bytes of static RAM (12K dual port)
- Multimaster/Intelligent Slave Multibus Interface
- Nine Interrupt Levels 8259A
- Two serial channels through 8274
- One Serial channel through 8273
- S/W programmable baud rate generator
- Interfaces: RS232, RS422/449, CCITT V.24
- 8237A DMA controller
- Baud Rate to 800K Baud

```
INITIALIZE_8274: PROCEDURE PUBLIC;

/*************************************************************************/
/*                                                                       */
/*        INITIALIZE THE 8274 FOR SDLC MODE                              */
/*                                                                       */
/*        1.  RESET CHANNEL                                              */
/*        2.  EXTERNAL INTERRUPTS ENABLED                                */
/*        3.  NO WAIT                                                    */
/*        4.  PIN 10 = RTS                                               */
/*        5.  NON-VECTORED INTERRUPT-8086 MODE                           */
/*        6.  CHANNEL A DMA, CH B INT                                    */
/*        7.  TX AND RX = 8 BITS/CHAR                                    */
/*        9.  ADDRESS SEARCH MODE                                        */
/*        10. CD AND CTS AUTO ENABLE                                     */
/*        11. X1 CLOCK                                                   */
/*        12. NO PARITY                                                  */
/*        13. SDLC/HDLC MODE                                             */
/*        14. RTS AND DTR                                                */
/*        15. CCITT - CRC                                                */
/*        16. TRANSMITTER AND RECEIVER ENABLED                           */
/*        17. 7EH = FLAG                                                 */
/*                                                                       */
/*************************************************************************/

DECLARE C BYTE;

/* TABLE TO INITIALIZE THE 8274 CHANNEL A AND B    */
/* FORMAT IS: WRITE REGISTER, REGISTER DATA        */
/* INITIALIZE CHANNEL A ONLY                       */

DECLARE TABLE_74_A(*) BYTE DATA
        (00H, 18H,          /* CHANNEL RESET */
         00H, 80H,          /* RESET TX CRC */
         02H, 11H,          /* PIN 10=RTSB, A DMA, B INT */
         04H, 20H,          /* SDLC/HDLC MODE, NO PARITY */
         07H, 07EH,         /* SDLC FLAG */
         01H, 08H,          /* RX DMA ENABLE */
         05H, 0EBH,         /* DTR, RTS, 8 TX BITS, TX ENABLE, */
                            /* SDLC CRC, TX CRC ENABLE */
         06H, 55H,          /* DEFAULT ADDRESS */
         03H, 0D9H,         /* 8 RX BITS, AUTO ENABLES, HUNT MODE, */
                            /* RX CRC ENABLE */
         0FFH);             /* END OF INITIALIZATION TABLE */

DECLARE TABLE_74_B(*) BYTE DATA
        (02H, 00H,          /* INTERRUPT VECTOR */
         01H, 1CH,          /* STATUS AFFECTS VECTOR */
         0FFH);             /* END */

/* INITIALIZE THE 8274 */

C=0;
DO WHILE TABLE_74_B(C) <> 0FFH;
        OUTPUT(COMMAND_B_74) = TABLE_74_B(C);
        C=C+1;
        OUTPUT(COMMAND_B_74) = TABLE_74_B(C);
        C=C+1;
END;

C=0;
DO WHILE TABLE_74_A(C) <> 0FFH;
        OUTPUT(COMMAND_A_74) = TABLE_74_A(C);
        C=C+1;
        OUTPUT(COMMAND_A_74) = TABLE_74_A(C);
        C=C+1;
END;
RETURN;
END INITIALIZE_8274;
```

210403-7

Figure 18. Typical MPSC SDLC Initialization Sequence

For this application, the CPU is run at 8 MHz. The board is configured to operate the 8274 in SDLC operation with the data transfer in DMA mode using the 8237A. 8274 is configured first in non-vectored mode in which case the INTEL Priority Interrupt Controller 8259A is used to resolve priority between various interrupting sources on the board and subsequently interrupt the CPU. However, the vectored mode of the 8274 is also verified by disabling the 8259A and reading the vectors from the 8274. Software examples for each case will be shown later.

The application example is interrupt driven and uses DMA for all data transfers under 8237A control. The 8254 provides the transmit and receive clocks for the 8274. The 8274 was run at 400K baud with a local loopback (jumper wire) on Channel A data. The board was also run at 800K baud by modifying the software as will be discussed later in the Special Applications section. One detail to note is that the Rx Channel DMA request line from the 8274 has higher priority than the Tx Channel DMA request line. The 8274 master clock was 4.0 MHz. The on-board RAM is used to define transmit and receive data buffers. In this application, the data is read from memory location 800H through 810H and transferred to memory location 900H to 910H through the 8274 Serial Link. The operation is full duplex. 8274 modem control pins, $\overline{CTS}$ and $\overline{CD}$ have been tied low (active).

Software

The software consists of a monitor program and a program to exercise the 8274 in the SDLC mode. Appendix A contains the entire program listing. For the sake of clarity, each source module has been rewritten in a simple language and will be discussed here individually. Note that some labels in the actual listings in the Appendix will not match with the labels here. Also the listing in the Appendix sets up some flags to communicate with the monitor. Some of these flags are not explained in detail for the reason that they are not pertinent to this discussion. The monitor takes the command from a keyboard and executes this program, logging any error condition which might occur.

8274 Initialization

The MPSC is initialized in the SDLC mode for Channel A. Channel B is disabled. See Figure 18 for the initialization routine. Note that WR4 is initialized before setting up the transmitter and receive parameters. However, it may also be pointed out that other than WR4, all the other registers may be programmed in any order. Also SDLC-CRC has been programmed for correct operation. An incorrect CRC selection will result in incorrect operation. Also note that receive interrupt on first receive character has been programmed although Channel A is in the DMA mode.

Interrupt Routines

The 8274 interrupt routines will be discussed here. On an 8274 interrupt, program branches off to the "Main Interrupt Routine". In main interrupt routine, status register RR2 is read. RR2 contains the modified vector. The cause of the interrupt is determined by reading the modified bits of the vector. Note that the 8274 has been programmed in the non-vectored mode and status affects vector bit has been set. Depending on the value of the modified bits, the appropriate interrupt routine is called. See Figure 19 for the flow diagram and Figure 20 for the source code. Note that an End of Interrupt Command is issued after servicing the interrupt. This is necessary to enable the lower priority interrupts.

Figure 21 shows all the interrupt routines called by the Main Interrupt Routine. "Ignore-Interrupt" as the name implies, ignores any interrupts and sets the FAIL flag. This is done because this program is for Channel A only and we are ignoring any Channel B interrupts. The important thing to note is the Channel A Receiver Character available routine. This routine is called after receiving the first character in the SDLC frame. Since the transfer mode is DMA, we have a maximum of three character times to service this interrupt by enabling the DMA controller.

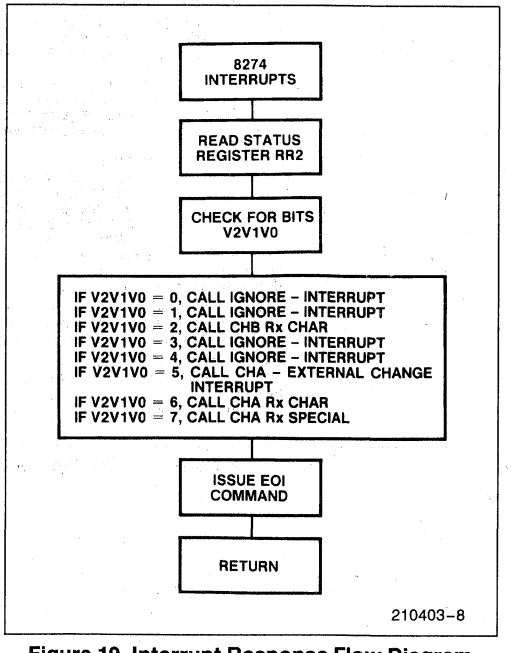


Figure 19. Interrupt Response Flow Diagram

```
/*************************/
/* MAIN INTERRUPT ROUTINE */
/*************************/
OUTPUT(COMMAND_B_74) = 2;                  /* SET POINTER TO 2*/
TEMP = INPUT(STATUS_B_74) AND 07H;         /* READ INTERRUPT VECTOR */
                                           /* CHECK FOR CHA INT ONLY*/
/* FOR THIS APPLICATION CH B INTERRUPTS ARE IGNORED*/
DO CASE TEMP;
        CALL    IGNORE_INT;                /* V2V1V0 = 000*/
        CALL    IGNORE_INT;                /* V2V1V0 = 001*/
        CALL    CHB_RX_CHAR;               /* V2V1V0 = 010*/
        CALL    IGNORE_INT;                /* V2V1V0 = 011*/
        CALL    IGNORE_INT;                /* V2V1V0 = 100*/
        CALL    CHA_EXTERNAL_CHANGE;       /* V2V1V0 = 101*/
        CALL    CHA_RX_CHAR;               /* V2V1V0 = 110*/
        CALL    CHA_RX_SPECIAL;            /* V2V1V0 = 111*/
END;
OUTPUT(COMMAND_A_74) =38H;       /* END OF INTERRUPT FOR 8274 */
RETURN;
END INTERRUPT_8274;
```

210403-9

Figure 20. Typical Main Interrupt Routine

```
/*******************************************************/
/* CHANNEL A EXTERNAL/STATUS CHANGE INTERRUPT HANDLER */
/*******************************************************/
CHA_EXTERNAL_CHANGE: PROCEDURE;
TEMP = INPUT(STATUS_A_74);       /* STATUS REG 1*/
IF (TEMP AND END_OF_TX_MESSAGE) = END_OF_TX_MESSAGE THEN
        TXDONE_S=DONE;
ELSE DO;
        TXDONE_S=DONE;
        RESULTS_S=FAIL;
     END;
OUTPUT(COMMAND_A_74) = 10H;      /* RESET EXT/STATUS INTERRUPTS */
RETURN;
END CHA_EXTERNAL_CHANGE;
/*************************************************************/
/* CHANNEL A SPECIAL RECEIVE CONDITIONS INTERRUPT HANDLER */
/*************************************************************/
CHA_RX_SPECIAL: PROCEDURE;
        OUTPUT(COMMAND_A_74) = 1;
        TEMP = INPUT(STATUS_A_74);
        IF (TEMP AND END_OF_FRAME) = END_OF_FRAME THEN
              DO;
                IF(TEMP AND 040H) = 040H THEN
                  RESULTS_S = FAIL;          /* CRC ERROR */
                RXDONE_S = DONE;
                OUTPUT(COMMAND_A_74) = 30H; /*ERROR RESET*/
              END;
              ELSE DO;
                IF (TEMP AND 20H) = 20H THEN DO;
                RESULTS_S = FAIL;            /* RX OVERRUN ERROR*/
                RXDONE_S = DONE;
                OUTPUT(COMMAND_A_74) = 30H; /*ERROR RESET*/
                END;
              END;
RETURN;
END CHA_RX_SPECIAL;
/*************************************/
/* CHANNEL A RECEIVE CHARACTER AVAILABLE */
/*************************************/
CHA_RX_CHAR: PROCEDURE;
OUTPUT(SINGLE_MASK) = CH0_SEL;           /*ENABLE RX DMA CHANNEL*/
RETURN;
END CHA_RX_CHAR;
```

210403-10

Figure 21. 8274 Typical Interrupt Handling Routines

It may be recalled that the receiver buffer is three bytes deep in addition to the receiver shift register. At very high data rates, it may not be possible to have enough time to read RR2, enable the DMA controller without overrunning the receiver. In a case like this, the DMA controller may be left enabled before receiving the Receive Character Interrupt. Remember, the Rx DMA request and interrupt for the receive character appears at the same time. If the DMA controller is enabled, it would service the DMA request by reading the received character. This will make the 8274 interrupt line go inactive. However, the 8259A has latched the interrupt and a regular interrupt acknowledge sequence still occurs after the DMA controller has completed the transfer and given up the bus. The 8259A will return Level 7 interrupt since the 8274 interrupt has gone away. The user software must take this into account, otherwise the CPU will hang up.

The procedure shown for the Special Receive Condition Interrupt checks if the interrupt is due to the End of Frame. If this is not TRUE, the FAIL flag is set and the program aborted. For a real life system, this must be followed up by error recovery procedures which obviously are beyond the scope of this Application Note.

The transmission is terminated when the End of Message (RR0, D6) interrupt is generated. This interrupt is serviced in the Channel A External/Status Change interrupt procedure. For any other change in external status conditions, the program is aborted and a FAIL flag set.

Main Program

Finally, we will briefly discuss the main program. Figure 22 shows the source program. It may be noted that the Transmit Under-run latch is reset after loading the first character into the 8274. This is done to ensure CRC transmission at the end of the frame. Also, the first character is loaded from the CPU to start DMA transfer of subsequent data. This concludes our discussion on hardware and software example. Appendix A also includes the software written to exercise the 8274 in the vectored mode by disabling the 8259A.

```
CHA_SDLC_TEST: PROCEDURE BYTE PUBLIC;

        CALL    ENABLE_INTERRUPTS_S;
        CALL    INIT_8274_SDLC_S;
        ENABLE;
        OUTPUT(COMMAND_A_74) = 28H;  /* RESET TX INT/DMA          */
        OUTPUT(COMMAND_B_74) = 28H;  /* BEFORE INITIALIZING 8237*/
        CALL    INIT_8237_S;
        OUTPUT(DATA_A_74) = 55H;     /*LOAD FIRST CHARACTER FROM */
                                     /*CPU */
        /* TO ENSURE CRC TRANSMISSION, RESET TX UNDERRUN LATCH   */
        OUTPUT(COMMAND_A_74) = 0C0H;
        RXDONE_S,TXDONE_S=NOT_DONE;  /* CLEAR ALL FLAGS          */
        RESULTS_S=PASS;              /* FLAG SET FOR MONITOR     */
        DO WHILE TXDONE_S=NOT_DONE;  /* DO UNTIL TERMINAL COUNT  */
        END;

        DO WHILE(INPUT(STATUS_A_74) AND 04H) <> 04H;
        /* WAIT FOR CRC TO GET TRANSMITTED */
        /* TEST FOR TX BUFFFER EMPTY TO VERIFY THIS*/
        END;
        DO WHILE RXDONE_S=NOT_DONE;  /* DO UNTIL TERMINAL COUNT  */
        END;
        CALL    STOP_8237_S;
END CHA_SDLC_TEST;
```

210403-11

Figure 22. Typical 8274 Transmit/Receive Set-Up in SDLC Mode

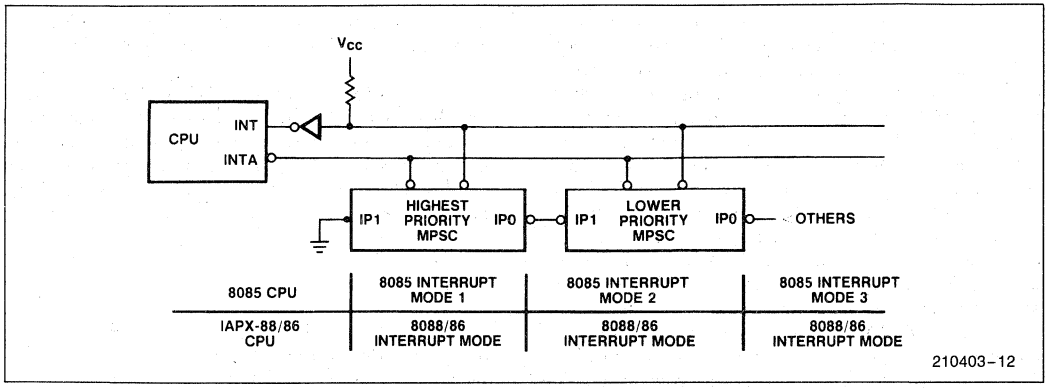


Figure 23. 8274 Daisy Chain Vectored Mode

SPECIAL APPLICATIONS

In this section, some special application issues will be discussed. This will be useful to a user who may be using a mode which is possible with the 8274 but not explicitly explained in the data sheet.

MPSC Daisy Chain Operation

Multiple MPSCs can be connected in a daisy-chain configuration (see Figure 23). This feature may be useful in an application where multiple communication channels may be required and because of high data rates, conventional interrupt controller is not used to avoid long interrupt response times. To configure the MPSCs for the daisy chain operation, the interrupt priority input pins ($\overline{IPI}$) and interrupt priority output pins ($\overline{IPO}$) of the MPSC should be connected as shown. The highest priority device has its $\overline{IPI}$ pin connected to ground. Each MPSC is programmed in a vectored mode with status affects vector bit set. In the 8085 basic systems, only one MPSC should be programmed in the 8085 Mode 1. This is the MPSC which will put the call vector (CD Hex) on the data bus in response to the first $\overline{INTA}$ pulse (see Figure 15). It may be pointed out that the MPSC in 8085 Mode 1 will provide the call vector irrespective of the state of $\overline{IPI}$ pin. Once a higher priority MPSC generates an interrupt, its $\overline{IPO}$ pin goes inactive thus preventing lower priority MPSCs from interrupting the CPU. Preferably the highest priority MPSC should be programmed in 8085 Mode 1. It may be recalled that the Priority Resolve Time on a given MPSC extends from the falling edge of the first $\overline{INTA}$ pulse to the falling edge of the second $\overline{INTA}$ pulse. During this period, no new internal interrupt requests are accepted. The maximum number of the MPSCs that can be connected in a daisy chain is limited by the Priority Resolution Time. Figure 24 shows a maximum number of MPSCs that can be connected in various CPU systems.

It may be pointed out that $\overline{IOP}$ to $\overline{IPI}$ delay time specification is 100 ns.

System Configuration	Priority Resolution Time Min (ns)	Number of 8274s Daisy Chained (Max)
8086-1	400	4
8086-2	500	5
8086	800	8
8088	800	8
8085-2	1200	12
8085A	1920	19

NOTE:
Zero wait states have been assumed.

Figure 24. 8274 Daisy Chain Operation

Bisync Transparent Communication

Bisync applications generally require that data transparency be established during communication. This requires that the special control characters may not be included in the CRC accumulation. Refer to the Synchronous Protocol Overview section for a more detailed discussion on data transparency. The 8274 can be used for transparent communication in Bisync communications. This is made possible by the capability of the MPSC to selectively turnon/turnoff the CRC accumulation while transmitting or receiving. In bisync transparent transmit mode, the special characters (DLE, DLE SYN, etc) are excluded from CRC calculation. This can be easily accomplished by turning off the transmit CRC calculation (WR5: D5 = 0) before loading the special character into the transmit buffer. If the next character is to be included in the CRC accumulation, then the CRC can be enabled (WR5: D5 = 1). See Figure 25 for a typical flow diagram.

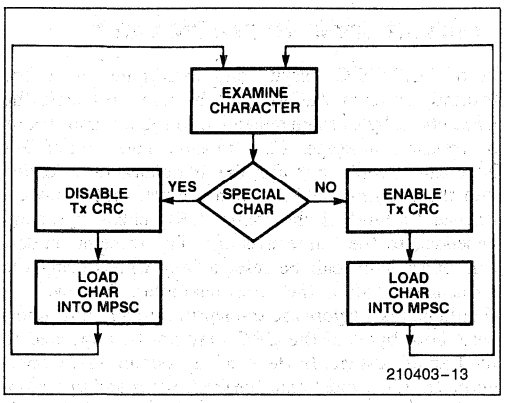


Figure 25. Transmit in Bisync Transparent Mode

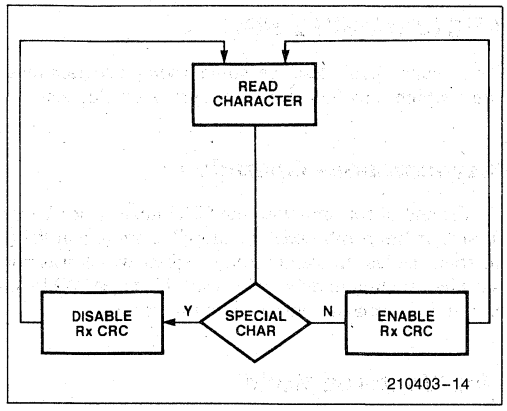


Figure 26. Receive in Bisync Transparent Mode

During reception, it is possible to exclude received character from CRC calculation by turning off the Receive CRC after reading the special character. This is made possible by the fact that the received data is presented to receive CRC checker 8 bit times after the character has been received. During this 8 bit times, the CPU must read the character and decide if it wants to be included in the CRC calculation. Figure 26 shows the typical flow diagram to achieve this.

It should be noted that the CRC generator must be enabled during CRC reception. Also, after reading the CRC bytes, two more characters (SYNC) must be read before checking for CRC check result in RR1.

Auto Enable Mode

In some data communication applications, it may be required to enable the transmitter or the receiver when the $\overline{CTS}$ or the $\overline{CD}$ lines respectively, are activated by the modems. This may be done very easily by programming the 8274 into the Auto Enable Mode. The auto enable mode is set by writing a '1' to WR3,D5. The function of this mode is to enable the transmitter automatically when $\overline{CTS}$ goes active. The receiver is enabled when $\overline{CD}$ goes active. An in-active state of $\overline{CTS}$ or $\overline{CD}$ pin will disable the transmitter or the receiver respectively. However, the Transmit Enable bit (WR5:D3) and Receive Enable bit (WR3:D1) must be set in order to use the auto enable mode. In non-auto mode, the transmitter or receiver is enabled if the corresponding bits are set in WR5 and WR3, irrespective of the state $\overline{CTS}$ or $\overline{CD}$ pins. It may be recalled that any transition on $\overline{CTS}$ or $\overline{CD}$ pin will generate External/Status Interrupt with the corresponding bits set in RR1. This interrupt can be cleared by issuing a Reset External/Status interrupt command as discussed earlier.

Note that in auto enable mode, the character to be transmitted must be loaded into the transmit buffer after the $\overline{CTS}$ becomes active, not before. Any character loaded into the transmit buffer before the $\overline{CTS}$ became active will not be transmitted.

High Speed DMA Operation

In the section titled Application Example, the MPSC has been programmed to operate in DMA mode and receiver is programmed to generate an interrupt on the first receive character. You may recall that the receive FIFO is three bytes deep. On receiving the interrupt on the first receive character, the CPU must enable the DMA controller within three received byte times to avoid receiver over-run condition. In the application example, at 400K baud, the CPU had approximately 60 μs to enable the DMA controller to avoid receiver buffer overflow. However, at higher baud rates, the CPU may not have enough time to enable the DMA controller in time. For example, at 1M baud, the CPU should enable the DMA controller within approximately 24 μs to avoid receiver buffer overrun. In most applications, this is not sufficient time. To solve this problem, the DMA controller should be left enabled before getting the interrupt on the first receive character (which is accompanied by the Rx DMA request for the appropriate channel). This will allow he DMA controller to start DMA transfer as soon as the Rx DMA request becomes active without giving the CPU enough time to respond to the interrupt on the first receive character. The CPU will respond to the interrupt *after* the DMA transfer has been completed and will find the 8259A (see Application Example) responding with interrupt level 7, the lowest priority level. Note that the 8274 interrupt request was satisfied by the DMA controller, hence the interrupt on the first receive character was cleared and the 8259A had no pending interrupt. Because of no pending interrupt, the 8259A returned interrupt level 7 in response to the INTA sequence from the CPU. The user software should take care of this interrupt.

PROGRAMMING HINTS

This section will describe some useful programming hints which may be useful in program development.

Asynchronous Operation

At the end of transmission, the CPU must issue "Reset Transmit Interrupt/DMA Pending" command in WR0 to reset the last transmit empty request which was not satisfied. Failing to do so will result in the MPSC locking up in a transmit empty state forever.

Non-Vectored Mode

In non-vectored mode, the Interrupt Acknowledge pin (INTA) on the MPSC must be tied high through a pull-up resistor. Failing to do so will result in unpredictable response from the 8274.

HDLC/SDLC Mode

When receiving data in SDLC mode, the CRC bytes must be read by the CPU (or DMA controller) just like any other data field. Failing to do so will result in receiver buffer overflow. Also, the End of Frame Interrupt indicates that the entire frame has been received. At this point, the CRC result (RR1:D6) and residue code (RR1:D3, D2, D1) may be checked.

Status Register RR2

ChB RR2 contains the vector which gets modified to indicate the source of interrupt (see the section titled MPSC Modes of Operation). However, the state of the vector does not change if no new interrupts are generated. The contents of ChB RR2 are only changed when a new interrupt is generated. In order to get the correct information, RR2 must be read only after an interrupt is generated, otherwise it will indicate the previous state.

Initialization Sequence

The MPSC initialization routine must issue a channel Reset Command at the beginning. WR4 should be defined before other registers. At the end of the initialization sequence, Reset External/Status and Error Reset commands should be issued to clear any spurious interrupts which may have been caused at power up.

Transmit Under-Run/EOM Latch

In SDLC/HDLC, bisync and monosync mode, the transmit underrun/EOM must be reset to enable the CRC check bytes to be appended to the transmit frame or transmit message. The transmit under-run/EOM latch can be reset only after the first character is loaded into the transmit buffer. When the transmitter underruns at the end of the frame, CRC check bytes are appended to the frame/message. The transmit underrun/EOM latch can be reset at any time during the transmission after the first character. However, it should be reset *before* the transmitter under-runs otherwise, both bytes of the CRC may not be appended to the frame/message. In the receive mode in bisync operation, the CPU must read the CRC bytes and two more SYNC characters before checking for valid CRC result in RR1.

Sync Character Load Inhibit

In bisync/monosync mode only, it is possible to prevent loading sync characters into the receive buffers by setting the sync character load inhibit bit (WR3:D1 = 1). Caution must be exercised in using this option. It may be possible to get a CRC character in the received message which may match the sync character and not get transferred to the receive buffer. However, sync character load inhibit should be enabled during all pre-frame sync characters so the software routine does not have to read them from the MPSC.

In SDLC/HDLC mode, sync character load inhibit bit must be reset to zero for proper operation.

EOI Command

EOI command can only be issued through channel A irrespective of which channel had generated the interrupt.

Priority in DMA Mode

There is no priority in DMA mode between the following four singals: TxDRQ(CHA), RxDRQ(CHA), TxDRQ(CHB), RxDRQ(CHB). The priority between these four signals must be resolved by the DMA controller. At any given time, all four DMA channels from the 8274 are capable of going active.

APPENDIX A
APPLICATION EXAMPLE: SOFTWARE LISTINGS

```
PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE INIT_8274_S
OBJECT MODULE PLACED IN :F1:SINI74.OBJ
COMPILER INVOKED BY:  PLM86.86 :F1:SINI74.PLM TITLE(iSBC 88/45 8274 CHANNEL
A SDLC TEST) COMPACT NOINTVECTOR ROM


                /*************************************************/
                /*                                               */
                /*       INITIALIZE THE 8274 FOR SDLC MODE       */
                /*                                               */
                /*       1. RESET CHANNEL                        */
                /*       2. EXTERNAL INTERRUPTS ENABLED          */
                /*       3. NO WAIT                              */
                /*       4. PIN 10 = RTS                         */
                /*       5. NON-VECTORED INTERRUPT-8086 MODE     */
                /*       6. CHANNEL A DMA, CH B INT              */
                /*       7. TX AND RX = 8 BITS/CHAR              */
                /*       9. ADDRESS SEARCH MODE                  */
                /*       10. CD AND CTS AUTO ENABLE              */
                /*       11. X1 CLOCK                            */
                /*       12. NO PARITY                           */
                /*       13. SDLC/HDLC MODE                      */
                /*       14. RTS AND DTR                         */
                /*       15. CCITT - CRC                         */
                /*       16. TRANSMITTER AND RECEIVER ENABLED    */
                /*       17. 7EH = FLAG                          */
                /*                                               */
                /*************************************************/

                INIT_8274_S: DO;

                $INCLUDE (:F1:PORTS.PLM)

          =     /*********************************************/
          =     /*                                           */
          =     /*       ISBC 88/45 PORT ASSIGNMENTS         */
          =     /*                                           */
          =     /*********************************************/

   2    1 =     DECLARE LIT LITERALLY 'LITERALLY';

          =     /* 8237A-5  PORTS */

   3    1 =     DECLARE CH0_ADDR          LIT     '080H',
          =             CH0_COUNT         LIT     '081H',
          =             CH1_ADDR          LIT     '082H',
          =             CH1_COUNT         LIT     '083H',
          =             CH2_ADDR          LIT     '084H',
          =             CH2_COUNT         LIT     '085H',
          =             CH3_ADDR          LIT     '086H',
          =             CH3_COUNT         LIT     '087H',
          =             STATUS_37         LIT     '088H',
          =             COMMAND_37        LIT     '088H',
          =             REQUEST_REG_37    LIT     '089H',
          =             SINGLE_MASK       LIT     '08AH',
          =             MODE_REG_37       LIT     '08BH',

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST

          =             CLR_BYTE_PTR_37 LIT     '08CH',
          =             TEMP_REG_37     LIT     '08DH',
          =             MASTER_CLEAR_37 LIT     '08DH',
          =             ALL_MASK_37     LIT     '08FH';

          =     /* 8254-2 PORTS */

   4    1 =     DECLARE CTR_00          LIT     '090H',
          =             CTR_01          LIT     '091H',
          =             CTR_02          LIT     '092H',
```

210403-15

```
                =           CONTROL0_54     LIT     '093H',
                =           STATUS0_54      LIT     '093H',
                =           CTR_10          LIT     '098H',
                =           CTR_11          LIT     '099H',
                =           CTR12           LIT     '09AH',
                =           CONTROL1_54     LIT     '09BH',
                =           STATUS1_54      LIT     '09BH';

                =   /* 8255 PORTS */

      5   1     =   DECLARE PORTA_55        LIT     '0A0H',
                =           PORTB_55        LIT     '0A1H',
                =           PORTC_55        LIT     '0A2H',
                =           CONTROL_55      LIT     '0A3H';

                =   /* 8274 PORTS */

      6   1     =   DECLARE DATA_A_74       LIT     '0D0H',
                =           DATA_B_74       LIT     '0D1H',
                =           STATUS_A_74     LIT     '0D2H',
                =           COMMAND_A_74    LIT     '0D2H',
                =           STATUS_B_74     LIT     '0D3H',
                =           COMMAND_B_74    LIT     '0D3H';

                =   /* 8259A PORTS */

      7   1     =   DECLARE STATUS_POLL_59  LIT     '0E0H',
                =           ICW1_59         LIT     '0E0H',
                =           OCW2_59         LIT     '0E0H',
                =           OCW3_59         LIT     '0E0H',
                =           OCW1_59         LIT     '0E1H',
                =           ICW2_59         LIT     '0E1H',
                =           ICW3_59         LIT     '0E1H',
                =           ICW4_59         LIT     '0E1H';

                =   /* 8274 REGISTER BIT ASSIGNMENTS */
                =   /* READ REGISTER 0 */

      8   1     =   DECLARE RX_AVAIL        LIT     '01H',
                =           INT_PENDING     LIT     '02H',
                =           TX_EMPTY        LIT     '04H',
                =           CARRIER_DETECT  LIT     '08H',
                =           SYNC_HUNT       LIT     '10H',
                =           CLEAR_TO_SEND   LIT     '20H',

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST

                =           END_OF_TX_MESSAGE LIT   '40H',
                =           BREAK_ABORT     LIT     '80H';

                =   /* READ REGISTER 1 */

      9   1     =   DECLARE ALL_SENT        LIT     '01H',
                =           PARITY_ERROR    LIT     '10H',
                =           RX_OVERRUN      LIT     '20H',
                =           CRC_ERROR       LIT     '40H',
                =           END_OF_FRAME    LIT     '80H';

                =   /* READ REGISTER 2 */

     10   1     =   DECLARE TX_B_EMPTY      LIT     '00H',
                =           EXT_B_CHANGE    LIT     '01H',
                =           RX_B_AVAIL      LIT     '02H',
                =           RX_B_SPECIAL    LIT     '03H',
                =           TX_A_EMPTY      LIT     '04H',
                =           EXT_A_CHANGE    LIT     '05H',
                =           RX_A_AVAIL      LIT     '06H',
                =           RX_A_SPECIAL    LIT     '07H';
```

210403-16

```
          =   /* 8237 BIT ASSIGNMENTS */

   11   1 =   DECLARE CH0_SEL          LIT     '00H',
          =           CH1_SEL          LIT     '01H',
          =           CH2_SEL          LIT     '02H',
          =           CH3_SEL          LIT     '03H',
          =           WRITE_XFER       LIT     '04H',
          =           READ_XFER        LIT     '08H',
          =           DEMAND_MODE      LIT     '00H',
          =           SINGLE_MODE      LIT     '40H',
          =           BLOCK_MODE       LIT     '80H',
          =           SET_MASK         LIT     '04H';

   12   1       DELAY_S: PROCEDURE PUBLIC;
   13   2       DECLARE D WORD;
   14   2       D=0;
   15   2       DO WHILE D<800H;
   16   3       D=D+1;
   17   3       END;
   18   2       END DELAY_S;


   19   1       INIT_8274_SDLC_S:   PROCEDURE PUBLIC;

   20   2       DECLARE C   BYTE;

                $EJECT



PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


                /* TABLE TO INITIALIZE THE 8274 CHANNEL A AND B */

                /*  FORMAT IS: WRITE REGISTER, REGISTER DATA  */
                /*       INITIALIZE CHANNEL ONLY                  */

   21   2       DECLARE TABLE_74_A(*) BYTE DATA
                        (00H,18H,       /* CHANNEL RESET */
                        00H,80H,        /* RESET TX CRC */
                        02H,11H,        /* PIN 10=RTSB, A DMA, B INT */
                        04H,20H,        /* SDLC/HDLC MODE, NO PARITY */
                        07H,07EH,       /* SDLC FLAG */
                        01H,0BH,        /* RX DMA ENABLE */
                        05H,0EBH,       /* DTR, RTS, 8 TX BITS, TX ENABLE, TX CRC ENABLE */
                        06H,55H,        /* DEFAULT ADDRESS */
                        03H,0D9H,       /* 8 RX BITS, AUTO ENABLES, HUNT MODE, */
                                        /* RX CRC ENABLE */
                        0FFH);          /* END OF INITIALIZATION TABLE */

   22   2       DECLARE TABLE_74_B(*) BYTE DATA
                        (02H,00H,       /* INTERRUPT VECTOR */
                        01H,1CH,        /* STATUS AFFECTS VECTOR */
                        0FFH);          /* END */


                /* INITIALIZE THE 8254 */

   23   2       OUTPUT(CONTROL0_54)=36H;
   24   2       OUTPUT(CTR_00) = LOW(20);       /* BAUD RATE = 400K_BAUD*/
   25   2       OUTPUT(CTR_00) = HIGH(20);      /* BAUD RATE = 400K_BAUD*/


                /* INITIALIZE THE 8274 */

   26   2       C=0;
   27   2       DO WHILE TABLE_74_B(C) <> 0FFH;
   28   3           OUTPUT(COMMAND_B_74) = TABLE_74_B(C);
   29   3           C=C+1;
   30   3           OUTPUT(COMMAND_B_74) = TABLE_74_B(C);
   31   3           C=C+1;
   32   3       END;
```

210403-17

```
  33   2       C=0;
  34   2       DO WHILE TABLE_74_A(C) <> 0FFH;
  35   3           OUTPUT(COMMAND_A_74) = TABLE_74_A(C);
  36   3           C=C+1;
  37   3           OUTPUT(COMMAND_A_74) = TABLE_74_A(C);
  38   3           C=C+1;
  39   3       END;
  40   2           CALL     DELAY_S;

  41   2       RETURN;
  42   2       END INIT_8274_SDLC_S;
  43   1       END INIT_8274_S;




PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


MODULE INFORMATION:

     CODE AREA SIZE      = 00A8H     168D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 0003H       3D
     MAXIMUM STACK SIZE  = 0006H       6D
     213 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION



PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE INIT_8237_CHA
OBJECT MODULE PLACED IN :F1:SINI37.OBJ
COMPILER INVOKED BY:  PLM86.86 :F1:SINI37.PLM TITLE(iSBC 88/45 8274 CHANNEL A SDLC
TEST) COMPACT NOINTVECTOR ROM


                /*************************************************************************/
                /*                                                                       */
                /*      8237    INITIALIZATION ROUTINE  FOR DMA TRANSFER                 */
                /*                                                                       */
                /*************************************************************************/

   1            INIT_8237_CHA: DO;

                $NOLIST

  12   1       INIT_8237_S: PROCEDURE PUBLIC;


  13   2       OUTPUT(MASTER_CLEAR_37)=0;
  14   2       OUTPUT(COMMAND_37) = 20H;        /* EXTENDED WRITE */
  15   2       OUTPUT(ALL_MASK_37) = 0FH;       /* MASK ALL REQUESTS */
  16   2       OUTPUT(MODE_REG_37) = (SINGLE_MODE OR WRITE_XFER OR CH0_SEL);
  17   2       OUTPUT(MODE_REG_37) = (SINGLE_MODE OR READ_XFER OR CH1_SEL);
  18   2       OUTPUT(CLR_BYTE_PTR_37) = 0;
  19   2       OUTPUT(CH0_ADDR) = 00;       /* RECEIVE BUFF AT 900H */
  20   2       OUTPUT(CH0_ADDR) = 09H;
  21   2       OUTPUT(CH0_COUNT) = 0H;
  22   2       OUTPUT(CH0_COUNT) = 01;
  23   2       OUTPUT(CH1_ADDR) = 00;       /* TRANSMIT BUFF AT 800H */
  24   2       OUTPUT(CH1_ADDR) = 08H;
  25   2       OUTPUT(CH1_COUNT) = 010H;
  26   2       OUTPUT(CH1_COUNT) = 00H;
```

210403-18

```
                /* ENABLE TRANSFER */
 27   2         OUTPUT(SINGLE_MASK) = CH1_SEL;     /* ENABLE TX DMA */
 28   2         RETURN;

 29   2         END INIT_8237_S;

                /* TURN OFF THE 8237 CHANNELS 0 AND 1 */

 30   1         STOP_8237_S: PROCEDURE PUBLIC;
 31   2         OUTPUT(SINGLE_MASK) = CH1_SEL OR SET_MASK;
 32   2         OUTPUT(SINGLE_MASK) = CH0_SEL OR SET_MASK;
 33   2         RETURN;
 34   2         END STOP_8237_S;
 35   1         END INIT_8237_CHA;


MODULE INFORMATION:

     CODE AREA SIZE      = 004CH     76D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D




PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


     MAXIMUM STACK SIZE = 0002H      2D
     163 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION


PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE INTR_8274_S
OBJECT MODULE PLACED IN :F1:SINTR.OBJ
COMPILER INVOKED BY:  PLM86 86 :F1:SINTR.PLM TITLE(iSBC 88/45 8274 CHANNEL
A SDLC TEST) COMPACT NOINTVECTOR ROM

                /*****************************************/
                /*                                       */
                /*      8274 INTERRUPT ROUTINE           */
                /*                                       */
                /*****************************************/

                INTR_8274_S  DO;
                $NOLIST
  12   1        DECLARE TEMP BYTE;
  13   1        DECLARE (RESULTS_S,TXDONE_S,RXDONE_S) BYTE EXTERNAL;
  14   1        DECLARE INT_VEC POINTER AT (140);
  15   1        DECLARE INT_VEC_STORE POINTER;
  16   1        DECLARE MASK_59 BYTE;
  17   1        DECLARE DONE            LIT     'OFFH',
                        NOT_DONE        LIT     '00H',
                        PASS            LIT     'OFFH',
                        FAIL            LIT     '00H';

                /*****************************/
                /* IGNORE INTERRUPT HANDLER */
                /*****************************/

  18   1        IGNORE_INT:  PROCEDURE;

  19   2        RESULTS_S = FAIL;
  20   2        RETURN;
  21   2        END IGNORE_INT;
```

210403-19

```
                /*******************************************************/
                /* CHANNEL A EXTERNAL/STATUS CHANGE INTERRUPT HANDLER */
                /*******************************************************/

  22    1       CHA_EXTERNAL_CHANGE: PROCEDURE;

  23    2       TEMP = INPUT(STATUS_A_74);          /* STATUS REG 1*/
  24    2       IF (TEMP AND END_OF_TX_MESSAGE) = END_OF_TX_MESSAGE THEN
  25    2           TXDONE_S=DONE;
  26    2       ELSE DO;
  27    3            TXDONE_S=DONE;
  28    3            RESULTS_S=FAIL;
  29    3            END;
  30    2       OUTPUT(COMMAND_A_74) = 10H; /* RESET EXT/STATUS INTERRUPTS */
  31    2       RETURN;
  32    2       END CHA_EXTERNAL_CHANGE;

                $EJECT

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


                /*******************************************************/
                /* CHANNEL A SPECIAL RECEIVE CONDITIONS INTERRUPT HANDLER */
                /*******************************************************/

  33    1       CHA_RX_SPECIAL: PROCEDURE;

  34    2           OUTPUT(COMMAND_A_74) = 1;
  35    2           TEMP = INPUT(STATUS_A_74);
  36    2           IF (TEMP AND END_OF_FRAME) = END_OF_FRAME THEN
  37    2                       DO;
  38    3                         IF(TEMP AND 040H) = 040H THEN
  39    3                           RESULTS_S = FAIL;          /* CRC ERROR */
  40    3                         RXDONE_S = DONE;
  41    3                         OUTPUT(COMMAND_A_74) = 30H; /*ERROR RESET*/
  42    3                       END;
  43    2                       ELSE DO;
  44    3                         IF (TEMP AND 20H) = 20H THEN DO;
  46    4                         RESULTS_S = FAIL;           /* RX OVERRUN ERROR*/
  47    4                         RXDONE_S = DONE;
  48    4                         OUTPUT(COMMAND_A_74) = 30H; /*ERROR RESET*/
  49    4                         END;
  50    3                       END;
  51    2       RETURN;
  52    2       END CHA_RX_SPECIAL;


                /***************************************/
                /* CHANNEL A RECEIVE CHARACTER AVAILABLE */
                /***************************************/

  53    1       CHA_RX_CHAR: PROCEDURE;
  54    2       OUTPUT(SINGLE_MASK) = CH0_SEL;          /*ENABLE RX DMA CHANNEL*/
  55    2       RETURN;
  56    2       END CHA_RX_CHAR;

                $EJECT


PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST

                /* ENABLE 8274 INTERRUPTS - SET UP THE 8259A */

  57    1       ENABLE_INTERRUPTS_S: PROCEDURE PUBLIC;

  58    2       DECLARE CHA_INT_ON LIT '0F7H';

  59    2       DISABLE;

  60    2       CALL    SET$INTERRUPT(39, INT_39);
```

210403-20

```
 61   2      INT_VEC_STORE = INT_VEC;
 62   2      INT_VEC = INTERRUPT$PTR(INT_8274_S);
 63   2      MASK_59 = INPUT(OCW1_59);

 64   2      OUTPUT(OCW1_59) = MASK_59 AND CHA_INT_ON;

 65   2      RETURN;
 66   2      END ENABLE_INTERRUPTS_S;


              /* DISABLE 8274 INTERRUPTS - SET UP THE 8259A */

 67   1      DISABLE_INTERRUPTS_S: PROCEDURE PUBLIC;

 68   2      DISABLE;

 69   2      INT_VEC = INT_VEC_STORE;

 70   2      OUTPUT(OCW1_59) = MASK_59;
 71   2      ENABLE;

 72   2      RETURN;
 73   2      END DISABLE_INTERRUPTS_S;


              /* CHANNEL B RECEIVE CHARACTER AVAILABLE */

 74   1      CHB_RX_CHAR: PROCEDURE;

 75   2      TEMP=INPUT(DATA_B_74);

 76   2      OUTPUT(COMMAND_B_74) = 38H;
 77   2      RETURN;
 78   2      END CHB_RX_CHAR;


              $EJECT

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


              /*************************/
              /* MAIN INTERRUPT ROUTINE */
              /*************************/

 79   1      INT_8274_S: PROCEDURE INTERRUPT 35 PUBLIC;

 80   2      OUTPUT(COMMAND_B_74) = 2;                  /* SET POINTER TO 2*/
 81   2      TEMP = INPUT(STATUS_B_74) AND 07H;         /* READ INTERRUPT VECTOR */
                                                        /* CHECK FOR CHA INT ONLY*/
              /* FOR THIS APPLICATION CH B INTERRUPTS ARE IGNORED*/
 82   2      DO CASE TEMP;
 83   3          CALL     IGNORE_INT;                  /* V2V1V0 = 000*/
 84   3          CALL     IGNORE_INT;                  /* V2V1V0 = 001*/
 85   3          CALL     CHB_RX_CHAR;                 /* V2V1V0 = 010*/
 86   3          CALL     IGNORE_INT;                  /* V2V1V0 = 011*/
 87   3          CALL     IGNORE_INT;                  /* V2V1V0 = 100*/
 88   3          CALL     CHA_EXTERNAL_CHANGE;         /* V2V1V0 = 101*/
 89   3          CALL     CHA_RX_CHAR;                 /* V2V1V0 = 110*/
 90   3          CALL     CHA_RX_SPECIAL;              /* V2V1V0 = 111*/
 91   3      END;
 92   2      OUTPUT(COMMAND_A_74) =38H;   /* END OF INTERRUPT FOR 8274 */
 93   2      OUTPUT(OCW2_59) = 63H;       /* 8259 EOI */
 94   2      OUTPUT(OCW1_59) = INPUT(OCW1_59) AND 0F7H;
 95   2      RETURN;
 96   2      END INT_8274_S;



              /* DEFAULT INTERRUPT ROUTINE - 8259A INTERRUPT 7 */
              /* REQUIRED ONLY WHEN DMA CONTROLLER IS ENABLED  */
              /* BEFORE RECEIVING FIRST CHARACTER WHICH IS     */
              /* AT HIGH BAUD RATES  LIKE BOOK BAUD. READ APP  */
              /* NOTE SECTION 6 FOR DETAILS                    */
```

210403-21

```
 97   1       INT_39: PROCEDURE INTERRUPT 39;
 98   2               OUTPUT(OCW2_59) = 20H;    /* NON-SPECIFIC EOI */
 99   2               OUTPUT(OCW1_59) = INPUT(OCW1_59) AND OF7H;
100   2               RESULTS_S = FAIL;
101   2       END INT_39;

102   1       END INTR_8274_S;


MODULE INFORMATION:

     CODE AREA SIZE     = 01BFH    447D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0006H      6D
     MAXIMUM STACK SIZE = 0022H     34D
     295 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE STEST
OBJECT MODULE PLACED IN :F1:STEST.OBJ
COMPILER INVOKED BY:  PLM86.86 :F1:STEST.PLM TITLE(iSBC 88/45 8274 CHANNEL A SDLC TEST)
COMPACT NOINTVECTOR ROM

              /*************************************************************************/
              /*                                                                       */
              /*       iSBC 545 PORT A (8274) SDLC TEST                                */
              /*                                                                       */
              /*************************************************************************/

  1           STEST: DO;


  2   1       DELAY_S: PROCEDURE EXTERNAL;
  3   2       END DELAY_S;

  4   1       ENABLE_INTERRUPTS_S: PROCEDURE EXTERNAL;
  5   2       END ENABLE_INTERRUPTS_S;

  6   1       DISABLE_INTERRUPTS_S: PROCEDURE EXTERNAL;
  7   2       END DISABLE_INTERRUPTS_S;

  8   1       INIT_8274_SDLC_S: PROCEDURE EXTERNAL;
  9   2       END INIT_8274_SDLC_S;

 10   1       INIT_8237_S: PROCEDURE EXTERNAL;
 11   2       END INIT_8237_S;

 12   1       STOP_8237_S: PROCEDURE EXTERNAL;
 13   2       END STOP_8237_S;

 14   1       VERIFY_TRANSFER_S: PROCEDURE EXTERNAL;
 15   2       END VERIFY_TRANSFER_S;

 16   1       INT_8274_S: PROCEDURE INTERRUPT 35 EXTERNAL;
 17   2       END INT_8274_S;
              $NOLIST
              $EJECT

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


 28   1       DECLARE (RESULTS_S,TXDONE_S,RXDONE_S) BYTE PUBLIC;
 29   1       DECLARE DONE            LIT     'OFFH',
                      NOT_DONE        LIT     '00H',
                      PASS            LIT     'OFFH',
                      FAIL            LIT     '00H';
```

210403-22

```
            $EJECT

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


  30   1       CHA_SDLC_TEST: PROCEDURE BYTE PUBLIC;


  31   2               CALL     ENABLE_INTERRUPTS_S;
  32   2               CALL     INIT_8274_SDLC_S;
  33   2               ENABLE;
  34   2               OUTPUT(COMMAND_A_74) = 28H;   /* RESET TX INT/DMA          */
  35   2               OUTPUT(COMMAND_B_74) = 28H;   /* BEFORE INITIALIZING 8237*/
  36   2               CALL     INIT_8237_S;
  37   2               OUTPUT(DATA_A_74) = 55H;   /* LOAD FIRST CHARACTER FROM CPU*/

                   /* TO ENSURE CRC TRANSMISSION RESET TX UNDERRUN LATCH*/
  38   2               OUTPUT(COMMAND_A_74) = 0C0H;
  39   2               RXDONE_S, TXDONE_S=NOT_DONE;          /* CLEAR ALL FLAGS */
  40   2               RESULTS_S=PASS;                       /* FLAG SET FOR MONITOR*/

  41   2               DO WHILE TXDONE_S=NOT_DONE;           /* DO UNTIL TERMINAL COUNT*/
  42   3               END;

  43   2               DO WHILE(INPUT(STATUS_A_74) AND 04H) <> 04H;
                   /* WAIT FOR CRC TO GET TRANSMITTED */
                   /* TEST FOR TX BUFFFER EMPTY TO VERIFY THIS*/
  44   3               END;
  45   2               DO WHILE RXDONE_S=NOT_DONE;           /* DO UNTIL TERMINAL COUNT*/
  46   3               END;

  47   2               CALL     STOP_8237_S;

  48   2               CALL     DISABLE_INTERRUPTS_S;

  49   2               CALL     VERIFY_TRANSFER_S;

  50   2               RETURN RESULTS_S;

  51   2       END CHA_SDLC_TEST;
  52   1       END STEST;


MODULE INFORMATION:

     CODE AREA SIZE     = 0063H     99D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0003H      3D
     MAXIMUM STACK SIZE = 0004H      4D
     198 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION


PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE VECTOR_MODE
OBJECT MODULE PLACED IN :F1:VECTOR.OBJ
COMPILER INVOKED BY:  PLM86.86 :F1:VECTOR.PLM TITLE(iSBC 88/45 8274 CHANNEL A SDLC TEST)


             /*****************************************************************/
             /*                                                               */
             /*            8274 INTERRUPT HANDLING ROUTINE FOR                */
             /*                  8274 VECTOR MODE                             */
             /*               STATUS AFFECTS VECTOR                           */
             /*                                                               */
             /*****************************************************************/
```

210403-23

```
                /* THIS IS AN EXAMPLE OF HOW 8274 CAN BE USED IN VECTORED MODE.      */
                /* THE iSBC88/45 BOARD WAS REWIRED TO DISABLE THE PIT 8259A AND      */
                /* ENABLE THE 8274 TO PLACE ITS VECTOR ON THE DATABUS IN RESPONSE    */
                /* TO THE INTA SEQUENCE FROM THE 8088.  OTHER MODIFICATIONS INCLUDED */
                /* CHANGES TO 8274 INITIALIZATION PROGRAM (SINI74) TO PROGRAM 8274   */
                /* INTO VECTORED MODE (WRITE REGISTER 2A D5=1).                      */


 1              VECTOR_MODE: DO;
                $NOLIST

12   1          DECLARE TEMP BYTE;
13   1          DECLARE (RESULTS_S,TXDONE,RXDONE) BYTE EXTERNAL;
14   1          DECLARE DONE  LITERALLY 'OFFH',
                           NOT_DONE LITERALLY '00H',
                           PASS     LITERALLY 'OFFH',
                           FAIL     LITERALLY '00H';

                /*************************************************************/
                /*      TRANSMIT INTERRUPT CHANNEL A INTERRUPT WILL NOT BE SEEN IN THE */
                /*      DMA OPERATION.                                       */
                /*************************************************************/

15   1              TX_INTERRUPT_CHA:PROCEDURE INTERRUPT 84;
16   2              OUTPUT(COMMAND_A_74) = 00101000B;         /*RESET TXINT PENDING*/
17   2              OUTPUT(COMMAND_A_74) = 00111000B;         /*EOI*/
18   2              END TX_INTERRUPT_CHA;


                /*************************************************************/
                /*      EXTERNAL/STATUS INTERRUPT PROCEDURE:  CHECKS FOR END OF MESSAGE */
                /*      ONLY.  IF THIS IS NOT TRUE THEN THE FAIL FLAG IS SET.  HOWEVER,  */
                /*      A USER PROGRAM SHOULD CHECK FOR OTHER EXT/STATUS CONDITIONS      */
                /*      ALSO IN RR1 AND THEN TAKE APPROPRIATE ACTION BASED ON THE        */
                /*      APPLICATION.                                         */
                /*************************************************************/

19   1          EXT_STAT_CHANGE_CHA:PROCEDURE INTERRUPT 85;
20   2                  TEMP  = INPUT(STATUS_A_74);
21   2                  IF (TEMP AND END_OF_TX_MESSAGE) = END_OF_TX_MESSAGE THEN
22   2                     TXDONE = DONE;
23   2                  ELSE DO;
24   3                     TXDONE = DONE;

PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


25   3                        RESULTS_S = FAIL;
26   3                     END;

27   2               OUTPUT(COMMAND_A_74) = 00010000B;         /*RESET EXT STAT INT*/
28   2                   OUTPUT(COMMAND_A_74) = 00111000B;         /*EOI*/
29   2                   RETURN;
30   2          END EXT_STAT_CHANGE_CHA;

                /*************************************************************/
                /*      RECEIVER CHARACTER AVAILABLE INTERRUPT WILL APPEAR ONLY ON FIRST*/
                /*      RECEIVE CHARACTER.  SINCE DMA CONTROLLER HAS BEEN ENABLED BEFORE */
                /*      THE FIRST CHARACTER IS RECEIVED, THE RECEIVER REQUEST IS         */
                /*      SERVICED BY THE DMA CONTROLLER.                      */
                /*************************************************************/

31   1          RX_CHAR_AVAILABLE_CHA:PROCEDURE INTERRUPT 86;
32   2               OUTPUT(COMMAND_A_74) = 00111000B;         /*EOI*/
33   2                  RETURN;
34   2          END RX_CHAR_AVAILABLE_CHA;
                $EJECT
```

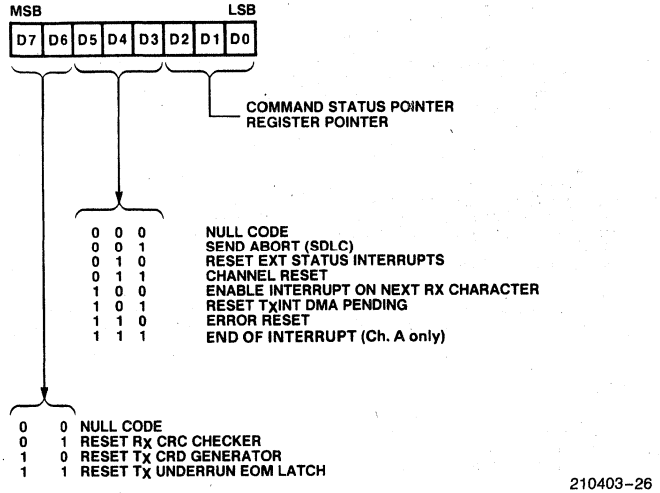
210403-24

```
PL/M-86 COMPILER     iSBC 88/45 8274 CHANNEL A SDLC TEST


                /************************************************************************/
                /*      SPECIAL RECEIVE CONDITION INTERRUPT SERVICE ROUTINE CHECKS FOR */
                /*      END OF FRAME BIT ONLY.  SEE SPECIAL SERVICE ROUTINE FOR NON-    */
                /*      VECTORED MODE FOR CRC CHECK AND OVERRUN ERROR CHECK             */
                /************************************************************************/

  35    1       SPECIAL_RX_CONDITION_CHA: PROCEDURE INTERRUPT 87;

  36    2               OUTPUT(COMMAND_A_74) = 1;               /*POINTER 1*/
  37    2               TEMP = INPUT(STATUS_A_74);
  38    2               IF (TEMP AND END_OF_FRAME) = END_OF_FRAME THEN
  39    2                     RXDONE = DONE;
  40    2               ELSE DO;
  41    3                     RXDONE = DONE;
  42    3                     RESULTS_S = FAIL;
  43    3                    END;
  44    2           OUTPUT(COMMAND_A_74) = 00110000B;        /*ERROR RESET*/
  45    2           OUTPUT(COMMAND_A_74) = 00111000B;        /*EOI*/
  46    2               RETURN;
  47    2       END SPECIAL_RX_CONDITION_CHA;


  48    1       ENABLE_INTERRUPTS: PROCEDURE PUBLIC;
  49    2       DISABLE;
  50    2       CALL SET$INTERRUPT(84, TX_INTERRUPT_CHA);
  51    2       CALL SET$INTERRUPT(85, EXT_STAT_CHANGE_CHA);
  52    2       CALL SET$INTERRUPT(86, RX_CHAR_AVAILABLE_CHA);
  53    2       CALL SET$INTERRUPT(87, SPECIAL_RX_CONDITION_CHA);
  54    2       RETURN;
  55    2       END ENABLE_INTERRUPTS;

  56    1       END VECTOR_MODE;
                /************************************************************************/
                /************************************************************************/


MODULE INFORMATION:

     CODE AREA SIZE     = 012EH    302D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0001H      1D
     MAXIMUM STACK SIZE = 001EH     30D
     226 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

210403-25

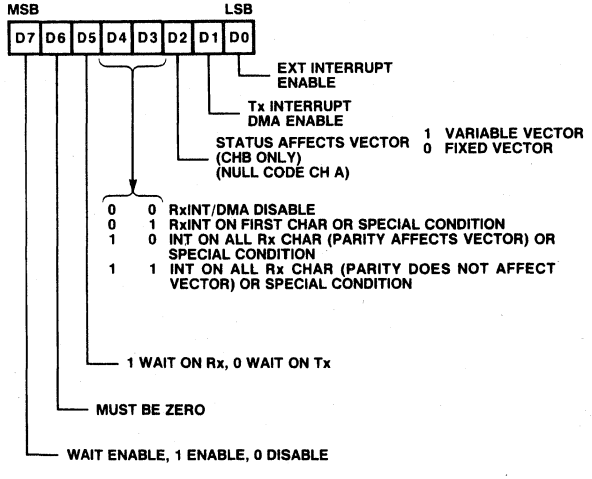
APPENDIX B
MPSC READ/WRITE REGISTER DESCRIPTIONS

WRITE REGISTER 0 (WR0)



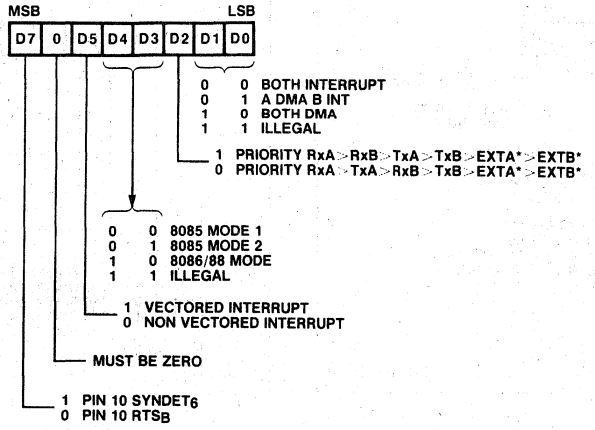
210403-26

WRITE REGISTER 1 (WR1)



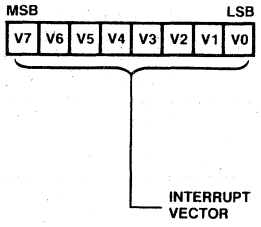
210403-27

WRITE REGISTER 2 (WR2): CHANNEL A

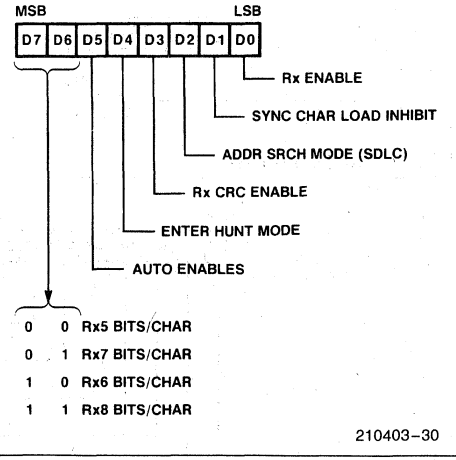


*External Status Interrupt only if EXT Interrupt Enable (WR1: D0) is set.

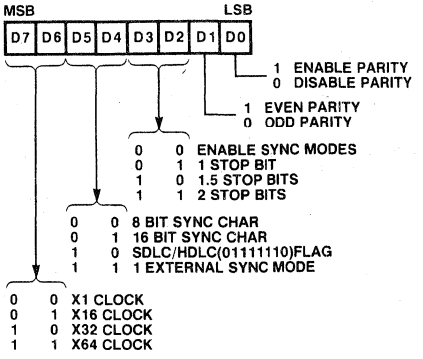
WRITE REGISTER 2 (WR2): CHANNEL B



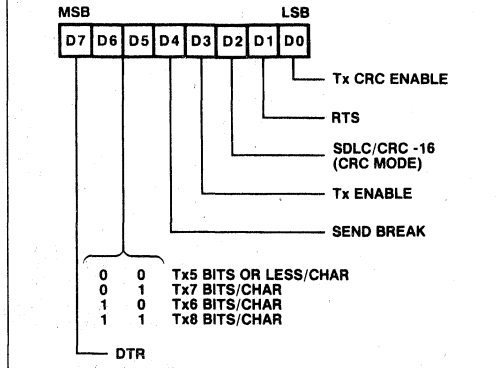
WRITE REGISTER 3 (WR3)



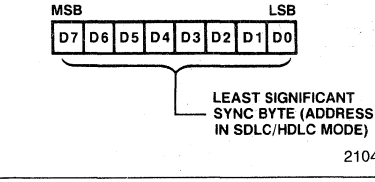
WRITE REGISTER 4 (WR4)



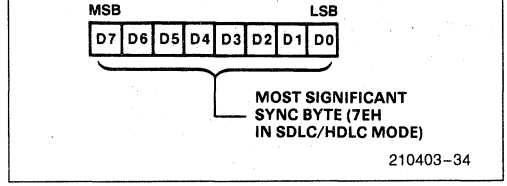
WRITE REGISTER 5 (WR5)



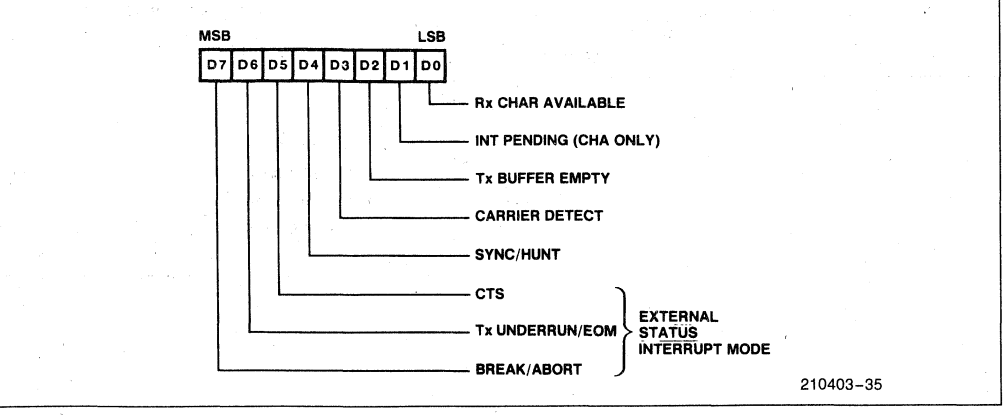
WRITE REGISTER 6 (WR6)



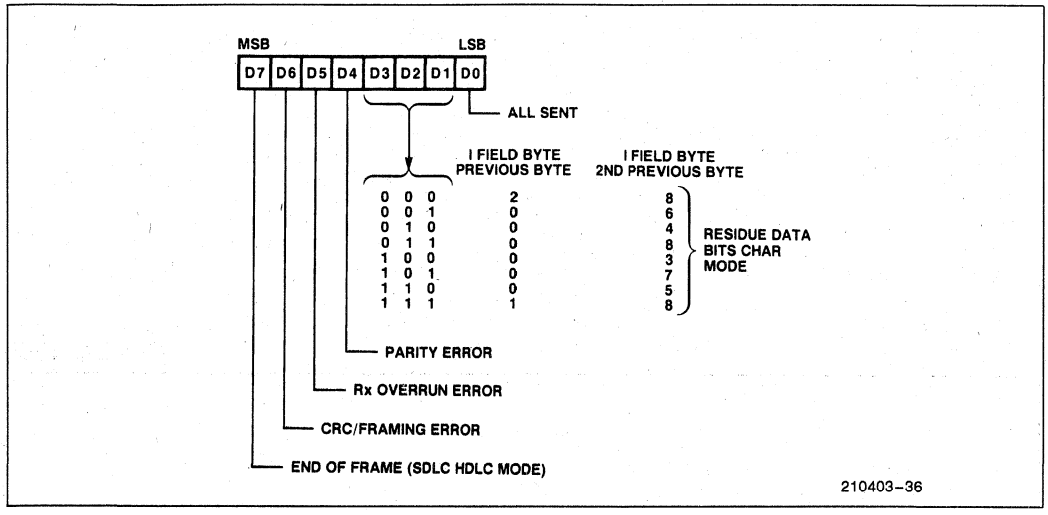
WRITE REGISTER (WR7)



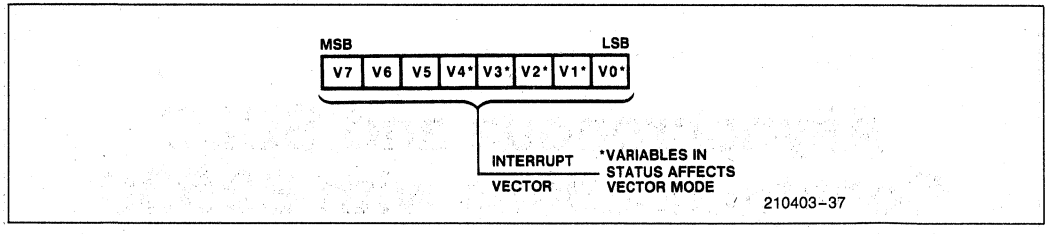
READ REGISTER 0 (RR0)



READ REGISTER 1 (RR1): (SPECIAL RECEIVE CONDITION MODE)



READ REGISTER 2 (RR2) CHANNEL B ONLY



REFERENCES

1. *IBM Document No. GA27-3004-2: General Information—Binary Synchronous Communications*
2. *Application Note AP134: Asynchronous Communication with the 8274 Multiple Protocol Serial Controller.* Intel Corp., Ca.
3. *8274 MPSC Data Sheet,* Intel Corporation, Ca.
4. *iSBC 88/45 Hardware Reference Manual,* Intel Corp., Ca.
5. *Computer Networks and Distributed Processing by James Martin.* Prentice Hall, Inc., N.J.

intel®

APPLICATION NOTE

AP-222

October 1989

Asynchronous and SDLC Communications with 82530

DFG TECHNICAL MARKETING

Order Number: 231262-004

INTRODUCTION

INTEL's 82530, Serial Communications Controller (SCC), is a dual channel, multi-protocol data communications peripheral. It is designed to interface to high speed communications lines using asynchronous, byte synchronous, and bit synchronous protocols. It runs up to 1.5 Mbits/sec, has on-chip baud rate generators and on-chip NRZI encoding and decoding circuits—very useful for SDLC communication. This application note shows how to write I/O drivers for the 82530 to do initialization and data links using asynchronous (ASYNC) and SDLC protocols. The appendix includes sections to show how the on-chip baud rate generators could be programmed, how the modem control pins could be used, and how the 82530 could be interfaced to INTEL's 80186/188 processors.

This article deals with the software for the following:

1. SCC port definition
2. Accessing the SCC registers
3. Initialization for ASYNC communication
4. ASYNC communication in polling mode
5. ASYNC communication in interrupt mode
6. Initialization for SDLC communication
7. SDLC frame reception
8. SDLC frame transmission
9. SDLC interrupt routines

The description is written around illustrations of the actual software written in PLM86 for a 80186 - 82530 system.

I. SCC Port Definition

The Figure 1 shows how the 4 ports (2 per channel) of the SCC can be defined. Note that the sequence of ports in the ascending order of addresses is *not* the one that is normally expected. In the ascending order it is: command (B), data (B), command (A) and data (A). In an 80186 - 82530 system, the interconnection is as follows:

	PCSn	—	CS	
	A1	—	D/C	
80186 pins	A2	—	A/B	82530 pins
	RD	—	RD	
	WR	—	WR	

2. Accessing the SCC Registers

The SCC has 16 registers on each of the channels (A and B). For each channel there is only one port, the command port, to access all the registers. The register #0 can be always accessed directly through the command port. All other registers are accessed indirectly through register #0. First, the number of the register to be accessed is written to the register #0 - see the statement, in Figure 2: 'output (ch_a_command) = reg_ no and 0fh'. Then, the desired register is written to or read out. The Figure 2 shows 4 procedures: rra and wra, for reading and writing channel A registers; rrb and wrb, for reading and writing channel B registers. The read procedures are of the type 'byte' - they return the contents of the register being read. The write procedures require two parameters - the register number and the value to be written.

```
/*-------------------------------------------------------------------------*/

declare ch_b_command   literally 'pcs5 + 0', /* scc channel_b command word*/
        ch_b_data      literally 'pcs5 + 2', /* scc channel_b data word */
        ch_a_command   literally 'pcs5 + 4', /* scc channel_a command word */
        ch_a_data      literally 'pcs5 + 6'; /* scc channel_a data word */

/*-------------------------------------------------------------------------*/
                                                              231262-1
```

Figure 1. SCC Port Definition

```
/*-------------------------------------------------------------------------*/

/* read selected scc register */

rra: procedure (reg_no) byte;
     declare reg_no byte;

     if (reg_no and 0fh) <> 0
     then output(ch_a_command) = reg_no and 0fh;
     return input(ch_a_command);
end rra;

rrb: procedure (reg_no) byte;
     declare reg_no byte;

     if (reg_no and 0fh) <> 0
     then output (ch_b_command) = reg_no and 0fh;
     return input(ch_b_command);
end rrb;


/* write selected scc register */

wra: procedure (reg_no, value);
     declare reg_no byte;
     declare value byte;

     if (reg_no and 0fh) <> 0
     then output (ch_a_command) = reg_no and 0fh;
     output (ch_a_command) = value;
end wra;

wrb: procedure (reg_no, value);
     declare reg_no byte;
     declare value byte;

     if (reg_no and 0fh) <> 0
     then output (ch_b_command) = reg_no and 0fh;
     output (ch_b_command) = value;
end wrb;

/*-------------------------------------------------------------------------*/
```

231262-2

Figure 2. Accessing the SCC Registers

3. Initialization for ASYNC Operation

In the following example, channel B of the SCC is used to perform ASYNC communication. Figure 3 shows how the channel B is initialized and configured for ASYNC operation. This is done by writing the various channel B registers with the proper parameters as shown. The comments in the program show what is achieved by each statement. After a software reset of the channel, register #4 should be written before writing to the other registers. The on-chip Baud Rate Generator is used to generate a 1200 bits/sec clock for both the transmitter and the receiver. The interrupts for transmitter and/or receiver are enabled only for the interrupt mode of operation; for polling, interrupts must be kept disabled.

4. ASYNC Communication in Polling Mode

Figure 4 shows the procedures for reading in a received character from the 82530 (scc__in) and for writing out a character to the 82530 (scc__out) in the polling mode.

The scc__in procedure returns a byte value which is the character read in. The receiver is polled to find if a character has been received by the SCC. Only when a character has been received, the character is read in from the data port of the SCC channel B.

The scc__out procedure requires a byte parameter which is the character being written out. The transmit-

```
/*-------------------------------------------------------------------------*/

scc_init_b: procedure;

/* scc ch B register initialization for ASYNC mode */

     call wrb(09, 01000000b);     /* channel B reset */
     call wrb(04, 11001110b);     /* 2 stop, no parity, brf = 64x */
     call wrb(02, 00100000b);     /* vector = 20h */
     call wrb(03, 11000000b);     /* rx 8 bits/char, no auto-enable */
     call wrb(05, 01100000b);     /* tx 8 bits/char */
     call wrb(06, 00000000b);
     call wrb(07, 00000000b);
     call wrb(09, 00000001b);     /* vector includes status */
     call wrb(10, 00000000b);
     call wrb(11, 01010110b);     /* rxc = txc = BRG , trxc = BRG out */
     call wrb(12, 00011000b);     /* to generate 1200 baud, x64 @ 4 mhz */
     call wrb(13, 00000000b);
     call wrb(14, 00000011b);     /* BRG source = SYS CLK, enable BRG */
     call wrb(15, 00000000b);     /* all ext status interrupts off */

    /* enables */

     call wrb(03, 11000001b);      /* scc-b receive enable */
     call wrb(05, 11101010b);      /* scc-b transmit enable, dtr on, rts on */

    /* enable interrupts - only for interrupt driven ASYNC I/O */

     call wrb(09, 00001001b);     /* master IE, vector includes status */
     call wrb(01, 00010011b);     /* tx ,rx, ext interrupts enable */

end scc_init_b;

/*-------------------------------------------------------------------------*/
                                                              231262-3
```

Figure 3. Initialization for ASYNC Communication

```
/*-------------------------------------------------------------------------*/

/* scc data character input from channel B */

scc_in: procedure byte;

     declare char byte;

     do while (input(ch_b_command) and 1h) = 0; end;
     char = input(ch_b_data);    /* if rx data character is available */
     return char;                /* then input it to buffer */

end scc_in;

/* scc data character output to channel B */

scc_out: procedure (char);

     declare char byte;

     do while (input(ch_b_command) and 4h) = 0; end;
     output(ch_b_data) = char;   /* if tx buff empty then transfer the */
                                 /* data character to tx buff */

end scc_out;

/*-------------------------------------------------------------------------*/
```

231262-4

Figure 4. ASYNC Communication in Polling Mode

ter is polled for being ready to transmit the next character before writing the character out to the data port of SCC channel B.

Typical calls to these procedures are:

```
abc_variable = scc_in;
call scc_out (xyz_variable);
```

5. ASYNC Communication in Interrupt Mode

In contrast to polling for the receiver and/or the transmitter to be ready with/for the next character, the 82530 can be made to interrupt when it is ready to do receive or transmit.

The on-chip interrupt controller of the SCC can be made to operate in the vectored mode. In this mode, it generates interrupt vectors that are characteristic of the event causing the interrupt. For the example here, the vector base is programmed at 20h and 'Vector Includes Status' (VIS) mode is set - WR9 = XXX0XX01. Vectors and the associated events are:

Vector	Procedure	Event Causing Interrupt
20h	txintr_b	ch_b - transmit buffer empty
22h	esi_b	ch_b - external/status change
24h	rxintr_b	ch_b - receive character available
26h	src_b	ch_b - special receive condition
28h	txintr_a	ch_a - transmit buffer empty
2ah	esi_a	ch_a - external/status change
2ch	rxintr_a	ch_a - receive character available
2eh	src_a	ch_a - special receive condition

NOTE:
Odd vector numbers do not exist.

Figure 5 shows the interrupt procedures for the channel B operating in ASYNC mode. The transmitter buffer empty interrupt occurs when the transmitter can accept one more character to output. In the interrupt procedure for transmit, the byte char_out_530 is output. Following this, is an epiloge that is *common to all the*

interrupt procedures; the first statement is an end of interrupt command to the 82530 - *note* that it is issued to *channel A* - and the second is an End of Interrupt (EOI) command to the 80186 interrupt controller which is, in fact, receiving the interrupt from the 82530.

The receive buffer full interrupt occurs when the receiver has at least one character in its buffer, waiting to be read in by the CPU.

The esi_b is not enabled to occur and src_b cannot occur in the ASYNC mode unless the receiver is overrun or a parity error occurs.

```
/*-------------------------------------------------------------------------*/

/* channel B interrupt procedures */

txintr_b:     procedure     interrupt 20h;

    output (ch_b_data) = char_out_530;

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end txintr_b;

esi_b:        procedure     interrupt 22h;

    call wrb(00,10h);               /* reset ESI */

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end esi_b;

rxintr_b:     procedure     interrupt 24h;

    char_in_530 = input (ch_b_data);

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end rxintr_b;

src_b:        procedure     interrupt 26h;

    call wrb(00,30h);               /* error reset */

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end src_b;

/*-------------------------------------------------------------------------*/
```

231262-5

Figure 5. ASYNC Communication in Interrupt Mode

6. Initialization for SDLC Communication

Channel A of the SCC is programmed for being used for SDLC operation. It uses the DMA channels on the 80186. Figure 6 shows the initialization procedure for channel A. The comments in the software show the effect of each statement. The on-chip Baud Rate Generator is used to generate a clock of 125 kHz both for reception and transmission. This procedure is just to prepare the channel A for SDLC operation. The actual transmission and reception of frames is done using the procedures described further.

7. SDLC Frame Reception

Figure 7 shows the entire set-up necessary to receive a SDLC frame. First the DMA controller is programmed with the receive buffer address (@rx_buff), byte count, mode etc and is also enabled. Then a flag indicating reception of the frame is reset. An Error Reset command is issued to clear up any pending error conditions. The receive interrupt is enabled to occur at the end of frame reception (Special Receive Condition); lastly, the receiver is enabled and put in the Hunt mode (to detect the SDLC flag). When the first flag is detected on the RxDA pin, it goes from the Hunt to the Sync mode. It receives the frame and the end of frame interrupt (src_b, vector = 2eh) occurs.

8. SDLC Frame Transmission

Figure 8 shows the procedure for transmitting a SDLC frame once channel A is initialized. The DMA controller is initialized with the transmit buffer address (@tx_buff (1)) - *note*, it is the second byte of the transmit buffer - and the byte count - again one less than the total buffer length. This is done because the first byte in the buffer is output directly using an I/O instruction and not by DMA. Then the flag indicating frame transmitted is reset. The events following are very critical in sequence:

a. Reset external status interrupts

b. Enable the transmitter

c. Reset transmit CRC

d. Enable transmitter underrun interrupt

e. Enable the DMA controller

f. Output first byte of the transmit block to data port

g. Reset Transmit Underrun Latch

```
/*-------------------------------------------------------------------------*/

scc_init_a: procedure;

/* scc ch A register initialization for SDLC mode */

     call wra(09, 10000000b);     /* channel A reset */
     call wra(04, 00100000b);     /* SDLC mode */
     call wra(01, 01100000b);     /* DMA for Rx */
     call wra(03, 11000000b);     /* 8 bit Rx char, Rx disable */
     call wra(05, 01100000b);     /* 8 bit Tx char, Tx disable */
     call wra(06, 01010101b);     /* node address */
     call wra(07, 01111110b);     /* SDLC flag */
     call wra(10, 10000000b);     /* preset CRC, NRZ encoding */
     call wra(11, 01010110b);     /* rxc = txc = BRG , trxc = BRG out */
     call wra(12, 00001110b);     /* to generate 125 Kbaud, x1 @ 4 mhz */
     call wra(13, 00000000b);
     call wra(14, 00000110b);     /* BRG source = SYS CLK, DMA for Tx */
     call wra(15, 00000000b);     /* all ext status interrupts off */

    /* enables */

     call wra(14, 00000111b);      /* enable : BRG */
     call wra(01, 11100000b);      /* enable : dreq */
     call wra(09, 00001001b);      /* master IE, vector includes status */

end scc_init_a;

/*-------------------------------------------------------------------------*/
                                                              231262-6
```

Figure 6. Initialization for SDLC Communication

```
/*-------------------------------------------------------------------------*/

rx_init: procedure;

     declare dma_0_mode literally '1010001001000000b';
     /* src=IO, dest=M(inc), sync=src, TC, noint, priority, byte */

     outword(dma_0_dpl) = low16(@rx_buff);
     outword(dma_0_dph) = high16(@rx_buff);
     outword(dma_0_spl) = ch_a_data;
     outword(dma_0_sph) = 0;
     outword(dma_0_tc)  = block_length + 2;        /* +2 for CRC */
     outword(dma_0_cw)  = dma_0_mode or 0006h;     /* start DMA channel 0 */

     frame_recd = 0;                 /* reset frame received flag */

     call wra(00,30h);               /* error reset */
     call wra(01, 11111001b);        /* sp. cond intr only, ext int enable */
     call wra(03, 11010001b);        /* enable receiver, enter hunt mode */

end rx_init;

/*-------------------------------------------------------------------------*/
                                                              231262-7
```

Figure 7. SDLC-DMA Frame Reception

```
/*-------------------------------------------------------------------------*/

tx_init: procedure;

     declare dma_1_mode literally '0001011010000000b';
     /* src=M(inc), dest=IO, sync=dest, TC, noint, noprior, byte */

     outword(dma_1_spl) = low16(@tx_buff(1));
     outword(dma_1_sph) = high16(@tx_buff(1));
     outword(dma_1_dpl) = ch_a_data;
     outword(dma_1_dph) = 0;
     outword(dma_1_tc)  = block_length - 1;    /* -1 for first byte */

     frame_tx = 0;                   /* reset frame transmitted flag */

     call wra(00, 00010000b);        /* reset ESI */
     call wra(05, 01101011b);        /* enable transmitter */
     call wra(00, 10101000b);        /* reset tx CRC, TxINT pending */
     call wra(15, 01000000b);        /* enable : TxU int */

     outword(dma_1_cw)  = dma_1_mode or 0006h;    /* start DMA channel 1 */
     output(ch_a_data) = tx_buff(0);   /* first byte - address field */
     call wra(00, 11000000b);          /* Reset Tx Underrun latch */

end tx_init;

/*-------------------------------------------------------------------------*/
                                                              231262-8
```

Figure 8. SDLC-DMA Frame Transmission

```
/*-------------------------------------------------------------------------*/

/* channel A interrupt procedures */

txintr_a:     procedure    interrupt 28h;

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end txintr_a;

esi_a:        procedure    interrupt 2ah;

    call wra(00,10h);               /* reset ESI */
    tx_stat = rra(0);               /* read in status */
    frame_tx = 0ffh;                /* set frame transmitted flag */

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end esi_a;

rxintr_a:     procedure    interrupt 2ch;

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end rxintr_a;

src_a:        procedure    interrupt 2eh;

    rx_stat = rra(1);
    call wra(00,30h);               /* error reset */
    call wra(03,11000000b);         /* disable rx */
    frame_recd = 0ffh;              /* set frame received flag */

    call wra(00,38h);               /* reset highest IUS */
    output (eoir_186)  = 8000h;     /* non specific EOI */
    return;
end src_a;

/*-------------------------------------------------------------------------*/
```

231262-9

Figure 9. SDLC-DMA Interrupt Routines

The frame gets transmitted out with all bytes, except the first one, being fetched by the SCC using the DMA controller. At the end of the block the DMA controller stops supplying bytes to the SCC. This makes the transmitter underrun. Since the Transmitter Underrun Latch is in the reset state at this moment, the CRC bytes are appended by the SCC at the end of the transmit block going out. An External Status Change interrupt (esi__a, vector = 2ah) is generated with the bit for transmitter underrun set in RR0 register. This interrupt occurs when the CRC is being transmitted out and *not* when the frame is completely transmitted out.

9. SDLC Interrupt Routines

Figure 9 shows all the interrupt procedures for channel A when operating in the SDLC mode. The procedures of significance here are esi__a and src__a.

The end of frame reception results in the src__a procedure getting executed. Here the status in register RR1 is stored in a variable rx__stat for future examination. Any error bits set in status are reset, receiver is disabled and the flag indicating reception of a new frame is set.

The esi__a procedure is executed when CRC of the transmitted frame is just going out of the SCC. Reset External Status Interrupt command is executed, the external status is stored in a variable tx__stat for future examination and the flag indicating transmission of the frame is set.

End of frame processing is required after both of these interrupt procedures. It involves looking at rx__stat and tx__stat and checking if the desired operation was successful. The buffers used, may have to be recovered or new ones obtained to start another frame transmission or reception.

CONCLUSIONS

This article should ease the process of writing a complete data link driver for ASYNC and SDLC modes since most of the hardware dependent procedures are illustrated here. It was a conscious decision to make the procedures as small and easy to understand as possible. This had to be done at the expense of making the procedures general and not dealing with various exception conditions that can occur.

REFERENCES

1. 82530 Data Sheet, Order #230834-001
2. 82530 SCC Technical Manual, Order #230925-001

APPENDIX A
82530—BAUD RATE GENERATORS

The 82530 has two Baud Rate Generators (BRG) on chip—one for each channel. They are used to provide the baud rate or serial clock for receive and transmit operations. This article describes how the BRG can be programmed and used.

The BRG for each channel is totally independent of each other and have to be programmed separately for each channel. This article describes how any one of the two BRGs can be programmed for operation. To use the BRG, four steps have to be performed:

1. Determine the Baud Rate Time Constant (BRTC) to be programmed into registers WR12 (LSB) and WR13 (MSB).
2. Program in register WR11, to specify where the output of the BRG must go to.
3. Program the clock source to the BRG in register WR14.
4. Enable the BRG.

Step 1: Baud Rate Time Constant (BRTC)

The BRTC is determined by a simple formula:

$$\text{BRTC} = \frac{\text{Serial Clock Frequency}}{2 \times (\text{Baud Rate} \times \text{Baud Rate Factor})} - 2$$

Example:

For Serial Clock Frequency = 4 MHz

Baud Rate = 9600

Baud Rate Factor = 16

$$\text{BRTC} = \frac{4000000}{2 \times (9600 \times 16)} - 2$$

$$= 13.021 - 2 = 11.021$$

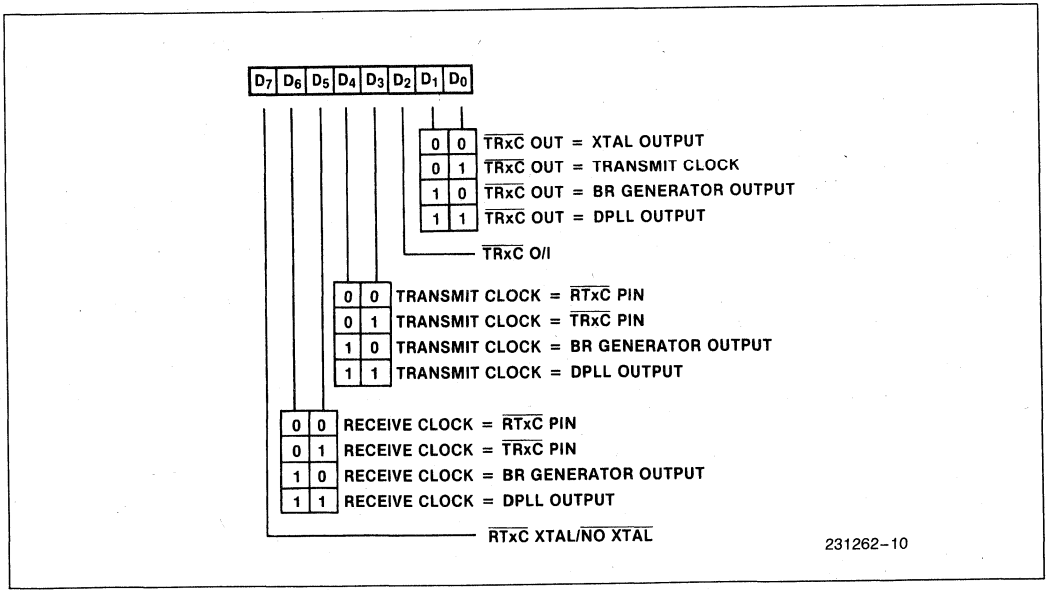


Figure 1. Write Register 11

Table 1. BRTC - Baud Rate Time Constant

		Baud Rate Factor			
		1	16	32	64
Baud Rate	9600	206.333	11.021	4.510	1.255
	4800	414.667	24.042	11.021	4.510
	2400	831.333	50.083	24.042	11.021
	1200	1664.667	102.167	50.083	24.042
	600	3331.333	206.333	102.167	50.083
	300	6664.667	414.667	206.333	102.167

Since only integers can be written into the registers WR12/WR13 this will have to be rounded off to 11 and it will result in an error of:

$$\frac{\text{fraction}}{\text{BRTC}} \times 100 = \frac{0.021}{11.021} \times 100 = 0.19\%$$

This error indicates that the baud rate signal generated by the BRG does not provide the exact frequency required by the system. This error is more serious for smaller baud rate factors. For asynchronous systems, errors up to 5% are considered acceptable.

Note that for BRTC = 0, BRG output frequency = 1/4 × Serial Clock Freq.

Table 1 shows the BRTC for a 4 MHz serial clock with various baud rates on the Y - axis and baud rate factors on the X - axis. The constant that is really programmed into registers WR12/WR13 is the integer closest to the BRTC value shown in the table.

Step 2: BRG Output

The output of the BRG can be directed to the Receiver, Transmitter and the TRxC output. This is programmed by setting bits D6 D5, bits D4 D3, and bits D1 D0 in register WR11 to 10. See Figure 1. The output of the BRG can also be directed to the Digital Phase Locked Loop (DPLL) for the on-chip decoding of the NRZI encoded received data signal. This is done by writing 100 into bits D7 D6 D5 of register WR14 as shown in Figure 2.

Step 3: BRG Source Clock

Register WR14 is used to select the input clock to the BRG. See Figure 2.

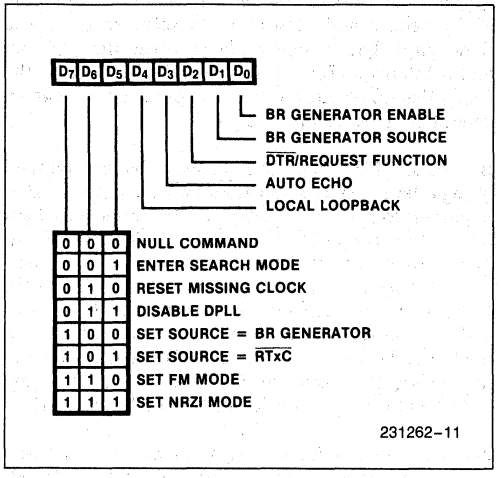


Figure 2. Write Register 14

WR14 / bit D1 = 0 → Clock comes from pin RTxC

WR14 / bit D1 = 1 → Clock comes from System Clock (PCLK)

On RESET WR14 / bit D1 = 0.

It should be noted that for the case of Bit D1 = 0, the clock comes either from:

a. Clock on pin RTxC - if WR11 / D7 = 0

or b. Crystal on pins RTxC & SYNC - if WR11 / D7 = 1

Step 4: BRG Enable

This is the last step where bit D0 of WR14 is set to start the BRG. The BRG can also be disabled by resetting this bit.

APPENDIX B
MODEM CONTROL PINS ON THE 82530

Introduction

This article describes how the $\overline{CTS}$ and $\overline{CD}$ pins on the 82530 behave and how to write software to service these pins. The article explains when the External Status Interrupt occurs and how and when to issue the Reset External/Status Interrupt command to reliably determine the state of these pins.

Bits D3 and D5 of register RR0 show the *inverted* state of logic levels on $\overline{CD}$ and $\overline{CTS}$ pins respectively. It is important to note that the register RR0 does *not* always reflect the *current* state of the $\overline{CD}$ and $\overline{CTS}$ pins. Whenever a Reset External/Status Interrupt ($\overline{RESI}$) command is issued, the (inverted) states of the $\overline{CD}$ and the $\overline{CTS}$ pins get updated and latched into the RR0 register and the register RR0 then reflect the inverted state of the $\overline{CD}$ and $\overline{CTS}$ pins at the time of the write operation to the chip. On channel or chip reset, the inverted state of $\overline{CD}$ and $\overline{CTS}$ pins get latched into RR0 register.

Normally, a transition on any of the pins does not necessarily change the corresponding bit(s) in RR0. In certain situations it does and in some cases it does not. A sure way of knowing the current state of the pins is to read the register RR0 *after* a RESI command.

There are two cases:

I. External/Status Interrupt (ESI) enabled.

II. Polling (ESI disabled).

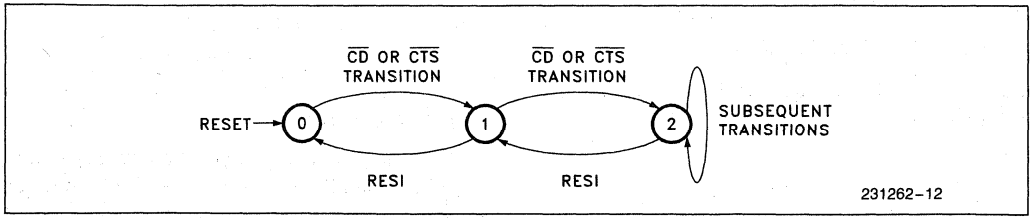
Case I: External Status Interrupt (ESI) Enabled

Whenever ESI is enabled, an interrupt can occur whenever there is a transition on $\overline{CD}$ or $\overline{CTS}$ pins - the IE bits for $\overline{CD}$ and/or $\overline{CTS}$ must also be set in WR15 for the interrupt to be enabled.

In this case, the first transition on any of these pins will cause an interrupt to occur and the corresponding bit in RR0 to change (even without the RESI command). A RESI command resets the interrupt line and also latches in the current state of both the $\overline{CD}$ and the $\overline{CTS}$ pins. If there was just one transition the RESI does not really change the contents of RR0.

If there are more than one transitions, either on the same pin or one each on both pins or multiple on both pins, the interrupt would get activated on the first transition and stay active. The bit in RR0 corresponding only to the very first transition is changed. All subsequent transitions have no effect on RR0. The first transition, in effect, freezes all changes in RR0. The first RESI command, as could be expected, latches the final (inverted) state of the $\overline{CD}$ and $\overline{CTS}$ pins into the RR0 register. Note that all the intermediate transitions on the pins are lost (because the response to the interrupt was not fast enough). The interrupt line gets reset for only a brief moment following the first RESI command. This brief moment is approximately 500 ns for the 82530. After that the interrupt becomes active again. A second RESI command is necessary to reset the interrupt. Two RESI commands resets the interrupt line independent of the number of transitions occurred.

Whenever operating with ESI enabled, it is recommendable to issue two back-to-back RESI commands and then read the RR0 register to reliably determine the state of the $\overline{CD}$ and $\overline{CTS}$ pins and also to reset the interrupt line in case multiple transitions may have occurred.



State Diagram

Case II: Polling RR0 for $\overline{CD}$ and $\overline{CTS}$ Pins

If RR0 is polled for determining the state of the $\overline{CD}$ and $\overline{CTS}$ pins, then the External/Status Interrupt (ESI) is kept disabled. In this case the bits in RR0 may not change even for the first transition. The best way to handle this case to always issue a RESI command before reading in the RR0 register to determine the state of $\overline{CD}$ and $\overline{CTS}$ pins. Note, however, if two back-to-back RESI commands were to be issued every time before reading in the RR0 register, the first subsequent transition will change the corresponding bit in RR0.

The state diagram above illustrates how each transition on $\overline{CD}$ and $\overline{CTS}$ pins affect the 82530 and what effect the RESI command has.

State 0

It is entered on reset. No ESI due to $\overline{CTS}$ or $\overline{CD}$ are pending in this state. Any transition on $\overline{CTS}$ or $\overline{CD}$ pins lead to the state 1 *accompanied by an immediate change in the RR0 register.*

State 1

Interrupt is active (if enabled). If a RESI command is issued, state 0 is reached where interrupt is again inactive. However, a further transition on $\overline{CTS}$ or $\overline{CD}$ pin leads to state 2 *without an immediate change in RR0 register.*

State 2

Interrupt is active (if enabled). Any further transitions have no effect. A RESI command leads to state 1, temporarily making the interrupt inactive.

CONCLUSIONS

Register RR0 does not always reflect the current (inverted) state of the $\overline{CD}$ and $\overline{CTS}$ pins. The most reliable way to determine the state of the pins in interrupt or polling mode is to issue two back-to-back RESI commands and then read RR0. While polling, the second RESI is redundant but harmless. When issuing the back-to-back RESI commands to 82530 note that the separation between the two write cycles should be at least 6 CLK + 200 ns; otherwise the second RESI will be ignored.

APPENDIX C
THE 82530 SCC - 80186 INTERFACE AP BRIEF

INTRODUCTION

The object of this document is to give the 82530 system designer an in-depth worst case design analysis of the typical interface to a 80186 based system. This document has been revised to include the new specifications for the 6 MHz 82530. The new specifications yield better margins and a 1 wait state interface to the CPU (2 wait states are required for DMA cycles). These new specifications will appear in the 1987 data sheet and advanced specification information can be obtained from your local Intel sales office. The following analysis includes a discussion of how the interface TTL is utilized to meet the timing requirements of the 80186 and the 82530. In addition, several optional interface configurations are also considered.

INTERFACE OVERVIEW

The 82530 - 80186 interface requires the TTL circuitry illustrated in Figure 1. Using five 14 pin TTL packages, 74LS74, 74AS74, 74AS08, 74AS04, and 74LS32, the following operational modes are supported:

- Polled
- Interrupt in vectored mode
- Interrupt in non-vectored mode
- Half-duplex DMA on both channels
- Full-duplex DMA on channel A

A brief description of the interface functional requirements during the five possible BUS operations follows below.

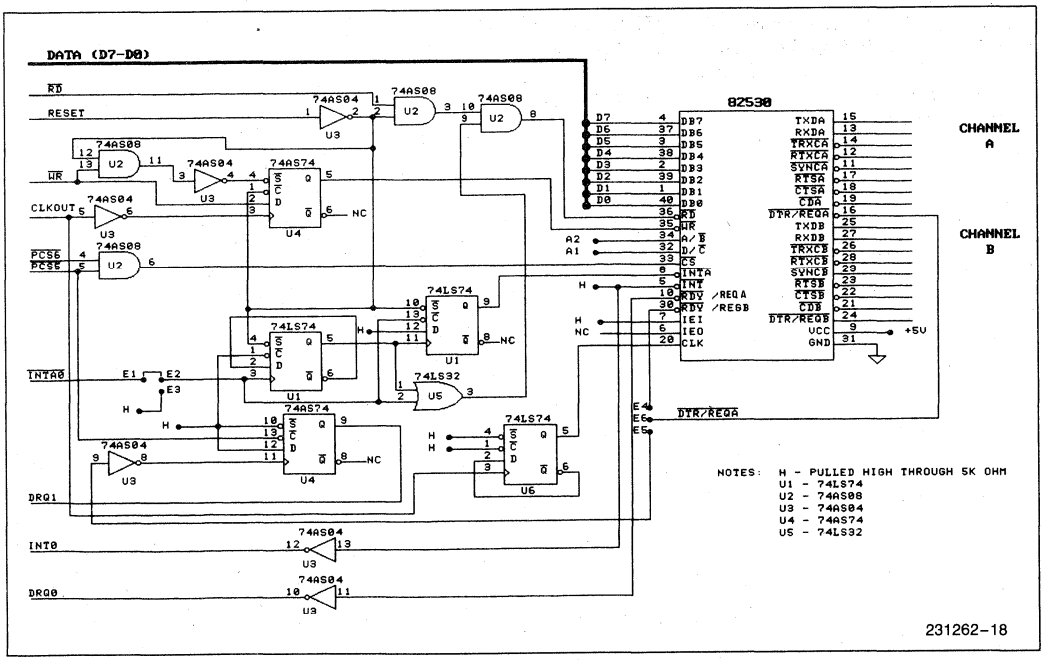


Figure 1. 82530-80186 Interface

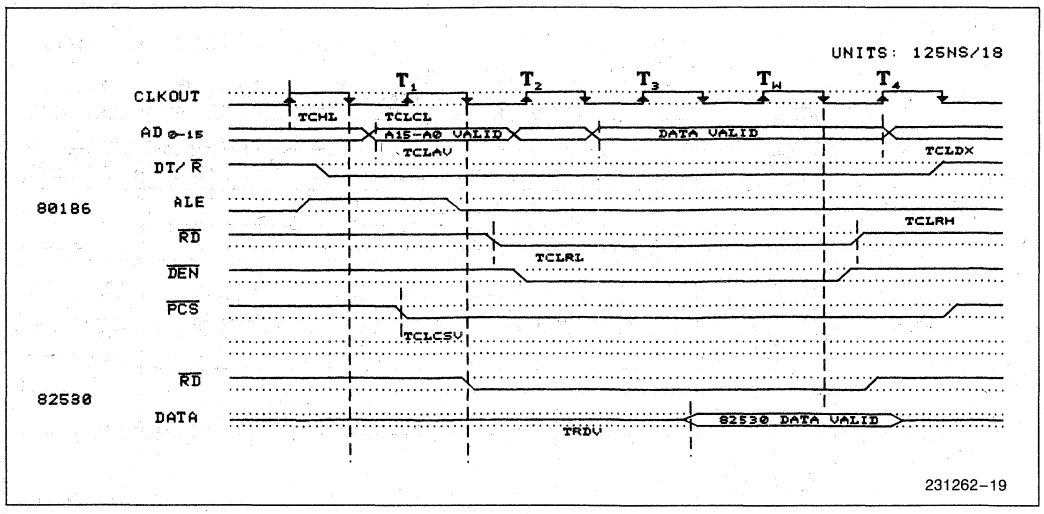


Figure 2. 80186–82530 Interface Read Cycle

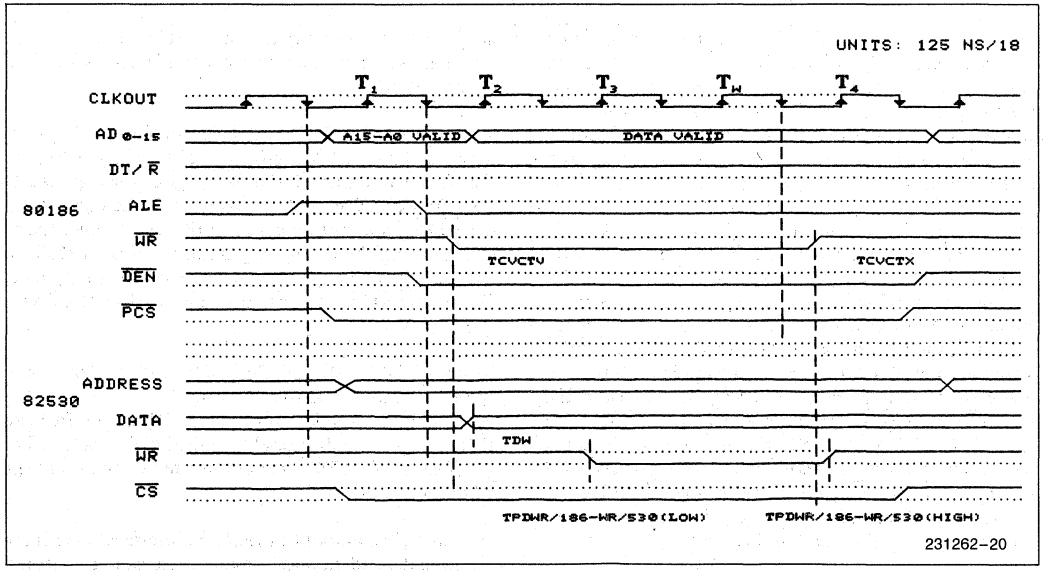


Figure 3. 80186–82530 Interface Write Cycle

READ CYCLE: The 80186 read cycle requirements are met without any additional logic, Figure 2. At least one wait state is required to meet the 82530 tAD access time.

WRITE CYCLE: The 82530 requires that data must be valid while the $\overline{WR}$ pulse is low, Figure 3. A D Flip-Flop delays the leading edge of $\overline{WR}$ until the falling edge of CLOCKOUT when data is guaranteed valid and $\overline{WR}$ is guaranteed active. The CLOCKOUT signal is inverted to assure that $\overline{WR}$ is active low before the D Flip-Flop is clocked. No wait states are necessary to meet the 82530's $\overline{WR}$ cycle requirements, but one is assumed from the $\overline{RD}$ cycle.

INTA CYCLE: During an interrupt acknowledge cycle, the 80186 provides two $\overline{INTA}$ pulses, one per bus cycle, separated by two idle states. The 82530 expects only one long $\overline{INTA}$ pulse with a $\overline{RD}$ pulse occurring only after the 82530 $\overline{IEI}/\overline{IEO}$ daisy chain settles. As

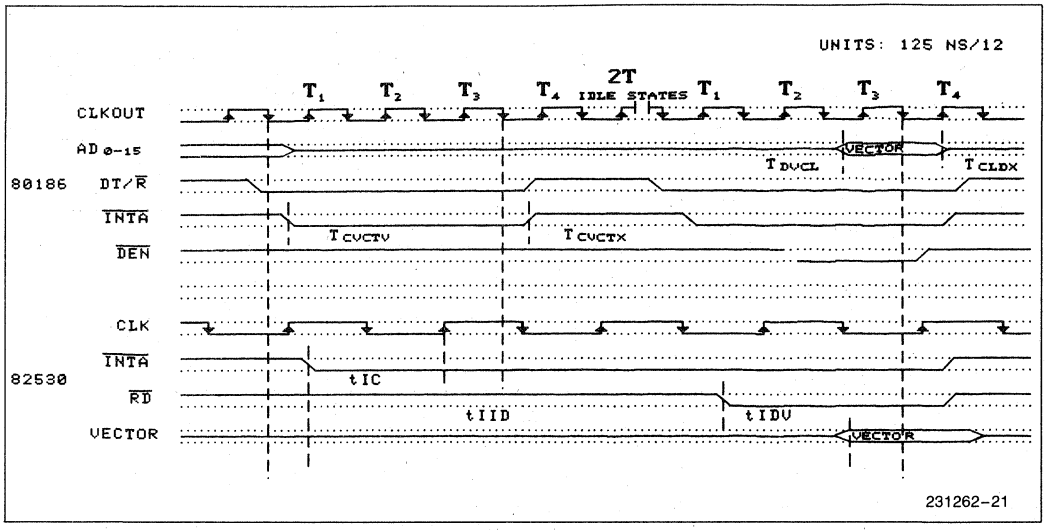


Figure 4. 82530–80186 INTA Cycle

illustrated in Figure 4, the INTA signal is sampled on the rising edge of CLK (82530). Two D Flip-Flops and two TTL gates, U2 and U5, are implemented to generate the proper INTA and RD pulses. Also, the INT signal is passively pulled high, through a 1 k resistor, and inverted through U3 to meet the 80186's active high requirement.

DMA CYCLE: Conveniently, the 80186 DMA cycle timings are the same as generic read and write operations. Therefore, with two wait states, only two modifications to the DMA request signals are necessary. First, the RDYREQA signal is inverted through U3 similar to the INT signal, and second the DTR/REQA signal is conditioned through a D Flip-Flop to prevent inadvertent back to back DMA cycles. Because the 82530 DTR/REQA signal remains active low for over five CLK (82530)'s, an additional DMA cycle could occur. This uncertain condition is corrected when U4 resets the DTR/REQ signal inactive high. Full Duplex on both DMA channels can easily be supported with one extra D Flip-Flop and an inverter.

RESET: The 82530 does not have a dedicated RESET input. Instead, the simultaneous assertion of both RD and WR causes a hardware reset. This hardware reset is implemented through U2, U3, and U4.

ALTERNATIVE INTERFACE CONFIGURATIONS

Due to its wide range of applications, the 82530 interface can have many varying configurations. In most of these applications the supported modes of operation need not be as extensive as the typical interface used in this analysis. Two alternative configurations are discussed below.

8288 BUS CONTROLLER: An 80186 based system implementing an 8288 bus controller will not require the preconditioning of the WR signal through the D Flip-Flop U4. When utilizing an 8288, the control signal IOWC does not go active until data is valid, therefore, meeting the timing requirements of the 82530. In such a configuration, it will be necessary to logically OR the IOWC with reset to accommodate a hardware reset operation.

NON-VECTORED INTERRUPTS: If the 82530 is to be operated in the non-vectored interrupt mode (B step only), the interface will not require U1 or U5. Instead, INTA on the 82530 should be pulled high, and pin 3 of U2 (RD AND RESET) should be fed directly into the RD input of the SCC.

Obviously, the amount of required interface logic is application dependent and in many cases can be considerably less than required by the typical configuration, supporting all modes of SCC operation.

DESIGN ANALYSIS

This design analysis is for a typical microprocessor system, pictured in Figure 5. The Timing analysis assumes an 8 MHz 80186 and a 6 MHz 82530 being clocked at 4 MHz. The 4 MHz clock is the 80186 CLKOUT divided by two by a flip-flop (U6). Also, included in the analysis are bus loading, and TTL-MOS compatibility considerations.

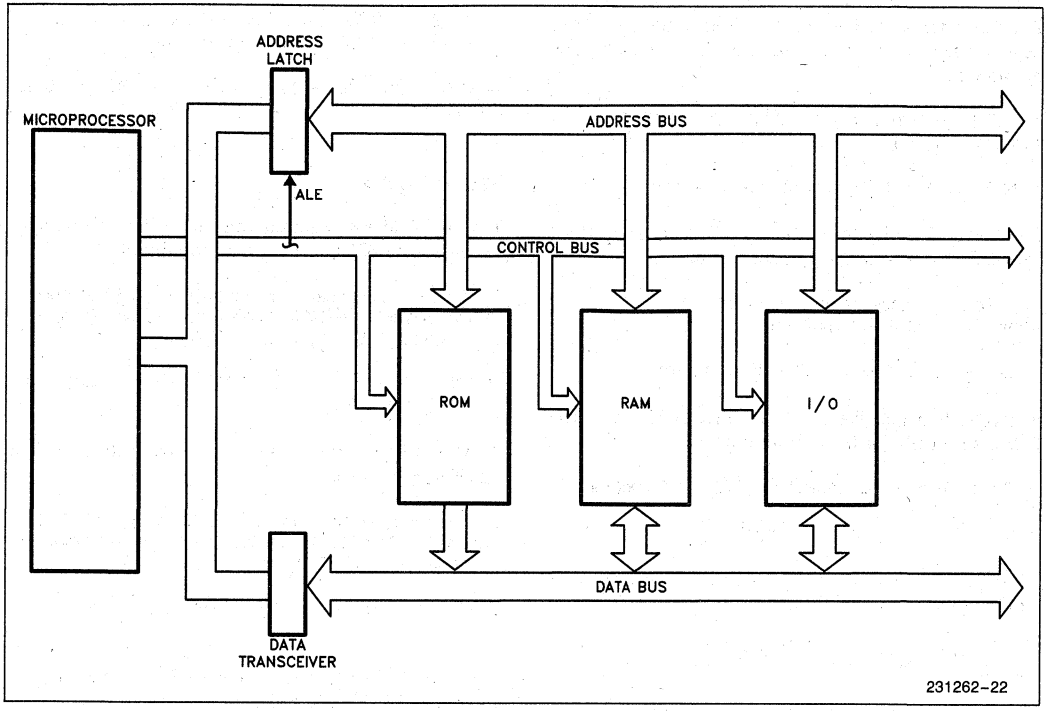


Figure 5. Typical Microprocessor System

Bus Loading and Voltage Level Compatabilities

The data and address lines do not exceed the drive capability of either 80186 or the 82530. There are several control lines that drive more than one TTL equivalent input. The drive capability of these lines are detailed below.

$\overline{WR}$: The $\overline{WR}$ signal drives U3 and U4.

* Iol (2.0 mA) > Iil (−0.4 mA + −0.5 mA)
Ioh (−400 μA) > Iih (20 μA + 20 μA)

$\overline{PCS5}$: The $\overline{PCS5}$ signal drives U2 and U4.

* Iol (2.0 mA) > Iil (−0.5 mA + −0.5 mA)
Ioh (−400 μA) > Iih (20 μA + 20 μA)

$\overline{INTA}$: The $\overline{INTA}$ signal drives 2(U1) and U5.

* Iol (2.0 mA) > Iil (−0.4 mA + −0.8 mA + −0.4 mA)
Ioh (−400 μA) > Iih (20 μA + 40 μA + 20 μA)

All the 82530 I/O pins are TTL voltage level compatible.

TIMING ANALYSIS

Certain symbolic conventions are adhered to throughout the analysis below and are introduced for clarity.

1. All timing variables with a lower case first letter are 82530 timing requirements or responses (i.e., tRR).
2. All timing variables with Upper case first letters are 80186 timing responses or requirements unless preceded by another device's alpha-numeric code (i.e., Tclcl or '373 Tpd).
3. In the write cycle analysis, the timing variable Tpd$\overline{WR}$186-$\overline{WR}$530 represents the propagation delay between the leading or trailing edge of the $\overline{WR}$ signal leaving the 80186 and the $\overline{WR}$ edge arrival at the 82530 $\overline{WR}$ input.

Read Cycle

1. **tAR:** Address valid to $\overline{RD}$ active set up time for the 82530. Since the propagation delay is the worst case path in the assumed typical system, the margin is calculated only for a propagation delay constrained and not an ALE limited path. The spec value is 0 ns minimum.

* 1 Tclcl − Tclav(max) − '245 Tpd(max) + Tclrl(min) + 2(U2) Tpd(min) − tAR(min)

= 125 − 55 − 20.8 + 10 + 2(2) − 0 = 63.2 ns margin

2. **tRA:** Address to $\overline{RD}$ inactive hold time. The ALE delay is the worst case path and the 82530 requires 0 ns minimum.

* 1 Tclcl − Tclrh (max) + Tchlh(min) + '373 LE Tpd(min) − 2(U2) Tpd(max)

= 55 − 55 + 5 + 8 − 2(5.5) = 2 ns margin

3. **tCLR:** $\overline{CS}$ active low to $\overline{RD}$ active low set up time. The 82530 spec value is 0 ns minimum.

* 1 Tclcl − Tclcsv(max) − Tclrl(min) − U2 skew($\overline{RD}$ − $\overline{CS}$) + U2 Tpd(min)

= 125 − 66 − 10 − 1 + 2 = 50 ns margin

4. **tRCS:** $\overline{RD}$ inactive to $\overline{CS}$ inactive hold time. The 82530 spec calls for 0 ns minimum.

* Tcscsx(min) − U2 skew($\overline{RD}$ − $\overline{CS}$) − U2 Tpd(max)

= 35 − 1 − 5.5 = 28.5 ns margin

5. **tCHR:** $\overline{CS}$ inactive to $\overline{RD}$ active set up time. The 82530 requires 5 ns minimum.

* 1 Tclcl + 1 Tchcl − Tchcsx(max) + Tclrl(min) − U2 skew ($\overline{RD}$ − $\overline{CS}$) + U2 Tpd(min) − tCHR

= 125 + 55 − 35 − 10 − 1 + 2 − 5 = 131 ns margin

6. **tRR:** $\overline{RD}$ pulse active low time. One 80186 wait state is included to meet the 150 ns minimum timing requirements of the 82530.

* Trlrh(min) + 1($\overline{Tclcl}$wait state) − 2(U2 skew) − tRR

= (250−50) + 1(125) − 2(1) − 150 = 173 ns margin

7. tRDV: $\overline{RD}$ active low to data valid maximum delay for 80186 read data set up time (Tdvcl = 20 ns). The margin is calculated on the Propagation delay path (worst case).

* 2 Tclcl + 1($\overline{Tclcl}$wait state) − Tclrl(max) − Tdvcl(min) − '245 Tpd(max) − 82530 tRDV(max) − 2(U2) Tpd(max)

= 2(125) + 1(125) − 70 − 20 − 14.2 − 105 − 2(5.5) = 154 ns margin

8. **tDF:** $\overline{RD}$ inactive to data output float delay. The margin is calculated to $\overline{DEN}$ active low of next cycle.

* 2 Tclcl + Tclch(min) − Tclrh(max) + Tchctv(min) − 2(U2) Tpd(max) − 82530 tDF(max)

= 250 + 55 −55 + 10 − 11 − 70 = 179 ns margin

9. **tAD:** Address required valid to read data valid maximum delay. The 82530 spec value is 325 ns maximum.

* 3 Tclcl + 1($\overline{Tclcl}$wait state) − Tclav(max) − '373 Tpd(max) − '245 Tpd − Tdvcl(min) − tAD

= 375 + 125 − 55 − 20.8 −14.2 − 20 −325 = 65 ns margin

Write Cycle

1. **tAW:** Address required valid to $\overline{WR}$ active low set up time. The 82530 spec is 0 ns minimum.

* Tclcl − Tclav(max) − Tcvctv(min) − '373 Tpd(max) + Tpd$\overline{WR}$186 − $\overline{WR}$530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)] − tAW

= 125 − 55 − 5 − 20.8 + [125 − 5 + 1 + 4.4] − 0 = 170.6 ns margin

2. **tWA:** $\overline{WR}$ inactive to address invalid hold time. The 82530 spec is 0 ns.

* Tclch(min) − Tcvctx(max) + Tchlh(min) + '373 LE Tpd(min) − Tpd$\overline{WR}$186=$\overline{WR}$530(HIGH) [U2 Tpd(max) + U3 Tpd(max) + U4 Tpd(max)]

= 55 − 55 + 5 + 8 − [5.5 + 3 + 7.1] = −2.6 ns margin

3. **tCLW:** Chip select active low to $\overline{WR}$ active low hold time. The 82530 spec is 0 ns.

* 1 Tclcl − Tclcsv(max) + Tcvctv(min) − U2 Tpd(max) + Tpd$\overline{WR}$186=$\overline{WR}$530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)]

= 125 − 66 + 5 − 5.5 + [125 − 5 + 1 + 4.4] = 183.9 ns margin

4. **tWCS:** $\overline{WR}$ invalid to Chip Select invalid hold time. 82530 spec is 0 ns.

* Tcxcsx(min) − U2 Tpd(max) − Tpd$\overline{WR}$186=$\overline{WR}$530(HIGH) [U2 Tpd(max) + U3 Tpd(max) + U4 Tpd(max)]

= 35 + 1.5 − [5.5 + 3 + 7.1] = 20.9 ns margin

5. **tCHW:** Chip Select inactive high to $\overline{WR}$ active low set up time. The 82530 spec is 5 ns.

* 1 Tclcl + Tchcl(min) + Tcvctv(min) − Tchcsx(max) − U2 Tpd(max) + Tpd$\overline{WR}$186=$\overline{WR}$530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)] − tCHW

= 125 + 55 + 5 − 35 − 5.5 + [125 −5 + 1 + 4.4] − 5 = 264 ns margin

6. **tWW:** $\overline{WR}$ active low pulse. 82530 requires a minimum of 60 ns from the falling to the rising edge of $\overline{WR}$. This includes one wait state.

* Twlwh [2Tclcl − 40] + 1 (Tclclwait state) − TpdWR/186−WR530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(max) + U4 Tpd(max)] + TpdWR/186=WR/530(HIGH) [U2 Tpd(min) U3 Tpd(min) + U4 Tpd(min)] − tWW

= 210 + 1(125) − [125 − 5 + 4.5 + 9.2] − [1.5 + 1 + 3.2] − 60 = 135.6 ns margin

7. **tDW:** Data valid to $\overline{WR}$ active low setup time. The 82530 spec requires 0 ns.

* Tcvctv(min) − Tcldv(max) − '245 Tpd(max) + TpdWR186−WR530(LOW) [Tclcl − Tcvctv(min) + U3 Tpd(min) + U4 Tpd(min)]

= 5 − 44 − 14.2 + 125 − 5 + 1.0 + 4.4 = 72.2 ns margin

8. **tWD:** Data valid to $\overline{WR}$ inactive high hold time. The 82530 requires a hold time of 0 ns.

* Tclch − skew {Tcvctx(max) + Tcvctx(min)} + '245 OE Tpd(min) − TpdWR186−WR530(HIGH) [U2 Tpd(max) + U3 Tpd(max) + U4 Tpd(max)]

= 55 − 5 + 11.25 − [5.5 + 3.0 + 7.1] = −50.6 ns margin

INTA Cycle:

1. **tIC:** This 82530 spec implies that the $\overline{INTA}$ signal is latched internally on the rising edge of CLK (82530). Therefore the maximum delay between the 80186 asserting $\overline{INTA}$ active low or inactive high and the 82530 internally recognizing the new state of $\overline{INTA}$ is the propagation delay through U1 plus the 82530 CLK period.

* U1 Tpd(max) + 82530 CLK period

= 45 + 250 = 295 ns

2. **tCI:** rising edge of CLK to $\overline{INTA}$ hold time. This spec requires that the state of $\overline{INTA}$ remains constant for 100 ns after the rising edge of CLK. If this spec is violated any change in the state of $\overline{INTA}$ may not be internally latched in the 82530. tCI becomes critical at the end of an $\overline{INTA}$ cycle when $\overline{INTA}$ goes inactive. When calculating margins with tCI, an extra 82530 CLK period must be added to the $\overline{INTA}$ inactive delay.

3. **tIW:** $\overline{INTA}$ inactive high to $\overline{WR}$ active low minimum setup time. The spec pertains only to 82530 $\overline{WR}$ cycle and has a value of 55 ns. The margin is calculated assuming an 82530 $\overline{WR}$ cycle occurs immediately after an $\overline{INTA}$ cycle. Since the CPU cycles following an 82530 $\overline{INTA}$ cycle are devoted to locating and executing the proper interrupt service routine, this condition should never exist. 82530 drivers should insure that at least one CPU cycle separates $\overline{INTA}$ and $\overline{WR}$ or $\overline{RD}$ cycles.

4. **tWI:** $\overline{WR}$ inactive high to $\overline{INTA}$ active low minimum hold time. The spec is 0 ns and the margin assumes CLK coincident with $\overline{INTA}$.

* Tclcl − Tcvctx(max) − TpdWR186 − WR530(HIGH) [U3 Tpd(max) + U4 Tpd(max)] + Tcvctv(min) + U1 Tpd(min)

= 125 − 55 − [5.5 + 3 + 7.1] + 5 + 10 = 69.4 ns margin

5. **tIR:** $\overline{INTA}$ inactive high to $\overline{RD}$ active low minimum setup time. This spec pertains only to 82530 $\overline{RD}$ cycles and has a value of 55 ns. The margin is calculated in the same manner as tIW.

6. **tRI:** $\overline{RD}$ inactive high to $\overline{INTA}$ active low minimum hold time. The spec is 0 ns and the margin assumes CLK coincident with $\overline{INTA}$.

* Tclcl − Tclrh(max) − 2 U2 Tpd(max) + Tcvctv(min) + U1 Tpd(min)

= 125 − 55 − 2(5.5) + 5 + 10 = 74 ns margin

7. **tIID:** $\overline{INTA}$ active low to $\overline{RD}$ active low minimum setup time. This parameter is system dependent. For any SCC in the daisy chain, tIID must be greater than the sum of tCEQ for the highest priority device in the daisy chain, tEI for this particular SCC, and tEIEO for each device separating them in the daisy chain. The typical system with only 1 SCC requires tIID to be greater than tCEQ. Since tEI occurs coincidently with tCEQ and it is smaller it can be neglected. Additionally, tEIEO does not have any relevance to a system with only one SCC. Therefore tIID > tCEQ = 250 ns.

* 4 Tclcl + 2 Tidle states − Tcvctv(max) − tIC [U1 Tpd(max) + 82530 CLK period] + Tcvctv(min) + U5 Tpd(min) + U2 Tpd(min) − tIID

= 500 + 250 − 70 − [45 + 250] + 5 + 6 + 2 − 250 = 148 ns margin

8. **tIDV:** $\overline{RD}$ active low to interrupt vector valid delay. The 80186 expects the interrupt vector to be valid on the data bus a minimum of 20 ns before T4 of the second acknowledge cycle (Tdvcl). tIDV spec is 100 ns maximum.

* 3 Tclcl − Tcvctv(max) − U5 Tpd(max) − U2 Tpd(max) − tIDV(max) − '245 Tpd(max) − Tdvcl(min)

= 375 − 70 − 25 − 5.5 − 100 − 14.2 − 20 = 140.3 ns margin

9. **tII:** $\overline{RD}$ pulse low time. The 82530 requires a minimum of 125 ns.

* 3 TclcI − Tcvctv(max) − U5 Tpd(max) − U2 Tpd(max) + Tcvctx(min) + U5 Tpd(min) + U2 Tpd(min) − tII(min)

= 375 − 70 − 25 − 5.5 + 5 + 6 + 1.5 − 125 = 162 ns margin

DMA Cycle

Fortunately, the 80186 DMA controller emulates CPU read and write cycle operation during DMA transfers. The DMA transfer timings are satisfied using the above analysis. Because of the 80186 DMA request input requirements, two wait states are necessary to prevent inadvertent DMA cycles. There are also CPUDMA intracycle timing considerations that need to be addressed.

1. **tDRD:** $\overline{RD}$ inactive high to $\overline{DTR}$REQ (REQUEST) inactive high delay. Unlike the $\overline{READY}$REQ signal, $\overline{DTR}$REQ does not immediately go inactive after the requested DMA transfer begins. Instead, the $\overline{DTR}$REQ remains active for a maximum of 5 tCY + 300 ns. This delayed request pulse could trigger a second DMA transfer. To avoid this undesirable condition, a D Flip Flop is implemented to reset the $\overline{DTR}$REQ signal inactive low following the initiation of the requested DMA transfer. To determine if back to back DMA transfers are required in a source synchronized configuration, the 80186 DMA controller samples the service request line 25 ns before T1 of the deposit cycle, the second cycle of the transfer.

* 4 TclcI − Tclcsv(max) − U4Tpd(max) − Tdrqcl(min)

= 500 − 66 − 10.5 − 25 = 398.5 ns margin

2. tRRI: 82530 $\overline{RD}$ active low to $\overline{REQ}$ inactive high delay. Assuming source synchronized DMA transfer, the 80186 requires only one wait state to meet the tRRI spec of 200 ns. Two are included for consistency with tWRI.

* 2 TclcI + 2($\overline{Tclcl}$wait state) − TclrI(max) − 2(U2) Tpd(max) − Tdrqcl − tRRI

=2(125) + 2(125) − 70 − 2(5.5) − 200 = 219 ns margin

3. tWRI: 82530 $\overline{WR}$ active low to $\overline{REQ}$ inactive high delay. Assuming destination synchronized DMA transfers, the 80186 needs two wait states to meet the tWRI spec. This is because the 80186 DMA controller samples requests two clocks before the end of the deposit cycle. This leaves only 1 TclcI + n(wait states) minus $\overline{WR}$ active delay for the 82530 to inactivate its $\overline{REQ}$ signal.

* TclcI + 2($\overline{Tclcl}$wait state) − Tcvctv(min) − Tpd$\overline{WR}$186−$\overline{WR}$530(LOW) [TclcI − Tcvctv(min) + U3 Tpd(max) + U4 Tpd(max)] − Tdrqcl − tWRI

=375 − 5 − [125 − 5 + 4.5 + 9.2] − 25 − 200 = 11.3 ns margin

NOTE:
If one wait state DMA interface is required, external logic, like that used on the $\overline{DTR}$REQ signal, can be used to force the 82530 $\overline{REQ}$ signal inactive.

4. **tREC:** CLK recovery time. Due to the internal data path, a recovery period is required between SCC bus transactions to resolve metastable conditions internal to the SCC. The DMA request lines are masked from requesting service until after the tREC has elapsed. In addition, the CPU should not be allowed to violate this recovery period when interleaving DMA transfers and CPU bus cycles. Software drivers or external logic should orchestrate the CPU and DMA controller operation to prevent tREC violation. In this example circuit, tREC could be improved by clocking the '530 with a 6 MHz clock.

Reset Operation

During hardware reset, the system RESET signal is asserted high for a minimum of four 80186 clock cycles (1000 ns). The 82530 requires $\overline{WR}$ and $\overline{RD}$ to be simultaneously asserted low for a minimum of 250 ns.

* 4 TclcI − U3 Tpd(max) − 2(U2) Tpd(max) + U4 Tpd(min) − tREC

= 1000 − 17.5 − 2(5.5) + 3.5 − 250 ns = 725 ns margin

82530 VALID ACCESS LOGIC

Due to the unique internal data path of the 82530, an intra-access recovery time must be provided to settle any internal metastable conditions. This internal metastble condition gives rise to the Clock Recovery [tREC] specification required by the 82530. This tREC is measured from the rising edge of a $\overline{RD}$ or $\overline{WR}$ to the falling edge of the next $\overline{RD}$ or $\overline{WR}$ intended for the 82530, and equates to 6 CLK's + 130 ns. Effectively, this specification implies that the system must provide 1130 ns (6 MHz 82530) between every CPU or other DMA access to the 82530. (Figure 1.)

Systems that only allow CPU access to the 82530 are not significantly impacted by this clock recovery time. In CPU access only designs, the software designer can insert NOP's to guarantee the tREC idle time in between successive CPU $\overline{RD}$ or $\overline{WR}$ cycles to the 82530. Unfortunately, systems that contain more than one direct memory access device, interfacing with the 82530, will require external hardware to arbitrate 82530 accesses and thereby guaranteeing the tREC restriction.

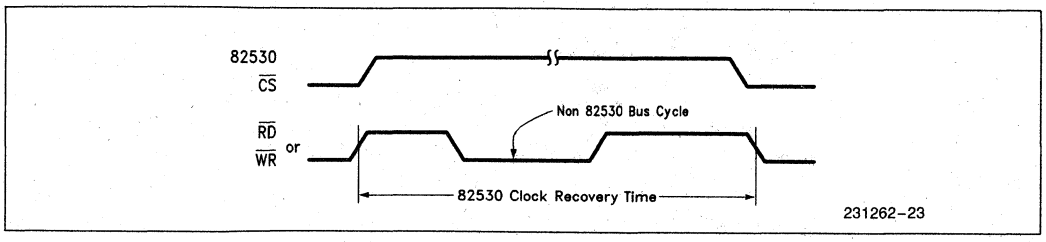


Figure 1

EXTERNAL VALID ACCESS HARDWARE

To accommodate this clock recovery specification, external hardware has been designed for the 82530 systems containing several DMA devices accessing the 82530 (ie., a CPU and a DMA controller). This logic has been tailored for an 80186 environment but can easily be modified to fit 8086 or 80286 systems.

LOGIC STATE MACHINE

There are two basic functions that need to be performed by the external logic. The first is to mask the $\overline{CS}$ signal from reaching the 82530 until the tREC intra-access idle time has elapsed. The second task is to generate a not ready condition to the CPU or DMA device until the tREC period has expired and the minimum wait state requirement for the particular access has been satisfied. The simple state machine, Figure 2, illustrates the required operation.

The TTL logic pictured in Figure 2 implements the state machine with some assorted gates, a flip-flop, and a shift register. $\overline{PCS}$ from the 80186 should be qualified with $\overline{RD}$ + $\overline{WR}$ to eliminate switching glitches during T1. The 'LS74 and 'LS00 perform rising edge detection to reset the shift register. The shift register clocks out the tREC period to enable $\overline{CS}$ and the additional 2 CLK's {82530} to satisfy the 82530 3 wait state requirement. The 80186 should be programmed to use the internal wait state generator {3 wait states for the 82530 and an 8 MHz 80186} and the external READY signal.

Note of caution: This hardware logic has not been verified on a bread board in an actual system. The hardware designer should verify that this logic fulfills his particular system timing requirements.

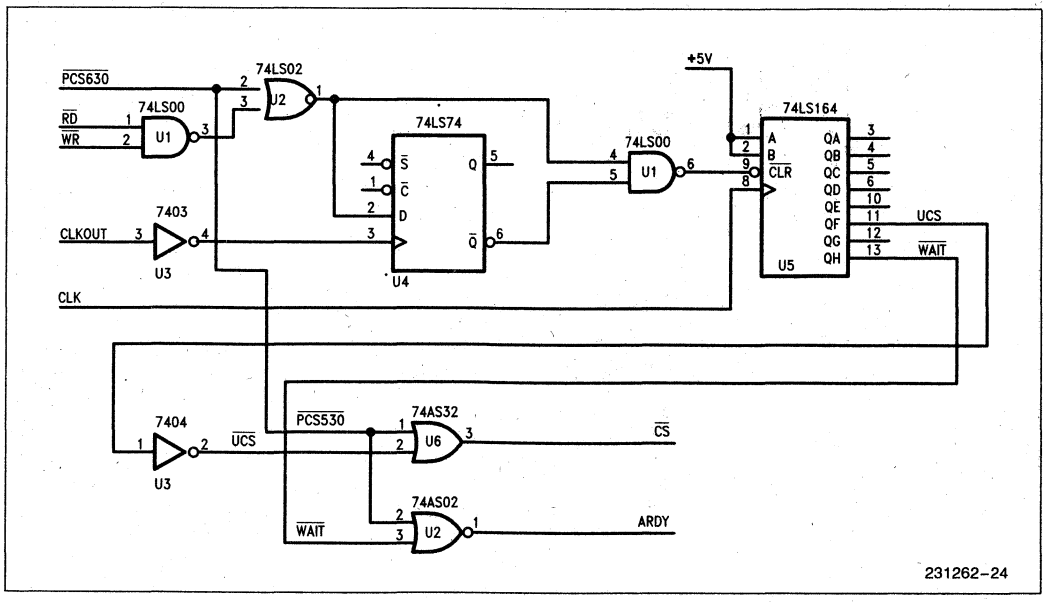


Figure 2

Other Components 3

intel®

APPLICATION NOTE

AP-166

April 1989

Using the 8291A GPIB Talker/Listener

Order Number: 230832-001

INTRODUCTION

This application note explains the Intel 8291A GPIB (General Purpose Interface Bus) Talker/Listener as a component, and shows its use in GPIB interface design tasks.

The first section of this note presents an overview of IEEE 488 (GPIB). The second section introduces the Intel GPIB component family. A detailed explanation of the 8291A follows. Finally, some application examples using the component family are presented.

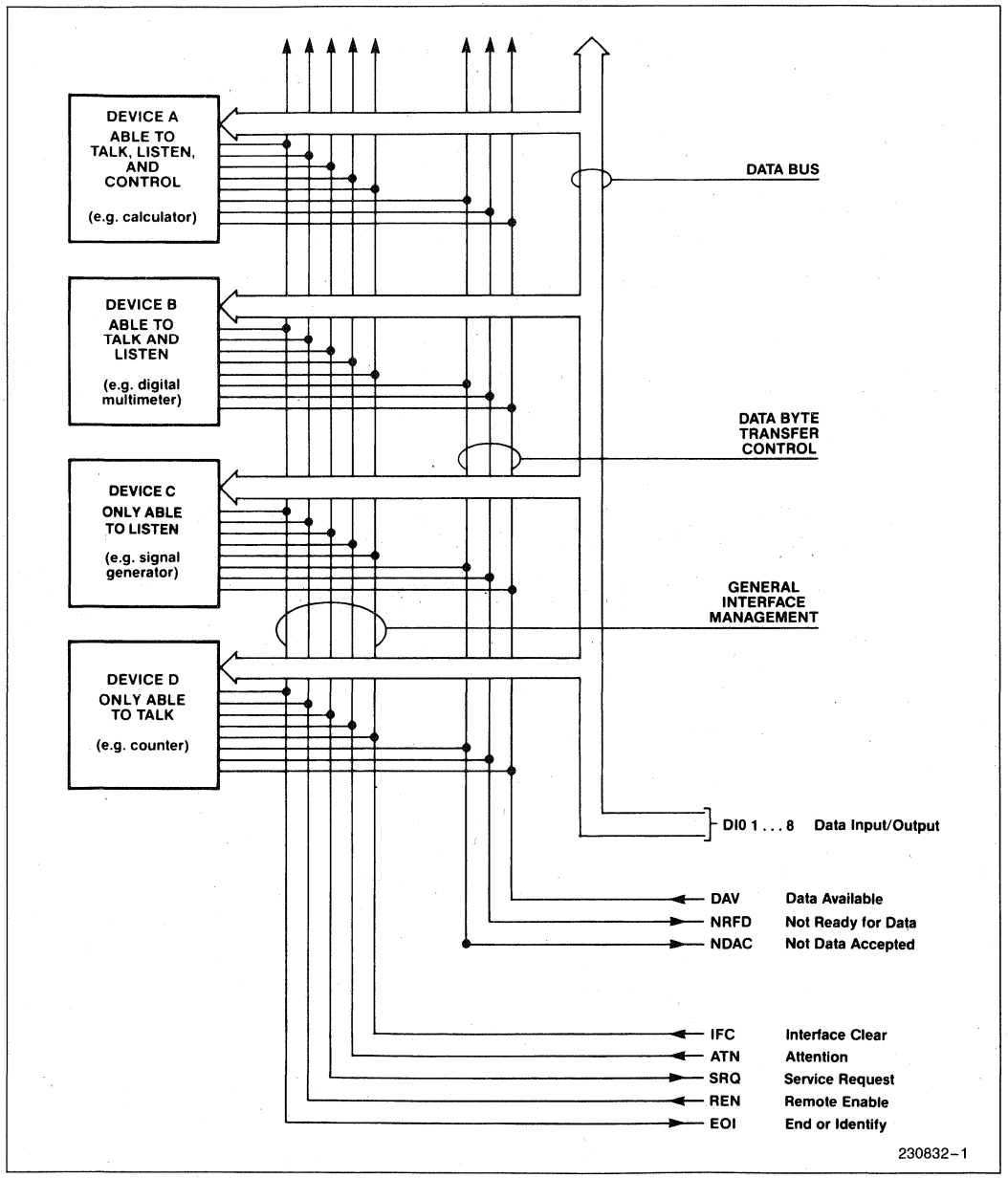


Figure 1. Interface Capabilities and Bus Structure

OVERVIEW OF IEEE 488/GPIB

The GPIB is a parallel interface bus with an asynchronous interlocking data exchange handshake mechanism. It is designed to provide a common communication interface among devices over a maximum distance of 20 meters at a maximum speed of 1 Mbps. Up to 15 devices may be connected together. The asynchronous interlocking handshake dispenses with a common synchronization clock, and allows intercommunication among devices capable of running at different speeds. During any transaction, the data transfer occurs at the speed of the slowest device involved.

The GPIB finds use in a diversity of applications requiring communication among digital devices over short distances. Common examples are: programmable instrumentation systems, computer to peripherals, etc.

The interface is completely defined in the IEEE STD.-488-1978.

A typical implementation consists of logical devices which talk (talker), listen (listeners), and control GPIB activity (controllers).

Interface Functions

The interface between any device and the bus may have a combination of several different capabilities (called 'functions'). Among a total of ten functions defined, the Talker, Listener, Source Handshake, Acceptor Handshake and Controller are the more common examples. The Talker function allows a device to transmit data. The Listener function allows reception. The Source and Acceptor Handshakes, synchronized with the Talker and Listener functions respectively, exchange the handshake signals that coordinate data transfer. The Controller function allows a device to activate the interface functions of the various devices through commands. Other interface functions are: Service request, Remote local, Parallel poll, Device clear and Device trigger. Each interface may not contain all these functions. Further, most of these functions may be implemented to various levels (called 'subsets') of capability. Thus, the overall capability of an interface may be tailored to the needs of the communicating device.

Electrical Signal Lines

As shown in Figure 1, the GPIB is composed of eight data lines (D08–D01), five interface management lines (IFC, ATN, SRQ, REN, EOI), and three transfer control lines (DAV, NRFD, NDAC).

The eight data lines are used to transfer data and commands from one device to another with the help of the management and control lines. Each of the five interface management lines has a specific function.

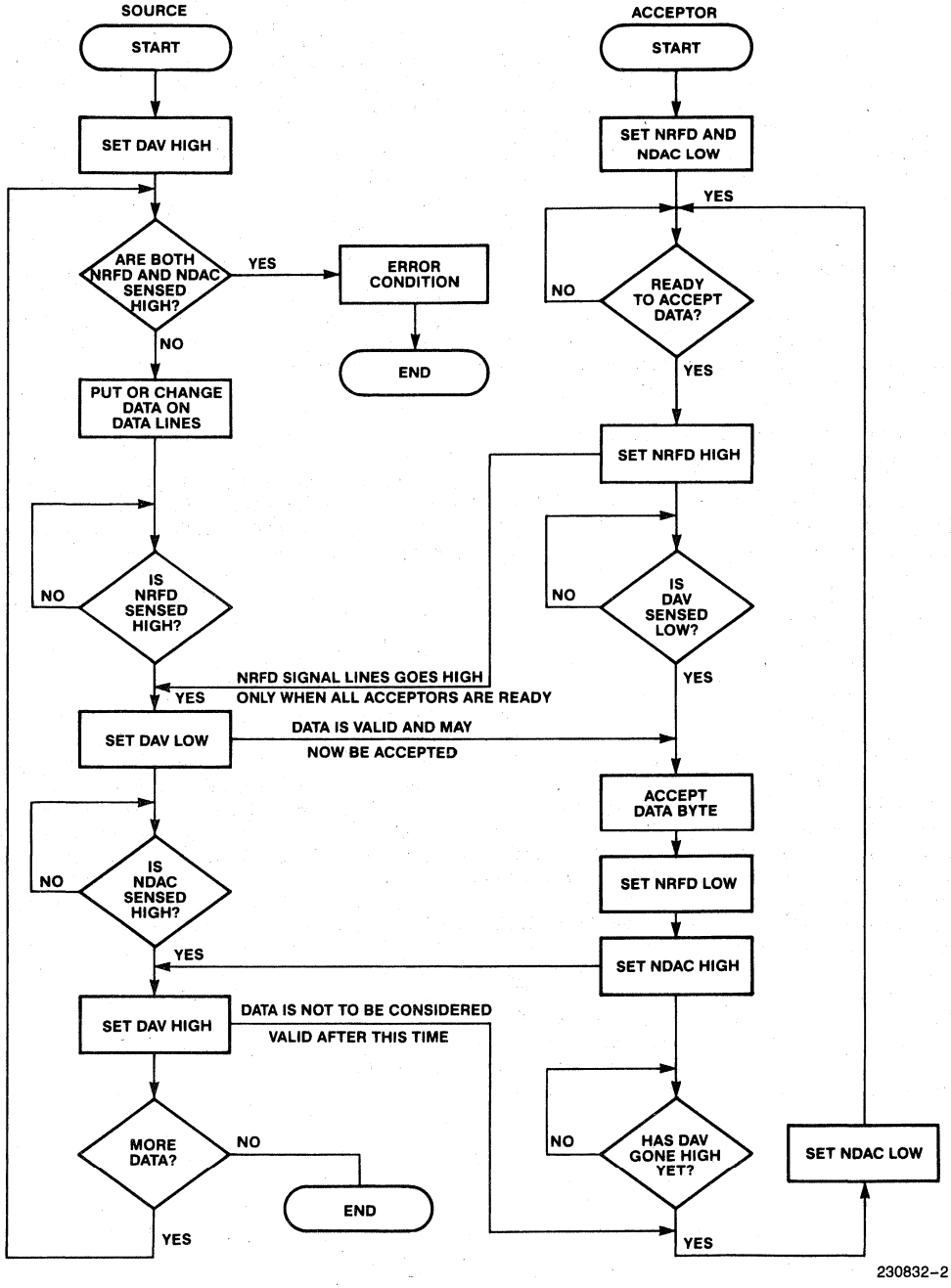
ATN (attention) is used by the Controller to indicate that it (the controller) has access to the GPIB and that its output on the data lines is to be interpreted as a command. ATN is also used by the controller along with EOI to indicate a parallel poll.

SRQ (service request) is used by a device to request service from the controller.

REN (remote enable) is used by the controller to specify the command source of a device. A device can be issued commands either locally through its front panel or by the controller.

EOI (end or identify) may be used by the controller as well as talker. A controller uses EOI along with ATN to demand a parallel poll. Used by a talker, EOI indicates the last byte of a data block.

IFC (interface clear) forces a complete GPIB interface to the idle state. This could be considered the GPIB's "interface reset." GPIB architecture allows for more than one controller to be connected to the bus simultaneously. Only one of these controllers may be in command at any given time. This device is known as the controller-in-charge. Control can be passed from one controller to another. Only one among all the controllers present on a bus can be the system controller. The system controller is the only device allowed to drive IFC.



NOTE:
Flow diagram outlines sequence of events during transfer of data byte. More than one listener at a time can accept data because of logical connection of NRFD and NDAC lines.

Figure 2. Handshake Flowchart

Transfer Control Lines

The transfer control lines conduct the asynchronous interlocking three-wire handshake.

DAV (data valid) is driven by a talker and indicates that valid data is on the bus.

NRFD (note ready for data) is driven by the listeners and indicates that not all listeners are ready for more data.

NDAC (not data accepted) is used by the listeners to indicate that not all listeners have read the GPIB data lines yet.

The asynchronous 3-wire handshake flowchart is shown in Figure 2. This is a concept fundamental to the asynchronous nature of the GPIB and is reviewed in the following paragraphs.

Assume that a talker is ready to start a data transfer. At the beginning of the handshake, NRFD is false indicating that the listener(s) is ready for data. NDAC is true indicating that the listener(s) has not accepted the data, since no data has been sent yet. The talker places data on the data lines, waits for the required settling time, and then indicates valid data by driving DAV true. All active listeners drive NRFD true indicating that they are not ready for more data. They then read the data and drive NDAC false to indicate acceptance. The talker responds by deasserting DAV and readies itself to transfer the next byte. The listeners respond to DAV false by driving NDAC true. The talker can now drive the data lines with a new data byte and wait for NRFD to be false to start the next handshake cycle.

Bus Commands

When ATN and DAV are true data patterns which have been placed by the controller on the GPIB, they are interpreted as commands by the other devices on the interface. The GPIB standard contains a repertory of commands such as MTA (My Talk Address), MSA (My Secondary Address), SPE (Serial Poll Enable), etc. All other patterns in conjunction with ATN and DAV are classified as undefined commands and their meaning is user-dependent.

Addressing Techniques

To allow the controller to issue commands selectively to specific devices, three types of addressing exist on the GPIB: talk only/listen only (ton/lon), primary, and secondary.

Ton/lon is a method where the ability of the GPIB interface to talk or listen is determined by the device and not by the GPIB controller. With this method, fixed poles can be easily designated in simple systems where reassignment is not necessary. This is appropriate and convenient for certain applications. For example, a logic analyzer might by interfaced via the GPIB to a line printer in order to document some type of failure. In this case, the line printer simply listens to the logic analyzer, which is a talker.

The controller addresses devices through three commands, MTA (my talk address), MLA (my listen address), and MSA (my secondary address). The device address is imbedded in the command bit pattern. The device whose address matches the imbedded pattern is enabled. Some devices may have the same logical talk and listen addresses. This is allowable since the talker and listener are separate functions. However, two of the same functions cannot have the same address.

In primary addressing, a device is enabled to talk (listen) by receiving the MTA (MLA) message.

Secondary addressing extends the address field from 5 to 10 bits by allowing an additional byte. This additional byte is passed via the MSA message. Secondary addressing can also be used to logically divide devices into various subgroups. The MSA message applies only to the device(s) whose primary address immediately precede it.

INTEL'S® GPIB COMPONENTS

The logic designer implementing a GPIB interface has, in the past, been faced with a difficult and complex discrete logic design. Advances in LSI technology have produced sophisticated microprocessor and peripheral devices which combine to reduce this once complex interface task to a system consisting of a small set of integrated circuits and some software drivers. A microprocessor hardware/software solution and a high-level language source code provide an additional benefit in end-product maintenance. Product changes are a simple matter of revising the product software. Field changes are as easy as exchanging EPROMS.

Intel has provided an LSI solution to GPIB interfacing with a talker/listener device (8291A), a controller device (8292), and a transceiver (8293). An interface with all capabilities except for the controller function can be built with an 8291A and a pair of 8293's. The addition of the 8292 produces a complex interface. Since most devices in a GPIB system will not have the controller function capability, this modular approach provides the least cost to the majority of interface designs.

Overview of the 8291A GPIB Talker/Listener

The Intel 8291A GPIB Talker/Listener operates over a clock range of 1 to 8 MHz and is compatible with the MCS-85, iAPX-86, and 8051 families of microprocessors.

A detailed description of the 8291A is given in the data sheet.

The 8291A implements the following functions: Source Handshake (SH), Acceptor Handshake (AH), Talker Extended (TE), Service Request (SRQ), Listener Extended (LE), Remote/Local (RL), Parallel Poll (PP2), Device Clear (DC), and Device Trigger (DT).

Current states of the 8291A can be determined by examining the device's status read registers. In addition, the 8291A contains 8 write registers. These registers are shown in Figure 3. The three register select pins RS3–RS0 are used to select the desired register.

The data-in register moves data from the GPIB to the microprocessor or to memory when the 8291A is addressed to listen. When the 8291A is addressed to talk, it uses the data-out register to move data onto the GPIB. The serial poll mode and status registers are used to request service and program the serial poll status byte.

A detailed description of each of the registers, along with state diagrams can be found in the 8291A data sheet.

Read Registers								Register Select Code RS2 RS1 RS0			Write Registers							
DI7	DI6	DI5	DI4	DI3	DI2	DI1	DI0	0	0	0	DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0
DATA IN											DATA OUT							
CPT	APT	GET	END	DEC	ERR	BO	BI	0	0	1	CPT	APT	GET	END	DEC	ERR	BO	BI
INTERRUPT STATUS 1											INTERRUPT ENABLE 1							
INT	SPAS	LLO	REM	SPC	LLOC	REMC	ADSC	0	1	0	0	0	DMAO	DMAI	SPC	LLOC	REMC	ADSC
INTERRUPT STATUS 2											INTERRUPT ENABLE 2							
S8	SRQS	S6	S5	S4	S3	S2	S1	0	1	1	S8	RSV	S6	S5	S4	S3	S2	S1
SERIAL POLL STATUS 2											SERIAL POLL MODE							
ton	Lon	EOI	LPAS	TPAS	LA	TA	MJMN	1	0	0	TO	LO	0	0	0	0	ADM1	ADM0
ADDRESS STATUS											ADDRESS MODE							
CPT7	CPT6	CPT5	CPT4	CPT3	CPT2	CPT1	CPT0	1	0	1	CNT2	CNT1	CNT0	COM4	COM3	COM2	COM1	COM0
COMMAND PASS THROUGH											AUX MODE							
INT	DTO	DLO	AD5-0	AD4-0	AD3-0	AD2-0	AD1-0	1	1	0	ARS	DT	DL	AD5	AD4	AD3	AD2	AD1
ADDRESS 0											ADDRESS 0/1							
X	DT1	DL1	AD5-1	AD4-1	AD3-1	AD2-1	AD1-1	1	1	1	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0
ADDRESS 1											EOS							

Figure 3. 8291A Registers

Address Mode

The address mode and status registers are used to program the addressing modes and track addressing states. The auxiliary mode register is used to select a variety of functions. The command pass through register is used for undefined commands and extended addresses. The address 0/1 register is used to program the addresses to which the 8291A will respond. The address 0 and address 1 registers allow reading of these programmed addresses plus trading of the interrupt bit. The EOS register is used to program the end of sequence character.

Detailed descriptions of the addressing modes available with the 8291A are described in the 8291A data sheet. Examples of how to program these modes are shown below.

1. MODE: Talker has single address of 01H
   Listener has single address of 02H

CPU Writes to:	Pattern	Comment
Address Mode Register	0000 0001	Select Mode 1 Addressing
Address 0/1 Register	0010 0001	Major is Talking. Address = 01H
Address 0/1 Register	1100 0010	Minor is Listener. Address = 02H

2. MODE: Talker has single address of 01H
   Listener has single address of 02H

CPU Writes to:	Pattern	Comment
Address Mode Register	0000 0001	Select Mode 1 Addressing
Address 0/1 Register	0100 0010	Major is Listener. Address = 02H
Address 0/1 Register	1010 0001	Minor is Talking. Address = 01H

Note that in both of the above examples, the listener will respond to a MLA message with five least significant bits equal to 02H and the talker to a 01H.

3. MODE: Talker and listener both share a single address of 03H

CPU Writes to:	Pattern	Comment
Address Mode Register	0000 0001	Select Mode 1 Addressing
Address 0/1 Register	0000 0011	Talker and Listener Address = 03
Address 0/1 Register	1110 0000	Minor Address is disabled

4. MODE: Talker and listener have a primary address of 04H and a secondary address of 05H

CPU Writes to:	Pattern	Comment
Address Mode Register	0000 0010	Select Mode 2 Addressing
Address 0/1 Register	0000 0100	Primary Address = 04H
Address 0/1 Register	1000 0101	Minor Address is disabled

5. MODE: Talker has a primary address of 06H. Listener has a primary address of 07H

CPU Writes to:	Pattern	Comment
Address Mode Register	0000 0011	Select Mode 3
Address 0/1 Register	0010 0110	Talker Address = 06
Address 0/1 Register	1100 0111	Listener Primary = 07

The CPU will verify the secondary addresses which could be the same or different.

APPLICATION OF THE 8291A

This phase of the application note will examine programming of the 8291A, corresponding bus commands and responses, CPU interruption, etc. for a variety of GPIB activities. This should provide the reader with a clear understanding of the role of the 8291A performs in a GPIB system. The talker function, listener function, remote message handling, and remote/local operations including local lockout, are discussed.

Talker Functions

TALK-ONLY (ton). In talk only mode the 8291A will not respond to the MTA message from a controller. Generally, ton is used in an environment which does not have a controller. Ton is also employed in an interface that includes the controller function.

When the 8291A is used with the 8292, the sequence of events for initialization are as follows:

1) The Interrupt/Enable registers are programmed.
2) Ton is selected.
3) Settling time is selected.
4) EOS character is loaded.
5) "Pon" local message is sent.
6) CPU waits for Byte Out (BO) and sends a byte to the data out register.

Addressed Talker (via MTA Message)

The GPIB controller will direct the 8291A to talk by sending a My Talk Address (MTA) message containing the 8291A's talk address. The sequence of events is as follows:

1) The interrupt enable and serial poll mode registers are programmed.
2) Mode 1 is selected.
3) Settling time is selected.
4) Talker and listener addresses are programmed.
5) Power on (pon) local message is sent.
6) CPU waits for an interrupt. When the controller has sent the MTA message for the 8291A an interrupt will be generated if enabled and the ADSC bit will be set.
7) CPU reads the Address Status register to determine if the 8291A has been addressed to talk (TA = 1).
8) CPU waits for an interrupt from either BO or ADSC
9) When BO is set, the CPU writes the data byte to the data out register.
10) CPU continues to poll the status registers.
11) When unaddressed ADSC, will be set and TA reset.

LISTENER FUNCTIONS

LISTEN-ONLY (lon). In listen-only mode the 8291 will not respond to the My Listen Address (MLA) message from the controller. The sequence of events is as follows:

1) The Interrupt Enable registers are programmed.
2) Lon is selected.
3) EOS character is programmed.
4) "Pon" local message is sent.
5) CPU waits for BI and reads the byte from the data-in register.

Note that enabling both ton and lon can create an internal loopback as long as another listener exists.

Addressed Listening (via the MLA Message)

The GPIB controller will direct the 8291A to listen by sending a MLA message containing the 8291A's listen address. The sequence of events is as follows:

1) The Interrupt Enable registers are programmed.
2) The serial poll mode register is loaded as desired.
3) Talker and listener addresses are loaded.
4) "Pon" local message is sent.
5) The CPU waits for an interrupt. When the controller has sent the MLA message for the 8291A, the ADSC bit will be set.
6) The CPU reads the Address Status Register to determine if the 8291A has been addressed to listen (LA = 1).
7) CPU waits for an interrupt for BI or ADSC.
8) When BI is set, the CPU reads the data byte from the data-in register.
9) The CPU continues to poll the status registers.
10) When unaddressed, ADSC will be set and LA reset.

Remote/Local and Lockout

Remote and local refer to the source of control of a device connected to the GPIB. Remote refers to control from the GPIB controller-in-charge. Local refers to control from the device's own system. Reference should be made to the RL state diagram in the 2891A data sheet.

Upon "pon" the 8291A is in the local state. In this state the REM bit in Interrupt Status 1 Register is reset. When the GPIB controller takes control of the bus it will drive the REN (remote enable) line true. This will cause the REM bit and REMC (remote/local change) bit to be set. The distinction between remote and local modes is necessary in that some types of devices will have local controls which have functions which are also controlled by remote messages.

In the local state the device is allowed to store, but not respond to, remote messages which control functions which are also controlled by local messages. A device which has been addressed to listen will exit the local state and go to the remote state if the REN message is true and the local rtl (return to local) message is false. The state of the "rtl" local message is ignored and the device is "locked" into the local state if the LLO remote message is true. In the Remote state the device is not allowed to respond to local messages which control function that are also controlled by remote messages. A device will exit the remote state and enter the local state when REN goes false. It will also enter the local state if the GTL (go to local) remote message is true and the device has been addressed to listen. It will also enter the local state if the rtl message is true and the LLO message is false or ACDS is inactive.

A device will exit the remote state and enter RWLS (remote with lockout state) if the LLO (local lockout) message is true and ACDS is active. In this mode, those local messages which control functions which are also controlled by remote messages are ignored. In other words, the "rtl" message is ignored. A device will exit RWLS and go to the local state if REN goes false. The device will exit RWLS and go to LWLS if the GTL message is true and the device is addressed to listen.

Polling

The IEEE-488 standard specifies two methods for a slave device to let the controller know that it needs service.

These two methods are called Serial and Parallel Poll. The controller performs one of these two polling methods after a slave device requests service. As implied in the name, a Serial Poll is when the controller sequentially asks each device if it requested service. In a Parallel Poll the controller asks all of the devices on the GPIB if they requested service, and they reply in parallel.

Serial Poll

When the controller performs a Serial Poll, each slave device sends back to the controller a Serial Poll Status Byte. One of the bits in the Serial Poll Status Byte indicates whether this device requested service or not. The remaining 7 bits are used defined, and they are used to indicate what type of service is required. The IEEE-488 spec only defines the service request bit, however HP has defined a few more bits in the Serial Poll Status Byte. This can be seen in Figure 4.

When a slave device needs service it drives the SRQ line on the GPIB bus true (low). For the 8291A this is done by setting bit 7 in the Serial Poll Status Byte. The CPU in the controller may be interrupted by SRQ or it may poll a register to determine the state of SRQ. Using the 8292 one could either poll the interrupt status register for the SRQ interrupt status bit, or enables SRQ to interrupt the CPU. After the controller recognizes a service request, it goes into the serial poll routine.

The first thing the controller does in the serial poll routine is assert ATN. When ATN is asserted true the controller takes control of the GPIB, and all slave de-

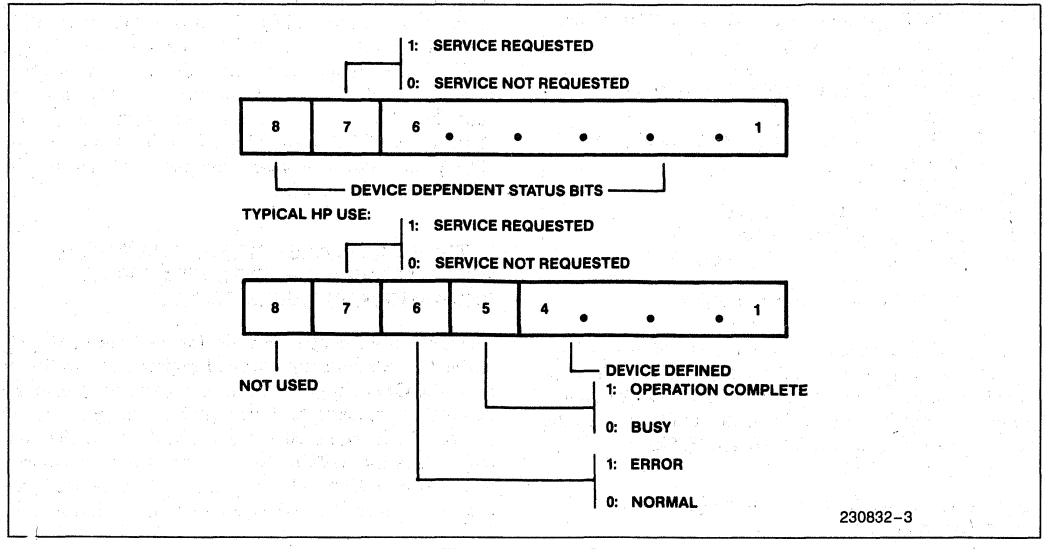


Figure 4. The Serial Poll Status Byte

vices on the bus must listen. All bytes sent over the bus while ATN is true are commands. After the controller takes control, it sends out a Universal Unlisten (UNL), which tells all previously addressed listeners to stop listening. The controller then sends out a byte called SPE (Serial Poll Enable). This command notifies all of the slaves on the bus that the controller has put the GPIB in the Serial Poll Mode State (SPMS). Now the controller addresses the first slave device to TALK and puts itself in the listen mode. When the controller resets ATN the device addressed to talk transmits to the controller its Serial Poll Status Byte. If the device just polled was the one requesting service, the SRQ line on the GPIB goes false, and bit 7 in the serial poll status byte of the 8291A is reset. If more than one device is requesting service, SRQ remains low until all of the devices requesting service have been polled, since SRQ is wire-ored. To continue the Serial Poll, the controller asserts ATN, addresses the next device to talk then reads the Serial Poll Status Byte. When the controller is finished polling it asserts ATN, sends the univeral untalk command (UNT), then sends the Serial Poll Disable command (SPD). The flow of the serial poll can be seen from the example in Figure 5.

0) DEVICE A REQUESTS SERVICE (SRQ)
1) ASSERT ATN
2) UNIVERSAL UNLISTEN (UNL)
3) SERIAL POLL ENABLE (SPE)
4) DEVICE A TALK ADDRESS (MTA)
5) RELEASE ATN
6) DEVICE A STATUS BYTE (STD) (RQS SET)
7) ASSERT ATN
8) DEVICE B TALK ADDRESS (MTA)
9) RELEASE ATN
10) DEVICE B STATUS BYTE (STB) (RQS CLEAR)
11) ASSERT ATN
12) DEVICE C TALK ADDRESS (MTA)
13) RELEASE ATN
14) DEVICE C STATUS BYTE (STB) (RQS CLEAR)
15) ASSERT ATN
16) UNIVERSAL UNTALK (UNT)
17) SERIAL POLL DISABLE (SPD)
18) GO PROCESS SERVICE REQUEST

Figure 5. Serial Polling

The following section describes the events which happen in a serial poll when 8291A and 8292 are the controller, and another 8291A is the slave device. While going through this section the reader should refer to the register diagrams for the 8291A and 8292.

A. DEVICE A REQUESTS SERVICE (SRQ BECOMES TRUE)

The slave devices rsv bit in the 2819A's serial poll mode register is set.

B. CONTROLLER RECOGNIZES SRQ AND ASSERTS ATN

The 8292's SPI pin 33 interrupts the CPU. The CPU reads the 8292's Interrupt status register and finds the SRQ bit set. The CPU tells the 8292 to 'Take Control Synchronously' by writing a OFDH to the 8292's command register.

C. THE CONTROLLER SENDS OUT THE FOLLOWING COMMANDS: UNIVERSAL UNLISTEN (UNL), SERIAL POLL ENABLE (SPE), MY TALK ADDRESS (MTA)

(MTA is a command which tells one of the devices on the bus to talk.)

The CPU in the controller waits for a BO (byte out) interrupts in the 8291A's interrupt status 1 register before it writes to the Data Out register a 3FH (UNL), 18H (SPE), 010XXXXX (MTA). The X represents the programmable address of a device on the GPIB. When the 8291A in the slave device receives its talk address, the ADSC bit in the Interrupt Status register 2 is set, and in the Address Status Register TA and TPAS bits are set.

D. CONTROLLER RECONFIGURES ITSELF TO LISTEN AND RESETS ATN

The CPU in the controller puts the 8291A in the listen only mode by writing a 40H to the Address Mode register of the 8291A, and then a OOH to the Aux Mode register. The second write is an 'Immediate Execute pon' which must be used when switching addressing modes such as talk only to listen only. To reset ATN the CPU tells the 8292 to 'Go To Standby' by writing a 0F6H to the command register. The moment ATN is reset, the 8219A in the slave device sets SPAS in Interrupt Status 2 register, and transmits the serial poll status byte. SRQS in the Serial Poll Status byte of the 8291A slave device is reset, and the SRQ line on the GPIB bus becomes false.

E. THE CONTROLLER READS THE SERIAL POLL STATUS BYTE, SETS ATN, THEN RECONFIGURES ITSELF TO TALK

The CPU in the controller waits for the Byte In bit (BI) in the 8291A's Interrupt Status 1 register. When this bit is set the CPU reads the Data In register to receive the Serial Poll Status Byte. Since bit 7 is set, this was the device which requested service. The CPU in the controller tells the 8292 to 'Take Control Synchronously' which asserts ATN. The moment ATN is asserted true the 8291A in the slave device resets SPAS, and sets the

Serial Poll Complete (SPC) bit in the Interrupt Status 2 register. The controller reconfigures itself to talk by setting the TO bit in the Address Mode register and then writing a OOH to the Aux Mode register.

F. THE CONTROLLER SENDS THE COMMANDS UNIVERSAL UNTALK (UNT), AND SERIAL POLL DISABLE (SPD) THEN RESETS THE SRQ BIT IN THE 8292 INTERRUPT STATUS REGISTER

The CPU in the controller waits for the BO Interrupt status bit to be set in the Interrupt Status 1 register of the 8291A before it writes 5FH (UNT) and 19H (SPD) to the Data Out register. The CPU then writes a 2BH to the 8292's command register to reset the SRQ status bit in the Interrupt Status register. When the 8291A in the slave device receives the UNT command the ADSC bit in the Interrupt Status 2 register is set, and the TA and TPAS bits in the Address Status register will be reset. At this point the controller can service the slave device's request.

Note that in the software listing of AP-66 (USING THE 8292 GPIB CONTROLLER) there is a bug in the serial poll routines. In the 'SRQ ROUTINE' when the CPU finds that the SRQ bit in the interrupt status register is set, it immediately writes the interrupt Acknowledge command to the 8292 to reset this bit. However the SRQ GPIB line will still be driven true until the slave device driving SRQ has been polled. Therefore, the SRQ status bit in the 8292 will become set and latched again, and as a result the SRQ status bit in the 8292 will still be set after the serial poll. The proper time to reset the SRQ bit in the 8292 is after SRQ on the GPIB becomes false.

Parallel Poll

The 8291A supports an additional method for obtaining status from devices known as parallel poll (PPOL). This method limits the controller to a maximum of 8 devices at a time since each device will produce a single bit response on the GPIB data lines. As shown in the state diagrams, there are three basic parallel poll states: PPIS (parallel poll idle state), PPSS (parallel poll standby state), and PPAS (parallel poll active state).

In PPIS, the device's parallel poll function is in the idle state and will not respond to a parallel poll. PPSS is the standby state, a state in which the device will respond to a parallel poll from the controller. The response is initiated by the controller driving both ATN and EOI true simultaneously.

The 8291A state diagram shows a transition from PPIS to PPSS with the "lpe" message. This is a PP2 implementation for a parallel poll. This "lpe" (local poll enable) local message is achieved by writing $011USP_3P_2P_1$ to the Aux Mode Register with $U=0$.

The S bit is the sense bit. If the "ist" (individual status) local message value matches the sense bit, then the 8291A will give a true response to a parallel poll. Bits P_3-P_1 identify which data line is used for a response.

For example, assume the programmer decides that the system containing the 8291A shall participate in parallel poll. The programmer, upon system initialization would write to the Aux Mode Register and reset the U bit and set the S bit plus identify a data line (P_3-P_1 bits). At "pon," the 8291A would not respond true to a parallel poll unless the parallel poll flag is set (via Aux Mode Register command).

When a status condition in the user system occurs and the programmer decides that this condition warrants a true response, then programmers software should set the parallel poll flag. Since the S bit value matches the "ist" (set) condition a true response will be given to all parallel polls.

An additional method of parallel polling reading exists known as a PP1 implementation. In this case the controller sends a PPE (parallel poll enable) message. PPE contains a bit pattern similar to the bit pattern used to program the "lpe" local message. The 8291A will receive this as an undefined command and use it to generate an "lpe" message. Thus the controller is specifying the sense bits and data lies for a response. A PPD (parallel poll disable) message exists which clears the bits $SP_3P_2P_1$ and sets the U bit. This also will be received by the 8291A and used to generate an "lpe" false local message.

The actual sequence of events is as follows. The controller sends a PPC (parallel poll configure) message. This is an undefined command which is received in the CPT register and the handshake is held off. The local CPU reads this bit pattern, decodes it, and sends a VSCMD message to the Aux Mode Register. The controller then sends a ppe message which is also received as an undefined command in the CPT register. The local CPU reads this, decodes it clears the MSB, and writes this to the Aux Mode Register generating the "lpe" message.

The controller then sends ATN and EOI true and the 8291A drives the appropriate data line if the "ist" (parallel poll flag) is true. The controller will then send a PPD (parallel poll disable) message (again, an undefined command). The CPU reads this from the CPT register and uses it to write new "lpe" message (this "lpe" message will be false). The controller then sends a PPU (parallel poll unconfigure) message. Since this is also an undefined command, it goes into the CPT register. When the local CPU decodes this, the CPU should clear the "ist" (parallel poll flag).

APPLICATION EXAMPLES

In the course of developing this application note, two complete and identical; GPIB systems were built. The schematics and block diagrams are contained in Appendix 1. These systems feature an 8088 CPU, 8237 DMA controller, serial I/O (8215a and 8253), RAM, EPROM, and a complete GPIB talker/listener controller. Jumper switches were provided to select between a controller function and a talker/listener function. This system design is based on the design of Intel's SDK-86 prototyping kit and thus shares the same I/O and memory addresses. This system uses the same download software to transfer object files from Intel development systems.

Two Software Drivers

Two software drivers were developed to demonstrate a ton/lon environment. These two programs (BOARD 1 and BOARD 2) are contained in Appendix 2.

In this example, one of the systems (BOARD 1) initially is programmed in talk-only mode and synchronization is achieved by waiting for the listening board to become active. This is sensed by the lack of a GPIB error since a condition of no active listener produces an ERR status condition. Board 1 upon detecting the presence of an active listener transmits a block of 100 bytes from a PROM memory across the bus. The second system (BOARD 2) receives this data and stores it in a buffer, EOI is sent true by the talker (BOARD 1) with the last byte of data. Upon detection of EOI, BOARD 2 switches to the talk only mode while BOARD 1 upon terminal count switches to the listen only mode. BOARD 2 then detects the presence of an active listener and transmits the contents of its buffer back to BOARD 1 which stores this data in the buffer. EOI again is sent with the last byte and BOARD 2 switches back to listen-only. BOARD 1 upon detecting EOI then compares the contents of its buffer with the contents of its PROM to ensure that no data transmission errors occurred. The process then repeats itself.

8291A with HP 9835A

An example of the 8291A used in conjunction with a bus controller is also included in this application note. In this example, the 8291A system used in previous experiments was connected via the GPIB to a Hewlett-Packard 9835A desktop computer. This computer contains, in addition to a GPIB interface, a black and white CRT, keyboard, tape drive for high quality data cassettes, and a calculator type printer. The software for the HP9835S is shown in Appendix 3. The user should refer to the operation manuals for the HP 9835A for information on the features and programming methods for the HP 9835A.

In this example, the 8292 was removed from its socket and the OPTA and OPTB pins of the two 8293 transceiver reconfigured to modes 0 and 1. Optionally, the mode pins could have been left wired for modes 2 and 3 and the 8292 left in its socket with its SYC pin wired to ground. This would have produced the same effect.

The first action performed is sending IFC. Generally, this is done when a controller first comes on line. This pulse is at least 100 μs in duration as specified by the IEEE-488 standard.

The software checks to see if active listeners are on line. For demonstration purposes, the HP 9835A will flag the operator to indicate that listeners are on line.

The HP 9835A then configures and performs a parallel poll (PPOL). The parallel poll indicates 1 bit of status of each device in a group of up to 8 devices. Such information could be used by an application program to determine whether optional devices are part of a system configuration. Such optional devices might include mass storage devices, printers, etc., where the application software for the controller might need to format data to match each type of device. Once the PPOL sequence is finished, the HP 9835A offers the user the opportunity to execute user commands from the keyboard. At this time the HP 9835A sits in a loop waiting for an SRQ condition. When the operator hits a key on the keyboard, the HP 9835A processor is interrupted and vectors to a service routine where the key is read and the appropriate routine is executed. The HP 9835A will then return to the loop checking for the SRQ true. For this application, the valid keys are G, D, R, H, and X. Pressing the "G" key causes the GET command to be sent across the bus. A message to this effect is printed in the CRT and the HP 9835A returns. The "D" key causes the SDC message to be sent with the 8291A being the addressed device. Again, an appropriate message is output on the HP 9835A CRT. The "R" key causes the GTL message to be sent. The CRT displays "REMOTE MESSAGE SENT." The "H" key causes a menu to be displayed on the HP 9835A CRT screen. This menu lists the allowed commands and their functions. NO GPIB commands are sent. The "X" key allows the operator to send one line of data across the bus. The line of data is terminated by a carriage return and line feed produced by pressing the "CONTINUE" key on the HP 9835A.

The characters are stored in the sequence entered into a buffer whose maximum size is 80 characters. Pressing the "CONTINUE" key terminates storing characters in the array and all characters including the carriage return and line feed are sent. EOI is then sent true with a false byte of OOH. This false byte is due to the 1975 standard which allows asynchronous sending and reception of EOI. (The 8291A supports the later 1978 standard which eliminates this false byte.)

After any key command is serviced control returns to the loop which checks for SRQ active. Should SRQ be active, then the keyboard interrupt is disabled and a message printed to indicate that SRQ has been received true.

The controller then performs a parallel poll.

This is an example of how parallel poll may be used to quickly check which group of devices contains a device sending SRQ. The eight devices in a group would, of course, have software drivers which allow a true response to a PPOL if that device is currently driving SRQ true. This would be a valuable method of isolation of the SRQ source in a system with a large number of devices. In this application program, only the response from the 8291A is of concern and only the 8291A's response is considered. It does, however, demonstrate the technique employed. If a true response from the 8291A is detected, then a message to this effect is printed on the HP 9835A CRT screen. From this process, the controller has identified the device requesting service and will use a serial poll (SPOL) to determine the reason for the service request. This method of using PPOL is not specifically defined by the IEEE-488 standard but is a use of the resources provided.

The controller software then prints a message to indicate that it is about to perform a serial poll. This serial poll will return to the controller the current status of the 2819A and clear the service request. The status byte received is then printed on the CRT screen of the HP 9835A. One of the 8291A status bits indicates that the 8291A system has a field (on line or less) of data to transfer to the HP 9835A. If this bit is set, then the HP 9835A addresses the 8291A system to talk. The data is sent by the 8291A system is then printed on the CRT screen of the HP 9835A. The HP 9835 then enables the keyboard interrupts and goes into its SRQ checking loop.

Appendix 4 contains the software for the 8291A system which is connected to the HP 9835A via the GPIB. This software throws away the first byte of data it receives since this transfer was used by the HP 9835A to test when the 8291A system came on line.

Next, both status registers are read and stored in the two variable STAT 1 and STAT 2. It is necessary to store the status since reading the status registers clears the status bits.

Initially, six status bits are evaluated (END, GET, CPT, DEC, REMC, ADSC). Some of these conditions require that additional status bits be evaluated.

If END is true, then the 8291A system has received a block from the HP 9835A and the contents of a buffer is printed on the CRT screen. Next, the CPT bit is checked. PPC and PPE are only valid undefined commands in this example.

Next, the GET bit is examined and if true, the CRT screen connected to the serial channel on the 8291A system prints a message to indicate that the trigger command has been received. A similar process occurs with the DEC and REMC status bits.

Address Status Chagne (ADSC) is checked to see if the 8291A has been addressed or unaddressed by the controller. If ADSC is false, then the software checks the keyboard at the CRT terminal. If ADSC is set, then the TA and LA bits are read and evaluated to determine whether the 8291A has been addressed to talk or listen. The DMA controller is set to start transfers at the start of the character buffer and the type of transfer is determined by whether the 8291A in in TADS or LADS. We only need to set up the DMA controller since the transfers will be transparent to the system processor. The keyboard from the CRT terminal is then checked. If a key has been hit, then this character is stored in the character buffer and the buffer printer set to the next character location. This process repeats until the received character is a line feed. The line feed is echoed to the CRT, the serial poll status byte updated and the SRQ line driven true. This allows the 8291A system to store up to one line of characters before requesting a transfer to the controller. Recall that upon receiving an SRQ, the controller will perform a serial poll and subsequently address the 8291A to talk. The 8291A system then goes back to reading the status register thus repeating the process.

CONCLUSION

This application note has shown a basic method to view the IEEE 488 bus, when used in conjunction with Intel's 8291A.

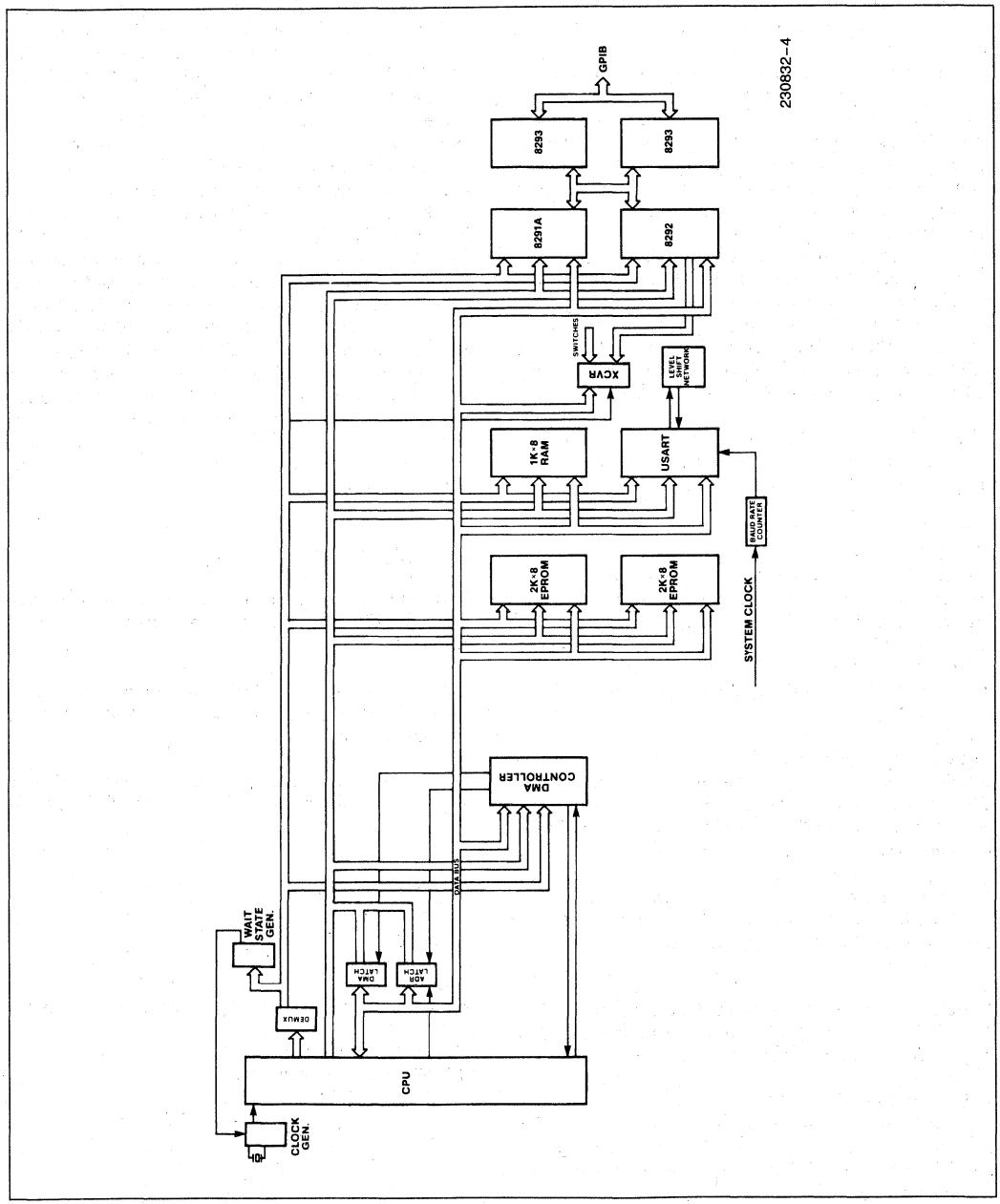
The main reference for GPIB questions is the IEEE Standard 488—1978. Reference 8291A's data sheet for detailed information on it.

Additional Intel GPIB products include iSBX-488, which is a multimode board consisting of the 8291A, 8292, and 8293.

REFERENCES

8291A Data Sheet
8292 Data Sheet
8293 Data Sheet
Application Note #66 "Using the 8292 GPIB Controller"
PLM-86 User Manual
HP 9835A User's Manual
IEEE—488—1978 Standard

APPENDIX A
SYSTEM BLOCK DIAGRAM WITH 8088



APPENDIX B
SOFTWARE DRIVERS FOR BLOCK DATA TRANSFER

```
PL/M-86 COMPILER     BOARD 1

ISIS-II PL/M-86  V1.1  COMPILATION OF MODULE BOARD 1
OBJECT MODULE PLACED IN:  F1:  BRD1  OBJ
COMPILER INVOKED BY:      PLM86.  F1:  BRD1.  SRC SYMBOLS MEDIUM

          /*    BOARD 1 TPT PROGRAM          */
          /*    THIS BOARD TALKS TO THE OTHER BOARD BY  */
          /*    TRANSFERRING A BLOCK OF DATA VIA THE 8237  */
          /*    COUPLED WITH THE 8291A.     THE 8291A IS PROGRAM- */
          /*    MED TO SEND EOI WHEN RECOGNIZING THE LAST     */
          /*    DATA BYTE'S BIT PATTERN.   WHILE DATA IS BEING */
          /*    TRANSFERRED, THE PROCESSOR PERFORMS  I/O READS */
          /*    OF THE 8237 COUNT REGISTERS TO SIMULATE BUS    */
          /*    ACTIVITY, AND TO DETERMINE WHEN TO TURN THE    */
          /*    LINE AROUND.  AFTER THE 8237 HAS REACHED  */
          /*    TERMINAL COUNT, THE 8291A IS PROGRAMMED TO      */
          /*    THE LISTENER STATE AND WAITS FOR THE BLOCK    */
          /*    TO BE TRANSMITTED BACK FROM THE SECOND BOARD.  */
          /*    THIS DATA IS PLACED IN A SECOND BUFFER AND     */
          /*    ITS CONTENTS COMPARED WITH THE ORIGINAL DATA  */
          /*    TO CHECK FOR INTERFACE INTEGRITY.   */

1         BOARD1:

          DO;
               /* PROCEDURES */

2    1         CO:  PROCEDURE (XXX) ;
3    2              DECLARE XXX BYTE,
                         SER$STAT  LITERALLY  'OFFF2H',
                         SER$DATA  LITERALLY  'OFFF0H',
                         TXRDY     LITERALLY  '01H',
4    2              DO WHILE (INPUT  (SER$STAT) AND TXRDY) <>     TXRDY;
5    3              END;
6    2              OUTPUT (SER$DATA)      = XXX;
7    2         END CO;

                    /*   SETUP BUFFERS */
8    1              DECLARE BUFF2 (100)  BYTE;   /*  RAM STORAGE AREA */
9    1                   DECLARE BUFF1 (100) BYTE DATA

                    (1,2,3,4,5,6,7,8,9,10H,
                    11H, 12H, 13H, 14H, 15H, 16H, 17H, 18H, 19H, 20H,
                    21H, 22H, 23H, 24H, 25H, 26H, 27H, 28H, 29H, 30H,
                    31H, 32H, 33H, 34H, 35H, 36H, 37H, 38H, 39H, 40H,
                    41H, 42H, 43H, 44H, 45H, 46H, 47H, 48H, 49H, 50H,
                    51H, 52H, 53H, 54H, 55H, 56H, 57H, 58H, 59H, 60H,
                    61H, 62H, 63H, 64H, 65H, 66H, 67H, 68H, 69H, 70H,
                    71H, 72H, 73H, 74H, 75H, 76H, 77H, 78H, 79H, 80H,
                    81H, 82H, 83H, 84H, 85H, 86H, 87H, 88H, 89H, 90H,
```

230832-5

```
PL/M-86  COMPILER  BOARD1

                    91H, 92H, 93H, 94H, 95H, 96H, 97H, 98H, 99H, ODH);
10  1               DECLARE BUFF3(17)  BYTE DATA
                    (ODH, OAH, 'COMPARE ERROR', ODH, OAH);   /* ROM STORAGE AREA */

                    /* 8237 PORT ADDRESSES  */

11  1               DECLARE

                    CLEAR$FF        LITERALLY 'OFFDDH', /* MASTER CLEAR */
                    START$0$LO      LITERALLY 'OFFD0H',
                    START$0$HI       LITERALLY 'OFFD0H',
                    0$COUNT$LO      LITERALLY 'OFFD1H',
                    0$COUNT$HI      LITERALLY 'OFFD1H',
                    SET$MODE        LITERALLY 'OFFDBH',
                    CMD$37          LITERALLY 'OFFD8H',
                    SET$MASK        LITERALLY 'OFFDFH';

                    /* 8237 COMMAND  - DATA BYTES */
12  1               DECLARE         DMA$ADR$TALK    POINTER;
13  1               DECLARE         DMA$ADR$LSTN    POINTER;

14  1               DECLARE

                    RD$TRANSFER     LITERALLY '48H',
                    WR$TRANSFER     LITERALLY '44H',
                    NORM$TIME       LITERALLY '20H',
                    TC$LO1          LITERALLY 'OFFH',
                    TC$HI1          LITERALLY '00H',
                    TC$LO2          LITERALLY '99D',          /* 100 XFERS  */
                    TC          LITERALLY '01H',
                    I               BYTE;

15  1               DECLARE

                    DMA$WRD$TALK (2)  WORD    AT    (@DMA$ADR$TALK),
                    DMA$WRD$LSTN(2)  WORD     AT    (@DMA$ADR$LSTN);

                    /* 8291A PORT ADDRESSES */

16  1               DECLARE

                    PORT$OUT        LITERALLY 'OFFC0H', /* DATA OUT*/
                    PORT$IN         LITERALLY 'OFFC0H'
                    STATUS$1        LITERALLY 'OFFC1H', /*INTR STAT 2*/
                    STATUS$2        LITERALLY 'OFFC2H', /* INTR STAT 2 */
                    ADDR$STATUS     LITERALLY 'OFFC4H',
                    COMMAND$MOD     LITERALLY 'OFFC5H', /*CMD PASS THRU */
                    ADDR$0          LITERALLY 'OFFC6H',
                    EOS$REG         LITERALLY 'OFFC7H', /*  EOS REGISTER */
```

230832-6

```
                /* 8291A COMMAND  - DATA BYTES */

PL/M-86 COMPILER    BOARD1

17  1                   DECLARE

                    END$EOI    LITERALLY        '88H',
                    DNE   LITERALLY        '10H',
                    PON   LITERALLY        '00H',
                    RESET      LITERALLY        '02H',
                    CLEAR      LITERALLY        '00H',
                    DMA$REQ$L  LITERALLY        '10H',
                    DMA$REQ$T  LITERALLY        '20H',
                    MOD1$TO    LITERALLY        '80H',
                    MOD1$LO    LITERALLY        '40H',
                    EOS   LITERALLY        'ODH',
                    PRESCALER LITERALLY        '23H',
                    HIGH$SPEED LITERALLY            '0A4H',
                    OKAY       LITERALLY        'OFFFFH',
                    XYZ   BYTE,
                    MATCH      WORD,
                    BO    LITERALLY        '02H',
                    BI    LITERALLY        '01H',
                    ERR   LITERALLY        '04H';

          /* CODE BEGINS */

18  1     START91:

                    OUTPUT   (STATUS$2)   =CLEAR;   /* SHUT-OFF DMA REQ BITS TO */
                                                    /* PREVENT EXTRA DMA REQS  */
                                                     /*FROM 8291A                */

                    /* MANIPULATE DMA ADDRESS VARIABLES */

19  1               DMA$ADR$TALK    =(@BUFF1);
20  1               DMA$ADR$LSTN    =(@BUFF2);
21  1               DMA$WRD$TALK(1)=SHL (DMA$WRD$TALK(1), 4);
22  1               DMA$WRD$TALK(0)=DMA$WRD$TALK (0) + DMA$WRD$TALK (1);
23  1               DMA$WRD$LSTN(1)=SHL (DMA$WRD$LSTN (1), 4);
24  1               DMA$WRD$LSTN(0)=DMA$WRD$LSTN (0) +DMA$WRD$LSTN (1);

25  1          INIT37:
                /* INIT 8237 FOR TALKER FUNCTIONS */

26  1                   OUTPUT     (CLEAR$FF)      =CLEAR;/* TOGGLE MASTER CLEAR */
27  1                   OUTPUT     (CMD$37)        =NORM$TIME;
28  1                   OUTPUT     (SET$MODE)      =RD$TRANSFER;
                        OUTPUT     (SET$MASK)      =CLEAR;
29  1                   OUTPUT     (START$0$LO)       =DMA$WRD$TALK (0);
30  1                   DMA$WRD$TALK (0)           =SHR (DMA$WRD$TALK (0), 8);
31  1                   OUTPUT     (START$0$HI)       =DMA$WRD$TALK (0);
32  1                   OUTPUT     (0$COUNT$LO)       =TC$LO2;
33  1                   OUTPUT     (0$COUNT$HI)       =TC$HI2;
                        /* INIT 8291A FOR TALKER FUNCTIONS */

PL/M-86 COMPILER      BOARD1
```

230832-7

```
34  1                OUTPUT  (EOS$REG)     =EOS;
35  1                OUTPUT  (COMMAND$MOD)      =END$EOI; /*  EOI ON EOS SENT */
36  1                OUTPUT  (ADDR$STATUS)      =MOD1$TO; /* TALK ONLY */
37  1                OUTPUT  (COMMAND$MOD)      =PRESCALER;
38  1                OUTPUT  (COMMAND$MOD)      =HIGH$SPEED;
39  1                OUTPUT  (COMMAND$MOD)      =PON;

40  1                DO WHILE (INPUT  (STATUS$1) AND BO)  =0;
41  2                END;     /*  WAIT FOR BO INTR  */
42  1                OUTPUT  (PORT$OUT)  = 0AAH;

43  1                DO WHILE  (INPUT  (STATUS$1) AND ERR) = ERR;
44  2                   DO WHILE  (INPUT  (STATUS$1)  AND BO) = 0;
45  3                   END;  /*  WAIT FOR BO INTR  */
46  2                   OUTPUT  (PORT$OUT)  =0AAH;
47  2                END;

48  1                OUTPUT  (STATUS$2)  =DMA$REQ$T;  /*  ENABLE DMA REQS */

49  1                    DO WHILE  (INPUT  (CMD$37)  AND TC) <>         TC;
                      /* WAIT FOR TC  = 0 */
50  2                    END;

51  1                INIT37L;

                      OUTPUT  (STATUS$2)  =CLEAR;  /*  DISABLE DMA REQS */

                      /* INIT 8237 FOR LISTENER FUNCTIONS */

52  1                OUTPUT  (CLEAR$FF) 0=CLEAR; /*  TOGGLE MASTER RESET */
53  2                OUTPUT  (CMD$37)     =NORM$TIME;
54  1                OUTPUT  (SET$MODE)  =WR$TRANSFER;
55  1                OUTPUT  (SET$MASK)  =CLEAR;
56  1                OUTPUT  (START$0$LO)    =DMA$WRD$LSTN (0);
57  1                DMA$WRD$LSTN (0)     =SHR (DMA$WRD$LSTN (0), 8);
58  1
      OUTPUT      (START $0$HI)      =DMA$WRD$LSTN (0);
59  1                OUTPUT  (0$COUNT$LO)      =TC$LO1;
60  1                OUTPUT  (0$COUNT$HI)      =TC$HI1;

                      /* INIT 8291A FOR LISTENER FUNCTIONS  */

61  1                OUTPUT  (COMMAND$MOD)     =RESET;
62  1                OUTPUT  (ADDR$STATUS)     =MOD1$LO; /*  LISTEN ONLY */
63  1                OUTPUT  (COMMAND$MOD)     =PON;

64  1                DO WHILE (INPUT  (STATUS$1)  AND BI)  =0;
65  2                END;  /*  WAIT FOR BI INTR */
66  1                XYZ  = INPUT (PORT$IN);

67  1                OUTPUT  (STATUS$2)  =DMA$REQ$L;  /*  ENABLE DMA REQS  */

68  1                DO WHILE  (INPUT  (STATUS$1)  AND DNE)<>          DNE;
                      /*  WAIT FOR EOI RECEIVED  */
```

230832-8

```
PL/M-86 COMPILER    BOARD 1

     70    1     CMPBLKS

                 /* COMPARE THE TWO BUFFERS CONTENTS */

                 MATCH=CMPB   (@BUFF1, @BUFF2, 100);

     71    1     IF MATCH = OKAY THEN GOTO START91;

                 /*  SEND ERROR MESSAGE IN BUFFER 3  */

     73    1     DO I=0 TO 16;
     74    2        CALL CO  (BUFF 3  (I)  );
     75    2     END;

     76    1     GOTO START91;

     77    1     END;


MODULE INFORMATION:

     CODE AREA SIZE          =01DBH    475D
     CONSTANT AREA SIZE      =0075H    117D
     VARIABLE AREA SIZE      =0070H    112D
     MAXIMUM STACK SIZE      =0006H      6D
     243 LINES READ
     0 PROGRAM ERROR (S)


END OF PL/M-86  COMPILATION
```

230832-9

```
PL/M-86 COMPILER    BOARD2

ISIS-II PL/M-86  V1.1 COMPILATION OF MODULE BOARD2
OBJECT MODULE PLACED IN  F1: BRD2.OBJ
COMPILER INVOKED BY:  PLM86  F1: BRD2.SRC


                        /* BOARD 2 TPT PROGRAM          */
                        /*                              */
                        /* THIS BOARD LISTENS TO THE OTHER BOARD (1)    */
                        /* AND DMA'S DATA INTO A BUFFER, WHILE WAITING */
                        /* FOR THE END INTERRUPT BIT TO BECOME ACTIVE   */
                        /* UPON END ACTIVE, THE DATA IN THE BUFFER IS   */
                        /* SENT BACK TO THE FIRST BOARD VIA THE GPID    */
                        /* WHEN THE BLOCK IS FINISHED THE 8291A IS   */
                        /* PROGRAMMED BACK INTO THE LISTENER MODE     */

     1          BOARD2:

                DO;
                        /* 8237 PORT ADDRESSES */

     2    1             DECLARE

                        CLEAR$FF        LITERALLY       'OFFDDH',     /*MASTER CLEAR */
                        START$0$LO      LITERALLY       'OFFD0H',
                        START$0$HI      LITERALLY       'OFFD0H',
                        0$COUNT$LO      LITERALLY       'OFFD1H',
                        0$COUNT$HI      LITERALLY       'OFFD1H',
                        SET$MODE        LITERALLY       'OFFD8H',
                        CMD$37          LITERALLY       'OFFD8H',
                        SET$MASK        LITERALLY       'OFFDFH',

                        /* 8237 COMMAND  - DATA BYTES   */

     3    1             DECLARE

                        RD$TRANSFER     LITERALLY       '48H',
                        WR$TRANSFER     LITERALLY       '44H',
                        ADDR$1A         LITERALLY       '00H',
                        ADDR$1B         LITERALLY       '01H',
                        NORM$TIME       LITERALLY       '20H',
                        TC$LO1          LITERALLY       'OFFH',
                        TC$HI1          LITERALLY       '00H',
                        TC$LO2          LITERALLY       '99D',
                        TC$HI2          LITERALLY       '00H',
                        TC      LITERALLY       '01H';

                        /* 8291A  PORT ADDRESSES */

     4    1             DECLARE

                        PORT$OUT        LITERALLY       'OFFC0H',
                        PORT$IN         LITERALLY       'OFFC0H',/* DATA IN */
                        STATUS$1        LITERALLY       'OFFC1H', /* INTR STAT 1 */
                        STATUS$2        LITERALLY       'OFFC2H', /* INTR STAT 2 */
                        ADDR$STATUS     LITERALLY       'OFFC4H', /* ADDR STAT   */
                        COMMAND$MOD     LITERALLY       'OFFC5H', /* CMD PASS THRU */
```

230832-10

```
PL/M-86 COMPILER  BOARD2

                    ADDR$0      LITERALLY       'OFFC6H',
                    EOS$REG     LITERALLY       'OFFC7H', /*  EOS REGISTER */

                    /* 8291A  COMMAND - DATA BYTES */

5   1               DECLARE

                    END$EOI     LITERALLY       '88H',
                    DNE   LITERALLY       '10H',
                    PON   LITERALLY       '00H',
                    RESET       LITERALLY       '02H',
                    CLEAR       LITERALLY       '00H',
                    DMA$REQ$L LITERALLY       '10H',
                    DMA$REQ$T LITERALLY       '20H',
                    MOD1$TO     LITERALLY       '80H',
                    MOD1$LO     LITERALLY       '40',
                    EOS   LITERALLY       'ODH',
                    PRESCALER LITERALLY       '23H',
                    HIGH$SPEED LITERALLY      'A4H',
                    XYZ         BYTE,
                    BO          LITERALLY '02H',
                    BI          LITERALLY '01H',
                    ERR         LITERALLY '04H',

6   1           START91;

                    OUTPUT  (STATUS$2)  =CLEAR;   /*  END INITILIZATION STATE */

                    /* INIT 8237 FOR LISTENER FUNCTION  */

7   1           INIT37L;

                    OUTPUT  (CLEAR$FF)  =CLEAR; /*  TOGGLE MASTER RESET  */
8   1               OUTPUT  (CMD$37)    =NORM$TIME;
9   1               OUTPUT  (SET$MODE)  =WR$TRANSFER;  /*  BLOCK XFER MODE  */
10  1               OUTPUT  (SET$MASK)  =CLEAR;
11  1               OUTPUT  (START$0$LO)      =ADDR$1A;
12  1               OUTPUT  (START$0$HI)      =ADDR$1B;
13  1               OUTPUT  (0$COUNT$LO)      =TC$LO1;
14  1               OUTPLUT (0$COUNT$HI)      =TC$HI1;

                    /*  INIT 8291A  FOR LISTENER FUNCTIONS   */

15  1               OUTPUT  (COMMAND$MOD)     =RESET;
16  1               OUTPUT  (ADDR$STATUS)     =MOD1$LO;
17  1               OUTPUT  (COMMAND$MOD)     =PON;
18  1               DO WHILE (INPUT  (STATUS$1)  AND BI)  =0;
19  2               END;  /*  WAIT FOR BI INTR */
20  1               XYZ= INPUT  (PORT$IN);
21  1               OUTPUT   (STATUS$2)     =DMA$REQ$L;

                    /* WAIT UNTIL EOI RCVD AND END INTR-BIT SET  */

22  1                  DO WHILE  (INPUT  (STATUS$1)  AND DNE ) <>   DNE;
```

230832-11

```
PL/M-86  COMPILER  BOARD2

23  1               END;

24  1          INIT37T:
                 /* INIT 8237 FOR TALKER FUNCTION  */

                 OUTPUT  (STATUS$2)  =CLEAR;    /* CLEAR  8291A  DRQ  */
25  1            OUTPUT  (CLEAR$FF)  =CLEAR;
26  1            OUTPUT  (CMD$37)    =NORM$TIME;
27  1            OUTPUT  (SET$MODE)  =RD$TRANSFER;   /*  BLOCK XFER MODE */
28  1            OUTPUT  (SET$MASK)  =CLEAR;
29  1            OUTPUT  (START$0$LO)      =ADDR$1A;
30  1            OUTPUT  (START$0$HI)      =ADDR$1B;
31  1            OUTPUT  (0$COUNT$LO)      =TC$LO2;
32  1            OUTPUT  (0$COUNT$HI)      =TC$HI2;

                 /*  INIT 9291A  FOR TALKER FUNCTION  */

33  1            OUTPUT  (EOS$REG)     =EOS;
34  1            OUTPUT  (COMMAND$MOD)         =END$EOI; /* EOI ON EOS SENT  */
35  1            OUTPUT  (ADDR$STATUS)         =MOD1$TO; /* TALK ONLY  */
36  1            OUTPUT  (COMMAND$MOD)         =PRESCALER;
37  1            OUTPUT  (COMMAND$MOD)         =HIGH$SPEED;
38  1            OUTPUT  (COMMAND$MOD)         =PON;

39  1            DO WHILE  (INPUT  (STATUS$1)   AND BO)    =0;
40  2            END;     /* WAIT FOR BO INTR  */
41  1            OUTPUT  (PORT$OUT)    =0AAH;

42  1            DO WHILE  (INPUT  (STATUS$1)  AND ERR)    =ERR;
43  2              DO WHILE   (INPUT  (STATUS$1)   AND BO)    =0;
44  3              END;     /* WAIT FOR BO INTR  */
45  2              OUTPUT  (PORT$OUT)  =0AAH;
46  2            END;

47  1            OUTPUT  (STATUS$2)     =DMA$REG$T;
                 /*  WAIT FOR TC=0  */

48  1                DO WHILE   (INPUT   (CMD$37)  AND TC) <>  TC;
49  2                END;

50  1            GOTO START91;

51  1     END;


MODULE INFORMATION

     CODE AREA SIZE       =0122H    290D
     CONSTANT AREA SIZE   =0000H      0D
     VARIABLE AREA SIZE   =0001H      1D
     MAXIMUM STACK SIZE   =0000H      0D
     152 LINES READ
     0 PROGRAM ERROR (S)
```

230832-12

APPENDIX C
SOFTWARE FOR HP 9835A

```
10   REM SEND IN
TERFACE CLEAR
20   ABORTIO 7
30   REM FORCE E
RRORS UNTIL LIST
ENERS ACTIVE
40 Frcerr:   OUT
PUT 704 USING "#
;K";"B"
50 Chkstst:   ST
ATUS 7;Stat1,Sta
t2,Stat3,Stat4
60   Err=Stat2 A
ND 1
70   IF Err=1 TH
EN GOTO Frcerr
80   PRINT CHR$(
12),"LISTENERS A
RE ON LINE "
90   REM CONFIGU
RE PPOLL
100  PPOLL CONF
GURE 704;"0000
00"
110

   ! response on
 bit 4
120  PRINT CHR$(
12),"PARALLEL PO
LL CONFIGURED"
130  REM ENABLE
KEYBOARD INTERRU
PT
140  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
150 Keyen:   ON K
BD GOSUB 610
160  STATUS 7;St
at1,Stat2,Stat3,
Stat4
170  Sra=BINAND(
Stat1,L$8)
180  IF Sra=0 TH
EN GOTO Keyen
190  OFF KBD
200  PRINT CHR$(
12),"SRQ RECEIVE
D"
210  PRINT "SEND
ING PARALLEL POL
L RESPONSE MESSA
GE"
220  REM EXECUTI
NG PARALLEL POLL
230  Ppollbyte=P
POLL(7)
240  PRINT "PARA
LLEL POLL BYTE =
 "iPpollbyte
250  PRINT "----
----------------
------"
260  Ppollbyte=B
INAND(Ppollbyte,
8)
270  IF Ppollbyt
e=0 THEN GOTO 29
291
280  PRINT "SR
NOT FROM 8291"
281  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
290  GOTO Keyen
300 F8291:   PRIN
T "SRQ IS FROM N
CC 8291 . . . THE
ENTERPRISE"
310  PRINT "PERF
ORMING SERIAL PO
LL TO GET STATUS
"
320  STATUS 704;
Stat
330  PRINT CHR$(
12),"Status = ";
Stat
340  Dxfer=BINAN
D(Stat,1)
520  IF Dxfer>0
THEN GOTO Rcvr
530  GOTO Keyen
531 Rcvr:   REM R
EADY TO RCV CHAR
S FROM GPIB
540  DIM G$[80]
550  ENTER 704 U
SING "%,T";G$
560  PRINT CHR$(
12),G$
570  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
580  GOTO Keyen
590  REM INTERRU
PT SERVICE ROUTI
NES
600  REM GET KEY
BOARD DATA
610 Whatkey:   DI
M K$[80]
620  K$=KBD$
630  IF K$="G" T
HEN GOTO Get
640  IF K$="D" T
HEN GOTO Dec
650  IF K$="R" T
HEN GOTO Rem
660  IF K$="H" T
HEN GOTO Help
670  IF K$="X" T
HEN GOTO Xmit
680 Get:   TRIGGE
R 704
690  PRINT CHR$(
12),"GROUP EXECU
TE TRIGGER SENT"
700  PRINT " "
710  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
720  RETURN
730 Dec:   RESET
704
740  PRINT CHR$(
12),"SELECTIVE D
EVICE CLEAR SENT
"
750  PRINT " "
760  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
770  RETURN
780 Rem:   LOCAL
704
790  PRINT CHR$(
12),"REMOTE MESS
AGE SENT"
800  PRINT " "
810  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
820  RETURN
830 Help:   PRINT
 CHR$(12)
840  PRINT " @@@
@ OPERATOR ALLOW
ABLE COMMANDS @@
@@ "
850  PRINT " hit
 key   result"
860  PRINT "   G
      Send GET m
essage"
870  PRINT "   D
      Send DEC m
essage"
880  PRINT "   R
      Send REM-L
OC message"
890  PRINT "   X
      Xmits keyb
oard input to 82
91"
900  PRINT "   H
      Prints thi
s table"
910  PRINT " "
920  PRINT "...
go ahead, TRY IT
 !"
930  RETURN
940 Xmit:   DIM A
$[80]
950  PRINT CHR$(
12),"Enter data
 to send and hit
 CONTINUE"
960  INPUT A$
970  OUTPUT 704;
A$
971  EOI 7;0
980  PRINT "COMM
AND = ?     (HIT
 'H' FOR LIST)"
990  RETURN
1000 END
```

230832-13

230832-14

230832-15

APPENDIX D
SOFTWARE FOR HP 8088/HP 9835A VIA GPIB

```
PL/M-86 COMPILER     HPIB


ISIS-II PL/M-86 V1.1 COMPILATION OF MODULE HPIB
OBJECT MODULE PLACED IN :F1:HPIB.OBJ
COMPILER INVOKED BY:  PLM86 :F1:HPIB.SRC LARGE


   1            HPIB:
                /*

                PARAMETER DECLARATIONS
                */

                DO;

   2    1       DECLARE

                ADDR$HI       LITERALLY    '01H',
                ADDR$LO       LITERALLY    '00H',
                ADSC          LITERALLY    '01H',
                BI         LITERALLY    '01H',
                BO         LITERALLY    '02H',
                CHAR$COUNT    BYTE,
                CHAR          BYTE,
                CHARS(80)     BYTE,
                CLEAR         LITERALLY    '00H',
                CPT        LITERALLY    '80H',
                CRLF          LITERALLY    '0AH',
                DEC        LITERALLY    '08H',
                DMA$ADR$LSTN      POINTER,
                DMA$ADR$TALK      POINTER,
                DMA$WRD$LSTN(2) WORD AT       (@DMA$ADR$LSTN),
                DMA$WRD$TALK(2) WORD AT       (@DMA$ADR$TALK),
                DMA$REG$L     LITERALLY    '10H',
                DMA$REG$T     LITERALLY    '20H',
                DNE        LITERALLY    '10H',
                END$EOI       LITERALLY    '88H',
                EOS        LITERALLY    '0DH',
                ERR        LITERALLY    '04H',
                GET        LITERALLY    '20H',
                I          BYTE,
                LISTEN        LITERALLY    '04H',
                MLA        LITERALLY    '04H',
                MODE$1        LITERALLY    '01H',
                NO$DMA        LITERALLY    '00H',
                NO$RSV        LITERALLY    '00H',
                NORM$TIME     LITERALLY    '20H',
                PON        LITERALLY    '00H',
                PPC        LITERALLY    '05H',
                PPE$MASK      LITERALLY    '60H',
                PPOLL$CNFG$FLAG LITERALLY    '01H',
                PPOLL$EN$BYTE     BYTE,
                PRI$BUF(80) BYTE AT       (@CHARS),
                RD$XFER       LITERALLY    '48H',
                RESET         LITERALLY    '02H',
                REMC          LITERALLY    '02H',
                RSV        LITERALLY    '40H',
                RXRDY         LITERALLY    '02H',
```

230832-16

```
PL/M-86 COMPILER     HPIB


                SRQS            LITERALLY   '40H',
                STAT1           BYTE,
                STAT2           BYTE,
                TALK            LITERALLY   '02H',
                TA$OR$LA        BYTE,
                TRQ         LITERALLY   '41H',
                TC          LITERALLY   '01H',
                TC$HI           LITERALLY   '00H',
                TC$LO           LITERALLY   '0FFH',
                TXRDY           LITERALLY   '01H',
                UDC         BYTE,
                WR$XFER         LITERALLY   '44H',
                XYZ         BYTE;

                /*

                PORT DECLARATIONS

                */

  3   1         DECLARE

                ADDR$0          LITERALLY   '0FFC6H',
                ADDR$STATUS     LITERALLY   '0FFC4H',
                CLEAR$FF        LITERALLY   '0FFDDH',
                CMD$37          LITERALLY   '0FFD8H',
                COMMAND$MOD     LITERALLY   '0FFC5H',
                COUNT$HI        LITERALLY   '0FFD1H',
                COUNT$LO        LITERALLY   '0FFD1H',
                CPT$REG         LITERALLY   '0FFC5H',
                EOS$REG         LITERALLY   '0FFC7H',
                PORT$IN         LITERALLY   '0FFC0H',
                PORT$OUT        LITERALLY   '0FFC0H',
                SER$DATA        LITERALLY   '0FFF0H',
                SER$STAT        LITERALLY   '0FFF2H',
                SET$MASK        LITERALLY   '0FFDFH',
                SET$MODE        LITERALLY   '0FFDBH',
                SPOLL$STAT      LITERALLY   '0FFC3H',
                START$HI        LITERALLY   '0FFD0H',
                START$LO        LITERALLY   '0FFD0H',
                STATUS$1        LITERALLY   '0FFC1H',
                STATUS$2        LITERALLY   '0FFC2H';

                /* crt  messages list */


  4   1         DECLARE GET$MSG(11) BYTE DATA (0DH,0AH,'TRIGGER',0AH,0DH);
  5   1         DECLARE DEC$MSG(16) BYTE DATA (0DH,0AH,'DEVICE CLEAR',0AH,0DH);
  6   1         DECLARE REMC$MSG(10) BYTE DATA (0DH,0AH,'REMOTE',0DH,0AH);
  7   1         DECLARE CPT$MSG(22) BYTE DATA (0DH,0AH,'UNDEF CMD RECEIVED',0AH,0DH);
  8   1         DECLARE HUH$MSG(11) BYTE DATA (0DH,0AH,'HUH ???',0DH,0AH);


                    /* called procedures */

  9   1         REGSER:     PROCEDURE;
```

230832-17

```
PL/M-86 COMPILER     HPIB

  10   2              OUTPUT (SPOLL$STAT)=TRG;

  11   2              DO WHILE (INPUT (SPOLL$STAT) AND SRQS)=SRQS;
  12   3              END;

  13   2              OUTPUT (SPOLL$STAT)=NO$RSV;

  14   2              END REQSER;

  15   1      CO: PROCEDURE(XXX);
  16   2              DECLARE
                        XXX         BYTE;

  17   2              DO WHILE (INPUT (SER$STAT) AND TXRDY)<>TXRDY;
  18   3              END;
  19   2              OUTPUT (SER$DATA)=XXX;
  20   2          END CO;
  21   1      HUH:        PROCEDURE;
  22   2                  DO I=0 TO 10;
  23   3                   CALL CO (HUH$MSG(I));
  24   3                  END;
  25   2              END HUH;

  26   1      CI:     PROCEDURE;

  27   2              IF (INPUT (SER$STAT) AND RXRDY)=RXRDY THEN
  28   2                  DO;
  29   3                      I=0;
  30   3                      CHAR$COUNT=0;
  31   3      STORE$CHAR:         CHAR=(INPUT (SER$DATA) AND 7FH);
  32   3                      CHAR$COUNT=CHAR$COUNT+1;
  33   3                      CALL CO (CHAR);
  34   3                      CHARS(I)=CHAR;
  35   3                      I=I+1;
  36   3                      IF CHAR <> CRLF THEN
  37   3                          DO;
  38   4                           DO WHILE (INPUT (SER$STAT) AND RXRDY) <>RXRDY;
  39   5                           END;
  40   4                           GOTO STORE$CHAR;
  41   4                          END;
  42   3                      CALL REQSER;
  43   3                  END;
  44   2              END CI;

  45   1      TALK$EXEC:  PROCEDURE;

  46   2              OUTPUT (STATUS$2)=CLEAR;

                      /*
                      manipulate address bits for DMA controller
                      */

  47   2              DMA$ADR$TALK=(@CHARS);
  48   2              DMA$WRD$TALK(1)=SHL(DMA$WRD$TALK(1),4);
  49   2              DMA$WRD$TALK(0)=DMA$WRD$TALK(0)+DMA$WRD$TALK(1);

  50   2              OUTPUT (CLEAR$FF)=CLEAR;
```

230832-18

```
PL/M-86 COMPILER     HPIB

  51   2              OUTPUT (CMD37)=NORM$TIME;
  52   2              OUTPUT (SET$MODE)=RD$XFER;
  53   2              OUTPUT (SET$MASK)=CLEAR;
  54   2              OUTPUT (START$LO)=DMA$WRD$TALK(0);
  55   2              DMA$WRD$TALK(0)=SHR(DMA$WRD$TALK(0),8);
  56   2              OUTPUT (START$HI)=DMA$WRD$TALK(0);
  57   2              OUTPUT (COUNT$LO)=CHAR$COUNT;
  58   2              OUTPUT (COUNT$HI)=0;


  59   2              OUTPUT (EOS$REG)=EOS;
  60   2              OUTPUT (COMMAND$MOD)=END$EOI;

  61   2              DO WHILE (INPUT (STATUS$1) AND BO)=0;
  62   3              END;
  63   2              OUTPUT (PORT$OUT)=0AAH;

  64   2              DO WHILE (INPUT (STATUS$1) AND ERR)=ERR;
  65   3                  DO WHILE (INPUT (STATUS$1) AND BO)=0;
  66   4                  END;
  67   3                  OUTPUT (PORT$OUT)=0AAH;
  68   3              END;
  69   2              OUTPUT (STATUS$2)=DMA$REQ$T;

  70   2              END TALK$EXEC;

  71   1      LISTEN$EXEC:    PROCEDURE;

  72   2              OUTPUT (STATUS$2)=CLEAR;
  73   2              OUTPUT (CLEAR$FF)=CLEAR;
  74   2              OUTPUT (CMD$37)=NORM$TIME;
  75   2              OUTPUT (SET$MODE)=WR$XFER;
  76   2              OUTPUT (SET$MASK)=CLEAR;
  77   2              DMA$ADR$LSTN=(@CHARS);
  78   2              DMA$WRD$LSTN(1)=SHL(DMA$WRD$LSTN(1),4);
  79   2              DMA$WRD$LSTN(0)=DMA$WRD$LSTN(0)+DMA$WRD$LSTN(1);
  80   2              OUTPUT (START$LO)=DMA$WRD$LSTN(0);
  81   2              DMA$WRD$LSTN(0)=SHR(DMA$WRD$LSTN(0),8);
  82   2              OUTPUT (START$HI)=DMA$WRD$LSTN(0);
  83   2              OUTPUT (COUNT$LO)=TC$LO;
  84   2              OUTPUT (COUNT$HI)=TC$HI;
  85   2              OUTPUT (STATUS$2)=DMA$REQ$L;

  86   2              END LISTEN$EXEC;

  87   1      PRINTER:    PROCEDURE;

  88   2              I=0;

  89   2              DO WHILE PRI$BUF(I) <>CRLF;
  90   3                  CALL CO (PRI$BUF(I));
  91   3                  I=I+1;
  92   3              END;
  93   2              CALL CO (PRI$BUF(I));

  94   2              END PRINTER;
```

230832-19

```
PL/M-86 COMPILER    HPIB


 95   1      ADSC$EXEC:  PROCEDURE;

 96   2              TA$OR$LA=INPUT (ADDR$STATUS);

 97   2              IF (TA$OR$LA AND TALK)=TALK THEN
 98   2                  CALL TALK$EXEC;
 99   2              IF (TA$OR$LA AND LISTEN)=LISTEN THEN
100   2                  CALL LISTEN$EXEC;

101   2              END ADSC$EXEC;

102   1      GET$EXEC:   PROCEDURE;
103   2                  DO I=0 TO 10;
104   3                   CALL CO (GET$MSG(I));
105   3                  END;
106   2              END GET$EXEC;

107   1      DEC$EXEC:   PROCEDURE;
108   2                  DO I=0 TO 15;
109   3                   CALL CO (DEC$MSG(I));
110   3                  END;
111   2              END DEC$EXEC;

112   1      REMC$EXEC:  PROCEDURE;
113   2                  DO I=0 TO 9;
114   3                   CALL CO (REMC$MSG(I));
115   3                  END;
116   2              END REMC$EXEC;

117   1      PPOLL$CON:  PROCEDURE;

118   2              OUTPUT (COMMAND$MOD)=PPOLL$CNFG$FLAG;

119   2              END PPOLL$CON;

120   1      PPOLL$EN:   PROCEDURE;

121   2              PPOLL$EN$BYTE=(UDC AND 6FH);
122   2              OUTPUT (COMMAND$MOD)=PPOLL$EN$BYTE;

123   2              END PPOLL$EN;

124   1      CPT$EXEC:   PROCEDURE;
125   2                  DO I=0 TO 21;
126   3                   CALL CO (CPT$MSG(I));
127   3                  END;

128   2                  UDC=INPUT (CPT$REG);
129   2                  UDC=(UDC AND 7FH);
130   2                  IF (UDC AND PPC)=PPC THEN
131   2                   CALL PPOLL$CON;

132   2                  IF (UDC AND PPE$MASK)=PPE$MASK THEN
133   2                   CALL PPOLL$EN;
```

230832-20

```
PL/M-86 COMPILER    HPIB


 134   2                END CPT$EXEC;
                 /*
                 BEGIN CODE
                 */

 135   1         INIT:

                     OUTPUT  (CLEAR$FF)  =CLEAR;
 136   1           OUTPUT  (COMMAND$MOD)   =RESET;
 137   1           OUTPUT  (ADDR$STATUS)   =MODE$1;
 138   1           OUTPUT  (ADDR$O)    =MLA;
 139   1           OUTPUT  (STATUS$2)  =NO$DMA;
 140   1           OUTPUT  (COMMAND$MOD)   =PON;

 141   1         LISTENERS:

                     /* response to listeners check */


                     DO WHILE (INPUT (STATUS$1) AND BI)=0;
 142   2           END;

 143   1           XYZ=INPUT (PORT$IN);
 144   1           XYZ=INPUT (STATUS$2);

 145   1         CMD:

                 RDSTAT:
                     /* read status registers and interpret command */

                     STAT1=INPUT (STATUS$1);
 146   1           STAT2=INPUT (STATUS$2);

 147   1           IF (STAT1 AND DNE)=DNE THEN
 148   1              CALL PRINTER;
 149   1           IF (STAT1 AND CPT)=CPT THEN
 150   1               DO;
 151   2               CALL CPT$EXEC;
 152   2               STAT2=(STAT2 AND OFEH);
 153   2               END;
 154   1           IF (STAT1 AND GET)=GET THEN
 155   1               DO;
 156   2               CALL GET$EXEC;
 157   2               STAT2=(STAT2 AND OFEH);
 158   2               END;
 159   1           IF (STAT1 AND DEC)=DEC THEN
 160   1               DO;
 161   2               CALL DEC$EXEC;
 162   2               STAT2=(STAT2 AND OFEH);
 163   2               END;
 164   1           IF (STAT2 AND REMC)=REMC THEN
 165   1               DO;
 166   2               CALL REMC$EXEC;
 167   2               STAT2=(STAT2 AND OFEH);
 168   2               END;
 169   1           IF (STAT2 AND ADSC)=ADSC THEN
```

230832-21

```
PL/M-86 COMPILER     HPIB

 170   1                DO;
 171   2                CALL ADSC$EXEC;
 172   2                STAT2=(STAT2 AND OFEH);
 173   2                END;

 174   1           CALL CI;

 175   1           GOTO CMD;



 176   1      END;


MODULE INFORMATION:

     CODE AREA SIZE      = 0475H   1141D
     CONSTANT AREA SIZE  = 0000H      0D
     VARIABLE AREA SIZE  = 0061H     97D
     MAXIMUM STACK SIZE  = 000AH     10D
     349 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

230832-22

intel®

APPLICATION NOTE

AP-66

June 1989

Using the 8292 GPIB Controller

Order Number: 231324-002

INTRODUCTION

The Intel® 8292 is a preprogrammed UPI™-41A that implements the Controller function of the IEEE Std 488-1978 (GPIB, HP-IB, IEC Bus, etc.). In order to function the 8292 must be used with the 8291 Talker/Listener and suitable interface and transceiver logic such as a pair of Intel 8293s. In this configuration the system has the potential to be a complete GPIB Controller when driven by the appropriate software. It has the following capabilities: System Controller, send IFC and Take Charge, send REN, Respond to SRQ, send Interface messages, Receive Control, Pass Control, Parallel Poll and Take Control Synchronously.

This application note will explain the 8292 only in the system context of an 8292, 8291, two 8293s and the driver software. If the reader wishes to learn more about the UPI-41A aspects of the 8292, Intel's Application Note AP-41 describes the hardware features and programming characteristics of the device. Additional information on the 8291 may be obtained in the data sheet. The 2893 is detailed in its data sheet. Both chips will be covered here in the details that relate to the GPIB controller.

The next section of this application note presents an overview of the GPIB in a tutorial, but comprehensive nature. The knowledgeable reader may wish to skip this section; however, certain basic semantic concepts introduced there will be used throughout this note.

Additional sections cover the view of the 8292 from the CPU's data bus, the interaction of the 3 chip types (8291, 8292, 8293), the 8292's software protocol and the system level hardware/software protocol. A brief description of interrupts and DMA will be followed by an application example. Appendix A contains the source code for the system driver software.

GPIB/IEEE 488 OVERVIEW

Design Objectives

WHAT IS THE IEEE 488 (GPIB)?

The experience of designing systems for a variety of applications in the early 1970's caused Hewlett-Packard to define a standard intercommunication mechanism which would allow them to easily assemble instrumentation systems of varying degrees of complexity. In a typical situation each instrument designer designed his/her own interface from scratch. Each one was inconsistent in terms of electrical levels, pin-outs on a connector, and types of connectors. Every time they built a system they had to invent new cables and new documentation just to specify the cabling and interconnection procedures.

Based on this experience, Hewlett-Packard began to define a new interconnection scheme. They went further than that, however, for they wanted to specify the typical communication protocol for systems of instruments. So in 1972, Hewlett-Packard came out with the first version of the bus which since has been modified and standardized by a committee of several manufacturers, coordinated through the IEEE, to perfect what is now known as the IEEE 488 Interface Bus (also known as the HPIB, the GPIB and the IEC bus). While this bus specification may not be perfect, it is a good compromise of the various desires and goals of instrumentation and computer peripheral manufacturers to produce a common interconnection mechanism. It fits most instrumentation systems in use today and also fits very well the microcomputer I/O bus requirements. The basic design objectives for the GPIB were to:

1) Specify a system that is easy to use, but has all of the terminology and the definitions related to that system precisely spelled out so that everyone uses the same language when discussing the GPIB.
2) Define all of the mechanical, electrical, and functional interface requirements of a system, yet not define any of the device aspects (they are left up to the instrument designer).
3) Permit a wide range of capabilities of instruments and computer peripherals to use a system simultaneously and not degrade each other's performance.
4) Allow different manufacturers' equipment to be connected together and work together on the same bus.
5) Define a system that is good for limited distance interconnections.
6) Define a system with minimum restrictions on performance of the devices.
7) Define a bus that allows asynchronous communication with a wide range of data rates.
8) Define a low cost system that does not require extensive and elaborate interface logic for the low cost instruments, yet provides higher capability for the higher cost instruments if desired.
9) Allow systems to exist that do not need a central controller; that is, communication directly from one instrument to another is possible.

Although the GPIB was originally designed for instrumentation systems, it became obvious that most of these systems would be controlled by a calculator or computer. With this in mind several modifications were made to the original proposal before its final adoption as an international standard. Figure 1 lists the salient characteristics of the GPIB as both an instrumentation bus and as a computer I/O bus.

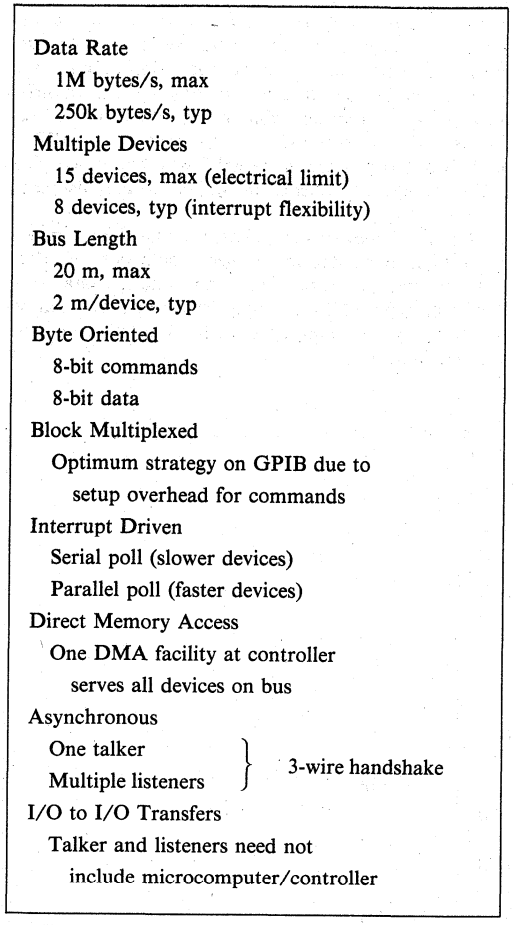


Figure 1. Major Characteristics of GPIB as Microcomputer I/O Bus

The bus can be best understood by examining each of these characteristics from the viewpoint of a general microcomputer I/O bus.

Data Rate—Most microcomputer systems utilize peripherals of differing operational rates, such as floppy discs at 31k or 62k bytes/s (single or double density), tape cassettes at 5k to 10k bytes/s, and cartridge tapes at 40k to 80k bytes/s. In general, the only devices that need high speed I/O are 0.5″ (1.3-cm) magnetic tapes and hard discs, operational at 30k to 781k bytes/s, respectively. Certainly, the 250k-bytes/s data rate that can be easily achieved by the IEEE 488 bus is sufficient for microcomputers and their peripherals, and is more than needed for typical analog instruments that take only a few readings per second. The 1M-byte/s maximum data rate is not easily achieved on the GPIB and requires special attention to considerations beyond the scope of this note. Although not required, data buffering in each device will improve the overall bus performance and allow utilization of more of the bus bandwidth.

Multiple Devices—Many microcomputer systems used as computers (not as components) service from three to seven peripherals. With the GPIB, up to 8 devices can be handled easily by 1 controller; with some slowdown in interrupt handling, up to 15 devices can work together. The limit of 8 is imposed by the number of unique parallel poll responses available; the limit of 15 is set by the electrical drive characteristics of the bus. Logically, the IEEE 488 Standard is capable of accommodating more device addresses (31 primary, each potentially with 31 secondaries).

Bus Length—Physically, the majority of microcomputer systems fit easily on a desk top or in a standard 19″ (48-cm) rack, eliminating the need for extra long cables. The GPIB is designed typically to have 2m of length per device, which accommodates most systems. A line printer might require greater cable lengths, but this can be handled at the lower speeds involved by using extra dummy terminations.

Byte Oriented—The 8-bit byte is almost universal in I/O applications; even 16-bit and 32-bit computers use byte transfers for most peripherals. The 8-bit byte matches the ASCII code for characters and is an integral submultiple of most computer word sizes. The GPIB has an 8-bit wide data path that may be used to transfer ASCII or binary data, as well as the necessary status and control bytes.

Block Multiplexed—Many peripherals are block oriented or are used in a block mode. Bytes are transferred in a fixed or variable length group; then there is a wait before another group is sent to that device, e.g., one sector of a floppy disc, one line on a printer or type punch, etc. The GPIB is, by nature, a block multiplexed bus due to the overhead involved in addressing various devices to talk and listen. This overhead is less bothersome if it only occurs once for a large number of data bytes (once per block). This mode of operation matches the needs of microcomputers and most of their peripherals. Because of block multiplexing, the bus works best with buffered memory devices.

Interrupt Driven—Many types of interrupt systems exist, ranging from complex, fast, vectored/priority networks to simple polling schemes. The main tradeoff is usually cost versus speed of response. The GPIB has two interrupt protocols to help span the range of applications. The first is a single service request (SRQ) line that may be asserted by all interrupting devices. The controller then polls all devices to find out which wants service. The polling mechanism is well defined and can

be easily automated. For higher performance, the parallel poll capability in the IEEE 488 allows up to eight devices to be polled at once—each device is assigned to one bit of the data bus. This mechanism provides fast recognition of an interrupting device. A drawback is the frequent need for the controller to explicitly conduct a parallel poll, since there is no equivalent of the SRQ line for this mode.

Direct Memory Access (DMA)—In many applications, no immediate processing of I/O data on a byte-by-byte basis is needed or wanted. In fact, programmed transfers slow down the data transfer rate unnecessarily in these cases, and higher speed can be obtained using DMA. With the GPIB, one DMA facility at the controller serves all devices. There is no need to incorporate complex logic in each device.

Asynchronous Transfers—An asynchronous bus is desirable so that each device can transfer at its own rate. However, there is still a strong motivation to buffer the data at each device when used in large systems in order to speed up the aggregate data rate on the bus by allowing each device to transfer at top speed. The GPIB is asynchronous and uses a special 3-wire handshake that allows data transfers from one talker to many listeners.

I/O to I/O Transfers—In practice, I/O to I/O transfers are seldom done due to the need for processing data and changing formats or due to mismatched data rates. However, the GPIB can support this mode of operation where the microcomputer is neither the talker nor one of the listeners.

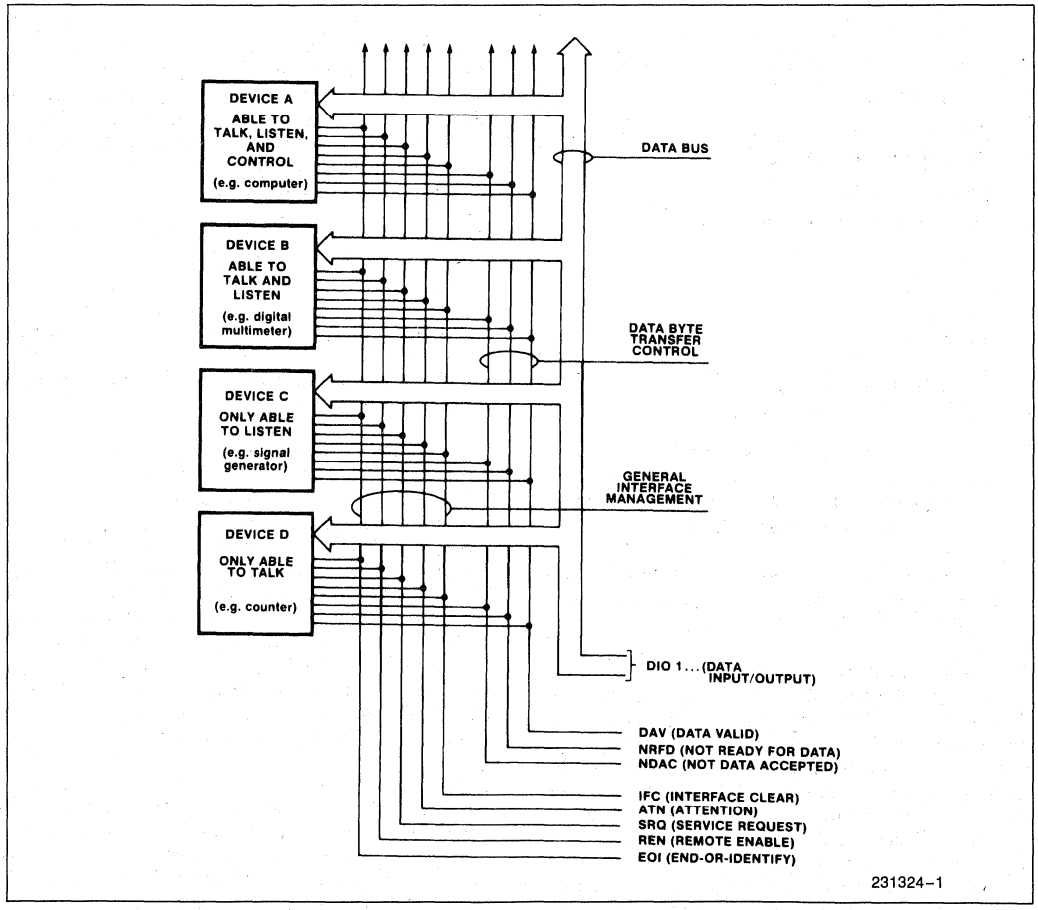


Figure 2. Interface Capabilities and Bus Structure

GPIB Signal Lines

DATA BUS

The lines DI01 through DI08 are used to transfer addresses, control information and data. The formats for addresses and control bytes are defined by the IEEE 488 standard (see Appendix C). Data formats are undefined and may be ASCII (with or without parity) or binary. D101 is the Least Significant bit (note that this will correspond to bit 0 on most computers).

MANAGEMENT BUS

ATN—Attention. This signal is asserted by the Controller to indicate that it is placing an address or control byte on the Data Bus. ATN is de-asserted to allow the assigned Talker to place status or data on the Data Bus. The Controller regains control by reasserting ATN; this is normally done synchronously with the handshake to avoid confusion between control and data bytes.

EOI—End or Identify. This signal has two uses as its name implies. A talker may assert EOI simultaneously with the last byte of data to indicate end of data. The Controller may assert EOI along with ATN to initiate a Parallel Poll. Although many devices do not use Parallel Poll, all devices *should* use EOI to end transfers (many currently available ones do not).

SRQ—Service Request. This line is like an interrupt: it may be asserted by any device to request the Controller to take some action. The Controller must determine which device is asserting SRQ by conducting a Serial Poll at its earliest convenience. The device deasserts SRQ when polled.

IFC—Interface Clear. This signal is asserted only by the System Controller in order to initialize all device interfaces to a known state. After deasserting IFC, the System Controller is the active controller of the system.

REN—Remote Enable. This signal is asserted only by the System Controller. Its assertion does not place devices into Remote Control mode; REN only *enables* a device to go remote when addressed to listen. When in Remote, a device should ignore its front panel controls.

TRANSFER BUS

NRFD—Not Ready For Data. This handshake line is asserted by a listener to indicate it is not yet ready for the next data or control byte. Note that the Controller will not see NRFD deasserted (i.e., ready for data) until all devices have deasserted NRFD.

NDAC—Not Data Accepted. This handshake line is asserted by a Listener to indicate it has not yet accepted the data or control byte on the DIO lines. Note that the Controller will not see NDAC deasserted (i.e., data accepted) until all devices have deasserted NDAC.

DAV—Data Valid. This handshake line is asserted by the Talker to indicate that a data or control byte has been placed on the DIO lines and has had the minimum specified settling time.

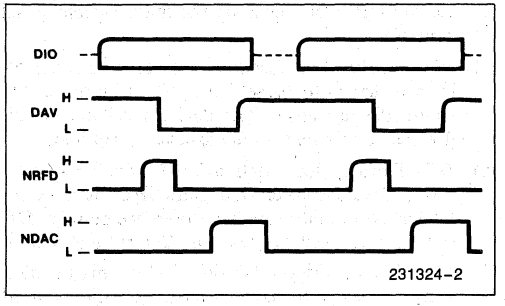


Figure 3. GPIB Handshake Sequence

GPIB Interface Functions

There are ten (10) interface functions specified by the IEEE 488 standard. Not all devices will have all functions and some may only have partial subsets. The ten functions are summarized below with the relevant section number from the IEEE document given at the beginning of each paragraph. For further information please see the IEEE standard.

1) *SH*—Source Handshake (section 2.3). This function provides a device with the ability to properly transfer data from a Talker to one or more Listeners using the three handshake lines.
2) *AH*—Acceptor Handshake (section 2.4). This function provides a device with the ability to properly receive data from the Talker using the three handshake lines. The AH function may also delay the beginning (NRFD) or end (NDAC) of any transfer.
3) *T*—Talker (section 2.5). This function allows a device to send status and data bytes when addressed to talk. An address consists of one (Primary) or two (Primary and Secondary) bytes. The latter is called an extended Talker.

4) *L*—Listener (section 2.6). This function allows a device to receive data when addressed to listen. There can be extended Listeners (analogous to extended Talkers above).
5) *SR*—Service Request (section 2.7). This function allows a device to request service (interrupt) the Controller. The SRQ line may be asserted asynchronously.
6) *RL*—Remote Local (section 2.8). This function allows a device to be operated in two modes: Remote via the GPIB or Local via the manual front panel controls.
7) *PP*—Parallel Poll (section 2.9). This function allows a device to present one bit of status to the Controller-in-charge. The device need not be addressed to talk and no handshake is required.
8) *DC*—Device Clear (section 2.10). This function allows a device to be cleared (initialized) by the Controller. Note that there is a difference between DC (*device* clear) and the IFC line (*interface* clear).
9) *DT*—Device Trigger (section 2.11). This function allows a device to have its basic operation started either individually or as part of a group. This capability is often used to synchronize several instruments.
10) *C*—Controller (section 2.12). This function allows a device to send addresses, as well as universal and addressed commands to other devices. There may be more than one controller on a system, but only one may be the controller-in-charge at any one time.

At power-on time the controller that is hardwired to be the System Controller becomes the active controller-in-charge. The System Controller has several unique capabilities including the ability to send Interface Clear (IFC—clears all device interfaces and returns control to the System Controller) and to send Remote Enable (REN—allows devices to respond to bus data once they are addressed to listen). The System Controller may optionally Pass Control to another controller, if the system software has the capability to do so.

GPIB Connector

The GPIB connector is a standard 24-pin industrial connector such as Cinch or Amphenol series 57 Micro-Ribbon. The IEEE standard specifies this connector, as well as the signal connections and the mounting hardware.

The cable has 16 signal lines and 8 ground lines. The maximum length is 20 meters with no more than two meters per device.

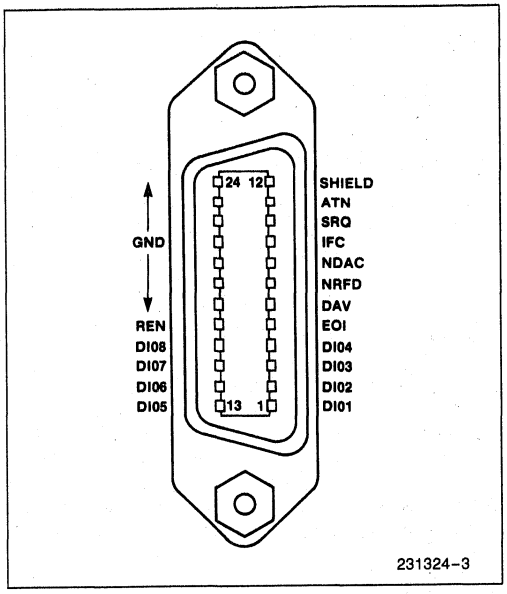


Figure 4. GPIB Connector

GPIB Signal Levels

The GPIB signals are all TTL compatible, low true signals. A signal is asserted (true) when its electrical voltage is less than 0.5 volts and is deasserted (false) when it is greater than 2.4 volts. Be careful not to become confused with the two handshake signals, NRFD and NDAC which are also low true (i.e. > 0.5 volts implies the device is Not Ready For Data).

The Intel 8293 GPIB transceiver chips ensure that all relevant bus driver/receiver specifications are met. Detailed bus electrical specifications may be found in Section 3 of the IEEE Std 488-1978. The Standard is the ultimate reference for all GPIB questions.

GPIB Message Protocols

The GPIB is a very flexible communications medium and as such has many possible variations of protocols. To bring some order to the situation, this section will discuss a protocol similar to the one used by Ziatech's ZT80 GPIB controller for Intel's MULTIBUS™ computers. The ZT80 is a complete high-level interface processor that executes a set of high level instructions that map directly into GPIB actions. The sequences of commands, addresses and data for these instructions provide a good example of how to use the GPIB (additional information is available in the ZT80 Instruction Manual). The 'null' at the end of each instruction is for cosmetic use to remove previous information from the DIO lines.

DATA—Transfer a block of data from device A to devices B, C . . .

1) Device A Primary (Talk) Address
   Device A Secondary Address (if any)
2) Universal Unlisten
3) Device B Primary (Listen) Address
   Device B Secondary Address (if any)
   Device C Primary (Listen) Address
   etc.
4) First Data Byte
   Second Data Byte
   .
   .
   .
   Last Data Byte (EOI)
5) Null

TRIGR—Trigger devices A, B . . . to take action

1) Universal Unlisten
2) Device A Primary (Listen) Address
   Device A Secondary Address (if any)
   Device B Primary (Listen) Address
   Device B Secondary Address (if any)
   etc.
3) Group Execute Trigger
4) Null

PSCTL—Pass control to device A

1) Device A Primary (Talk) Address
   Device A Secondary Address (if any)
2) Talk Control
3) Null

CLEAR—Clear all devices

1) Device Clear
2) Null

REMAL—Remote Enable

1) Assert REN continuously

GOREM—Put devices A, B, . . . into Remote

1) Assert REN continuously
2) Device A Primary (Listen) Address
   Device A Secondary Address (if any)
   Device B Primary (Listen) Address
   Device B Secondary Address (if any)
   etc.
3) Null

GOLOC—Put devices A, B, . . . into Local

1) Device A Primary (Listen) Address
   Device A Secondary Address (if any)
   Device B Primary (Listen) Address
   Device B Secondary Address (if any)
   etc.
2) Go To Local
3) Null

LOCAL—Reset all devices to Local

1) Stop asserting REN

LLKAL—Prevent all devices from returning to Local

1) Local Lock Out
2) Null

SPOLL—Conducts a serial poll of devices A, B, . . .

1) Serial Poll Enable
2) Universal Unlisten
3) ZT 80 Primary (Listen) Address
   ZT 80 Secondary Address
4) Device Primary (Talk) Address
   Device Secondary Address (if any)
5) Status byte from device
6) Go to Step 4 until all devices on list have been polled
7) Serial Poll Disable
8) Null

PPUAL—Unconfigure and disable Parallel Poll response from all devices

1) Parallel Poll Unconfigure
2) Null

ENAPP—Enable Parallel Poll response in devices A, B, . . .

1) Universal Unlisten
2) Device Primary (Listen) Address
   Device Secondary Address (if any)
3) Parallel Poll Configure
4) Parallel Poll Enable
5) Go to Step 2 until all devices on list have been configured.
6) Null

DISPP—Disable Parallel Poll response from devices A, B, . . .

1) Universal Unlisten
2) Device A Primary (Listen) Address
   Device A Secondary Address (if any)
   Device B Primary (Listen) Address
   Device B Secondary Address (if any)
   etc.
3) Disable Parallel Poll
4) Null

This Ap Note will detail how to implement a useful subset of these controller instructions.

HARDWARE ASPECTS OF THE SYSTEM

8291 GPIB Talker/Listener

The 8291 is a custom designed chip that implements many of the non-controller GPIB functions. It provides hooks so the user's software can implement additional features to complete the set. This chip is discussed in detail in its data sheet. The major features are summarized here:

— Designed to interface microprocessors to the GPIB
— Complete Source and Acceptor Handshake
— Complete Talker and Listener Functions with extended addressing
— Service Request, Parallel Poll, Device Clear, Device Trigger, Remote/Local functions
— Programmable data transfer rate
— Maskable interrupts
— On-chip primary and secondary address recognition
— 1–8 MHz clock range
— 16 registers (8 read, 8 write) for CPU interface
— DMA handshake provision
— Trigger output pin
— On-chip EOS (End of Sequence)

The pinouts and block diagram are shown in Figure 5. One of eight read registers is for data transfer to the CPU; the other seven allow the microprocessor to monitor the GPIB states and various bus and device conditions. One of the eight write registers is for data transfer from the CPU; the other seven control various features of the 8291.

The 8291 interface functions will be software configured in this application example to the following subsets for use with the 8292 as a controller that does not pass control. The 8291 is used only to provide the handshake logic and to send and receive data bytes. It is not acting as a normal device in this mode, as it never sees ATN asserted.

SH1 Source Handshake
AH1 Acceptor Handshake
T3 Basic Talk-Only
L1 Basic Listen-Only
SR0 No Service Requests
RL0 No Remote/Local
PP0 No Parallel Poll Response
DC0 No Device Clear
DT0 No Device Trigger

If control is passed to another controller, the 8291 must be reconfigured to act as a talker/listener with the following subsets:

SH1 Source Handshake
AH1 Acceptor Handshake
T5 Basic Talker and Serial Poll
L3 Basic Listener
SR1 Service Requests
RL1 Remote/Local with Lockout
PP2 Reconfigured Parallel Poll
DC1 Device Clear
DT1 Device Trigger
C0 Not a Controller

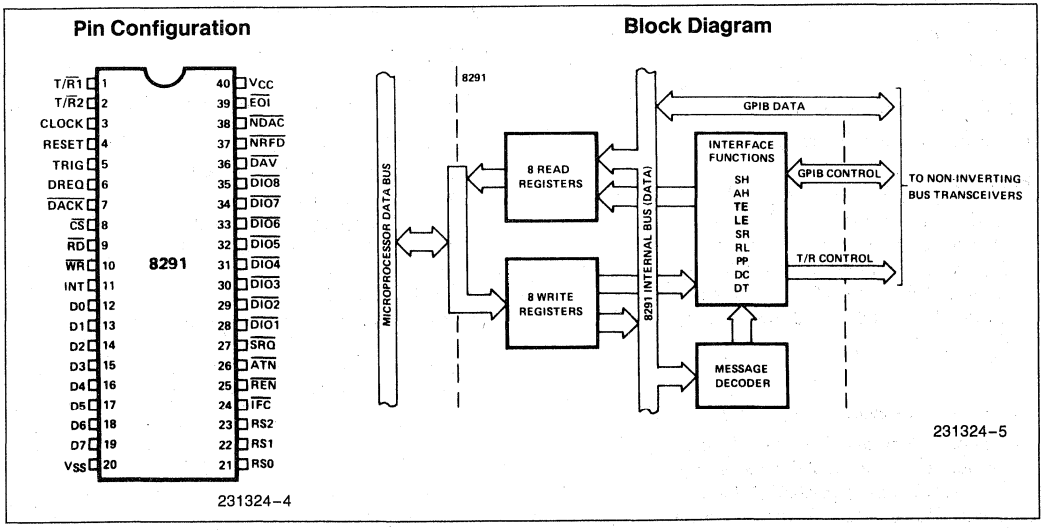


Figure 5. 8291 Pin Configuration and Block Diagram

Most applications do not pass control and the controller is always the system controller (see 8292 commands below).

8292 GPIB Controller

The 8292 is a preprogrammed Intel 8051A that provides the additional functions necessary to implement a GPIB controller when used with an 8291 Talker/Listener. The 8041A is documented in both a user's manual and in AP-41. The following description will serve only as an outline to guide the later discussion.

The 8292 acts as an intelligent slave processor to the main system CPU. It contains a processor, memory, I/O and is programmed to perform a variety of tasks associated with GPIB controller operation. The on-chip RAM is used to store information about the state of the Controller function, as well as a variety of local variables, the stack and certain user status information. The timer/counter may be optionally used for several time-out functions or for counting data bytes transferred. The I/O ports provide the GPIB control signals, as well as the ancillary lines necessary to make the 8291, 2, 3 work together.

The 8292 is closely coupled to the main CPU through three on-chip registers that may be independently accessed by both the master and the 8292 (UPI-41A). Figure 6 shows this Register Interface. Also refer to Figure 12.

The status register is used to pass Interrupt Status information to the master CPU (A0 = 1 on a read).

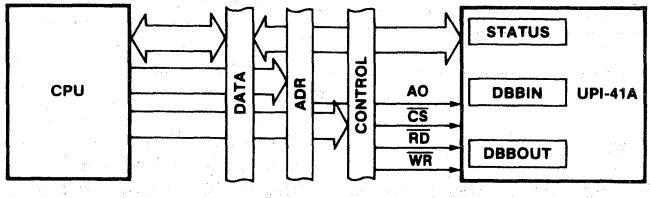
The DBBOUT register is used to pass one of five other status words to the master based on the last command written into DBBIN. DBBOUT is accessed when A0 = 0 on a Read. The five status words are Error Flag, Controller Status, GPIB Status, Event Counter Status or Time Out Status.

DBBIN receives either commands (A0 = 1 on a Write) or command related data (A0 = 0 on a write) from the master. These command related data are Interrupt Mask, Error Mask, Event Counter or Time Out.

8293 GPIB Transceivers

The 8293 is a multi-use HMOS chip that implements the IEEE 488 bus transceivers and contains the additional logic required to make the 8291 and 8292 work together. The two option strapping pins are used to internally configure the chip to perform the specialized gating required for use with 8291 as a device or with 8291/92 as a controller.

In this application example the two configurations used are shown in Figure 7a and 7b. The drivers are set to open collector or three state mode as required and the special logic is enabled as required in the two modes.



231324-48

$\overline{CS}$	AO	$\overline{RD}$	$\overline{WR}$	REGISTER
0	0	0	1	READ DBBOUT
0	1	0	1	READ STATUS
0	0	1	0	WRITE DBBIN (DATA)
0	1	1	0	WRITE DBBIN (COMMAND)
1	X	X	X	NO ACTION

Figure 6. UPI-41A Registers

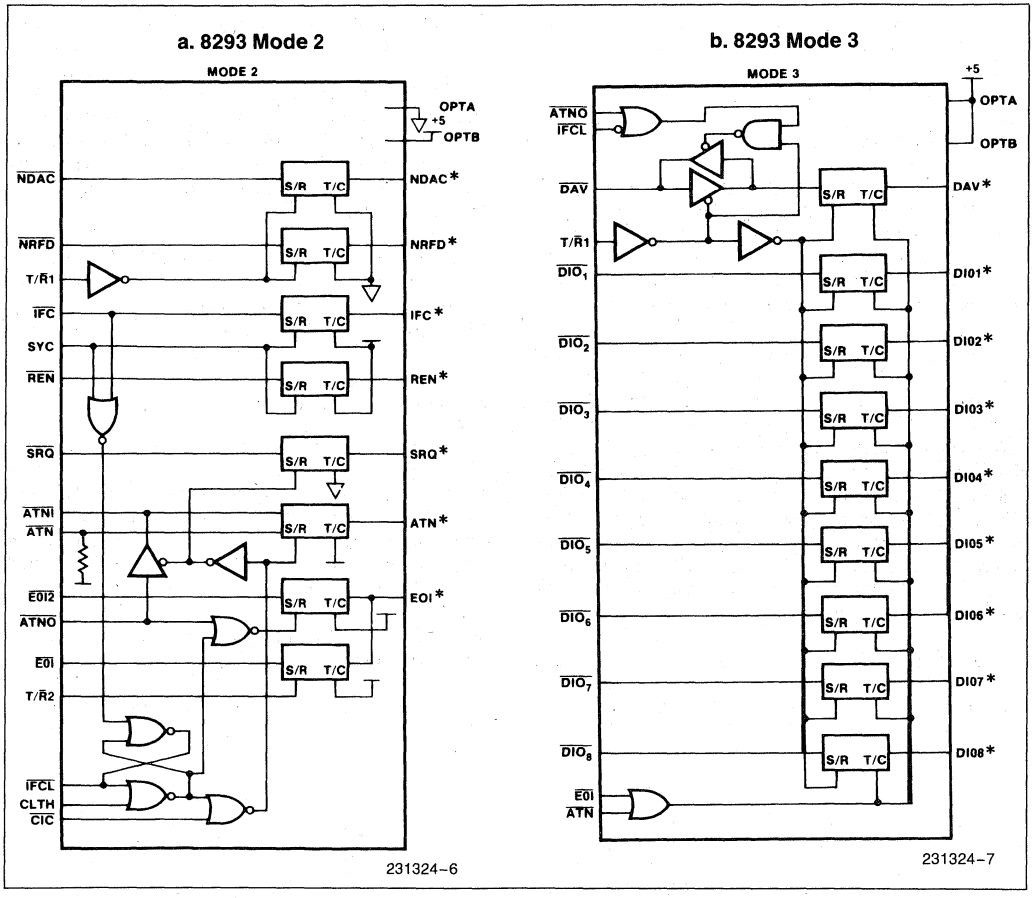


Figure 7

8291/2/3 Chip Set

Figure 8 shows the four chips interconnected with the special logic explicitly shown.

The 8291 acts only as the mechanism to put commands and addresses on the bus while the 8292 is asserting ATN. The 8291 is tricked into believing that the ATN line is not asserted by the ATN2 output of the ATN transceiver and is placed in Talk-only mode by the CPU. The 8291 then acts as though it is sending data, when in reality it is sending addresses and/or commands. When the 8292 deasserts ATN, the CPU software must place the 8291 in Talk-only, Listen-only or Idle based on the implicit knowledge of how the controller is going to participate in the data transfer. In other words, the 8291 does not respond directly to addresses or commands that it sends on the bus on behalf of the Controller. The user software, through the use of Listen-only or Talk-only, makes the 8291 behave as though it were addressed.

Although it is not a common occurrence, the GPIB specification allows the Controller to set up a data transfer between two devices and not directly participate in the exchange. The controller must know when to go active again and regain control. The chip set accomplishes this through use of the "Continuous Acceptor Handshake cycling mode" and the ability to detect EOI or EOS at the end of the transfer. See XFER in the Software Driver Outline below.

If the 8292 is not the System Controller as determined by the signal on its SYC pin, then it must be able to respond to an IFC within 100 μsec. This is accomplished by the cross-coupled NORs in Figure 7a which deassert the 8293's internal version of $\overline{CIC}$ (Not Controller-in-Charge). This condition is latched until the 8292's firmware has received the IFCL (interface clear received latch) signal by testing the $\overline{IFCL}$ input. The firmware then sets its signals to reflect the inactive condition and clears the 8293's latch.

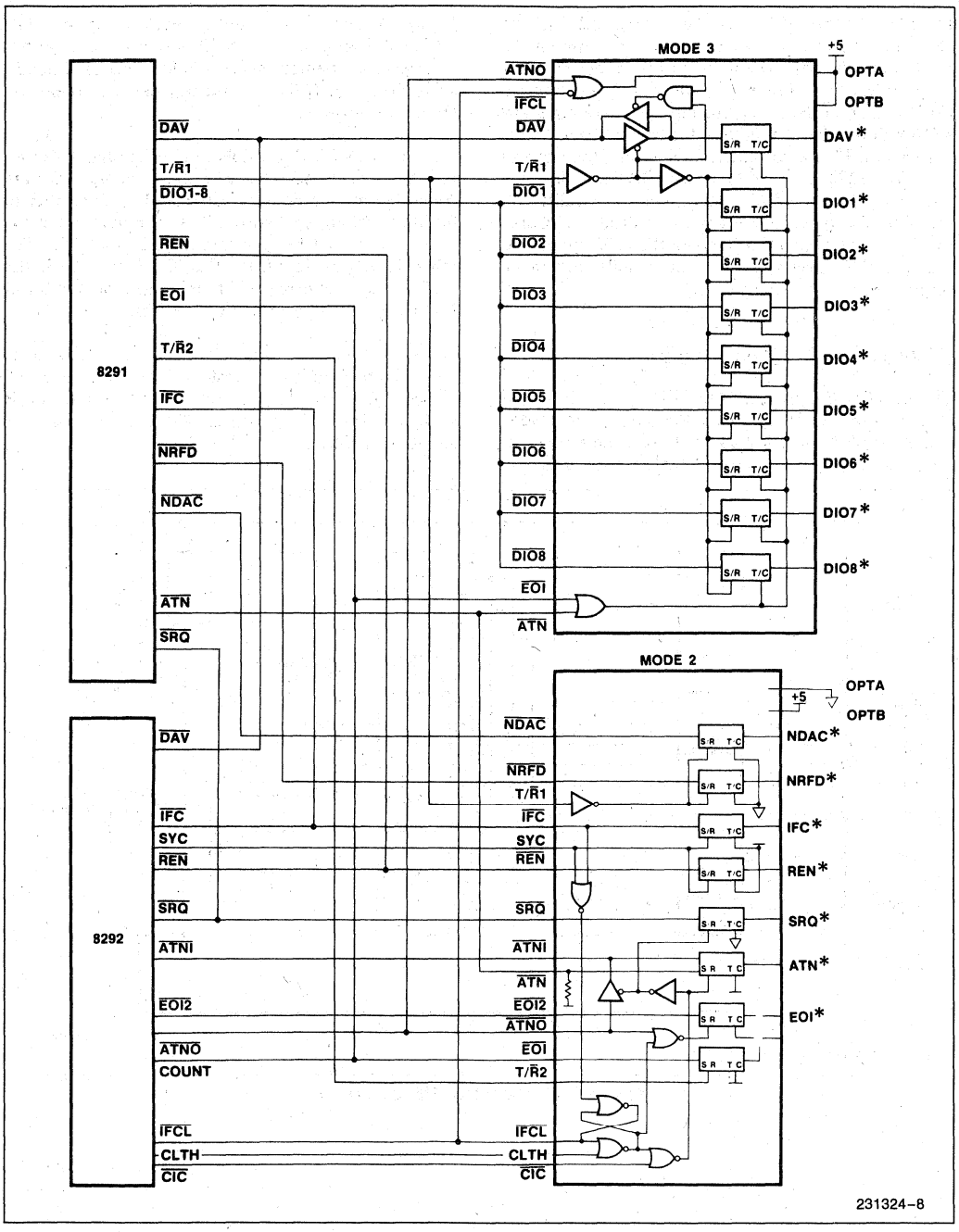


Figure 8. Talker/Listener/Controller

In order for the 8292 to conduct a Parallel Poll the 8291 must be able to capture the PP response on the DIO lines. The only way to do this is to fool the 8291 by putting it into Listen-only mode and generating a DAV condition. However, the bus spec does not allow a DAV during Parallel Poll, so the back-to-back 3-state buffers (see Figure 7b) in the 8293 isolate the bus and allow the 8292 to generate a local DAV for this purpose. Note that the 8291 cannot assert a Parallel Poll response. When the 8292 is not the controller-in-charge the 8291 may respond to PPs and the 8293 guarantees that the DIO drivers are in "open collector" mode through the OR gate (Figure 7b).

ZT7488/18 GPIB Controller

Ziatech's GPIB Controller, the ZT7488/18 will be used as the controller hardware in this Application Note. The controller consists of an 8291, 8292, an 8 bit input port and TTL logic equivalent to that shown in Figure 8.

Figure 9 shows the card's block diagram. The ZT7488/18 plugs into the STD bus, a 56 pin 8 bit microprocessor oriented bus. An 8085 CPU card is also available on the STD bus and will be used to execute the driver software.

The 8291 uses I/O Ports 60H to 67H and the 8292 uses I/O Ports 68H and 69H. The five interrupt lines are connected to a three-state buffer at I/O Port 6FH to facilitate polling operation. This is required for the TCI, as it cannot be read internally in the 8292. The other three 8229 lines (SPI, IBF, OBF) and the 8291's INT line are also connected to minimize the number of I/O reads necessary to poll the devices.

$\overline{\text{NDAC}}$ is connected to $\overline{\text{COUNT}}$ on the 8292 to allow byte counting on data transfers. The example driver software will not use this feature, as the software is simpler and faster if an internal 8085 register is used for counting in software.

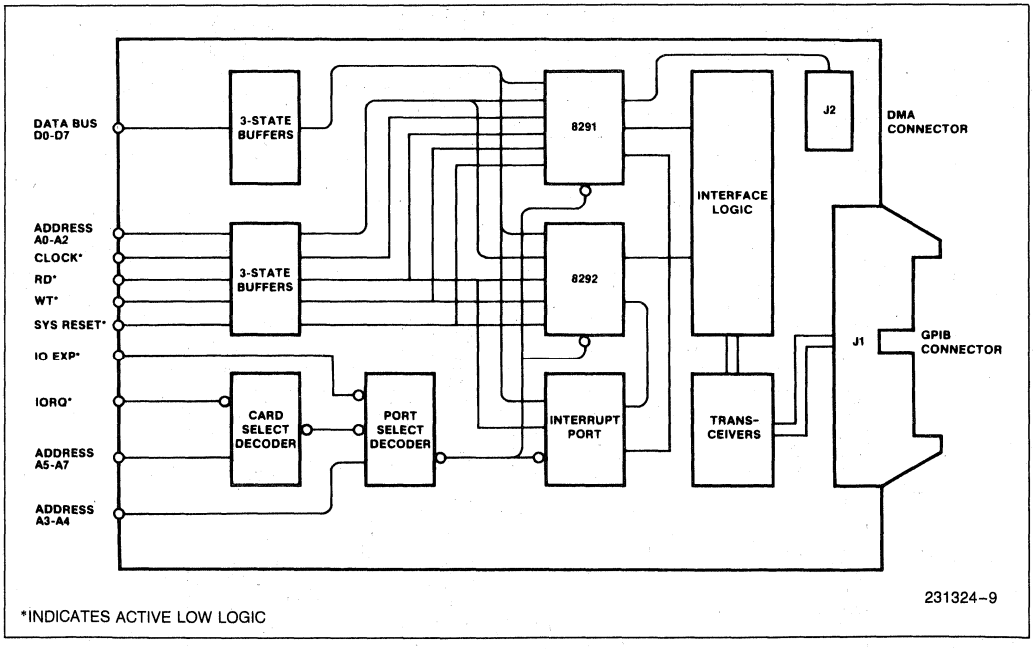


Figure 9. ZT7488/18 GPIB Controller

READ REGISTERS								PORT #	WRITE REGISTERS							
DI7	DI6	DI5	DI4	DI3	DI2	DI1	DI0	6ϕH	DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0
DATA IN									DATA OUT							
CPT	APT	GET	END	DEC	ERR	BO	BI	61H	CPT	APT	GET	END	DEC	ERR	BO	BI
INTERRUPT STATUS 1									INTERRUPT MASK 1							
INT	SPAS	LLO	REM	SPASC	LLOC	REMC	ADSC	62H	0	0	DMAO	DMAI	SPASC	LLOC	REMC	ADSC
INTERRUPT STATUS 2									INTERRUPT MASK 2							
S8	SRQS	S6	S5	S4	S3	S2	S1	63H	S8	rsv	S6	S5	S4	S3	S2	S1
SERIAL POLL STATUS									SERIAL POLL MODE							
ton	lon	EOI	LPAS	TPAS	LA	TA	MJMN	64H	TO	LO	0	0	0	0	ADM1	ADM0
ADDRESS STATUS									ADDRESS MODE							
CPT7	CPT6	CPT5	CPT4	CPT3	CPT2	CPT1	CPT0	65H	CNT2	CNT1	CNT0	COM4	COM3	COM2	COM1	COM0
COMMAND PASS THROUGH									AUX MODE							
X	DT0	DL0	AD5-0	AD4-0	AD3-0	AD2-0	AD1-0	66H	ARS	DT	DL	AD5	AD4	AD3	AD2	AD1
ADDRESS 0									ADDRESS 0/1							
X	DT1	DL1	AD5-1	AD4-1	AD3-1	AD2-1	AD1-1	67H	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0
ADDRESS 1									EOS							

Figure 10. 8291 Registers

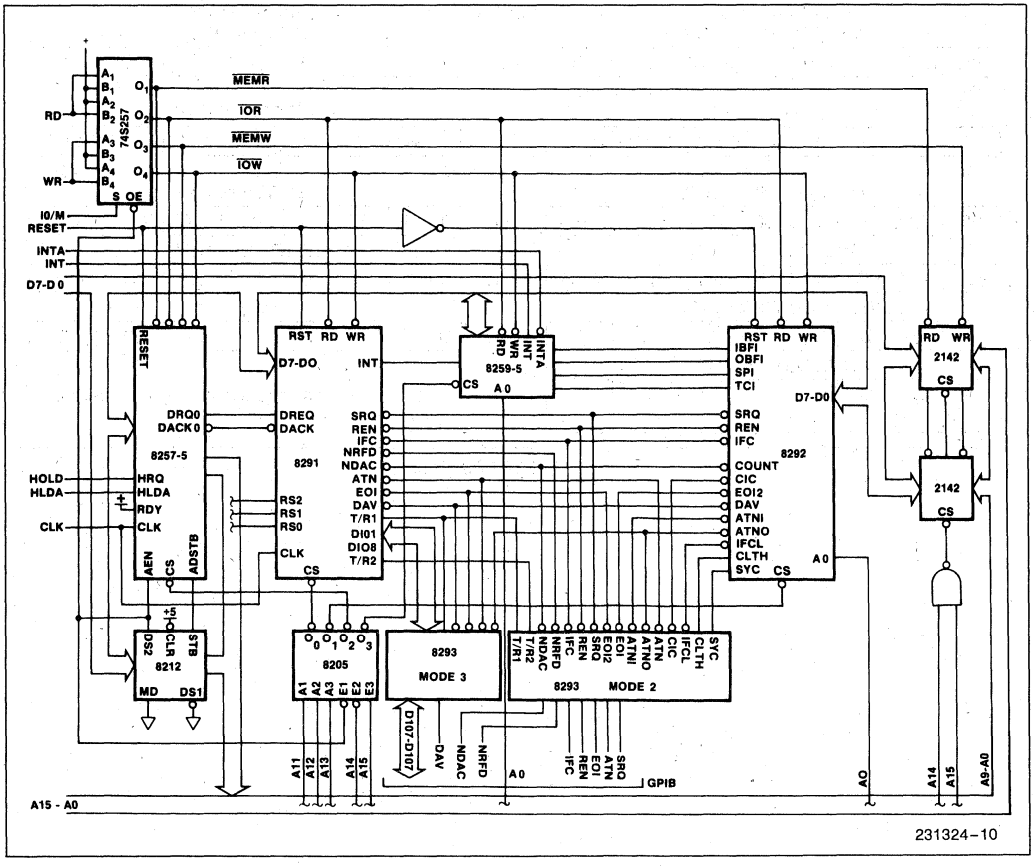


Figure 11. DMA/Interrupt GPIB Controller Block Diagram

The application example will not use DMA or interrupts; however, the Figure 11 block diagram includes these features for completeness.

The 8257-5 DMA chip can be used to transfer data between the RAM and the 8291 Talker/Listener. This mode allows a faster data rate on the GPIB and typically will depend on the 8291's EOS or EOI detection to terminate the transfer. The 8259-5 interrupt controller is used to vector the five possible interrupts for rapid software handling of the various conditions.

8292 COMMAND DESCRIPTION

This section discusses each command in detail and relates them to a particular GPIB activity. Recall that although the 8041A has only two read registers and one write register, through the magic of on-chip firmware the 8292 appears to have six read registers and five write registers. These are listed in Figure 12. Please see the 8292 data sheet for detailed definitions of each register. Note the two letter mnemonics to be used in later discussions. The CPU must not write into the 8292 while IBF (Input Buffer Full) is a one, as information will be lost.

Direct Commands

Both the Interrupt Mask (IM) and the Error Mask (EM) register may be directly written with the LSB of the address bus (A0) a "0". The firmware uses the MSB of the data written to differentiate between IM and EM.

LOAD INTERRUPT MASK

This command loads the Interrupt Mask with D7–D0. Note that D7 must be a "1" and that interrupts are enabled by a corresponding "1" bit in this register. IFC interrupt cannot be masked off; however, when the 8292 is the System Controller, sending an ABORT command will not cause an IFC interrupt.

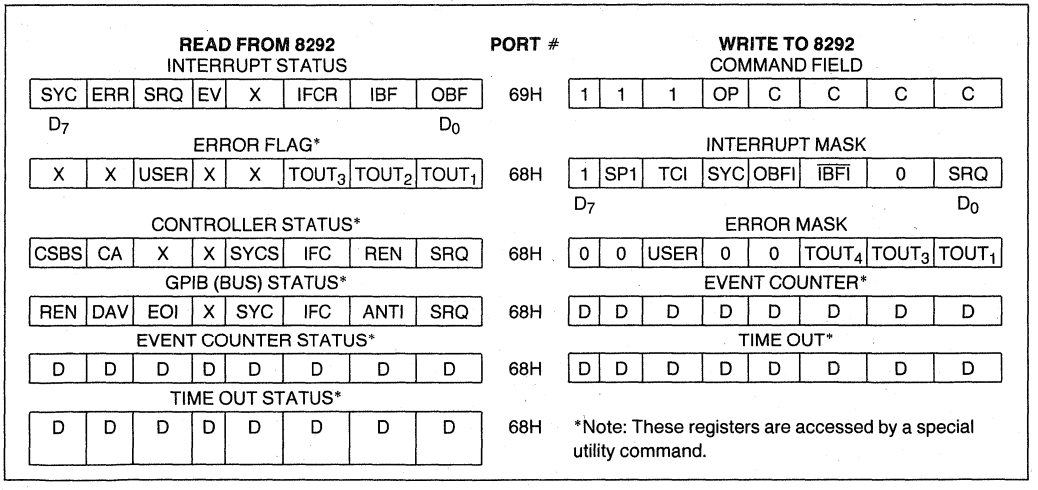


Figure 12. 8292 Registers

LOAD ERROR MASK

This command loads the Error Mask with D7–D0. Note that D7 must be a zero and that interrupts are enabled by a corresponding "1" bit in this register.

Utility Commands

These commands are used to read or write the 8292 registers that are not directly accessible. All utility commands are written with A0 = 1, D7 = D6 = D5 = 1, D4 = 0. D3–D0 specify the particular command. For writing into registers the general sequence is:

1) wait for IBF = 0 in Interrupt Status Register
2) write the appropriate command to the 8292,
3) write the desired register value to the 8292 with A0 = 1 with no other writes intervening,
4) wait for indication of completion from 8292 (IBF = 0).

For reading a register the general sequence is:

1) wait for IBF = 0 in Interrupt Status Register
2) write the appropriate command to the 8292
3) wait for a TCI (Task Complete Interrupt)
4) Read the value of the accessed register from the 8292 with A0 = 0.

WEVC—Write to Event Counter
(Command = 0E2H)

The byte written following this command will be loaded into the event counter register and event counter status for byte counting. The internal counter is incremented on a high to low transition of the COUNT (T1) input. In this application example $\overline{\text{NDAC}}$ is connected to count. The counter is an 8 bit register and therefore can count up to 256 bytes (writing 0 to the EC implies a count of 256). If longer blocks are desired, the main CPU must handle the interrupts every 256 counts and carefully observe the timing constraints.

Because the counter has a frequency range from 0 to 133 kHz when using a 6 MHz crystal, this feature may not be usable with all devices on the GPIB. The 8291 can easily transfer data at rates up to 250 kHz and even faster with some tuning of the system. There is also a 500 ns minimum high time requirement for COUNT which can potentially be violated by the 8291 in continuous acceptor handshake mode (i.e., TNDDV1 + TDVND2 − C = 350 + 350 = 700 max). When cable delays are taken into consideration, this problem will probably never occur.

When the 8292 has completed the command, IBF will become a "0" and will cause an interrupt if masked on.

WTOUT—Write to Time Out Register
(Command = 0E1H)

The byte written following this command will be used to determine the number of increments used for the time out functions. Because the register is 8 bits, the maximum time out is 256 time increments. This is probably enough for most instruments on the GPIB but is not enough for a manually stepped operation using a GPIB logic analyzer like Ziatech's ZT488. Also, the 488 Standard does not set a lower limit on how long a device may take to do each action. Therefore, any use of a time out must be able to be overridden (this is a good general design rule for service and debugging considerations).

The time out function is implemented in the 8292's firmware and will not be an accurate time. The counter counts backwards to zero from its initial value. The function may be enabled/disabled by a bit in the Error mask register. When the command is complete IBF will be set to a "0" and will cause an interrupt if masked on.

REVC—Read Event Counter Status
(Command = 0E3H)

This command transfers the content of the Event Counter to the DBBOUT register. The firmware then sets TCI = 1 and will cause an interrupt if masked on. The CPU may then read the value from the 8292 with A0 = 0.

RINM—Read Interrupt Mask Register
(Command = 0E5H)

This command transfers the content of the Interrupt Mask register to the DBBOUT register. The firmware sets TCI = 1 and will cause an interrupt if masked on. The CPU may then read the value.

RERM—Read Error Mask Register
(Command = 0EAH)

This command transfers the content of the Error Mask register to the DBBOUT register. The firmware sets TCI = 1 and will cause an interrupt if masked on. The CPU may then read the value.

RCST—Read Controller Status Register
(Command = 0E6H)

This command transfers the content of the Controller Status register to the DBBOUT register. The firmware sets TCI = 1 and will cause an interrupt if masked on. The CPU may then read the value.

RTOUT—Read Time Out Status Register
(Command = 0E9H)

This command transfers the content of the Time Out Status register to the DBBOUT register. The firmware sets TCI = 1 and will cause an interrupt if masked on. The CPU may then read the value.

If this register is read while a time-out function is in process, the value will be the time remaining before time-out occurs. If it is read after a time-out, it will be zero. If it is read when no time-out is in process, it will be the last value reached when the previous timing occurred.

RBST—Read Bus Status Register
(Command = 0E7H)

This command causes the firmware to read the GPIB management lines, DAV and the SYC pin and place a copy in DBBOUT. TCI is set to "1" and will cause an interrupt if masked on. The CPU may read the value.

RERF—Read Error Flag Register
(Command = 0E4H)

This command transfers the content of the Error Flag register to the DBBOUT register. The firmware sets TCI = 1 and will cause an interrupt if masked on. The CPU may then read the value.

This register is also placed in DBBOUT by an IACK command if ERR remains set. TCI is set to "1" in this case also.

IACK—Interrupt Acknowledge
(Command = A1 A2 A3 A4 1 A5 1 1)

This command is used to acknowledge any combinations of the five SPI interrupts (A1–A5): SYC, ERR, SRQ, EV, and IFCR. Each bit A1–A5 is an individual acknowledgement to the corresponding bit in the Interrupt Status Register. The command clears SPI but it will be set again if all of the pending interrupts were not acknowledged.

If A2 (ERR) is "1", the Error Flag register is placed in DBBOUT and TCI is set. The CPU may then read the Error Flag without issuing an RERF command.

Operation Commands

The following diagram (Figure 13) is an attempt to show the interrelationships among the various 8292 Operation Commands. It is not meant to replace the complete controller state diagram in the IEEE Standard.

RST—Reset (Command = 0F2H)

This command has the same effect as an external reset applied to the chip's pin #4. The 8292's actions are:

1) All outputs go to their electrical high state. This means that SPI, TCI, OBFI, $\overline{\text{IBFI}}$, CLTH will be TRUE and all other GPIB signals will be FALSE.
2) The 8292's firmware will cause the above mentioned five signals to go FALSE after approximately 17.5 μs (at 6 MHz).
3) These registers will be cleared: Interrupt Status, Interrupt Mask, Error Mask, Time Out, Event Counter, Error Flag.
4) If the 8292 is the System Controller (SYC is TRUE), then IFC will be sent TRUE for approximately 100 μs and the Controller function will end up in charge of the bus. If the 8292 is not the System Controller then it will end up in an Idle state.
5) TCI will not be set.

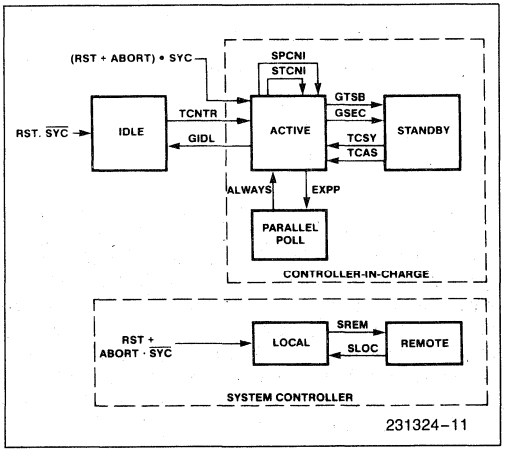


Figure 13. 8292 Command Flowchart

RSTI—Reset Interrupts (Command = 0F3)

This command clears all pending interrupts and error flags. The 8292 will stop waiting for actions to occur (e.g., waiting for ATN to go FALSE in a TCNTR command or waiting for the proper handshake state in a TCSY command). TCI will not be set.

ABORT—Abort all operations and Clear Interface
(Command = 0F9H)

If the 8292 is not the System Controller this command acts like a NOP and flags a USER ERROR in the Error Flag Register. No TCI will occur.

If the 8292 is the system Controller then IFC is set TRUE for approximately 100 μs and the 8292 becomes the Controller-in-Charge and asserts ATN. TCI will be set, only if the 8292 was NOT the CIC.

STCNI—Start Counter Interrupts
(Command = 0FEH)

Enables the EV Counter Interrupt. TCI will not be set. Note that the counter must be enabled by a GSEC command.

SPCNI—Stop Counter Interrupts
(Command = 0F0H)

The 8292 will not generate an EV interrupt when the counter reaches 0. Note that the counter will continue counting. TCI will not be set.

SREM—Set Interface to Remote Control
(Command = 0F8H)

If the 8292 is the System Controller, it will set REN and TCI TRUE. Otherwise it only sets the User Error Flag.

SLOC—Set Interface to Local Mode
(Command = 0F7H)

If the 8292 is the System Controller, it will set REN FALSE and TCI TRUE. Otherwise, it only sets the User Error Flag.

EXPP—Execute Parallel Poll
(Command = 0F5H)

If not Controller-in-Charge, the 8292 will treat this as a NOP and does not set TCI. If it is the Controller-in-Charge then it sets IDY (EOI & ATN) TRUE and generates a local DAV pulse (that never reaches the GPIB because of gates in the 8293). If the 8291 is configured as a listener, it will capture the Parallel Poll Response byte in its data register. TCI is not generated, the CPU must detect the BI (Byte In) from the 8291. The 8292 will be ready to accept another command before the BI occurs; therefore the 8291's BI serves as a task complete indication.

GTSB—Go To Standby (Command = 0F6H)

If the 8292 is not the Controller-in-Charge, it will treat this command as a NOP and does not set TCI TRUE. Otherwise, it goes to Controller Standby State (CSBS), sets ATN FALSE and TCI TRUE. This command is used as part of the Send, Receive, Transfer and Serial Poll System commands (see next section) to allow the addressed talker to send data/status.

If the data transfer does not start within the specified Time-Out, the 8292 sets TOUT2 TRUE in the Error Flag Register and sets SPI (if enabled). The controller continues waiting for a new command. The CPU must decide to wait longer or to regain control and take corrective action.

GSEC—Go To Standby and Enable Counting
(Command = 0F4H)

This command does the same things as GTSB but also initializes the event counter to the value previously stored in the Event Counter Register (default value is 256) and enables the counter. One may wire the count input to $\overline{\text{NDAC}}$ to count bytes. When the counter reaches zero, it sets EV (and SPI if enabled) in Interrupt Status and will set EV every 256 bytes thereafter. Note that there is a potential loss of count information if the CPU does not respond to the EV/SPI before another 256 bytes have been transferred. TCI will be set at the end of the command.

TCSY—Take Control Synchronously
(Command = 0FDH)

If the 8292 is not in Standby, it treats this command as a NOP and does not set TCI. Otherwise, it waits for the proper handshake state and sets ATN TRUE. The 8292 will set TOUT3 if the handshake never assumes the correct state and will remain in this command until the handshake is proper or a RSTI command is issued. If the 8292 successfully takes control, it sets TCI TRUE.

This is the normal way to regain control at the end of a Send, Receive, Transfer or Serial Poll System Command. If TCSY is not successful, then the controller must try TCAS (see warning below).

TCAS—Take Control Asynchronously
(Command = 0FCH)

If the 8292 is not in Standby, it treats this command as a NOP and does not set TCI. Otherwise, it arbitrarily sets ATN TRUE and ECI TRUE. Note that this action may cause devices on the bus to lose a data byte or cause them to interpret a data byte as a command byte. Both Actions can result in anomalous behavior. TCAS should be used only in emergencies. If TCAS fails, then the System Controller will have to issue an ABORT to clean things up.

GIDL—Go to Idle (Command = 0F1H)

If the 8292 is not the Controller in Charge and Active, then it treats this command as a NOP and does not set TCI. Otherwise, it sets ATN FALSE, becomes Not Controller in Charge, and sets TCI TRUE. This command is used as part of the Pass Control System Command.

TCNTR—Take (Receive) Control
(Command = 0FAH)

If the 8292 is not Idle, then it treats this command as a NOP and does not set TCI. Otherwise, it waits for the current Controller-in-Charge to set ATN FALSE. If this does not occur within the specified Time Out, the 8292 sets TOUT1 in the Error Flag Register and sets SPI (if enabled). It will not proceed until ATN goes false or it receives an RSTI command. Note that the Controller in Charge must previously have sent this controller (via the 8291's command pass through register) a Pass Control message. When ATN goes FALSE, the 8292 sets CIC, ATN and TCI TRUE and becomes Active.

SOFTWARE DRIVER OUTLINE

The set of system commands discussed below is shown in Figure 14. These commands are implemented in software routines executed by the main CPU.

The following section assumes that the Controller is the System Controller and will not Pass Control. This is a valid assumption for 99+% of all controllers. It also assumes that no DMA or Interrupts will be used. SYC (System Control Input) should not be changed after Power-on in any system—it adds unnecessary complexity to the CPU's software.

In order to use polling with the 8292 one must enable TCI but not connect the pin to the CPU's interrupt pin. TCI must be readable by some means. In this application example it is connected to bit 1 port 6FH on the ZT7488/18. In addition, the other three 8292 interrupt lines and the 8291 interrupt are also on that port (SPI-Bit 2, $\overline{\text{IBFI}}$-Bit 4, OBFI-Bit 3, 8291 INT-Bit 0).

These drivers assume that only primary addresses will be used on the GPIB. To use secondary addresses, one must modify the test for valid talk/listen addresses (range macro) to include secondaries.

```
  INIT      INITIALIZATION
Talker/Listener
  SEND      SEND DATA
  RECV      RECEIVE DATA
  XFER      TRANSFER DATA
Controller
  TRIG      GROUP EXECUTE TRIGGER
  DCLR      DEVICE CLEAR
  SPOL      SERIAL POLL
  PPEN      PARALLEL POLL ENABLE
  PPDS      PARALLEL POLL DISABLE
  PPUN      PARALLEL POLL UNCONFIGURE
  PPOL      PARALLEL POLL
  PCTL      PASS CONTROL
  RCTL      RECEIVE CONTROL
  SRQD      SERVICE REQUESTED
System Controller
  REME      REMOTE ENABLE
  LOCL      LOCAL
  IFCL      ABORT/INTERFACE CLEAR
```

Figure 14. Software Drive Routines

Initialization

8292—Comes up in Controller Active State when SYC is TRUE. The only initialization needed is to enable the TCI interrupt mask. This is done by writing 0A0H to Port 68H.

8291—Disable both the major and minor addresses because the 8291 will never see the 8292's commands/addresses (refer to earlier hardware discussion). This is done by writing 60H and 0E0H to Port 66H.

Set Address Mode to Talk-only by writing 80H to Port 64H.

Set internal counter to 3 MHz to match the clock input coming from the 8085 by writing 23H to Port 65H. High speed mode for the handshakes will not be used here even though the hardware uses three-state drivers.

No interrupts will be enabled now. Each routine will enable the ones it needs for ease of polling operation. The INT bit may be read through Port 6FH. Clear both interrupt mask registers.

Release the chip's initialization state by writing 0 to Port 65H.

```
INIT:
 Enable-8292                    ;Set up In. pins for Port 6FH
  Enable TCI                    ;Task complete must be on
 Enable-8291
  Disable major address         ;In controller usage, the 8291
  Disable minor address         ;Is set to talk only and/or listen only
  ton                           ;Talk only is our rest state
  Clock frequency               ;3 MHz in this ap note example
  All interrupts off
  Immediate execute pon         ;Releases 8291 from init. state
```

Talker/Listener Routines

SEND DATA

SEND <listener list pointer> <count> <EOS> <data buffer pointer>

This system command sends data from the CPU to one or more devices. The data is usually a string of ASCII characters, but may be binary or other forms as well. The data is device-specific.

My Talk Address (MTA) must be output to satisfy the GPIB requirement of only one talker at a time (any other talker will stop when MTA goes out). The MTA is not needed as far as the 8291 is concerned—it will be put into talk-only mode (ton).

This routine assumes a non-null listener list in that it always sends Univeral Unlisten. If it is desired to send data to the listeners previously addressed, one could add a check for a null list and not send UNL. Count must be 255 or less due to an 8 bit register. This routine also always uses an EOS character to terminate the string output; this could easily be eliminated and rely on the count. Items in brackets ( ) are optional and will not be included in the actual code in Appendix A.

```
SEND:
 Output-to-8291 MTA, UNL              ;We will talk, nobody listen
 Put EOS into 8291                    ;End of string compare character
 While 20H ≤ listener ≤ 3EH           ;GPIB listen addresses are
  output-to-8291 listener             ;"space" thru ">" ASCII
  Increment listen list pointer       ;Address all listeners
 Output-to-8292 GTSB                  ;8292 stops asserting ATN, go to standby
 Enable-8291
  Output EOI on EOS sent              ;Send EOI along with EOS character
 If count < > 0 then
  While not (end or count = 0)        ;Wait for EOS or end of count
  (could check tout 2 here)           ;Optionally check for stuck bus-tout 2
   Output-to-8291 data                ;Output all data, one byte at a time
   Increment data buffer pointer      ;8085 CREG will count for us
   Decrement count
 Output-to-8292 TCSY                  ;8292 asserts ATN, take control sync.
 (If tout3 then take control async)   ;If unable to take control sync.
 Enable 8291                          ;Restore 8291 to standard condition
  No output EOI on EOS sent
 Return
```

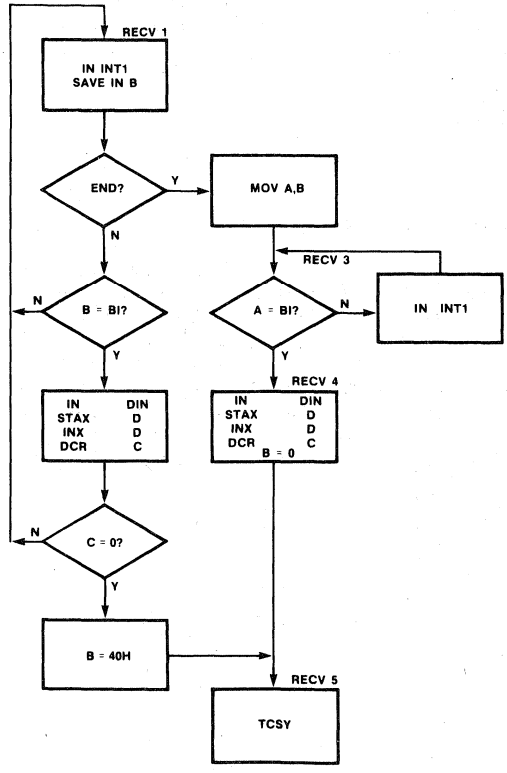


Figure 15. Flowchart for Receive Ending Conditions

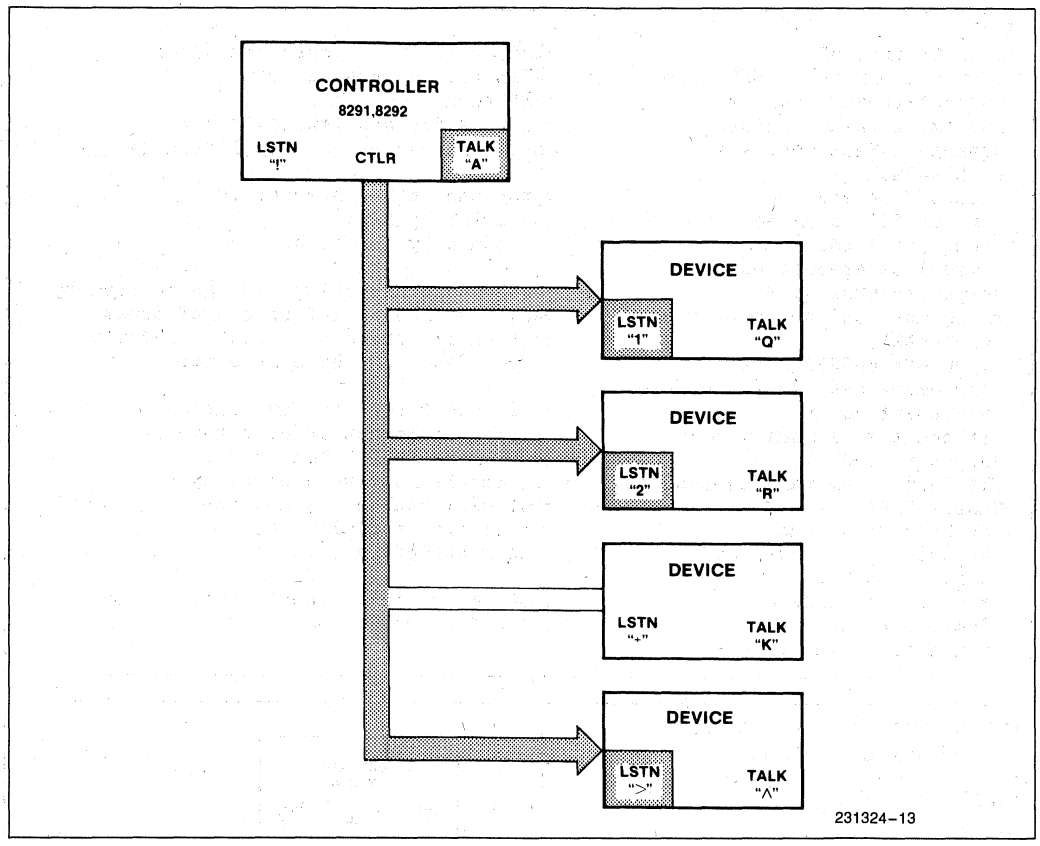


Figure 16. SEND to "1", "2", ">"; "ABCD"; EOS = "D"

RECEIVE DATA

RECV <talker> <count> <EOS> <data buffer pointer>

This system command is used to input data from a device. The data is typically a string of ASCII characters.

This routine is the dual of SEND. It assumes a new talker will be specified, a count of less than 257, and an EOS character to terminate the input. EOI received will also terminate the input. Figure 15 shows the flow chart for the RECV ending conditions. My Listen Address (MLA) is sent to keep the GPIB transactions totally regular to facilitate analysis by a GPIB logic analyzer like the Ziatech ZT488. Otherwise, the bus would appear to have no listener even though the 8291 will be listening.

Note that although the count may go to zero before the transmission ends, the talker will probably be left in a strange state and may have to be cleared by the controller. The count ending of RECV is therefore used as an error condition in most situations.

```
RECV:
 Put EOS into 8291                    ;End of string compare character
 If 40H ≤ talker ≤ 5EH then           ;GPIB talk addresses are
  Output-to-8291 talker               ;"@" thru "^" ASCII
 Increment talker pointer             ;Do this for consistency's sake
 Output-to-8291 UNL, MLA              ;Everyone except us stop listening
 Enable-8291
  Holdoff on end                      ;Stop when EOS character is
  End on EOS received                 ;Detected by 8291
  lon, reset ton                      ;Listen only (no talk)
  Immediate execute pon
 Output-to-8292 GTSB                  ;8292 stops asserting ATN, go to standby
 While not (end or count = 0          ;wait for EOS or EOI or end of count
 (or tout2))                          ;optionally check for stuck bus-tout2
  Input-from-8291 data                ;input data, one byte at a time
  Increment data buffer pointer
  Decrement count                     ;Use 8085 C register as counter
 (If count = 0 then error)            ;Count should not occur before end
 Output-to-8292 TCSY                  ;8292 asserts ATN take control
 (If Tout3 then take control async.)  ;If unable to take control sync.
 Enable-8291                          ;Put 8291 back as needed for
  No holdoff on end                   ;Controller activity and
  No end on EOS received              ;Clear holdoff due to end
  ton, reset lon
  Finish handshake                    ;Complete holdoff due to end, if any
  Immediate execute pon               ;Needed to reset lon
 Return error-indicator
```

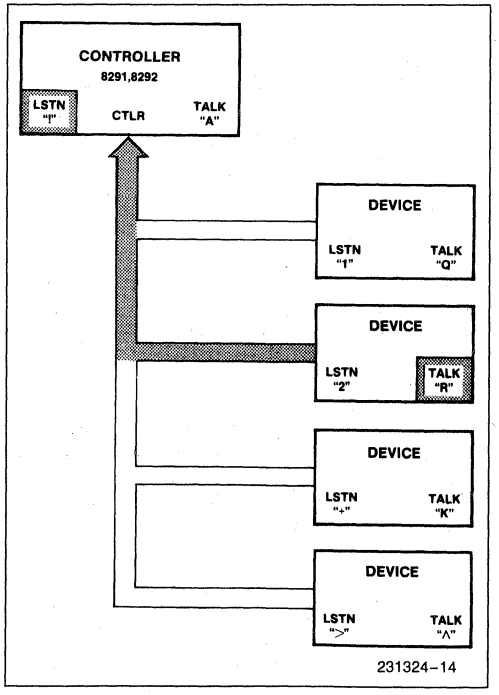


Figure 17. RECV from "R"; EOS = 0DH

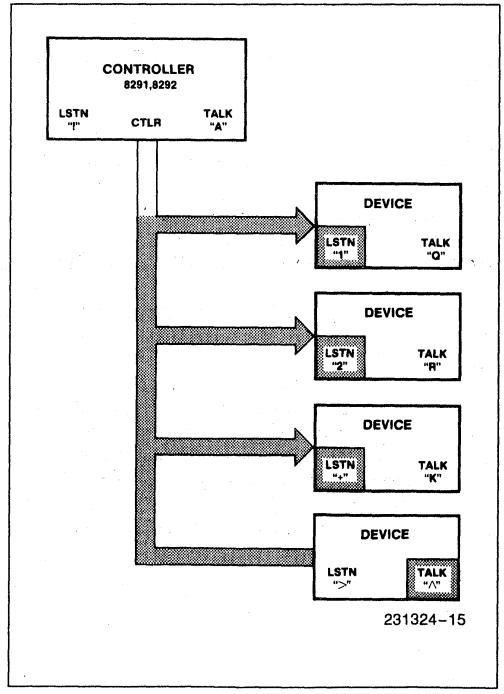


Figure 18. XFER from "^" to "1", "2", "+"; EOS = 0DH

TRANSFER DATA

XFER <Talker> <Listener list> <EOS>

This system command is used to transfer data from a talker to one or more listeners where the controller does not participate in the transfer of the ASCII data. This is accomplished through the use of the 8291's continuous acceptor handshake mode while in listen-only.

This routine assumes a device list that has the ASCII talker address as the first byte and the string (one or more) or ASCII listener addresses following. The EOS character or an EOI will cause the controller to take take control synchronously and thereby terminate the transfer.

```
XFER:
 Output-to-8291: Talker, UNL          ;Send talk address and unlisten
 While 20H ≤ listen ≤ 3EH
  Output-to-8291: Listener            ;Send listen address
  Increment listen list pointer
 Enable-8291
  lon, no ton                         ;Controller is pseudo listener
  Continuous AH mode                  ;Handshake but don't capture data
  End on EOS received                 ;Capture EOS as well as EOI
  Immediate execute PON               ;Initialize the 8291
 Put EOS into 8291                    ;Set up EOS character
 Output-to-8292: GTSB                 ;Go to standby
                                      ;8292 waits for EOS or EOI and then

 Upon end (or tout2) then
  Take control synchronously          ;Regains control
 Enable-8291                          ;Go to Ready for Data
  Finish handshake
  Not continuous AH mode
  Not END on EOS received
  ton
  Immediate execute pon
 Return
```

Controller

GROUP EXECUTE TRIGGER

TRIG <Listener list>

This system command causes a group execute trigger (GET) to be sent to all devices on the listener list. The intended use is to synchronize a number of instruments.

```
TRIG:
 Output-to-8291 UNL                   ;Everybody stop listening
 While 20H ≤ listener ≤ 3EH           ;Check for valid listen address
  Output-to-8291 Listener             ;Address each listener
  Increment listen list pointer       ;Terminate on any non-valid character
 Output-to-8291 GET                   ;Issue group execute trigger
 Return
```

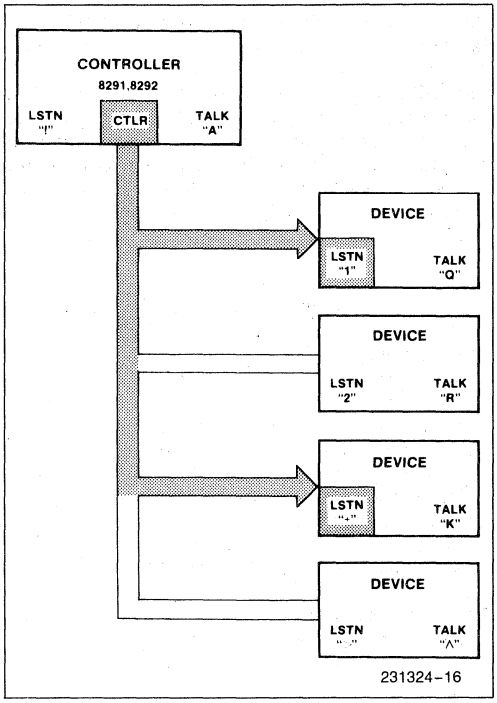


Figure 19. TRIG "1", "+"

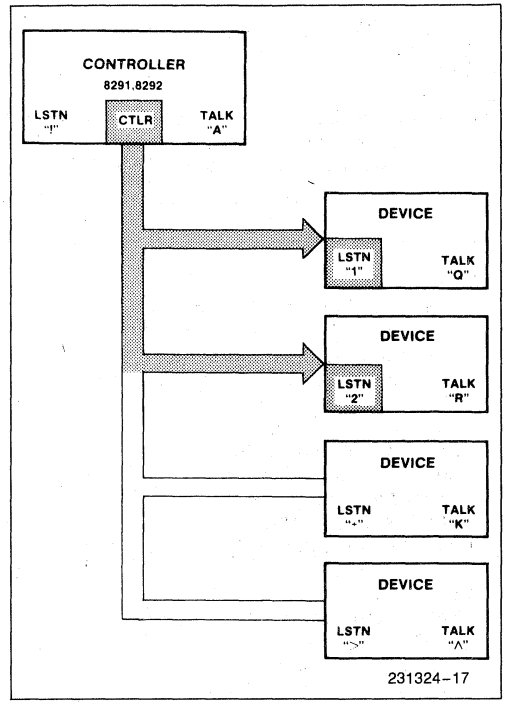


Figure 20. DCLR "1", "2"

DEVICE CLEAR

DCLR <Listener list>

This system command causes a device clear (SDC) to be sent to all devices on the listener list. Note that this is not intended to clear the GPIB interface of the device, but should clear the device-specific logic.

```
DCLR:
 Output-to-8291 UNL                    ;Everybody stop listening
 While 20H ≤ listener ≤ 3EH            ;Check for valid listen address
  Output-to-8291 Listener              ;Address each listener
  Increment listen list pointer        ;Terminate on any non-valid character
 Output-to-8291 SDC                    ;Selective device clear
 Return
```

SERIAL POLL

SPOL <Talker list> <status buffer pointer>

This system command sequentially addresses the designated devices and receives one byte of status from each. The bytes are stored in the buffer in the same order as the devices appear on the talker list. MLA is output for completeness.

```
SPOL:
 Output-to-8291 UNL, MLA, SPE          ;Unlisten, we listen, serial poll enable
                                       ;Only one byte of serial poll
                                       ;Status wanted from each talker
 While 40H ≤ talker ≤ 5 EH             ;Check for valid transfer
  Output-to-8291 talker                ;Address each device to talk
  Increment talker list pointer        ;One at a time
  Enable-8291
   lon, reset ton                      ;Listen only to get status
   Immediate execute pon               ;This resets ton
  Output-to-8292 GTSB                  ;Go to standby
  Wait for data in (BI)                ;Serial poll status byte into 8291
  Output-to-8292 TCSY                  ;Take control synchronously
  Input-from-8291 data                 ;Actually get data from 8291
  Increment buffer pointer
  Enable 8291
   ton, reset lon
   Immediate execute pon               ;Reset lon
  Output-to-8291 SPD                   ;Send serial poll disable after all
                                         devices polled
  Return
```

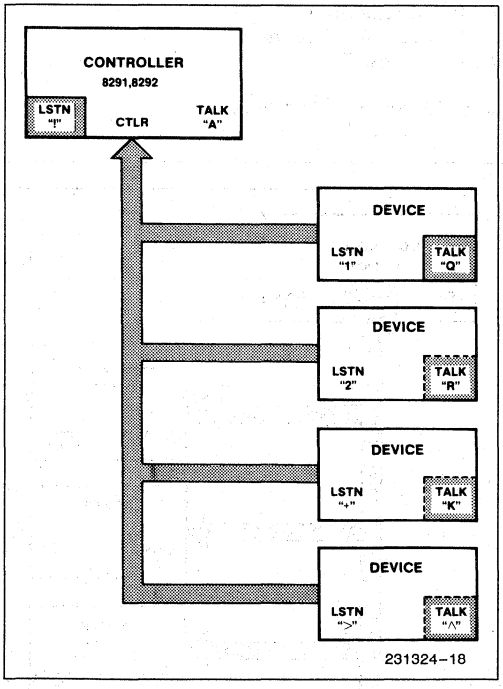


Figure 21. SPOL "Q", "R", "K", "∧"

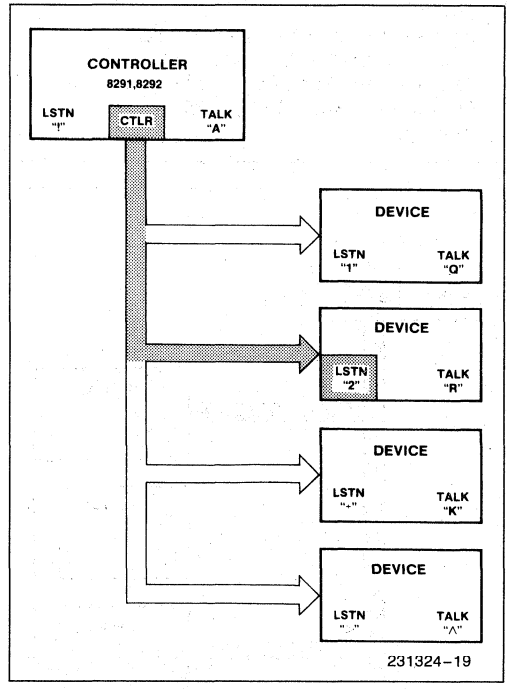


Figure 22. PPEN "2"; $iP_3P_2P_1 = 0111B$

PARALLEL POLL ENABLE

PPEN<Listener list> <Configuration Buffer pointer>

This system command configures one or more devices to respond to Parallel Poll, assuming they implement subset PP1. The configuration information is stored in a buffer with one byte per device in the same order as

devices appear on the listener list. The configuration byte has the format XXXXIP3P2P1 as defined by the IEEE Std. P3P2P1 indicates the bit # to be used for a response and I indicates the assertion value. See Sec. 2.9.3.3 of the Std. for more details.

```
PPEN:
 Output-to-8291 UNL                    ;Universal unlisten
 While 20H ≤ Listener ≤ 3EH            ;Check for valid listener
  Output-to-8291 listener              ;Stop old listener, address new
  Output-to-8291 PPC, (PPE or data)    ;Send parallel poll info
  Increment listener list pointer      ;Point to next listener
  Increment buffer pointer             ;One configuration byte per listener
 Return
```

PARALLEL POLL DISABLE

PPDS<listener list>

This system command disables one or more devices from responding to a Parallel Poll by issuing a Parallel Poll Disable (PPD). It does not deconfigure the devices.

```
PPDS:
 Output-to-8291 UNL                    ;Universal Unlisten
 While 20H ≤ Listener ≤ 3EH            ;Check for valid listener
  Output-to-8291 listener              ;Address listener
  Increment listener list pointer
 Output-to-8291 PPC, PPD               ;Disable PP on all listeners
 Return
```

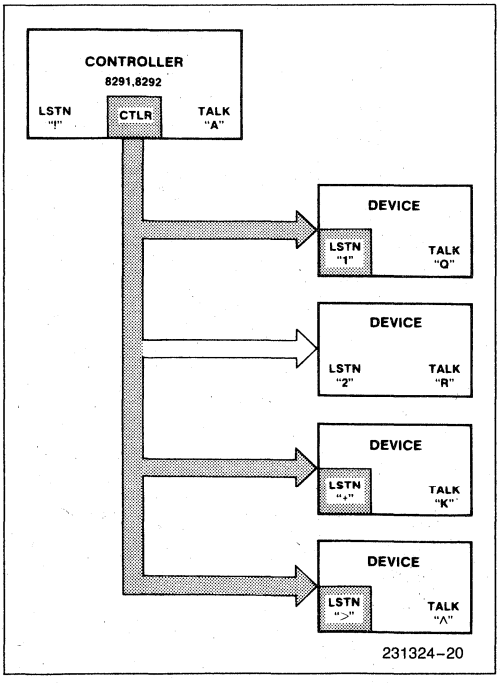


Figure 23. PPDS "1", "+", ">"

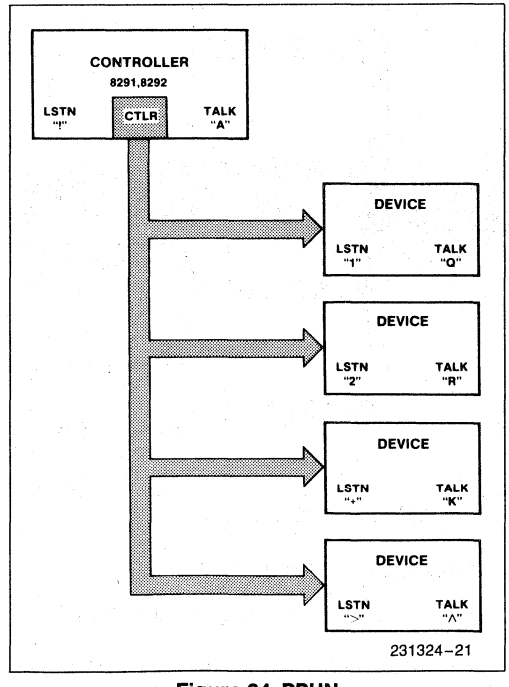


Figure 24. PPUN

PARALLEL POLL UNCONFIGURE

PPUN

This system command deconfigures the Parallel Poll response of all devices by issuing a Parallel Poll Unconfigure message.

```
PPUN:
 Output-to-8291 PPU                    ;Unconfigure all parallel poll
 Return
```

CONDUCT A PARALLEL POLL

PPOL

This system command causes the controller to conduct a Parallel Poll on the GPIB for approximately 12.5 μsec (at 6 MHz). Note that a parallel poll does not use the handshake; therefore, to ensure that the device knows whether or not its positive response was observed by the controller, the CPU should explicitly acknowledge each device by a device-dependent data string. Otherwise, the response bit will still be set when the next Parallel Poll occurs. This command returns one byte of status.

```
PPOL:
 Enable-8291
  lon                                  ;Listen only
  Immediate execute pon                ;This resets ton
 Output-to-8292 EXPP                   ;Execute parallel poll
 Upon BI                               ;When byte is input
  Input-from-8291 data                 ;Read it
 Enable-8291
  ton                                  ;Talk only
  Immediate execute pon                ;This resets lon
 Return Data (status byte)
```

PASS CONTROL

PCTL<talker>

This system command allows the controller to relinquish active control of the GPIB to another controller. Normally some software protocol should already have informed the controller to expect this, and under what conditions to return control. The 8291 must be set up to become a normal device and the CPU must handle all commands passed through, otherwise control cannot be returned (see Receive Control below). The controller will go idle.

```
PCTL:
 If 40H ≤ talker ≤ 5EH then
  if talker < > MTA then               ;Cannot pass control to myself
   output-to-8291 talker, TCT          ;Take control message to talker
   Enable-8291                         ;Set up 8291 as normal device
    not ton, not lon
    Immediate execute pon              ;Reset ton and lon
    My device address, mode 1          ;Put device number in Register 6
    Undefined command pass through     ;Required to receive control
    (Parallel Poll Configuration)      ;Optional use of PP
  Output-to-8292 GIDL                  ;Put controller in idle
 Return
```

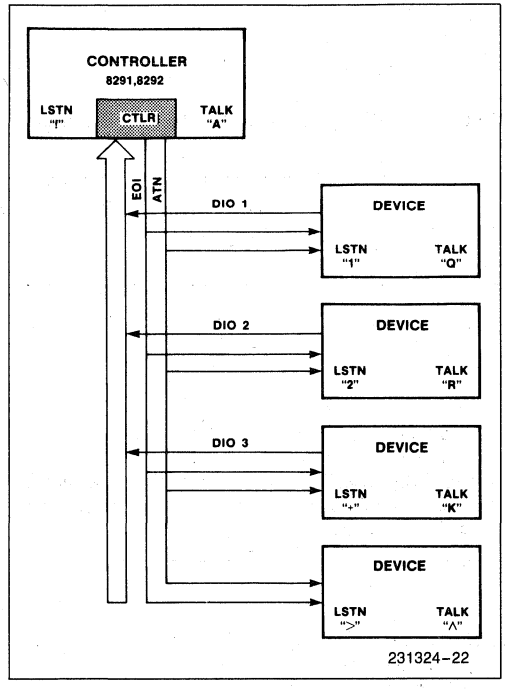


Figure 25. PPOL

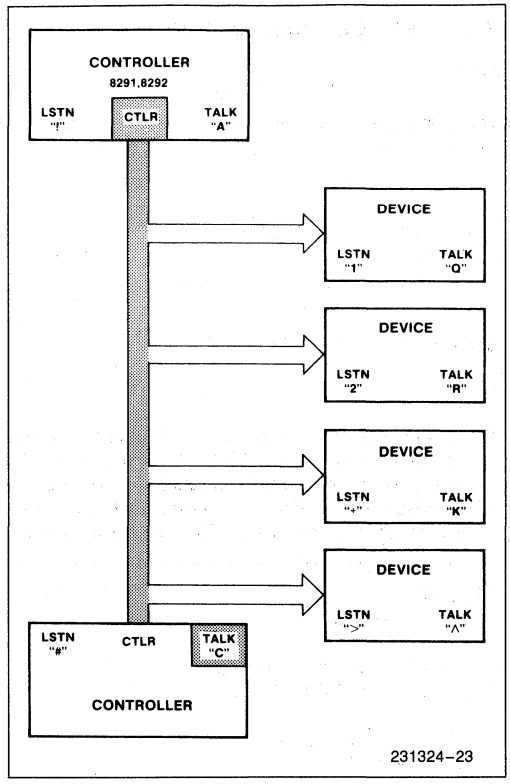


Figure 26. PCTL "C"

RECEIVE CONTROL

RCTL

This system command is used to get control back from the current controller-in-charge if it has passed control to this inactive controller. Most GPIB systems do not use more than one controller and therefore would not need this routine.

To make passing and receiving control a manageable event, the system designer should specify a protocol whereby the controller-in-charge sends a data message to the soon-to-be-active controller. This message should give the current state of the system, why control is being passed, what to do, and when to pass control back. Most of these issues are beyond the scope of this Ap Note.

```
RCTL:
 Upon CPT                              ;Wait for command pass through bit in 8291
  If (command=TCT) then                ;If command is take control and
   If TA then                          ;We are talker addressed
    Enable-8291
     Disable major device number       ;Controller will use ton and lon
     ton                               ;Talk only mode
     Mask off interrupts
     Immediate execute pon
    Output-to-8292 TCNTR               ;Take (receive) control
    Enable-8291
     Valid command                     ;Release handshake
    Return valid
   Else
    Enable-8291
     Invalid command                   ;Not talker addr. so TCT not for us
  Else
   Enable-8291
    Invalid command                    ;Not TCT, so we don't care
 Return invalid
```

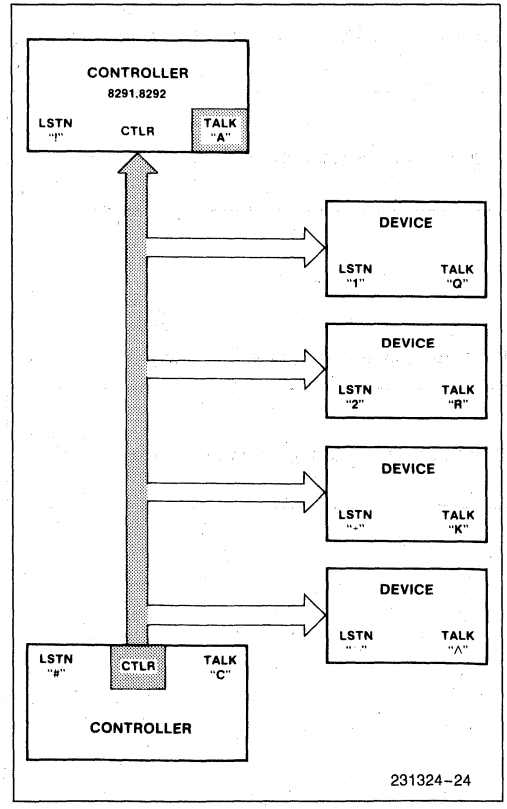


Figure 27. RCTL

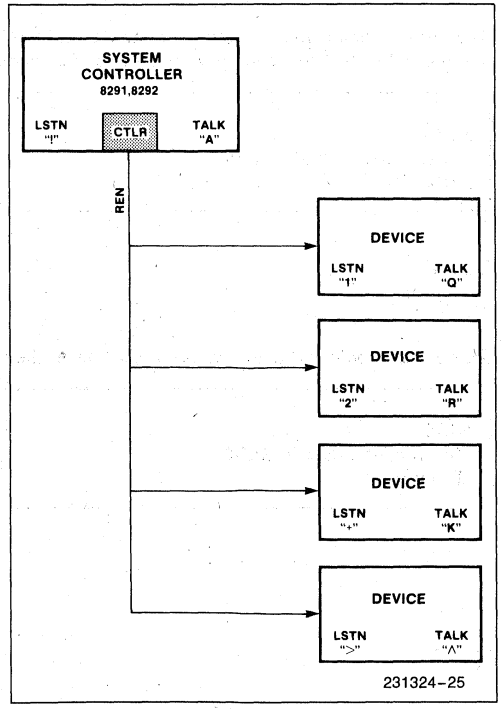


Figure 28. REME

SERVICE REQUEST

SRQD

This system command is used to detect the occurrence of a Service Request on the GPIB. One or more devices may assert SRQ simultaneously, and the CPU would normally conduct a Serial Poll after calling this routine to determine which devices are SRQing.

```
SRQD:
 If SRQ then                          ;Test 92 status bit
  Output-to-8292 IACK.SRQ             ;Acknowledge it
  Return SRQ
  Else return no SRQ
```

System Controller

REMOTE ENABLE

REME

This system command asserts the Remote Enable line (REN) on the GPIB. The devices will not go remote until they are later addressed to listen by some other system command.

```
REME:
 Output-to-8292 SREM                  ;8292 asserts remote enable line
 Return
```

LOCAL

LOCL

This system command deasserts the REN line on the GPIB. The devices will go local immediately.

```
LOCL:
 Output-to-8292 SLOC                  ;8292 stops asserting remote enable
 Return
```

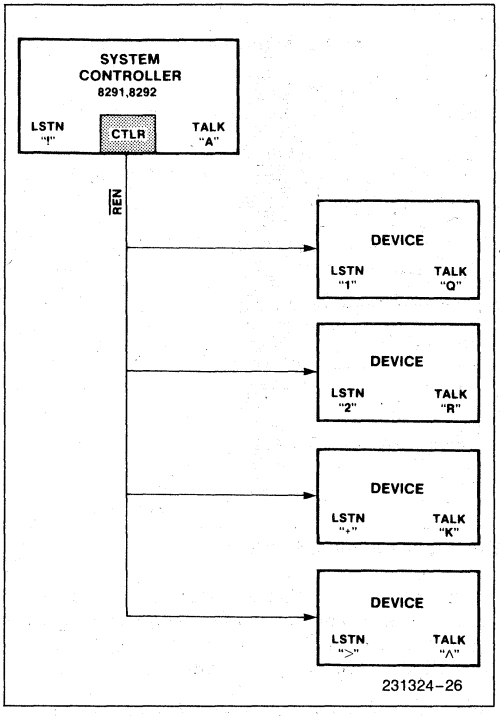


Figure 29. LOCL

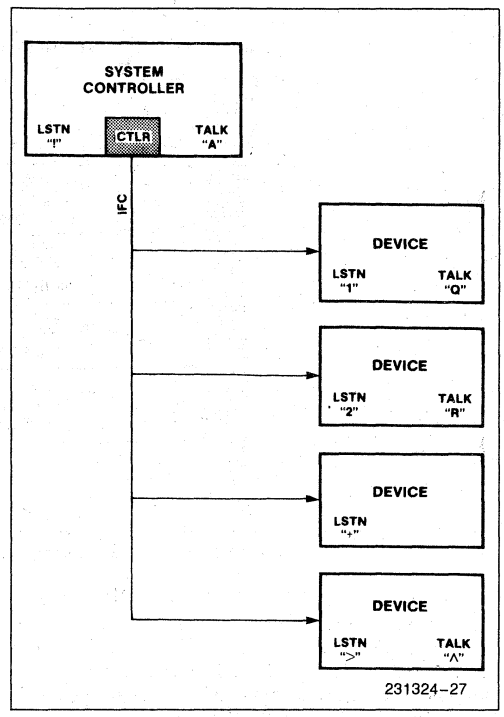


Figure 30. IFCL

INTERFACE CLEAR/ABORT

IFCL

This system command asserts the GPIB's Interface Clear (IFC) line for at least 100 microseconds. This causes all interface logic in all devices to go to a known state. Note that the device itself may or may not be reset, too. Most instruments do totally reset upon IFC. Some devices may require a DCLR as well as an IFCL to be completely reset. The (system) controller becomes Controller-in-Charge.

```
IFCL:
 Output-to-8292 ABORT              ;8292 asserts Interface Clear
 Return                            ;For 100 microseconds
```

INTERRUPTS AND DMA CONSIDERATIONS

The previous sections have discussed in detail how to use the 8291, 8292, 8293 chip set as a GPIB controller with the software operating in a polling mode and using programmed transfer of the data. This is the simplest mode of use, but it ties up the microprocessor for the duration of a GPIB transaction. If system design constraints do not allow this, then either Interrupts and/or DMA may be used to free up processor cycles.

The 8291 and 8292 provide sufficient interrupts that one may return to do other work while waiting for such things as 8292 Task Completion, 8291 Next Byte In, 8291 Last Byte Out, 8292 Service Request In, etc. The only difficulty lies in integrating these various interrupt sources and their matching routines into the overall system's interrupt structure. This is highly situation-specific and is beyond the scope of this Ap Note.

The strategy to follow is to replace each of the WAIT routines (see Appendix A) with a return to the main code and provide for the corresponding interrupt to bring the control back to the next section of GPIB

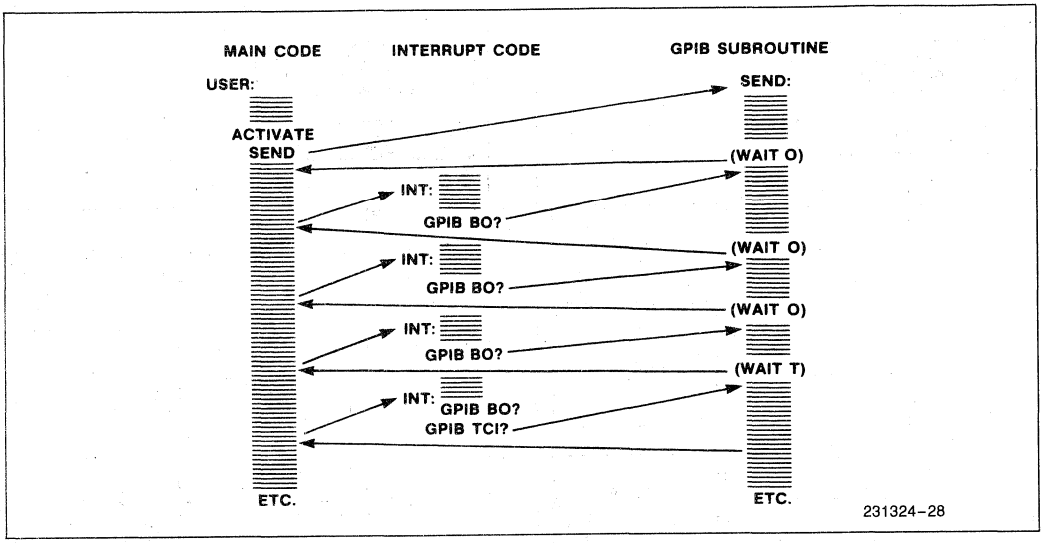


Figure 31. GPIB Interrupt and Co-Routine Flow of Control

code. For example WAITO (Wait for Byte Out of 8291) would be replaced by having the BO interrupt enabled and storing the (return) address of the next instruction in a known place. This co-routine structure will then be activated by a BO interrupt. Figure 31 shows an example of the flow of control.

DMA is also useful in relieving the processor if the average length of a data buffer is long enough to overcome the extra time used to set up a DMA chip. This decision will also be a function of a data rate of the instrument. The best strategy is to use the DMA to handle only the data buffer transfers on SEND and RECV and to do all the addressing and control just as shown in the driver descriptions.

Another major reason for using a DMA chip is to increase the data rate and therefore increase the overall transaction rate. In this case the limiting factor becomes the time used to do the addressing and control of the GPIB using software like that in Appendix A. The data transmission time becomes insignificant at DMA speeds unless extremely long buffers are used.

Refer to Figure 11 for a typical DMA and interrupt based design using the 8291, 8292, 8293. A system like this can achieve a 250K byte transfer rate while under DMA control.

APPLICATION EXAMPLE

This section will present the code required to operate a typical GPIB instrument set up as shown in Figure 32. The HP5328A universal counter and the HP3325 function generator are typical of many GPIB devices; however, there are a wide variety of software protocols to be found on the GPIB. The Ziatech ZT488 GPIB analyzer is used to single step the bus to facilitate debugging the system. It also serves as a training/familiarization aid for newcomers to the bus.

This example will set up the function generator to output a specific waveform, frequency and amplitude. It will then tell the counter to measure the frequency and Request Service (SRQ) when complete. The program will then read in the data. The assembled source code will be found at the end of Appendix A.

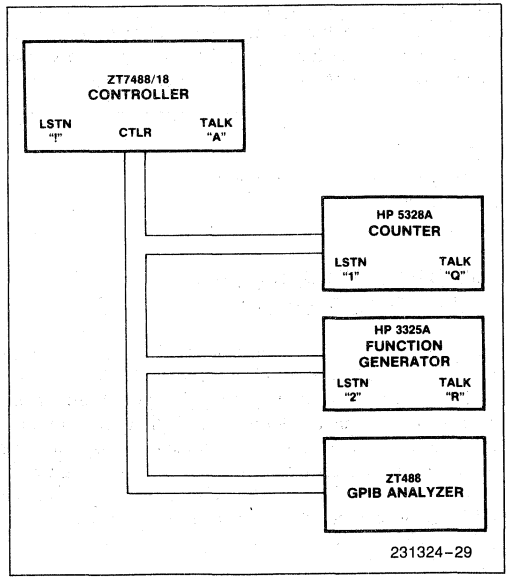


Figure 32. GPIB Example Configuration

```
SEND
 LSTN: "2", COUNT: 15, EOS: ODH, DATA: "FU1FR37KHAM2VO (CR)"
 ;SETS UP FUNCTION GEN. TO 37 KHZ SINE, 2 VOLTS PP
 ;COUNT EQUAL TO # CHAR IN BUFFER
 ;EOS CHARACTER IS (CR) = ODH = CARRIAGE

SEND
 LSTN: "1", COUNT: 6, EOS: "T" DATA: "PR4G7T"
 ;SETS UP COUNTER FOR P:INITIALIZE, F4: FREQ CHAN A
 ;         G7:0.1 HZ RESOLUTION, T:TRIGGER AND SRQ
 ;COUNT IS EQUAL TO # CHAR

WAIT FOR SRQ
SPOL TALK: "Q", DATA: STATUS 1
 ;CLEARS THE SRO_IN THIS EXAMPLE ONLY FREQ CTR ASSERTS SRQ

RECV TALK: "Q", COUNT: 17, EOS: OAH,
 DATA: "+   37000.0E+0" (CR) (LF)
 ;GETS 17 BYTES OF DATA FROM COUNTER
 ;COUNT IS EXACT BUFFER LENGTH
 ;DATA SHOWN IS TPYICAL HP5328A READING THAT WOULD BE RECEIVED
```

CONCLUSION

This Application Note has shown a structured way to view the IEEE 488 bus and has given typical code sequences to make the Intel 8291, 8292, and 8293's behave as a controller of the GPIB. There are other ways to use the chip set, but whatever solution is chosen, it must be integrated into the overall system software.

The ultimate reference for GPIB questions is the IEEE Std 488 -1978 which is available from IEEE, 345 East 47th St., New York, NY, 10017. The ultimate reference for the 8292 is the source listing for it (remember it's a pre-programmed UPI-41A) which is available from INSITE, Intel Corp., 3065 Bowers Ave., Santa Clara, CA 95051.

APPENDIX A

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0
GPIB CONTROLLER SUBROUTINES

LOC  OBJ          LINE        SOURCE STATEMENT

                     1 $TITLE('GPIB CONTROLLER SUBROUTINES')
                     2 ;
                     3 ; GPIB CONTROLLER SUBROUTINES
                     4
                     5 ;
                     6 ;       for Intel 8291, 8292 on ZT 7488/18
                     7 ;       Bert Forbes, Ziatech Corporation
                     8 ;       2410 Broad Street
                     9 ;       San Luis Obispo, CA, USA 93401
                    10 ;
                    11 ;
                    12 ;       General Definitions & Equates
                    13 ;       8291 Control Values
                    14 ;
1000                15         ORG     1000H   ; For ZT7488/18 w/8085
                    16 ;
0060                17 PRT91   EQU     60H     ;8291 Base Port #
                    18 ;
                    19 ;       Reg #0 Data in & Data out
0060                20 DIN     EQU     PRT91+0 ;91 Data in reg
0060                21 DOUT    EQU     PRT91+0 ;91 Data out reg
                    22 ;
                    23 ;       Reg # 1 Interrupt 1 Constants
0061                24 INT1    EQU     PRT91+1 ;INT Reg 1
0061                25 INTM1   EQU     PRT91+1 ;INT Mask Reg. 1
0002                26 BOM     EQU     02      ;91 BO INTRP Mask
0001                27 BIM     EQU     01      ;91 BI INTRP Mask
0010                28 ENDMK   EQU     10H     ;91 END INTRP Mask
0080                29 CPT     EQU     80H     ;91 command pass thru int bit
                    30 ;
                    31 ;       Reg #2 Interrupt 2
0062                32 INT2    EQU     PRT91+2
                    33 ;
                    34 ;       Reg #4 Address Mode Constants
0064                35 ADRMD   EQU     PRT91+4 ;91 address mode register #
0080                36 TON     EQU     80H     ;91 talk only mode & not listen only
0040                37 LON     EQU     40H     ;91 listen only & not ton
00C0                38 TLON    EQU     0C0H    ;91 talk & listen only
0001                39 MODE1   EQU     01      ;mode 1 addressing for device
                    40
                    41 ;       Reg #4          (Read)  Address Status Register
0064                42 ADRST   EQU     PRT91+4 ;reg #4
0020                43 EOIST   EQU     20H
0002                44 TA      EQU     2
0001                45 LA      EQU     1       ;listener active
                    46 ;
                    47 ;       Reg #5  (Write) Auxillary Mode Register
0065                48 AUXMD   EQU     PRT91+5 ;91 auxillary mode register #
0023                49 CLKRT   EQU     23H     ;91 3 Mhz clock input
```

231324-30

```
0003              50 FNHSK   EQU     03      ;91 fininsh handshake command
0006              51 SDEOI   EQU     06      ;91 send EOI with next byte
00B0              52 AXRA    EQU     80H     ;91 aux. reg A pattern
0001              53 HOHSK   EQU      1      ;91 hold off handshake on all bytes
0002              54 HOEND   EQU      2      ;91 hold off handshake on end
0003              55 CAHCY   EQU      3      ;91 continuous AH cycling
0004              56 EDEOS   EQU      4      ;91 end on EOS received
0008              57 EOIS    EQU      8      ;91 output EOI on EOS sent
00DF              58 VSCMD   EQU     0FH     ;91 valid command pass through
0007              59 NVCMD   EQU     07H     ;91 invalid command pass through
00A0              60 AXRB    EQU     0A0H    ;Aux. reg. B pattern
0001              61 CPTEN   EQU     01H     ;command pass thru enable
                  62 ;
                  63 ;       Reg #5  (Read)
0005              64 CPTRG   EQU     PRT91+5
                  65 ;
                  66 ;       Reg #6  Address 0/1 reg. constants
0006              67 ADR01   EQU     PRT91+6
0060              68 DTDL1   EQU     60H     ;Disable major talker & listener
00E0              69 DTDL2   EQU     0E0H    ;Disable minor talker & listener
                  70 ;
                  71 ;       Reg #7  EOS     Character Register
0007              72 EOSR    EQU     PRT91+7
                  73 ;
                  74 ;
                  75 ;       8292    CONTROL VALUES
                  76 ;
                  77 ;
                  78 ;
0008              79 PRT92   EQU     PRT91+8 ;8292 Base Port # (CS7)
                  80 ;
0008              81 INTMR   EQU     PRT92+0 ;92 INTRP Mask Reg
00A0              82 INTM    EQU     0A0H    ;TCI
                  83 ;
0008              84 ERRM    EQU     PRT92+0 ;92 Error Mask Reg
0001              85 TOUT1   EQU     01      ;92 Time Out for Pass Control
0002              86 TOUT2   EQU     02      ;92 Time Out for Standby
0004              87 TOUT3   EQU     04      ;92 Time Out for Take Control Sync
0008              88 EVREG   EQU     PRT92+0 ;92 Event Counter Pseudo Reg
0008              89 TOREG   EQU     PRT92+0 ;92 Time Out Pseudo Reg
                  90 ;
0009              91 CMD92   EQU     PRT92+1 ;92 Command Register
                  92 ;
0009              93 INTST   EQU     PRT92+1 ;92 Interrupt Status Reg
0010              94 EVBIT   EQU     10H     ;Event Counter Bit
0002              95 IBFBT   EQU     02      ;Input Buffer Full Bit
0020              96 SRQBT   EQU     20H     ;Seq bit
                  97 ;
0008              98 ERFLG   EQU     PRT92+0 ;92 Error Flag Pseudo Reg
0008              99 CLRST   EQU     PRT92+0 ;92 Controller Status Pseudo Reg
0008             100 BUSST   EQU     PRT92+0 ;92 GPIB (Bus) Status Pseudo Reg
0008             101 EVCST   EQU     PRT92+0 ;92 Event Counter Status Pseudo Reg
0008             102 TOST    EQU     PRT92+0 ;92 Time Out Status Pseudo Reg
                 103 ;
                 104 ;       8292    OPERATION COMMANDS
                 105 ;
                 106 ;
00F0             107 SPCNI   EQU     0F0H    ;Stop Counter Interrupts
00F1             108 GIDL    EQU     0F1H    ;Go to idle
00F2             109 RSET    EQU     0F2H    ;Reset
00F3             110 RSTI    EQU     0F3H    ;Reset Interrupts
00F4             111 GSEC    EQU     0F4H    ;Goto standby, enable counting
00F5             112 EXPP    EQU     0F5H    ;Execute parallel poll
00F6             113 GTSB    EQU     0F6H    ;Goto standby
00F7             114 SLOC    EQU     0F7H    ;Set local mode
00F8             115 SREM    EQU     0F8H    ;Set interface to remote
00F9             116 ABORT   EQU     0F9H    ;Abort all operation, clear interface
00FA             117 TCNTR   EQU     0FAH    ;Take control (Receive control)
00FC             118 TCASY   EQU     0FCH    ;Take control asyncronously
00FD             119 TCSY    EQU     0FDH    ;Take control syncronously
00FE             120 STCNI   EQU     0FEH    ;Start counter interrupts
                 121 ;
                 122 ;
```

231324-31

```
                    123 ;       8292    UTILITY COMMANDS
                    124 ;
                    125 ;
00E1                126 WOUT    EQU     0E1H    ;Write to timeout reg
00E2                127 WEVC    EQU     0E2H    ;Write to event counter
00E3                128 REVC    EQU     0E3H    ;Read event counter status
00E4                129 RERF    EQU     0E4H    ;Read error flag reg
00E5                130 RINM    EQU     0E5H    ;Read interrupt mask reg
00E6                131 RCST    EQU     0E6H    ;Read controller status reg
00E7                132 RBST    EQU     0E7H    ;Read GPIB Bus status reg
00E9                133 RTOUT   EQU     0E9H    ;Read timeout status reg
00EA                134 RERM    EQU     0EAH    ;Read error mask reg
000B                135 IACK    EQU     0BH     ;Interrupt Acknowledge
                    136 ;
                    137 ;
                    138 ;       PORT F BIT ASSIGNMENTS
                    139 ;
                    140 ;
                    141 ;
006F                142 PRTF    EQU     PRT91+0FH       ;ZT7488 port 6F for interrupts
0002                143 TCIF    EQU     02H     ;Task complete interrupt
0004                144 SPIF    EQU     04H     ;Special interrupt
0008                145 OBFF    EQU     08H     ;92 Output (to CPU) Buffer full
0010                146 IBFF    EQU     10H     ;92 Input (from CPU) Buffer empty
0001                147 BOF     EQU     01H     ;91 Int line (BO in this case)
                    148 ;
                    149 ;       GPIB MESSAGES (COMMANDS)
                    150 ;
0001                151 MDA     EQU     1       ;My device address is 1
0041                152 MTA     EQU     MDA+40H ;My talk address is 1 ("A")
0021                153 MLA     EQU     MDA+20H ;My listen address is 1 ("!")
003F                154 UNL     EQU     3FH     ;Universal unlisten
0008                155 GET     EQU     08      ;Group Execute Trigger
0004                156 SDC     EQU     04H     ;Device Clear
0018                157 SPE     EQU     18H     ;Serial poll enable
0019                158 SPD     EQU     19H     ;Serial poll disable
0005                159 PPC     EQU     05      ;Parallel poll configure
0070                160 PPD     EQU     70H     ;Parallel poll disable
0060                161 PPE     EQU     60H     ;Parallel poll disable
0015                162 PPU     EQU     15H     ;Parallel poll unconfigured
0009                163 TCT     EQU     09      ;Take control (pass control)
                    164
                    165 ;       MACRO DEFINITIONS
                    166 ;
                    167 ;
                    168 ;
                    169 SETF    MACRO           ;Sets flags on A register
                    170         ORA     A
                    171         ENDM
                    172 ;
                    173 WAITO   MACRO           ;Wait for last 91 byte to be done
                    174         LOCAL   WAITL
                    175 WAITL:  IN      INT1    ;Get Int1 status
                    176         ANI     BOM     ;Check for byte out
                    177         JZ      WAITL   ;If not, try again
                    178         ENDM            ;until it is
                    179 ;
                    180 ;
                    181 WAITI   MACRO           ;Wait for 91 byte to be input
                    182         LOCAL   WAITL
                    183 WAITL:  IN      INT1    ;Get INT1 status
                    184         MOV     B,A     ;Save status in B
                    185         ANI     BIM     ;Check for byte in
                    186         JZ      WAITL   ;If not, just try again
                    187         ENDM            ;until it is
                    188 ;
                    189 WAITX   MACRO           ;Wait for 92's TCI to go false
                    190         LOCAL   WAITL
                    191 WAITL:  IN      PRTF
                    192         ANI     TCIF
                    193         JNZ     WAITL
                    194         ENDM
                    195 ;
```

231324-32

```
                        196 WAITT   MACRO
                        197         LOCAL   WAITL
                        198 WAITL:  IN      PRTF    ;Get task complete int,etc.
                        199         ANI     TCIF    ;Mask it
                        200         JZ      WAITL   ;Wait for task to be complete
                        201         ENDM
                        202
                        203 RANGE   MACRO   LOWER,UPPER,LABEL
                        204                         ;Checks for value in range
                        205                         ;branches to label if not
                        206                         ;in range. Falls through if
                        207                         ;lower <= ( (H)(L) ) <= upper.
                        208                         ;Get next byte.
                        209         MOV     A,M
                        210         CPI     LOWER
                        211         JM      LABEL
                        212         CPI     UPPER+1
                        213         JP      LABEL
                        214         ENDM
                        215 ;
                        216 CLRA    MACRO
                        217         XRA     A       ;A XOR A =0
                        218         ENDM
                        219 ;
                        220 ;       All of the following routines have these common
                        221 ;       assumptions about the state of the 8291 & 9292 upon entry
                        222 ;       to the routine and will exit the routine in an identical state.
                        223 ;
                        224 ;
                        225 ;       8291:   BO is or has been set,
                        226 ;               All interrupts are masked off
                        227 ;               TON mode, not LA
                        228 ;               No holdoffs in effect or enabled
                        229 ;               No holdoffs waiting for finish command
                        230 ;
                        231 ;       8292:   ATN asserted (active controller)
                        232 ;               note: RCTL is an exception--- it expects
                        233 ;               to not be active controller
                        234 ;               Any previous task is complete & 92 is
                        235 ;               ready to receive next command.
                        236 ;       8085:   Pointer registers (DE,HL) end one
                        237 ;               beyond last legal entry
                        238 ;*******************************************************
                        239 ;
                        240 ;
                        241 ;       INITIALIZATION ROUTINE
                        242 ;
                        243 ;INPUTS:        None
                        244 ;OUTPUTS:       None
                        245 ;CALLS:         None
                        246 ;DESTROYS:      A,F
                        247 ;
1000 3EA0               248 INIT:   MVI     A,INTM  ;Enable TCI
1002 D368               249         OUT     INTMR   ;Output to 92's intr. mask reg
1004 3E60               250         MVI     A,DTDL1 ;Disable major talker/listener
1006 D366               251         OUT     ADR01
1008 3EE0               252         MVI     A,DTDL2 ;Disable minor talker/listener
100A D365               253         OUT     ADR01
100C 3E80               254         MVI     A,TON   ;Talk only mode
100E D364               255         OUT     ADRMD
1010 3E23               256         MVI     A,CLKRT ;3 MHZ for delay timer
1012 D365               257         OUT     AUXMD
                        258         CLRA
1014 AF                 259+        XRA     A       ;A XOR A =0
1015 D361               260         OUT     INT1
1017 D362               261         OUT     INT2    ;Disable all 91 mask bits
1019 D365               262         OUT     AUXMD   ;Immediate execute PON
101B C9                 263         RET
                        264 ;
                        265 ;*******************************************************
                        266 ;
                        267 ;
                        268 ;       SEND ROUTINE
                        269 ;
```

231324-33

```
                        270 ;
                        271 ;
                        272 ;       INPUTS:         HL listener list pointer
                        273 ;                       DE data buffer pointer
                        274 ;                       C  count-- 0 will cause no data to be sent
                        275 ;                       b  EOS character-- software detected
                        276 ;       OUTPUTS:        none
                        277 ;       CALLS:          none
                        278 ;       DESTROYS:       A, C, DE, HL, F
                        279 ;
                        280 ;
                        281 ;
101C 3E41               282 SEND:   MVI     A,MTA   ;Send MTA to turn off any
101E D360               283         OUT     DOUT    ;previous talker
                        284         WAITO
1020 DB61               285+??0001:  IN     INT1    ;Get Intl status
1022 E602               286+        ANI     BOM     ;Check for byte out
1024 CA2010             287+        JZ      ??0001  ;If not, try again
1027 3E3F               288         MVI     A,UNL   ;Send universal unlisten
1029 D360               289         OUT     DOUT    ;to stop previous listeners
102B 78                 290         MOV     A,B     ;Get EOS character
102C D367               291         OUT     EOSR    ;Output it to 8291
                        292                         ;while listener.....
                        293 SEND1:  RANGE   20H,3EH,SEND2   ;Check next listen address
                        294+                        ;Checks for value in range
                        295+                        ;branches to label if not
                        296+                        ;in range. Falls through if
                        297+                        ;lower <= ( (H)(L) ) <= upper.
                        298+                        ;Get next byte.
102E 7E                 299+        MOV     A,M
102F FE20               300+        CPI     20H
1031 FA4710             301+        JM      SEND2
1034 FE3F               302+        CPI     3EH+1
1036 F24710             303+        JP      SEND2
                        304         WAITO                   ;Wait for previous listener sent
1039 DB61               305+??0002:  IN     INT1    ;Get Intl status
103B E602               306+        ANI     BOM     ;Check for byte out
103D CA3910             307+        JZ      ??0002  ;If not, try again
1040 7E                 308         MOV     A,M             ;Get this listener
1041 D360               309         OUT     DOUT            ;Output to GPIB
1043 23                 310         INX     H               ;Increment listener list pointer
1044 C32E10             311         JMP     SEND1           ;Loop till non-valid listener
                        312                                 ;Enable 91 ending conditions
                        313 SEND2:  WAITO           ;Wait for lstn addr accepted
1047 DB61               314+??0003:  IN     INT1    ;Get Intl status
1049 E602               315+        ANI     BOM     ;Check for byte out
104B CA4710             316+        JZ      ??0003  ;If not, try again
                        317                         ;WAITO required for early versions
                        318                         ;of 8292 to avoid GTSB before DAC
104E 3EF6               319         MVI     A,GTSB  ;Goto standby
1050 D369               320         OUT     CMD92   ;
1052 3E88               321         MVI     A,AXRA+EOIS     ;Send EOI with EOS character
1054 D365               322         OUT     AUXMD
                        323         WAITX           ;Wait for TCI to go false
1056 DB6F               324+??0004: IN      PRTF
1058 E602               325+        ANI     TCIF
105A C25610             326+        JNZ     ??0004
                        327         WAITT           ;Wait for TCI on GTSB
105D DB5F               328+??0005: IN      PRTF    ;Get task complete int,etc.
105F E602               329+        ANI     TCIF    ;Mask it
1061 CA5D10             330+        JZ      ??0005  ;Wait for task to be complete
                        331
                        332 ;       delete next 3 instructions to make count of 0=256
                        333 ;
1064 79                 334         MOV     A,C     ;Get count
                        335         SETF            ;Set flags
1065 B7                 336+        ORA     A
1066 CA8810             337         JZ      SEND6   ;If count=0, send no data
1069 1A                 338 SEND3:  LDAX    D       ;Get data byte
106A D360               339         OUT     DOUT    ;Output to GPIB
106C B8                 340         CMP     B       ;Test EOS ...this is faster
                        341                         ;and uses less code than using
                        342                         ;91's END or EOI bits
```

231324-34

```
106D CA7F10          343          JZ      SEND5   ;If char = EOS , go finish
                     344 SEND4:   WAITO
1070 DB61            345+??0006:   IN     INT1    ;Get Intl status
1072 E602            346+         ANI     BOM     ;Check for byte out
1074 CA7010          347+         JZ      ??0006  ;If not, try again
1077 13              348          INX     D       ;Increment buffer pointer
1078 0D              349          DCR     C       ;Decrement count
1079 C26910          350          JNZ     SEND3   ;If count < > 0, go send
107C C38810          351          JMP     SEND6   ;Else go finish
107F 13              352 SEND5:   INX     D       ;for consistency
1080 0D              353          DCR     C       ; "       "
                     354          WAITO

                                                  ;This ensures that the standard entry
1081 DB61            355+??0007:   IN     INT1    ;Get Intl status
1083 E602            356+         ANI     BOM     ;Check for byte out
1085 CA8110          357+         JZ      ??0007  ;If not, try again
                     358                          ;assumptions for the next subroutine are met
1088 3EFD            359 SEND6:   MVI     A,TCSY  ;Take control syncronously
108A D369            360          OUT     CMD92
108C 3E80            361          MVI     A,AXRA  ;Reset send EOI on EOS
108E D365            362          OUT     AUXMD
                     363          WAITX           ;Wait for TCI false
1090 DB6F            364+??0008: IN       PRTF
1092 E602            365+         ANI     TCIF
1094 C29010          366+         JNZ     ??0008
                     367          WAITT           ;Wait for TCI
1097 DB6F            368+??0009: IN       PRTF    ;Get task complete int,etc.
1099 E602            369+         ANI     TCIF    ;Mask it
109B CA9710          370+         JZ      ??0009  ;Wait for task to be complete
109E C9              371          RET
                     372 ;*****************************************************************
                     373 ;
                     374 ;        RECEIVE ROUTINE
                     375 ;
                     376 ;
                     377 ;INPUT:           HL talker pointer
                     378 ;                 DE data buffer pointer
                     379 ;                 C  count (max buffer size) 0 implies 256
                     380 ;                 B  EOS character
                     381 ;OUTPUT:          Fills buffer pointed at by DE
                     382 ;CALLS:           None
                     383 ;DESTROYS:        A, BC, DE, HL, F
                     384 ;
                     385 ;RETURNS:         A=0 normal termination--EOS detected
                     386 ;                 A=40 Error--- count overrun
                     387 ;                 A<40 or A>5EH Error--- bad talk address
                     388 ;
                     389 ;
109F 78              390 RECV:    MOV     A,B     ;Get EOS character
10A0 D367            391          OUT     EOSR    ;Output it to 91
                     392          RANGE   40H,5EH,RECV6
                     393+                         ;Checks for value in range
                     394+                         ;branches to label if not
                     395+                         ;in range. Falls through if
                     396+                         ;lower <= ( (H)(L) ) <= upper.
                     397+                         ;Get next byte.
10A2 7E              398+         MOV     A,M
10A3 FE40            399+         CPI     40H
10A5 FA3911          400+         JM      RECV6
10A8 FE5F            401+         CPI     5EH+1
10AA F23911          402+         JP      RECV6
                     403                          ;valid if 40H<= talk <=5EH
10AD D360            404          OUT     DOUT    ;Output talker to GPIB
10AF 23              405          INX     H       ;Incr pointer for consistency
                     406          WAITO
10B0 DB61            407+??0010:   IN     INT1    ;Get Intl status
10B2 E602            408+         ANI     BOM     ;Check for byte out
10B4 CAB010          409+         JZ      ??0010  ;If not, try again
10B7 3E3F            410          MVI     A,UNL   ;Stop other listeners
10B9 D360            411          OUT     DOUT
                     412          WAITO
10BB DB61            413+??0011:   IN     INT1    ;Get Intl status
10BD E602            414+         ANI     BOM     ;Check for byte out
10BF CABB10          415+         JZ      ??0011  ;If not, try again
```

231324-35

```
10C2 3E21          416           MVI     A,MLA    ;For completeness
10C4 D360          417           OUT     DOUT
10C6 3E86          418           MVI     A,AXRA+HOEND+EDEOS       ;End when
10C8 D365          419           OUT     AUXMD    ;EOS or EOI & Holdoff
                   420           WAITO
10CA DB61          421+??0012:    IN     INT1     ;Get Intl status
10CC E602          422+          ANI     BOM      ;Check for byte out
10CE CACA10        423+          JZ      ??0012   ;If not, try again
10D1 3E40          424           MVI     A,LON    ;Listen only
10D3 D364          425           OUT     ADRMD
                   426           CLRA              ;Immediate XEQ PON
10D5 AF            427+          XRA     A        ;A XOR A =0
10D6 D365          428           OUT     AUXMD
10D8 3EF6          429           MVI     A,GTSB   ;Goto standby
10DA D359          430           OUT     CMD92
                   431           WAITX             ;Wait for TCI=0
10DC DB6F          432+??0013:    IN     PRTF
10DE E602          433+          ANI     TCIF
10E0 C2DC10        434+          JNZ     ??0013
                   435           WAITT             ;Wait for TCI=1
10E3 DB6F          436+??0014:    IN     PRTF     ;Get task complete int,etc.
10E5 E602          437+          ANI     TCIF     ;Mask it
10EA DB61          439 RECV1:     IN     INT1     ;Get 91 Int status (END &/or BI)
10EC 47            440           MOV     B,A      ;Save it in B for BI check later
10ED E610          441           ANI     ENDMK    ;Check for EOS or EOI
10EF C20511        442           JNZ     RECV2    ;Yes end--- go wait for BI
10F2 78            443           MOV     A,B      ;NO, retrieve status &
10F3 E601          444           ANI     BIM      ;check for BI
10F5 CAEA10        445           JZ      RECV1    ;NO, go wait for either END or BI
10F8 DB60          446           IN      DIN      ;YES, BI--- get data
10FA 12            447           STAX    D        ;Store it in buffer
10FB 13            448           INX     D        ;Increment buffer pointer
10FC 0D            449           DCR     C        ;Decrement counter
10FD C2EA10        450           JNZ     RECV1    ;If count < > 0 go back & wait
1100 0640          451           MVI     B,40H    ;Else set error indicator
1102 C31711        452           JMP     RECV5    ;And go take control
                   453 ;
1105 78            454 RECV2:     MOV     A,B      ;Retreive status
1106 E601          455 RECV3:     ANI     BIM      ;Check for BI
1108 C21011        456           JNZ     RECV4    ;If BI then go input data
110B DB61          457           IN      INT1     ;Else wait for last BI
110D C30611        458           JMP     RECV3    ;In loop
1110 DB60          459 RECV4:     IN      DIN      ;Get data byte
1112 12            460           STAX    D        ;Store it in buffer
1113 13            461           INX     D        ;Incr data pointer
1114 0D            462           DCR     C        ;Decrement count, but ignore it
1115 0600          463           MVI     B,0      ;Set normal completion indicators
                   464 ;
1117 3EFD          465 RECV5:     MVI     A,TCSY   ;Take control synchronously
1119 D359          466           OUT     CMD92
                   467           WAITX             ;Wait for TCI=0 (7 tcy)
111B DB6F          468+??0015:    IN     PRTF
111D E602          469+          ANI     TCIF
111F C21B11        470+          JNZ     ??0015
                   471           WAITT             ;Wait for TCI=1
1122 DB6F          472+??0016:    IN     PRTF     ;Get task complete int,etc.
1124 E602          473+          ANI     TCIF     ;Mask it
1126 CA2211        474+          JZ      ??0016   ;Wait for task to be complete
                   475 ;
                   476 ;if timeout 3 is to be checked, the above WAITT should
                   477 ;be omitted & the appropriate code to look for TCI or
                   478 ;TOUT3 inserted here.
                   479 ;
1129 3E80          480           MVI     A,AXRA   ;Pattern to clear 91 END conditions
112B D365          481           OUT     AUXMD    ;
112D 3E80          482           MVI     A,TON    ;This bit pattern already in "A"
112F D364          483           OUT     ADRMD    ;Output TON
1131 3E03          484           MVI     A,FNHSK  ;Finish handshake
1133 D365          485           OUT     AUXMD
                   486           CLRA
1135 AF            487+          XRA     A        ;A XOR A =0
1136 D365          488           OUT     AUXMD    ;Immediate execute PON-Reset LON
1138 78            489           MOV     A,B      ;Get completion character
1139 C9            490 RECV6:     RET
```

231324-36

```
                    491 ;
                    492 ;*************************************************************
                    493 ;       XFER ROUTINE
                    494 ;
                    495 ;
                    496 ;INPUTS:          HL device list pointer
                    497 ;                 B  EOS character
                    498 ;OUTPUTS:         None
                    499 ;CALLS:           None
                    500 ;DESTROYS:        A, HL, F
                    501 ;RETURNS:         A=0 normal, A < > 0 bad talker
                    502 ;
                    503 ;
                    504 ;NOTE:            XFER will not work if the talker
                    505 ;                 uses EOI to terminate the transfer.
                    506 ;                 Intel will be making hardware
                    507 ;                 modifications to the 8291 that will
                    508 ;                 correct this problem. Until that time,
                    509 ;                 only EOS may be used without possible
                    510 ;                 loss of the last data byte transfered.
                    511 XFER:    RANGE   40H,5EH,XFER4   ;Check for valid talker
                    512+                                ;Checks for value in range
                    513+                                ;branches to label if not
                    514+                                ;in range. Falls through if
                    515+                                ;lower <= ( (H)(L) ) <= upper.
                    516+                                ;Get next byte.
113A 7E             517+            MOV     A,M
113B FE40           518+            CPI     40H
113D FABB11         519+            JM      XFER4
1140 FE5F           520+            CPI     5EH+1
1142 F2BB11         521+            JP      XFER4
1145 D360           522             OUT     DOUT    ;Send it to GPIB
1147 23             523             INX     H       ;Incr pointer
                    524             WAITO
1148 DB61           525+??0017:      IN     INT1    ;Get Intl status
114A E602           526+            ANI     BOM     ;Check for byte out
114C CA4811         527+            JZ      ??0017  ;If not, try again
114F 3E3F           528             MVI     A,UNL   ;Universal unlisten
1151 D360           529             OUT     DOUT
                    530 XFER1:  RANGE   20H,3EH,XFER2   ;Check for valid listener
                    531+                                ;Checks for value in range
                    532+                                ;branches to label if not
                    533+                                ;in range. Falls through if
                    534+                                ;lower <= ( (H)(L) ) <= upper.
                    535+                                ;Get next byte.
1153 7E             536+            MOV     A,M
1154 FE20           537+            CPI     20H
1156 FA6C11         538+            JM      XFER2
1159 FE3F           539+            CPI     3EH+1
115B F26C11         540+            JP      XFER2
                    541             WAITO
115E DB61           542+??0018:      IN     INT1    ;Get Intl status
1160 E602           543+            ANI     BOM     ;Check for byte out
1162 CA5E11         544+            JZ      ??0018  ;If not, try again
1165 7E             545             MOV     A,M     ;Get listener
1166 D360           546             OUT     DOUT
1168 23             547             INX     H       ;Incr pointer
1169 C35311         548             JMP     XFER1   ;Loop until non-valid listener
                    549 XFER2:  WAITO
116C DB61           550+??0019:      IN     INT1    ;Get Intl status
116E E602           551+            ANI     BOM     ;Check for byte out
1170 CA6C11         552+            JZ      ??0019  ;If not, try again
1173 3E87           553             MVI     A,AXRA+CAHCY+SDEOS      ;Invisible handshake
1175 D365           554             OUT     AUXMD   ;Continuous AH mode
1177 3E40           555             MVI     A,LON   ;Listen only
1179 D364           556             OUT     ADRMD
                    557             CLRA
117B AF             558+            XRA     A       ;A XOR A =0
117C D365           559             OUT     AUXMD   ;Immed. XEQ PON
117E 78             560             MOV     A,B     ;Get EOS
117F D367           561             OUT     EOSR    ;Output it to 91
1181 3EF6           562             MVI     A,GTSB  ;Go to standby
1183 D369           563             OUT     CMD92
```

231324-37

```
                        564             WAITX
1185 DB6F               565+??0020:     IN      PRTF
1187 E602               566+            ANI     TCIF
1189 C28511             567+            JNZ     ??0020
                        568             WAITT           ;Wait for TCS
118C DB6F               569+??0021:     IN      PRTF    ;Get task complete int,etc.
118E E602               570+            ANI     TCIF    ;Mask it
1190 CA8C11             571+            JZ      ??0021  ;Wait for task to be complete
1193 DB61               572 XFER3:      IN      INT1    ;Get END status bit
1195 E610               573             ANI     ENDMK   ;Mask it
1197 CA9311             574             JZ      XFER3
119A 3EFD               575             MVI     A,TCSY  ;Take control syncronously
119C D369               576             OUT     CMD92
                        577             WAITX
119E DB6F               578+??0022:     IN      PRTF
11A0 E602               579+            ANI     TCIF
11A2 C29E11             580+            JNZ     ??0022
                        581             WAITT           ;Wait for TCI
11A5 DB6F               582+??0023:     IN      PRTF    ;Get task complete int,etc.
11A7 E602               583+            ANI     TCIF    ;Mask it
11A9 CAA511             584+            JZ      ??0023  ;Wait for task to be complete
11AC 3E80               585             MVI     A,AXRA  ;Not cont AH or END on EOS
11AE D365               586             OUT     AUXMD
11B0 3E03               587             MVI     A,FNHSK ;Finish handshake
11B2 D365               588             OUT     AUXMD
11B4 3E80               589             MVI     A,TON   ;Talk only
11B6 D364               590             OUT     ADRMD
                        591             CLRA            ;Normal return A=0
11B8 AF                 592+            XRA     A       ;A XOR A =0
11B9 D365               593             OUT     AUXMD   ;Immediate XEQ PON
11BB C9                 594 XFER4:      RET
                        595 ;
                        596 ;*************************************************
                        597 ;
                        598 ;
                        599 ;           TRIGGER ROUTINE
                        600 ;
                        601 ;
                        602 ;INPUTS:            HL listener list pointer
                        603 ;OUTPUTS:           None
                        604 ;CALLS:             None
                        605 ;DESTROYS:          A, HL, F
                        606 ;
                        607 ;
11BC 3E3F               608 TRIG:       MVI     A,UNL   ;
11BE D360               609             OUT     DOUT    ;Send universal unlisten
                        610 TRIG1:      RANGE   20H,3EH,TRIG2   ;Check for valid listen
                        611+                                    ;Checks for value in range
                        612+                                    ;branches to label if not
                        613+                                    ;in range. Falls through if
                        614+                                    ;lower <= ( (H)(L) ) <= upper.
                        615+                                    ;Get next byte.
11C0 7E                 616+            MOV     A,M
11C1 FE20               617+            CPI     20H
11C3 FAD911             618+            JM      TRIG2
11C6 FE3F               619+            CPI     3EH+1
11C8 F2D911             620+            JP      TRIG2
                        621             WAITO           ;Wait for UNL to finish
11CB DB61               622+??0024:     IN      INT1    ;Get Intl status
11CD E602               623+            ANI     BOM     ;Check for byte out
11CF CACB11             624+            JZ      ??0024  ;If not, try again
11D2 7E                 625             MOV     A,M     ;Get listener
11D3 D360               626             OUT     DOUT    ;Send Listener to GPIB
11D5 23                 627             INX     H       ;Incr. pointer
11D6 C3C011             628             JMP     TRIG1   ;Loop until non-valid char
                        629 TRIG2:      WAITO           ;Wait for last listen to finish
11D9 DB61               630+??0025:     IN      INT1    ;Get Intl status
11DB E602               631+            ANI     BOM     ;Check for byte out
11DD CAD911             632+            JZ      ??0025  ;If not, try again
11E0 3E08               633             MVI     A,GET   ;Send group execute trigger
11E2 D360               634             OUT     DOUT    ;to all addressed listeners
                        635             WAITO
11E4 DB61               636+??0026:     IN      INT1    ;Get Intl status
11E6 E602               637+            ANI     BOM     ;Check for byte out
```

231324-38

```
11E8 CAE411          638+          JZ      ??0026  ;If not, try again
11EB C9              639           RET
                     640 ;
                     641 ;*****************************************
                     642 ;
                     643 ;DEVICE CLEAR ROUTINE
                     644 ;
                     645 ;
                     646 ;
                     647 ;INPUTS:          HL listener pointer
                     648 ;OUTPUT:          None
                     649 ;CALLS:           None
                     650 ;DESTROYS:        A, HL, F
                     651 ;
11EC 3E3F            652 DCLR:     MVI     A,UNL
11EE D360            653           OUT     DOUT
                     654 DCLR1:    RANGE   20H,3EH,DCLR2
                     655+                          ;Checks for value in range
                     656+                          ;branches to label if not
                     657+                          ;in range. Falls through if
                     658+                          ;lower <= ( (H)(L) ) <= upper.
                     659+                          ;Get next byte.
11F0 7E              660+          MOV     A,M
11F1 FE20            661+          CPI     20H
11F3 FA0912          662+          JM      DCLR2
11F6 FE3F            663+          CPI     3EH+1
11F8 F20912          664+          JP      DCLR2
                     665           WAITO
11FB DB61            666+??0027:    IN     INT1    ;Get Intl status
11FD E602            667+          ANI     BOM     ;Check for byte out
11FF CAF811          668+          JZ      ??0027  ;If not, try again
1202 7E              669           MOV     A,M
1203 D360            670           OUT     DOUT    ;Send listener to GPIB
1205 23              671           INX     H
1206 C3F011          672           JMP     DCLR1
                     673 DCLR2:    WAITO
1209 DB61            674+??0028:    IN     INT1    ;Get Intl status
120B E602            675+          ANI     BOM     ;Check for byte out
120D CA0912          676+          JZ      ??0028  ;If not, try again
1210 3E04            677           MVI     A,SDC   ;Send device clear
1212 D360            678           OUT     DOUT    ;To all addressed listeners
                     679           WAITO
1214 DB61            680+??0029:    IN     INT1    ;Get Intl status
1216 E602            681+          ANI     BOM     ;Check for byte out
1218 CA1412          682+          JZ      ??0029  ;If not, try again
121B C9              683           RET
                     684 ;
                     685 ;*****************************************************
                     686 ;
                     687 ;         SERIAL POLL ROUTINE
                     688 ;
                     689 ;INPUTS:          HL talker list pointer
                     690 ;                 DE status buffer pointer
                     691 ;OUTPUTS:         Fills buffer pointed to by DE
                     692 ;CALLS:           None
                     693 ;DESTROYS:        A, BC, DE, HL, F
                     694 ;
121C 3E3F            695 SPOL:     MVI     A,UNL   ;Universal unlisten
121E D360            696           OUT     DOUT
                     697           WAITO
1220 DB61            698+??0030:    IN     INT1    ;Get Intl status
1222 E602            699+          ANI     BOM     ;Check for byte out
1224 CA2012          700+          JZ      ??0030  ;If not, try again
1227 3E21            701           MVI     A,MLA   ;My listen address
1229 D360            702           OUT     DOUT
                     703           WAITO
122B DB61            704+??0031:    IN     INT1    ;Get Intl status
122D E602            705+          ANI     BOM     ;Check for byte out
122F CA2B12          706+          JZ      ??0031  ;If not, try again
1232 3E18            707           MVI     A,SPE   ;Serial poll enable
1234 D360            708           OUT     DOUT    ;To be formal about it
                     709           WAITO
1236 DB61            710+??0032:    IN     INT1    ;Get Intl status
```

231324-39

```
1238 E602          711+          ANI     BOM      ;Check for byte out
123A CA3612        712+          JZ      ??0032   ;If not, try again
                   713 SPOL1:    RANGE   40H,5EH,SPOL2   ;Check for valid talker
                   714+                          ;Checks for value in range
                   715+                          ;branches to label if not
                   716+                          ;in range. Falls through if
                   717+                          ;lower <= ( (H)(L) ) <= upper.
                   718+                          ;Get next byte.
123D 7E            719+          MOV     A,M
123E FE40          720+          CPI     40H
1240 FA9412        721+          JM      SPOL2
1243 FE5F          722+          CPI     5EH+1
1245 F29412        723+          JP      SPOL2
1248 7E            724           MOV     A,M      ;Get talker
1249 D360          725           OUT     DOUT     ;Send to GPIB
124B 23            726           INX     H        ;Incr talker list pointer
124C 3E40          727           MVI     A,LON    ;Listen only
124E D364          728           OUT     ADRMD
                   729           WAITO            ;Wait for talk address to complete
1250 DB61          730+??0033:    IN     INT1     ;Get Int1 status
1252 E602          731+          ANI     BOM      ;Check for byte out
1254 CA5012        732+          JZ      ??0033   ;If not, try again
                   733           CLRA             ;Pattern for immediate XEQ PON
1257 AF            734+          XRA     A        ;A XOR A =0
1258 D365          735           OUT     AUXMD
125A 3EF6          736           MVI     A,GTSB   ;Goto standby
125C D369          737           OUT     CMD92
                   738           WAITX            ;Wait for TCI false
125E DB6F          739+??0034:   IN      PRTF
1260 E602          740+          ANI     TCIF
1262 C25E12        741+          JNZ     ??0034
                   742           WAITT            ;Wait for TCI
1265 DB6F          743+??0035:   IN      PRTF     ;Get task complete int,etc.
1267 E602          744+          ANI     TCIF     ;Mask it
1269 CA6512        745+          JZ      ??0035   ;Wait for task to be complete
                   746           WAITI            ;Wait for status byte input
126C DB61          747+??0036:   IN      INT1     ;Get INT1 status
126E 47            748+          MOV     B,A      ;Save status in B
126F E601          749+          ANI     BIM      ;Check for byte in
1271 CA6C12        750+          JZ      ??0036   ;If not, just try again
1274 3EFD          751           MVI     A,TCSY   ;Take control sync
1276 D369          752           OUT     CMD92
                   753           WAITX            ;Wait for TCI false
1278 DB6F          754+??0037:   IN      PRTF
127A E602          755+          ANI     TCIF
127C C27812        756+          JNZ     ??0037
                   757           WAITT            ;Wait for TCI
127F DB6F          758+??0038:   IN      PRTF     ;Get task complete int,etc.
1281 E602          759+          ANI     TCIF     ;Mask it
1283 CA7F12        760+          JZ      ??0038   ;Wait for task to be complete
1286 DB60          761           IN      DIN      ;Get serial poll status byte
1288 12            762           STAX    D        ;Store it in buffer
1289 13            763           INX     D        ;Incr pointer
128A 3E80          764           MVI     A,TON    ;Talk only for controller
128C D364          765           OUT     ADRMD    ;
                   766           CLRA
128E AF            767+          XRA     A        ;A XOR A =0
128F D365          768           OUT     AUXMD    ;Immeditate XEQ PON
                   769                            ;CLR LA
1291 C33D12        770           JMP     SPOL1    ;Go on to next device on list
                   771 ;
1294 3E19          772 SPOL2:    MVI     A,SPD    ;Serial poll disable
1296 D360          773           OUT     DOUT     ;We know BO was set (WAITO above)
                   774           WAITO
1298 DB61          775+??0039:    IN     INT1     ;Get Int1 status
129A E602          776+          ANI     BOM      ;Check for byte out
129C CA9812        777+          JZ      ??0039   ;If not, try again
                   778           CLRA
129F AF            779+          XRA     A        ;A XOR A =0
12A0 D365          780           OUT     AUXMD    ;Immediate XEQ PON to clear LA
12A2 C9            781           RET
                   782 ;
                   783 ;*************************************************
                   784 ;
```

231324-40

```
                        785 ;       PARALLEL POLL ENABLE ROUTINE
                        786 ;
                        787 ;INPUTS:         HL listener list pointer
                        788 ;                DE configuration byte pointer
                        789 ;OUTPUTS:        None
                        790 ;CALLS:          None
                        791 ;DESTROYS:       A, DE, HL, F
                        792 ;
                        793 ;
12A3 3E3F               794 PPEN:   MVI     A,UNL   ;Universal unlisten
12A5 D360               795         OUT     DOUT
                        796 PPEN1:  RANGE   20H,3EH,PPEN2   ;Check for valid listener
                        797+                        ;Checks for value in range
                        798+                        ;branches to label if not
                        799+                        ;in range. Falls through if
                        800+                        ;lower <= ( (H)(L) ) <= upper.
                        801+                        ;Get next byte.
12A7 7E                 802+        MOV     A,M
12A8 FE20               803+        CPI     20H
12AA FAD812             804+        JM      PPEN2
12AD FE3F               805+        CPI     3EH+1
12AF F2D812             806+        JP      PPEN2
                        807         WAITO           ;Valid wait 91 data out req
12B2 DB61               808+??0040:  IN     INT1    ;Get Intl status
12B4 E602               809+        ANI     BOM     ;Check for byte out
12B6 CAB212             810+        JZ      ??0040  ;If not, try again
12B9 7E                 811         MOV     A,M     ;Get listener
12BA D360               812         OUT     DOUT
                        813         WAITO
12BC DB61               814+??0041:  IN     INT1    ;Get Intl status
12BE E602               815+        ANI     BOM     ;Check for byte out
12C0 CABC12             816+        JZ      ??0041  ;If not, try again
12C3 3E05               817         MVI     A,PPC   ;Parallel poll configure
12C5 D360               818         OUT     DOUT
                        819         WAITO
12C7 DB61               820+??0042:  IN     INT1    ;Get Intl status
12C9 E602               821+        ANI     BOM     ;Check for byte out
12CB CAC712             822+        JZ      ??0042  ;If not, try again
12CE 1A                 823         LDAX    D       ;Get matching configuration byte
12CF F660               824         ORI     PPE     ;Merge with parallel poll enable
12D1 D360               825         OUT     DOUT
12D3 23                 826         INX     H       ;Incr pointers
12D4 13                 827         INX     D
12D5 C3A712             828         JMP     PPEN1   ;Loop until invalid listener char
                        829 PPEN2:  WAITO
12D8 DB61               830+??0043:  IN     INT1    ;Get Intl status
12DA E602               831+        ANI     BOM     ;Check for byte out
12DC CAD812             832+        JZ      ??0043  ;If not, try again
12DF C9                 833         RET
                        834 ;
                        835 ;PARALLEL POLL DISABLE ROUTINE
                        836 ;
                        837 ;INPUTS:         HL listener list pointer
                        838 ;OUTPUTS:        None
                        839 ;CALLS:          None
                        840 ;DESTROYS:       A, HL, F
                        841 ;
12E0 3E3F               842 PPDS:   MVI     A,UNL   ;Universal unlisten
12E2 D360               843         OUT     DOUT
                        844 PPDS1:  RANGE   20H,3EH,PPDS2   ;Check for valid listener
                        845+                        ;Checks for value in range
                        846+                        ;branches to label if not
                        847+                        ;in range. Falls through if
                        848+                        ;lower <= ( (H)(L) ) <= upper.
                        849+                        ;Get next byte.
12E4 7E                 850+        MOV     A,M
12E5 FE20               851+        CPI     20H
12E7 FAFD12             852+        JM      PPDS2
12EA FE3F               853+        CPI     3EH+1
12EC F2FD12             854+        JP      PPDS2
                        855         WAITO
12EF DB61               856+??0044:  IN     INT1    ;Get Intl status
12F1 E602               857+        ANI     BOM     ;Check for byte out
12F3 CAEF12             858+        JZ      ??0044  ;If not, try again
```

231324-41

```
12F6 7E          859          MOV     A,M     ;Get listener
12F7 D360        860          OUT     DOUT
12F9 23          861          INX     H       ;Incr pointer
12FA C3E412      862          JMP     PPDS1   ;Loop until invalid listener
                 863 PPDS2:   WAITO
12FD DB61        864+??0045:   IN     INT1    ;Get Intl status
12FF E602        865+          ANI    BOM     ;Check for byte out
1301 CAFD12      866+          JZ     ??0045  ;If not, try again
1304 3E05        867           MVI    A,PPC   ;Parallel poll configure
1306 D360        868           OUT    DOUT
                 869           WAITO
1308 DB61        870+??0046:   IN     INT1    ;Get Intl status
130A E602        871+          ANI    BOM     ;Check for byte out
130C CA0813      872+          JZ     ??0046  ;If not, try again
130F 3E70        873           MVI    A,PPD   ;Parallel poll disable
1311 D360        874           OUT    DOUT
                 875           WAITO
1313 DB61        876+??0047:   IN     INT1    ;Get Intl status
1315 E602        877+          ANI    BOM     ;Check for byte out
1317 CA1313      878+          JZ     ??0047  ;If not, try again
131A C9          879           RET
                 880 ;
                 881 ;         PARALLEL POLL UNCONFIGURE ALL ROUTINE
                 882 ;
                 883 ;
                 884 ;INPUTS:          None
                 885 ;OUTPUTS:         None
                 886 ;CALLS:           None
                 887 ;DESTROYS:        A, F
                 888 ;
131B 3E15        889 PPUN:    MVI     A,PPU   ;Parallel poll unconfigure
131D D360        890          OUT     DOUT
                 891          WAITO
131F DB61        892+??0048:   IN     INT1    ;Get Intl status
1321 E602        893+          ANI    BOM     ;Check for byte out
1323 CA1F13      894+          JZ     ??0048  ;If not, try again
1326 C9          895           RET
                 896 ;
                 897 ;************************************************
                 898 ;
                 899 ;CONDUCT A PARALLEL POLL
                 900 ;
                 901 ;
                 902 ;INPUTS:          None
                 903 ;OUTPUTS:         None
                 904 ;CALLS:           None
                 905 ;DESTROYS:        A, B, F
                 906 ;RETURNS:         A= parallel poll status byte
                 907 ;
1327 3E40        908 PPOL:    MVI     A,LON   ;Listen only
1329 D364        909          OUT     ADRMD
                 910          CLRA            ;Immediate XEQ PON
132B AF          911+         XRA     A       ;A XOR A =0
132C D365        912          OUT     AUXMD   ;Reset TON
132E 3EF5        913          MVI     A,EXPP  ;Execute parallel poll
1330 D369        914          OUT     CMD92
                 915          WAITI           ;Wait for completion= BI on 91
1332 DB61        916+??0049: IN       INT1    ;Get INTl status
1334 47          917+         MOV     B,A     ;Save status in B
1335 E601        918+         ANI     BIM     ;Check for byte in
1337 CA3213      919+         JZ      ??0049  ;If not, just try again
133A 3E80        920          MVI     A,TON   ;Talk only
133C D364        921          OUT     ADRMD
                 922          CLRA            ;Immediate XEQ PON
133E AF          923+         XRA     A       ;A XOR A =0
133F D365        924          OUT     AUXMD   ;Reset LON
1341 DB60        925          IN      DIN     ;Get PP byte
1343 C9          926          RET
                 927 ;
                 928 ;********************************************
                 929 ;PASS CONTROL ROUTINE
                 930 ;
                 931 ;INPUTS:          HL pointer to talker
                 932 ;OUTPUTS:         None
```

231324-42

```
                        933 ;CALLS:          None
                        934 ;DESTROYS:       A, HL, F
                        935 PCTL:    RANGE   40H,5EH,PCTL1   ;Is it a valid talker ?
                        936+                                 ;Checks for value in range
                        937+                                 ;branches to label if not
                        938+                                 ;in range. Falls through if
                        939+                                 ;lower <= ( (H)(L) ) <= upper.
                        940+                                 ;Get next byte.
1344 7E                 941+         MOV     A,M
1345 FE40               942+         CPI     40H
1347 FA8A13             943+         JM      PCTL1
134A FE5F               944+         CPI     5EH+1
134C F28A13             945+         JP      PCTL1
134F FE41               946          CPI     MTA      ;Is it my talker address
1351 CA8A13             947          JZ      PCTL1    ;Yes, just return
1354 D360               948          OUT     DOUT     ;Send on GPIB
                        949          WAITO
1356 DB61               950+??0050:   IN      INT1     ;Get Intl status
1358 E602               951+         ANI     BOM      ;Check for byte out
135A CA5613             952+         JZ      ??0050   ;If not, try again
135D 3E09               953          MVI     A,TCT    ;Take control message
135F D360               954          OUT     DOUT
                        955          WAITO
1361 DB61               956+??0051:   IN      INT1     ;Get Intl status
1363 E602               957+         ANI     BOM      ;Check for byte out
1365 CA6113             958+         JZ      ??0051   ;If not, try again
1368 3E01               959          MVI     A,MODE1  ;Not talk only or listen only
136A D364               960          OUT     ADRMD    ;Enable 91 address mode 1
                        961          CLRA
136C AF                 962+         XRA     A        ;A XOR A =0
136D D365               963          OUT     AUXMD    ;Immediate XEQ PON
136F 3E01               964          MVI     A,MDA    ;My device address
1371 D366               965          OUT     ADR01    ;enabled to talk and listen
1373 3EA1               966          MVI     A,AXRB+CPTEN     ;Command pass thru enable
1375 D365               967          OUT     AUXMD
                        968 ;********optional PP configuration goes here********
1377 3EF1               969          MVI     A,GIDL   ;92 go idle command
1379 D369               970          OUT     CMD92
                        971          WAITX
137B DB6F               972+??0052: IN       PRTF
137D E602               973+         ANI     TCIF
137F C27B13             974+         JNZ     ??0052
                        975          WAITT            ;Wait for TCI
1382 DB6F               976+??0053: IN       PRTF     ;Get task complete int,etc.
1384 E602               977+         ANI     TCIF     ;Mask it
1386 CA8213             978+         JZ      ??0053   ;Wait for task to be complete
1389 23                 979          INX     H
138A C9                 980 PCTL1:   RET
                        981 ;
                        982 ;
                        983 ;*****************************************
                        984 ;
                        985 ;RECEIVE CONTROL ROUTINE
                        986 ;
                        987 ;INPUTS:          None
                        988 ;OUTPUTS:         None
                        989 ;CALLS:           None
                        990 ;DESTROYS:        A, F
                        991 ;RETURNS:         0= invalid (not take control to us or CPT bit not on)
                        992 ;                 < > 0 = valid take control-- 92 will now be in control
                        993 ;NOTE:            THIS CODE MUST BE TIGHTLY INTEGRATED INTO ANY USER
                        994 ;                 SOFTWARE THAT FUNCTIONS WITH THE 8291 AS  A DEVICE.
                        995 ;                 NORMALLY SOME ADVANCE WARNING OF IMPENDING PASS
                        996 ;                 CONTROL SHOULD BE GIVEN TO US BY THE CONTROLLER
                        997 ;                 WITH OTHER USEFUL INFO. THIS PROTOCOL IS SITUATION
                        998 ;                 SPECIFIC AND WILL NOT BE COVERED HERE.
                        999 ;
                       1000 ;
138B DB61              1001 RCTL:    IN      INT1     ;Get INT1 req (i.e. CPT etc.)
138D E680              1002          ANI     CPT      ;Is command pass thru on ?
138F CACF13            1003          JZ      RCTL2    ;No, invalid-- go return
1392 DB65              1004          IN      CPTRG    ;Get command
1394 FE09              1005          CPI     TCT      ;Is it take control ?
```

231324-43

```
13A6 C2CA13       1005          JNZ     RCTL1   ;No, go return invalid
13A9 DB64         1006          IN      ADRST   ;Get address status
13AB E602         1007          ANI     TA      ;Is TA on ?
13AD CACA13       1008          JZ      RCTL1   ;No -- go return invalid
13A0 3E60         1009          MVI     A,DTDL1 ;Disable talker listener
13A2 D365         1010          OUT     ADR01
13A4 3E80         1011          MVI     A,TON   ;Talk only
13A6 D364         1012          OUT     ADRMD
                  1013          CLRA
13A8 AF           1014+         XRA     A       ;A XOR A =0
13A9 D361         1015          OUT     INT1    ;Mask off INT bits
13AB D362         1016          OUT     INT2
13AD D365         1017          OUT     AUXMD
13AF 3EFA         1018          MVI     A,TCNTR ;Take (receive) control 92 command
13B1 D369         1019          OUT     CMD92
13B3 3E0F         1020          MVI     A,VSCMD ;Valid command pattern for 91
13B5 D365         1021          OUT     AUXMD
                  1022 ;******** optional TOUT1 check could be put here ********
                  1023          WAITX
13B7 DB6F         1024+??0054:  IN      PRTF
13B9 E602         1025+         ANI     TCIF
13BB C2B713       1026+         JNZ     ??0054
                  1027          WAITT           ;Wait for TCI
13BE DB6F         1028+??0055:  IN      PRTF    ;Get task complete int,etc.
13C0 E602         1029+         ANI     TCIF    ;Mask it
13C2 CABE13       1030+         JZ      ??0055  ;Wait for task to be complete
13C5 3E09         1031          MVI     A,TCT   ;Valid return pattern
13C7 C3CF13       1032          JMP     RCTL2   ;Only one return per routine
13CA 3E0F         1033 RCTL1:   MVI     A,VSCMD ;Acknowledge CPT
13CC D365         1034          OUT     AUXMD
                  1035          CLRA            ;Error return pattern
13CE AF           1036+         XRA     A       ;A XOR A =0
13CF C9           1037 RCTL2:   RET
                  1038 ;
                  1039 ;*************************************************
                  1040 ;
                  1041 ;       SRQ ROUTINE
                  1042 ;
                  1043 ;INPUTS:        None
                  1044 ;OUTPUTS:       None
                  1045 ;CALLS:         None
                  1046 ;RETURNS:       A= 0 no SRQ
                  1047 ;                A < > 0 SRQ occured
                  1048 ;
                  1049 ;
13D0 DB69         1050 SRQD:    IN      INTST   ;Get 92's INTRQ status
13D2 E620         1051          ANI     SRQBT   ;Mask off SRQ
13D4 CAE213       1052          JZ      SRQD2   ;Not set--- go return
13D7 F60B         1053          ORI     IACK    ;Set--- must clear it with IACK
13D9 D369         1054          OUT     CMD92
13DB DB69         1055 SRQD1:   IN      INTST   ;Get IBF
13DD E602         1056          ANI     IBFBT   ;Mask it
13DF CADB13       1057          JZ      SRQD1   ;Wait if not set
13E2 C9           1058 SRQD2:   RET
                  1059 ;
                  1060 ;*************************************************
                  1061 ;
                  1062 ;REMOTE ENABLE ROUTINE
                  1063 ;
                  1064 ;INPUTS:        None
                  1065 ;OUTPUTS:       None
                  1066 ;CALLS:         NONE
                  1067 ;DESTROYS:      A, F
                  1068 ;
13E3 3EF8         1069 REME:    MVI     A,SREM
13E5 D369         1070          OUT     CMD92   ;92 asserts remote enable
                  1071          WAITX           ;Wait for TCI = 0
13E7 DB6F         1072+??0056:  IN      PRTF
13E9 E602         1073+         ANI     TCIF
13EB C2E713       1074+         JNZ     ??0056
                  1075          WAITT           ;Wait for TCI
13EE DB6F         1076+??0057:  IN      PRTF    ;Get task complete int,etc.
13F0 E602         1077+         ANI     TCIF    ;Mask it
13F2 CAEE13       1078+         JZ      ??0057  ;Wait for task to be complete
```

231324-44

```
13F5 C9          1080          RET
                 1081 ;
                 1082 ;*****************************************
                 1083 ;
                 1084 ;LOCAL ROUTINE
                 1085 ;
                 1086 ;
                 1087 ;INPUTS:          None
                 1088 ;OUTPUTS:         None
                 1089 ;CALLS:           None
                 1090 ;DESTROYS:        A, F
                 1091 ;
13F6 3EF7        1092 LOCL:    MVI      A,SLOC
13F8 D369        1093          OUT      CMD92    ;92 stops asserting remote enable
                 1094          WAITX              ;Wait for TCI =0
13FA DB6F        1095+??0058: IN        PRTF
13FC E602        1096+         ANI      TCIF
13FE C2FA13      1097+         JNZ      ??0058
                 1098          WAITT              ;Wait for TCI
1401 DB6F        1099+??0059: IN        PRTF     ;Get task complete int,etc.
1403 E602        1100+         ANI      TCIF     ;Mask it
1405 CA0114      1101+         JZ       ??0059   ;Wait for task to be complete
1408 C9          1102          RET
                 1103 ;
                 1104 ;*********************************************
                 1105 ;
                 1106 ;INTERFACE CLEAR / ABORT ROUTINE
                 1107 ;
                 1108 ;
                 1109 ;INPUTS:                 None
                 1110 ;OUTPUTS:                None
                 1111 ;CALLS:                 None
                 1112 ;DESTROYS:              A, F
                 1113 ;
                 1114 ;
1409 3EF9        1115 IFCL:    MVI      A,ABORT
140B D369        1116          OUT      CMD92    ;Send IFC
                 1117          WAITX              ;Wait for TCI =0
140D DB6F        1118+??0060: IN        PRTF
140F E602        1119+         ANI      TCIF
1411 C20D14      1120+         JNZ      ??0060
                 1121          WAITT              ;Wait for TCI
1414 DB6F        1122+??0061: IN        PRTF     ;Get task complete int,etc.
1416 E602        1123+         ANI      TCIF     ;Mask it
1418 CA1414      1124+         JZ       ??0061   ;Wait for task to be complete
                 1125 ;Delete both WAITX & WAITT if this routine
                 1126 ;is to be called while the 9292 is
                 1127 ;Controller-in-Charge. If not C.I.C. then
                 1128 ;TCI is set, else nothing is set (IFC is sent)
                 1129 ;and the WAIT'S will hang forever
141B C9          1130          RET
                 1132 ;
```

231324-45

```
                    1133 ;APPLICATION EXAMPLE CODE FOR 8085
                    1134 ;
0032                1135 FGDNL   EQU     '2'     ;Func gen device num "2" ASCII,lstn
0031                1136 FCDNL   EQU     '1'     ;Freq ctr device num "1" ASCII,lstn
0051                1137 FCDNT   EQU     'Q'     ;Freq ctr talk address
000D                1138 CR      EQU     0DH     ;ASCII carriage return
000A                1139 LF      EQU     0AH     ;ASCII line feed
00FF                1140 LEND    EQU     0FFH    ;List end for Talk/Listen lists
0040                1141 SRQM    EQU     40H     ;Bit indicating device sent SRQ
                    1142 ;
141C 46553146       1143 FGDATA: DB      'FU1FR37KHAM2V0',CR     ;Data to set up func. gen
1420 5233374B
1424 48414D32
1428 564F
142A 0D
000F                1144 LIM1    EQU     15                      ;Buffer length
1428 50463447       1145 FCDATA: DB      'PF4G7T'                ;Data to set up freq ctr
142F 3754
0006                1146 LIM2    EQU     6                       ;Buffer length
1431 31             1147 LL1:    DB      FCDNL,LEND              ;Listen list for freq ctr
1432 FF
1433 32             1148 LL2:    DB      FGDNL,LEND              ;Listen list for func. gen
1434 FF
1435 51             1149 TL1:    DB      FCDNT,LEND              ;Talk list for freq ctr
1436 FF
                    1150 ;
                    1151 ;SETUP FUNCTION GENERATOR
1437 060D           1152         MVI     B,CR    ;EOS
1439 0E0F           1153         MVI     C,LIM1  ;Count
143B 111C14         1154         LXI     D,FGDATA        ;Data pointer
143E 213314         1155         LXI     H,LL2   ;Listen list pointer
1441 CD1C10         1156         CALL    SEND
                    1157 ;
                    1158 ;SETUP FREQ COUNTER
                    1159 ;
1444 0654           1160         MVI     B,'T'   ;EOS
1446 0E06           1161         MVI     C,LIM2  ;Count
1448 112814         1162         LXI     D,FCDATA        ;Data pointer
144B 213114         1163         LXI     H,LL1   ;Listen list pointer
144E CD1C10         1164         CALL    SEND
                    1165 ;
                    1166 ;WAIT FOR SRQ FROM FREQ CTR
                    1167 ;
1451 CDD013         1168 LOOP:   CALL    SRQD    ;Has SRQ occurred ?
1454 CA5114         1169         JZ      LOOP    ;No, wait for it
                    1170 ;
                    1171 ;SERIAL POLL TO CLEAR SRQ
                    1172 ;
1457 11003C         1173         LXI     D,SPBYTE        ;Buffer pointer
145A 213514         1174         LXI     H,TL1           ;Talk list pointer
145D CD1C12         1175         CALL    SPOL
1460 1B             1176         DCX     D       ;Backup buffer pointer to ctr byte
1461 1A             1177         LDAX    D       ;Get status byte
1462 E640           1178         ANI     SRQM    ;Did ctr assert SRQ ?
1464 CA7714         1179         JZ      ERROR   ;Ctr should have said yes
                    1180 ;
                    1181 ;RECEIVE READING FROM COUNTER
                    1182 ;
1467 060A           1183         MVI     B,LF    ;EOS
1469 0E11           1184         MVI     C,LIM3  ;Count
146B 213514         1185         LXI     H,TL1   ;Talk list pointer
146E 11013C         1186         LXI     D,FCDATI        ;Data in buffer pointer
1471 CD9F10         1187         CALL    RECV
1474 C27714         1188         JNZ     ERROR
                    1189 ;
                    1190 ;******* rest of user processing goes here *****
                    1191 ;
                    1192 ;
1477 00             1193 ERROR:  NOP             ;User dependant error handling
                    1194 ;       ETC.
3C00                1195 ORG     3C00H
3C00                1196 SPBYTE: DS      1       ;Location for serial poll byte
0011                1197 LIM3    EQU     17      ;Max freq counter input
```

231324-46

```
3C01                1198 FCDATI: DS      LIM3    ;Freq ctr input buffer
                    1199         END

PUBLIC SYMBOLS


EXTERNAL SYMBOLS


USER SYMBOLS
ABORT  A 00F9   ADR01  A 0056   ADRMD  A 0054   ADRST  A 0054   AUXMD  A 0055   AXRA   A 0080   AXRB   A 00A0
BIM    A 0001   BOF    A 0001   BOM    A 0002   BUSST  A 0058   CAHCY  A 0003   CLKRT  A 0023   CLRA   + 0007
CLRST  A 0068   CMD92  A 0069   CPT    A 0088   CPTEN  A 0001   CPTRG  A 0055   CR     A 0000   DCL    A 0014
DCLR   A 11EC   DCLR1  A 11F0   DCLR2  A 1209   DIN    A 0060   DOUT   A 0060   DTDL1  A 0050   DTDL2  A 00E0
EDEOS  A 0004   ENDMK  A 0010   EOIS   A 0008   EOIST  A 0020   EOSR   A 0007   ERFLG  A 0068   ERRM   A 0068
ERROR  A 1477   EVBIT  A 0010   EVCST  A 0068   EVREG  A 0068   EXPP   A 00F5   FCDATA A 1428   FCDATI A 3C01
FCONL  A 0031   FCONT  A 0051   FGDATA A 141C   FGONL  A 0032   FNHSK  A 0003   GET    A 0008   GIDL   A 00F1
GSEC   A 00F4   GTSB   A 00F6   HOEND  A 0002   HOHSK  A 0001   IACK   A 003B   IBFBT  A 0002   IBFF   A 0010
IFCL   A 1409   INIT   A 1000   INT1   A 0061   INT2   A 0062   INTM   A 00A0   INTM1  A 0061   INTMR  A 0068
INTST  A 0069   LA     A 0001   LEND   A 00FF   LF     A 000A   LIM1   A 000F   LIM2   A 0005   LIM3   A 0011
LL1    A 1431   LL2    A 1433   LOCL   A 13F6   LON    A 0040   LOOP   A 1451   MDA    A 0001   MLA    A 0021
MODE1  A 0001   MTA    A 0041   NVCMD  A 0007   OBFF   A 0008   PCTL   A 1344   PCTL1  A 138A   PPC    A 0005
PPD    A 0070   PPDS   A 1280   PPDS1  A 12B4   PPDS2  A 12FD   PPE    A 0060   PPEN   A 12A3   PPEN1  A 12A7
PPEN2  A 12D8   PPOL   A 1327   PPU    A 0015   PPUN   A 131B   PRT91  A 0050   PRT92  A 0068   PRTF   A 006F
RANGE  + 0005   RBST   A 00E7   RCST   A 00B6   RCTL   A 1398   RCTL1  A 13CA   RCTL2  A 13CF   RECV   A 109F
RECV1  A 10EA   RECV2  A 1105   RECV3  A 1106   RECV4  A 1110   RECV5  A 1117   RECV6  A 1139   RFME   A 13E3
RERF   A 00E4   RERM   A 00EA   REVC   A 00E3   RINM   A 00E5   RSET   A 00F2   RSTI   A 00F3   RTOUT  A 0089
SDEOI  A 0006   SEND   A 101C   SEND1  A 102E   SEND2  A 1047   SEND3  A 1059   SEND4  A 1070   SEND5  A 107F
SEND6  A 1088   SETF   + 0008   SLOC   A 00F7   SPBYTE A 3C00   SPCNI  A 00F0   SPD    A 0019   SPE    A 0018
SPIF   A 0004   SPOL   A 121C   SPOL1  A 123D   SPOL2  A 1294   SREM   A 00F8   SRQBT  A 0020   SRQD   A 13D0
SRQD1  A 13DB   SRQD2  A 13E2   SRQM   A 0040   STCNI  A 00FE   TA     A 0002   TCASY  A 00FC   TCIF   A 0002
TCNTR  A 00FA   TCSY   A 00FD   TCT    A 0009   TL1    A 1435   TLON   A 00C0   TON    A 0090   TOREG  A 0068
TOST   A 0058   TOUT1  A 0001   TOUT2  A 0002   TOUT3  A 0004   TRIG   A 11BC   TRIG1  A 11C0   TRIG2  A 11D9
UNL    A 003F   VSCMD  A 000F   WAITI  + 0002   WAITO  + 0001   WAITT  + 0004   WAITX  + 0003   XEVC   A 00E2
WOUT   A 0021   XFER   A 113A   XFER1  A 1153   XFER2  A 116C   XFER3  A 1193   XFER4  A 11BB

ASSEMBLY COMPLETE,  NO ERRORS
```

231324-47

APPENDIX B

Test Cases for the Software Drivers

The following test cases were used to exercise the software routines and to check their action. To provide another device/controller on the GPIB a ZT488 GPIB Analyzer was used. This analyzer acted as a talker, listener or another controller as needed to execute the tests. The sequence of outputs are shown with each test. All numbers are hexadecimal.

Send Test Cases

B=	44	44	44
C=	30	2	0
DE=	3E80	3E80	3E80
HL=	3E70	3E70	3E70
3E70:	20 30 3E 3F		
3E80:	11 44		
GPIB output:	41 ATN	41 ATN	41 ATN
	3F ATN	3F ATN	3F ATN
	20 ATN	20 ATN	20 ATN
	30 ATN	30 ATN	30 ATN
	3E ATN	3E ATN	3E ATN
	11	11	
	44 EOI	44 EOI	
Ending B=	44	44	44
Ending C=	2E	0	0
Ending DE=	3E82	3E82	3E80
Ending HL=	3E73	3E73	3E73

Receive Test Cases

B=	44	44	44	44	44	44	44
C=	30	30	30	30	4	4	0=256
DE=	3E80	3E80	3E80	3E80	3E80	3E80	3E80
HL=	3E70	3E70	3E70	3E70	3E70	3E70	3E70
3E70:	40	50	5E	5F	40	40	40
GPIB output:	40 ATN	50 ATN	5E ATN		40 ATN	40 ATN	40 ATN
	3F ATN	3F ATN	3F ATN		3F ATN	3F ATN	3F ATN
	21 ATN	21 ATN	21 ATN		21 ATN	21 ATN	21 ATN
ZT488 Data	1	1	1		1	11	1
In	2	2	2		2	22	2
	3	3	3		3	33	3
	4	4	44,EOI		4	44	44
	44	5,EOI					
Ending A =	0	0	0	5F	40	0	0
Ending B =	0	0	0	44	40	0	0
Ending C =	2B	2B	2C	30	0	0	FC
Ending DE=	3E85	3E85	3E84	3E80	3E84	3E84	3E84
Ending HL=	3E71	3E71	3E71	3E70	3E71	3E71	3E71

Serial Poll Test Cases

C=	30
DE=	3E80
HL=	3E70
3E70:	40
	50
	5E
	5F
GPIB output:	3F ATN
output:	21 ATN
output:	18 ATN
output:	40 ATN
input*:	00
output:	50 ATN
input*:	41
output:	5E ATN
input*:	7F
output:	19 ATN

*NOTE: leave ZT488 in single step mode even on input

Ending C =	30
Ending DE=	3E83
Ending HL=	3E73
Ending 3E80:	00 41 7F

C=	30
DE=	3E80
HL=	3E70
3E70:	5F
GPIB output:	3F ATN
	21 ATN
	18 ATN
	19 ATN
Ending C =	30
Ending DE=	3E80
Ending HL=	3E70

Pass Control Test Cases

HL=	3E70	3E70	3E70
3E70:	40	41(MTA)	5F
GPIB output:	40 ATN		
	09 ATN		
	—$\overline{\text{ATN}}$		
Ending HL=	3E71	3E70	3E70
Ending A =	02	41(MTA)	5F

Receive Control Test Cases

GPIB input	10 ATN	40 ATN	41 ATN
	$\overline{\text{ATN}}$	09 ATN	09 ATN
Run Receive Control			
GPIB Input		$\overline{\text{ATN}}$	$\overline{\text{ATN}}$
Ending A=	0	0	09

Parallel Poll Enable Test Cases

DE=	3E80	3E80
HL=	3E70	3E70
3E70:	20 30 3E 3F	3F
3E80:	01 02 03	
GPIB output:	3F ATN	3F ATN
	20 ATN	
	05 ATN	
	61 ATN	
	30 ATN	
	05 ATN	
	62 ATN	
	3E ATN	
	05 ATN	
	63 ATN	
Ending DE=	3E83	3E80
Ending HL=	3E73	3E70

Parallel Poll Disable Test Cases

HL=	3E70	3E70
3E70:	20 30 3E 3F	3F
GPIB output:	3F ATN	3F ATN
	20 ATN	05 ATN
	30 ATN	70 ATN
	3E ATN	
	05 ATN	
	70 ATN	
Ending HL=	3E73	3E70

Parallel Poll Unconfigure Test Case

GPIB output: 15 ATN

Parallel Poll Test Cases

Set DIO#	1	2	3	4	5	6	7	8	None
Ending A	1	2	4	8	10	20	40	80	0

SRQ Test

	Set SRQ momentarily	Reset SRQ
Ending A =	02	00

Trigger Test

```
HL=          3E70
DE=          3E80
BC=          4430
3E70:        20  30  3E  3F
GPIB output: 3F ATN
             20 ATN
             30 ATN
             3E ATN
             08 ATN
Ending HL=   3E73
DE=          3E80
BC=          4430
```

Device Clear Test

```
HL=          3E70
DE=          3E80
BC=          4430
3E70:        20  30  3E  3F
GPIB output: 3F ATN
             20 ATN
             30 ATN
             3E ATN
             14 ATN
Ending HL=   3E73
DE=          3E80
RC=          4430
```

XFER Test

```
B=           44
HL=          3E70:
3E70:        40  20  30  3E  3F
GPIB output: 40 ATN
             3F ATN
             20 ATN
             30 ATN
             3E ATN
GPIB input:  0
             1
             2
             3
             44
Ending A =   0
B=           44
HL =         3E74
```

Application Example
GPIB Output/Input

GPIB output:	41 ATN
	3F ATN
	32 ATN
	46
	55
	31
	46
	52
	33
	37
	4B
	48
	41
	4D
	32
	56
	4F
	0D EOI
	41 ATN
	3F ATN
	31 ATN
	50
	46
	34
	47
	37
	54 EOI
GPIB input:	SRQ
GPIB output:	3F ATN
	21 ATN
	18 ATN
	51 ATN
GPIB input:	40 $\overline{\text{SRQ}}$
GPIB output:	19 ATN
	51 ATN
	3F ATN
	21 ATN

GPIB input:	20
	2B
	20
	20
	20
	33
	37
	30
	30
	30
	2E
	30
	45
	2B
	30
	0D
	0A
GPIB output:	XX ATN

APPENDIX C

REMOTE MESSAGE CODING

Mnemonic	Message Name		Type	Class	DIO 8	7	6	5	4	3	2	DIO 1	NDAC VDC	ATN	EOI	SRQ	IFC	REN
					Bus Signal Line(s) and Coding That Asserts the True Value of the Message													
ACG	addressed command group		M	AC	Y	0	0	0	X	X	X	X	XXX	1	X	X	X	X
ATN	attention		U	UC	X	X	X	X	X	X	X	X	XXX	1	X	X	X	X
DAB	data byte	(Notes 1, 9)	M	DD	D 8	D 7	D 6	D 5	D 4	D 3	D 2	D 1	XXX	0	X	X	X	X
DAC	data accepted		U	HS	X	X	X	X	X	X	X	X	XX0	X	X	X	X	X
DAV	data valid		U	HS	X	X	X	X	X	X	X	X	1XX	X	X	X	X	X
DCL	device clear		M	UC	Y	0	0	1	0	1	0	0	XXX	1	X	X	X	X
END	end		U	ST	X	X	X	X	X	X	X	X	XXX	0	1	X	X	X
EOS	end of string	(Notes 2, 9)	M	DD	E 8	E 7	E 6	E 5	E 4	E 3	E 2	E 1	XXX	0	X	X	X	X
GET	group execute trigger		M	AC	Y	0	0	0	1	0	0	0	XXX	1	X	X	X	X
GTL	go to local		M	AC	Y	0	0	0	0	0	0	1	XXX	1	X	X	X	X
IDY	identify		U	UC	X	X	X	X	X	X	X	X	XXX	X	1	X	X	X
IFC	interface clear		U	UC	X	X	X	X	X	X	X	X	XXX	X	X	X	1	X
LAG	listen address group		M	AD	Y	0	1	X	X	X	X	X	XXX	1	X	X	X	X
LLO	local lock out		M	UC	Y	0	0	1	0	0	0	1	XXX	1	X	X	X	X
MLA	my listen address	(Note 3)	M	AD	Y	0	1	L 5	L 4	L 3	L 2	L 1	XXX	1	X	X	X	X
MTA	my talk address	(Note 4)	M	AD	Y	1	0	T 5	T 4	T 3	T 2	T 1	XXX	1	X	X	X	X
MSA	my secondary address	(Note 5)	M	SE	Y	1	1	S 5	S 4	S 3	S 2	S 1	XXX	1	X	X	X	X
NUL	null byte		M	DD	0	0	0	0	0	0	0	0	XXX	X	X	X	X	X
OSA	other secondary address		M	SE	(OSA = SCG ∧ $\overline{\text{MSA}}$)													
OTA	other talk address		M	AD	(OTA = TAG ∧ $\overline{\text{MTA}}$)													
PCG	primary command group		M	—	(PCG = ACG ∨ UCG ∨ LAG ∨ TAG)													
PPC	parallel poll configure		M	AC	Y	0	0	0	0	1	0	1	XXX	1	X	X	X	X
PPE	parallel poll enable	(Note 6)	M	SE	Y	1	1	0	S	P 3	P 2	P 1	XXX	1	X	X	X	X
PPD	parallel poll disable	(Note 7)	M	SE	Y	1	1	1	D 4	D 3	D 2	D 1	XXX	1	X	X	X	X
PPR1	parallel poll response 1	(Note 10)	U	ST	X	X	X	X	X	X	X	1	XXX	1	1	X	X	X
PPR2	parallel poll response 2	(Note 10)	U	ST	X	X	X	X	X	X	1	X	XXX	1	1	X	X	X

REMOTE MESSAGE CODING (Continued)

Mnemonic	Message Name		Type	Class	Bus Signal Line(s) and Coding That Asserts the True Value of the Message: DIO8	7	6	5	4	3	2	DIO1	NDAC VDC	NRFD	ATN	EOI	SRQ	IFC	REN
PPR3	parallel poll response 3	}	U	ST	X	X	X	X	X	1	X	X	XXX	1	1	X	X	X
PPR4	parallel poll response 4	(Note 10)	U	ST	X	X	X	X	1	X	X	X	XXX	1	1	X	X	X
PPR5	parallel poll response 5	}	U	ST	X	X	X	1	X	X	X	X	XXX	1	1	X	X	X
PPR6	parallel poll response 6	}	U	ST	X	X	1	X	X	X	X	X	XXX	1	1	X	X	X
PPR7	parallel poll response 7	(Note 10)	U	ST	X	1	X	X	X	X	X	X	XXX	1	1	X	X	X
PPR8	parallel poll response 8	}	U	ST	1	X	X	X	X	X	X	X	XXX	1	1	X	X	X
PPU	parallel poll unconfigure		M	UC	Y	0	0	1	0	1	0	1	XXX	1	X	X	X	X
REN	remote enable		U	UC	X	X	X	X	X	X	X	X	XXX	X	X	X	X	1
RFD	ready for data		U	HS	X	X	X	X	X	X	X	X	X0X	X	X	X	X	X
RQS	request service	(Note 9)	U	ST	X	1	X	X	X	X	X	X	XXX	0	X	X	X	X
SCG	secondary command group		M	SE	Y	1	1	X	X	X	X	X	XXX	1	X	X	X	X
SDC	selected device clear		M	AC	Y	0	0	0	0	1	0	0	XXX	1	X	X	X	X
SPD	serial poll disable		M	UC	Y	0	0	1	1	0	0	1	XXX	1	X	X	X	X
SPE	serial poll enable		M	UC	Y	0	0	1	1	0	0	0	XXX	1	X	X	X	X
SRQ	service request		U	ST	X	X	X	X	X	X	X	X	XXX	X	X	1	X	X
STB	status byte	(Notes 8, 9)	M	ST	S	X	S	S	S	S	S	S	XXX	0	X	X	X	X
					8		6	5	4	3	2	1						
TCT	take control		M	AC	Y	0	0	0	1	0	0	1	XXX	1	X	X	X	X
TAG	talk address group		M	AD	Y	1	0	X	X	X	X	X	XXX	1	X	X	X	X
UCG	universal command group		M	UC	Y	0	0	1	X	X	X	X	XXX	1	X	X	X	X
UNL	unlisten		M	AD	Y	0	1	1	1	1	1	1	XXX	1	X	X	X	X
UNT	untalk	(Note 11)	M	AD	Y	1	0	1	1	1	1	1	XXX	1	X	X	X	X

The 1/0 coding on ATN when sent concurrent with multiline messages has been added to this revision for interpretive convenience.

NOTES:

1. D1–D8 specify the device dependent data bits.
2. E1–E8 specify the device dependent code used to indicate the EOS message.
3. L1–L5 specify the device dependent bits of the device's listen address.
4. T1–T5 specify the device dependent bits of the device's talk address.
5. S1–S5 specify the device dependent bits of the device's secondary address.
6. S specifies the sense of the PPR.

S	Response
0	0
1	1

P1–P3 specify the PPR message to be sent when a parallel poll is executed.

P3	P2	P1	PPR Message
0	0	0	PPR1
.	.	.	.
.	.	.	.
.	.	.	.
1	1	1	PPR8

7. D1–D4 specify don't-care bits that shall not be decoded by the receiving device. It is recommended that all zeroes be sent.
8. S1–S6, S8 specify the device dependent status. (DIO7 is used for the RQS message.)
9. The source of the message on the ATN line is always the C function, whereas the messages on the DIO and EOI lines are enabled by the T function.
10. The source of the messages on the ATN and EOI lines is always the C function, whereas the source of the messages on the DIO lines is always the PP function.
11. This code is provided for system use, see 6.3.

Modem Products

4

APPLICATION BRIEF

AB-24

May 1989

89024 Modem Customization for V.23 Data Transmission

BRIAN D. WALSH
APPLICATIONS ENGINEER
INTEL CORPORATION

Order Number: 292058-001

INTRODUCTION

This application brief will illustrate the steps involved in customizing a modem application using the 89024 modem chip set. Specifically, it will show how one may add V.23 capability to an 89024 modem design as embodied in the MEK II (Intel Modem Evaluation Kit) running software version 3.2.

GENERAL DESCRIPTION

This design consists of using the 89026 processor to control a separate V.23 Data Pump IC (Texas Instruments TCM3105) to support V.23 modulation in addition to the currently supported V.22bis/V.22/V.21/Bell212/Bell103.

The modem is placed in V.23 mode using the "AT&A1" command and is returned to normal operation with the "AT&A0" command. The originating modem dials normally using "AT" commands and then 2 seconds after completion of dialing, the modem sends 75 bps V.23 carrier. The answering modem, upon detecting a ring signal, goes off hook and sends 1200 bps V.23 carrier. The originate modem sends data at 75 bps and receives data at 1200 bps, while the answer modem sends at 1200 bps and receives at 75 bps. Both respond to "escape" at 1200 bps and command mode is always at 1200/1200 bps. The V.23 transmit level is fixed. Backward channel CCITT circuits are not supported, data is always transmitted from pin 2 and received at pin 3.

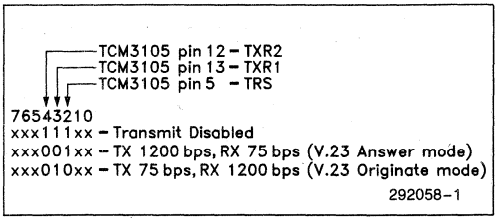
This application brief does not address the issues of V.25 calling tones or V.25 calling station identification.

HARDWARE DESCRIPTION

The MEK II is modified by adding a Texas Instruments TCM3105 FSK Modem IC. This Modem chip does not have an on-chip 4-wire to 2-wire hybrid circuit, so we use a dual op-amp MC1458 for this purpose. In order to control the TCM3105 we use 3 additional outputs of the 74LS373 latch that is already used to latch the /JS and AA signals from the microcontroller address/data bus. A 74LS157 2- to 1-line data selector is used to select the source of received data and the source of "energy detect" signal to the microcontroller.

V.23 Modem IC

The TCM3105 (U102) is a CMOS V.23 modem in a 16-pin package that consumes only 40 mW. It requires an external 4.4336 MHz crystal connected between pins 15 and 16 to derive timing. A resistor divider sets the carrier detect threshold by adjusting the voltage at pin 10. Bias distortion may be minimized by adjusting the voltage at pin 7. Pins 5, 13 and 12 together set the various modes of operation. These pins are connected to pins 6, 9 and 12 respectively of 74LS373 (U18) and are controlled through bits 2, 3 and 4 and executing a "STore" instruction to any even address of external memory (since this is the only external memory to be used). The modes of interest to us are:



74LS157 Data Selector

This IC is always enabled and the select signal is connected to the 6th output (bit 5) of the 74LS373 latch (U18). "SToring" a "0" to bit 5 of the latch selects "normal" mode of operation, while "SToring" a "1" to bit 5 selects V.23 mode. During "normal" mode, Receive Data (RXD) is routed from the 89026 microcontroller to the DTE and Energy Detect (ED) is routed from the 89027 AFE to the microcontroller. During V.23 mode RXD goes from the TCM3105 to the DTE and ED goes from the TCM3105 to the microcontroller. Transmit Data (TXD) is always connected from the DTE to both the 89026 and the TCM3105.

MC1458 Dual Op-Amp

This IC is configured as an active hybrid circuit, converting the 4-wire transmit and receive signals to 2-wire to drive the line transformer. The transmitted signal is also summed, but since only one of the transmitters will be active at a time, this will not be a problem. The 89027 has pin 10 tied low so as to disable the AFE's on-chip hybrid.

A schematic diagram of these changes is shown in Figure 1.

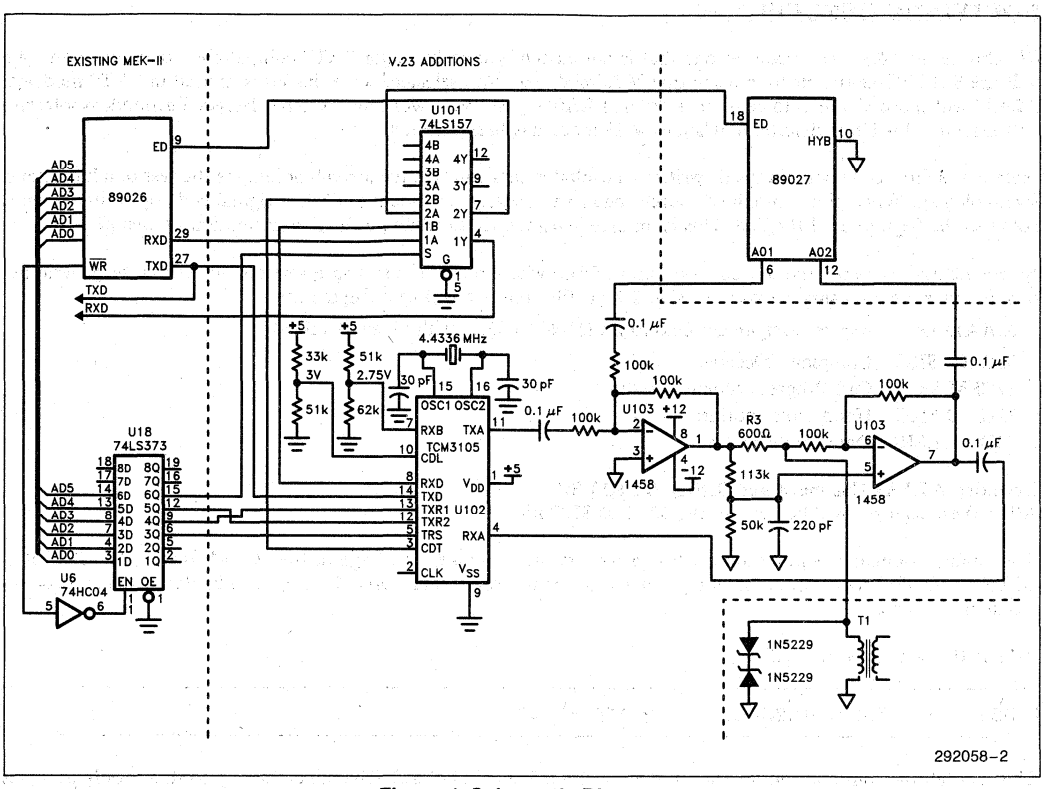


Figure 1. Schematic Diagram

SOFTWARE DESCRIPTION

We choose the "&A" command as one that is not currently used by major "AT" compatible modem vendors. We will use S23 bit 3 as the bit to indicate that V.23 mode has been selected, since this bit is unused in "AT" modems. "&A1" will cause S23 bit 3 to be set to a "1" and &A0 or just "&A" will cause it to be cleared. The modem software will examine this bit to determine whether V.23 mode has been selected.

Note that source code will always be written in capital letters and that the assembler ignores the rest of a line after a semi-colon (;). When giving modified source code I will usually "comment out" the original code by adding a semi-colon to the beginning of the line. This is an excellent practice to facilitate the documentation of changes.

By convention we name the source files: nmxxx.SRC (where n.m is the software version and xxx is the generic file name). Since we are using software version 3.2 the files that we will be changing are:

32AAD.SRC	register assignment definitions ($INCLUDEd with all source files)
32CMD.SRC	Command Decoder
32CPM.SRC	Call Progress Monitor routines
32HND.SRC	Handshake routines
32DATA.SRC	Data Mode routines

Decoding AT&A1 Command and Setting the S23 Bit
All of these changes will be done to the 32CMD.SRC file.

Since many commands simply modify S-register bits, we can take advantage of the "COMMON_REGISTER_OPERATIONS:" code by adding our command to the necessary tables and allowing it to be decoded as a register-modifying command.

Add as the last entry in TABLE_1:

```
DCB   (3 * 32) + (S23-S0)      ; AND_A_CMD
```

This will tell the common routine that this command affects bit 3 of S23. The table is set up so that it only occupies one byte per entry, with the bit number in the upper 3 bits and the register number in the lower 5 bits.

Add the command to the command list and the command vector table:

```
AND_CMDS:       DCB   "CJLPRSDG'
;               DCB   "MXFWZT',     0   ; was like this
                DCB   "MXAFWZT',    0   ; added &A command betw X and F
```

```
CMD_LU_TBL:     ....
                ....
;    DCB   AND_G_CMD-G1, AND_M_CMD-G1, AND_X_CMD-G1, AND_F_CMD-G2
     DCB   AND_G_CMD-G1, AND_M_CMD-G1, AND_X_CMD-G1, AND_A_CMD-G1
     DCB   AND_F_CMD-G2
```

The command vector table is the address offset of the command label from that of the first command (G1 EQU A_CMD). In the interests of saving space this offset table is only 1 byte per entry and so it has to be split into 2 groups as the range of addresses of command labels is more than 255 bytes. When modifying command code it is worth checking the list file to make sure that the CMD_LU_TBL: entries do not get bigger than 0FFH and wrap around through 0, causing those commands to branch to the wrong address.

Fix the branch vector calculator and the dial command offset calculator because the 1st group of commands are now 33 instead of 32:

```
GENERATE_BRANCH_VECTOR:
     ADD  TEMP_CMD_3, #G1              ; ADD OFFSET TO 1ST CMD GROUP
;    CMPB TEMP_CMD_2, #32              ; FIRST 32 CMDS FIT IN
     CMPB TEMP_CMD_2, #33              ; FIRST 33 CMDS FIT IN
```

```
D_R_CMD:
;    SUBB  CPM_CONTROL, TEMP_CMD_2, #36
     SUBB  CPM_CONTROL, TEMP_CMD_2, #37
```

Add the command label with the rest of the register modifying commands:

```
Y_CMD:
AND_A_CMD:    ; added &A command for V.23 operation
AND_C_CMD:
```

Updating the Output Pins to Control the TCM3105 and Data Selector

The IO_CONTROL: section of code in file 32CMD.SRC runs all the time and could be considered the "background routine". This is where the RS232 leads are updated, the health of the other routines is checked and the 74LS373 latch (U18) is written and is thus an appropriate place for the TCM3105 chip and the Data Selector (Data Mux) to be updated.

Add the following code after END_JS_UPDATE:

```
END_JS_UPDATE:

V_23_UPDATE:
  ANDB TEMP_CMD_1, #11011111B       ; MUX TO NON-V.23 POSN
  ORB  TEMP_CMD_1, #00011100B       ; SET V.23 CHIP OFF
  JBC  S23,  3, END_V_23_UPDATE     ; JMP IF NOT IN V.23 MODE

  JBC  CNTRL_C, 1, END_V_23_UPDATE ; JMP IF NOT IN HND OR DATA MODE
  ANDB TEMP_CMD_1, #11100111B       ; SET V.23 CHIP TO ANS MODE
  ORB  TEMP_CMD_1, #00000100B

  JBC  S14, 7, NOT_ORIG_MODE        ; JMP IF S REG SET TO ANS MODE
  ANDB TEMP_CMD_1, #11101011B       ; SET V.23 CHIP TO ORIG MODE
  ORB  TEMP_CMD_1, #00001000B

NOT_ORIG_MODE:
  JBC  CNTRL_C, 0, END_V_23_UPDATE ; JMP IF NOT IN DATA MODE
  JBS  CNTRL_C, 2, END_V_23_UPDATE ; JMP IF CMD FUNCTS ENABLED
  ORB  TEMP_CMD_1, #00100000B       ; DATA MODE, SO MUX TO V.23 POSN

END_V_23_UPDATE:
```

The next instruction in the source code STores the contents of TEMP_CMD_1 to PORT3, and so updates the Data Mux.

In order to ensure that the Data Mux gets set before the "OK" message is sent when entering the on-line escape state (response to "+ + +"), add a line of code after the three "ORB" instructions:

```
VALID_ESCAPE_SEQUENCE:
  ORB  CNTRL_F, #00010000B          ; ENABLE ESCAPE STATE
  ORB  CNTRL_C, #00000100B          ; ENABLE CMD FUNCTIONS
  ORB  MSG_RQST, #00100000B         ; SEND "OK" MESSAGE WITH MSG RQST
  JBS  S23, 3, ESCAPE_DETECT_END ; TRICK TO FORCE 1 MORE PASS THRU
       ; IO_CONTROL FOR MUX SETUP BEFORE GOING TO COMMAND DECODER
```

After a dial command is executed by the Command routine, it will activate the Call Progress routines.

The V.23 Call Progress Monitor Routines
The 32CPM.SRC routines check for call progress signals on the phone line and also for answer tone from the remote answering modem. Since a V.23 modem will answer with a 1300 Hz tone (1200 bps mark frequency), the AFE receive filter must be set to V.22 answer mode so as to pass this frequency to the energy detect circuitry.

Add three lines of code at the label SET_ANSWER_CONT:

```
SET_ANSWER_CONT:
  ANDB CPM_FLAG, #11101111B ; FLAG ANSWER PROCESSING FOR HOUSEKEEPING

  JBC  S23,  3, SET_ANSWER_CONT_1 ; IF V23 MODE THEN
  LDB  AFE_BYTE3, #01000000B      ; SET FILTER TO QAM ANS FOR
SET_ANSWER_CONT_1:                ; 1300Hz CARRIER DETECTION

  SJMP SIGNAL_MONITOR_INIT
```

The CPM routines will hand over control to the Handshake routines which we need to modify for V.23 handshake.

The V.23 Handshake Routines
The Handshake mode 32HND.SRC is entered for the first time after successful completion of the Call Progress routines. The first time that HANDSHAKE_MODE: is called, it goes through the Initialization code before the main routine is executed, thereafter the Initialization is skipped. The Main routine is entered at a rate of 600 times per second or more and consists of checking for Energy Detect and then branching to the routine address saved in TX_RTN_ADDR. The logical flow of the handshaking is controlled by changing the contents of TX_RTN_ADDR to the address of the routine to be executed the next time Handshake is called.

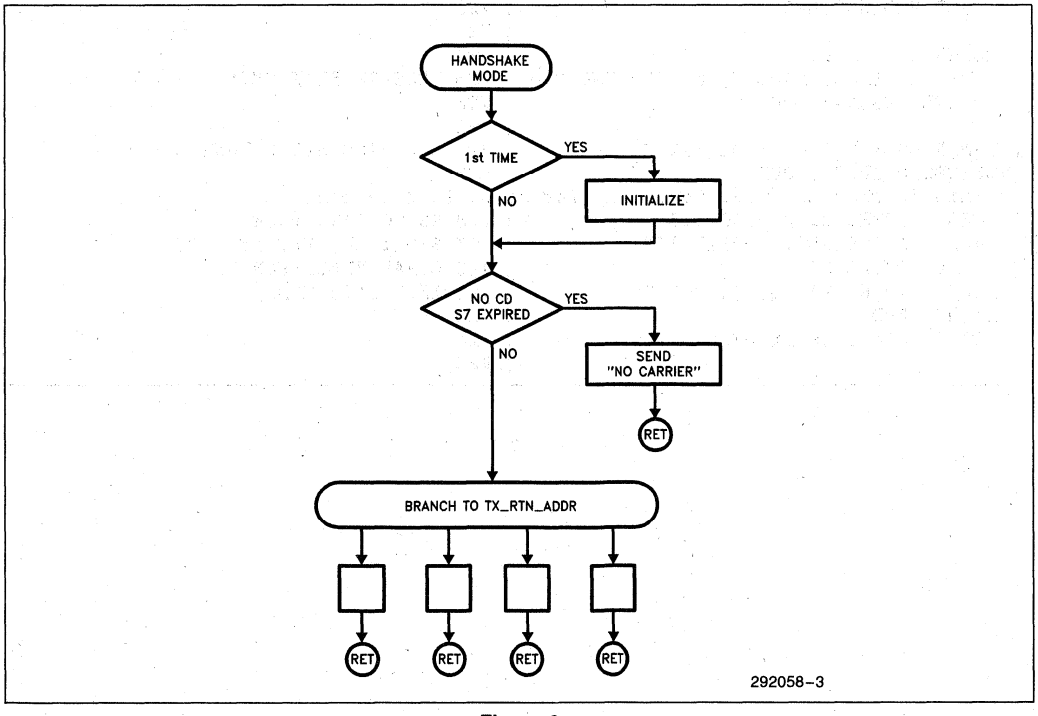


Figure 2

The Initialization required for V.23 consists of starting the S7 wait-for-carrier timer, starting a 2 second timer and loading a return address for the next time the routine executes. The following lines of source code are added (identified by "V23" at the start of the comment field) to the Handshake Initialization:

```
HANDSHAKE_INIT:
  ANDB MODE_STATUS, #10111111B          ; CLEAR INIT FLAG
                                        ; V23
  JBC  S23,  3,  NOT_V23_INIT           ; V23
V23_HND_INIT:                           ; V23
  ADDB S7_TIMER, TIME_BASE_SECOND, S7   ; V23 INIT S7 DCD TIMER
  ADDB TX_TIMER, TIME_BASE_100MS, #20D  ; V23 INIT 2 SEC TIMER
  LD   TX_RTN_ADDR,   #V23_HND_WAIT     ; V23
  SJMP HND_INIT_END                     ; V23
                                        ; V23
NOT_V23_INIT:                           ; V23
```

After the initialization code is executed once, the software will keep branching to V23__HND__WAIT: until the 2-second timer has expired, then it will initiate a "CONNECT" message. While the Connect message is being sent, the software will branch to V23__HND__MESSAGE:, then it will set up the Data mode and thereafter the Data Mode will be called instead of the Handshake mode.

```
V23_HND_WAIT:                           ; V23
  CMPB TIME_BASE_100MS, TX_TIMER        ; V23 TIMER EXPIRED YET?
  JNE  V23_HND_END                      ; V23
V23_HND_MESSAGE_INIT:                   ; V23
  LDB  MESSAGE_REQUEST, #00100001B      ; V23 START CONNECT MESSAGE
  LD   TX_RTN_ADDR,   #V23_HND_MESSAGE  ; V23
  SJMP V23_HND_END                      ; V23
                                        ; V23
V23_HND_MESSAGE:                        ; V23
  JBS  MESSAGE_REQUEST, 5, V23_HND_END  ; V23 MESSAGE SENT YET?
V23_HND_MESSAGE_END:                    ; V23
                                        ; V23
  ANDB COPY_PORT4, #10111111B           ; V23 DCD HIGH AFTER CONNECT
V23_SET_UP_DATA_MODE:                   ; V23
; This is where we need to set up for going to data mode
  ORB   CNTRL_C, #00000011B             ; V23 GO TO DATA MODE
  ANDB  AFE_BYTE4, #00111111B           ; V23 TXMITTER OFF, AFE OFF
  CLRB  DM_FLAGS                        ; V23 CLEAR FLAGS FOR DM
  ORB   MODE_STATUS, #10000000B         ; V23 INIT DATA MODE
V23_HND_END:                            ; V23
  LJMP  HANDSHAKE_MODE_END              ; V23
                                        ; V23
```

V.23 Data Mode

The modifications required in the Data Mode consist of checking for V23 mode and skipping past:

Initialization
Send space disconnect (twice)
Receive space disconnect
Loss of carrier disconnect
Retrain request
Test mode

```
DATA_MODE_INIT:
; DM FLAGS ALREADY CLEARED IN HANDSHAKE MODE
  ANDB MODE_STATUS,#7FH                ; CLEAR INITIALIZE FLAG
  JBS S23, 3, DATA_MODE_INIT_END       ; IF V23 THEN INIT DONE
```

```
DISCONNECT_INIT:
  ANDB DM_FLAGS, #11111101B            ; CLEAR DISCONNECT INIT FLAG
  JBS  S23,  3,  HANG_UP               ; V23 FORGET SPACE DISCONNECT
```

```
SEND_SPACE:
  JBC  S21, 7, HANG_UP                 ; IF BREAK_DISCONNECT DISABLED
  JBS  S23, 3,  HANG_UP                ; V23 FORGET BREAK
```

```
CHECK_DISCONNECT:                      ; CHECK FOR LONG SPACE DISC
CHECK_BREAK:
  JBS  S23, 3,  SET_BREAK_TIME         ; V23 FORGET BREAK
```

```
CHECK_CARRIER_LOSS:
  JBS  PORT0, 7, CHECK_CARRIER_LOSS_END ; SKIP IF ED IS HIGH
  ORB  DM_FLAGS, #01000000B             ; SET CDLOSS FLAG
  ADDB EDOFF_TIME,TIME_BASE_100MS,S10 ; CDOFF THRESHOLD IN REGISTER
  INCB EDOFF_TIME                  ; PUT AN OFFSET IN TIME FOR PROPER
                                        ; OPERATION DURING TM EXIT
  JBS S23, 3, CARRIER_LOSS_END          ; ALL DONE IF V23 MODE
```

```
QAM_RETRAIN:
  JBS S23, 3, SJMP_CHECK_TEST_MODE    ; SKIP RETRAIN IF V23 MODE
```

```
CHECK_S16_STATUS:
; EXAMINE S16 REGISTER FOR ANY TEST MODES AND SET FLAG
  JBS S23, 3, CHECK_S16_STATUS_END    ; SKIP RETRAIN IF V23 MODE
```

Assembling the Source Files
The source files can be assembled by issuing the following commands at the DOS prompt:

```
ASM96 32CMD.SRC
ASM96 32CPM.SRC
ASM96 32HND.SRC
ASM96 32DATA.SRC
```

Linking the Object Files
Link the object files by issuing the following command at the DOS prompt:

```
RL96 32HND.OBJ, 32INIT.OBJ, 32CMD.OBJ, 32CPM.OBJ, 32DATA.OBJ,
     32SOFT.OBJ, 32HSI.OBJ, 32HSO.OBJ, 32RX.OBJ TO 32ATR
```

Programming the EPROMs
After the code has been linked and located, the code must be split into low and high byte segments for programming into EPROMS. The following IPPS session illustrates that process (IPPS prompts are not shown):

```
IPPS                          ; invoke IPPS
I 80                          ; initialize file format
FORMAT 32ATR                  ; filename resulting from linking
3                             ; logical unit is byte
2                             ; input file is in words (2 bytes)
1                             ; output file is in bytes
0 to  32ATR.LO                ; low order bytes to one file
1 to  32ATR.HI                ; high order bytes to another
<enter>                       ; press "enter" to exit formatting
                              ;
                              ; the following assumes that an
                              ; INTEL PiUP 201A programmer is
                              ; connected to the PC
                              ;
TYPE                          ; display available EPROM types
27128                         ; specify EPROM type
                              ; insert blank EPROM into programmer
COPY 32ATR.LO TO PROM         ; copy low byte file to prom
                              ; insert blank EPROM into programmer
COPY 32ATR.HI TO PROM         ; copy low byte file to prom
EX                            ; exit IPPS
```

Custom routines can now be tested by placing EPROMS into target hardware.

REFERENCES

1. "FSK Modems: TCM3105 Designers Information" from Telecommunications Circuits Data Book, 1986. By Texas Instruments.
2. MEKII 89024 Enhanced Modem Evaluation Kit Users Manual, 1987. By Intel Corp.
3. 89024 Modem Reference Manual, 1987. By Intel Corp.
4. Developing MCS-96 Applications Using the SBE-96. Application Note AP-273 (Order Number 280249-001). By Intel Corp.

ISDN Products

5

intel®

APPLICATION NOTE

AP-282

January 1989

29C53 Transceiver Line Interfacing

JAGTINDER S. BOLARIA
TELECOM PRODUCT MARKETING

Order Number: 270209-003

INTRODUCTION

Presently, the majority of the transmission from the telephone to the Central Switching system is analog. For this purpose the circuitry interfacing to the twisted pair line is optimized to operate between 300 and 3400 Hz. The essential line interface functions consist of isolation, over voltage protection, signaling, power feeding and a ringing signal insertion. With the advent of ISDN (Integrated Services Digital Network) these functions have to be reassessed.

ISDN is implemented with digital transmission from the subscriber to the switch, which in turn offers the user various data services in addition to the voice service. CCITT has various recommendations for the implementation of the ISDN network. Of these, I.430 details the basic rate access i.e. the physical communications between a terminal and the first level of switching. For I.430, Intel offers a transceiver which is capable of operating at either end of the loop, namely the 29C53.

The 29C53 is a four wire (two for transmit and two for receive) transceiver operating over the "S" loop. The data transmitted by the 29C53 at the switch and the terminal is at a rate of 192 kb/s; the effective data throughput is 144 kb/s. This data consists of two bearer channels of 64 kb/s each (B1 + B2) and a 16 kb/s D channel. The 29C53, additionally, incorporates some protocol processing for the D channel. This transceiver has four interfaces, namely the microprocessor port, a general purpose I/O port, the SLD port and the "S" loop interface. It is the loop interface requirements that are addressed by this application note.

This note will analyze the line interface requirement at both the line card and the terminal, and will offer general implementations. These implementations will address power feeding, the protection circuitry, the line transformers and power extraction. Throughout this brief, the approach has been to present various alternate concepts which may assist the designer in addressing a specific application.

LINE INTERFACE

Both at the line card and the terminal, there is a need to provide isolation for the circuitry from the line itself. As well as isolation, it is also necessary to protect the equipment from any overvoltage conditions on the line. Additionally the system may be designed to provide phantom power feeding i.e. the switching system delivers power to the terminal over the "S" loop. Unlike its analog counter part the digital line card does not need to send a ringing signal owing to the fact that all signaling is accommodated via the D channel.

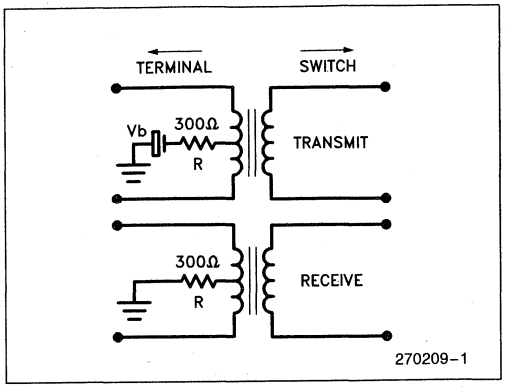


Figure 1. Voltage Feeding

POWER FEEDING

Figure 1 shows the CCITT recommended technique of phantom power feeding as described in section 9 of I.430. The current splits evenly between the two secondary windings. This in turn produces equal and opposite fluxes in the transformer, that cancel each other out, thus preventing the core from saturating. The equality of the fluxes in the secondary will depend on the longitudinal balance of the transformer and the transmission line.

The scheme shown on Figure 1 may be wasteful of power when feeding short lines. One way around this would be to have a constant current feed, which will make the power consumption independent of the length of line. Figure 2 shows such an implementation.

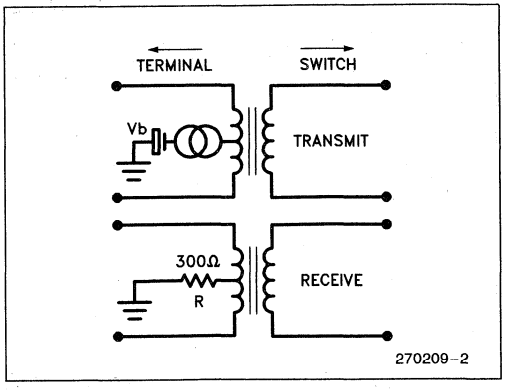


Figure 2. Current Feeding

One way of reducing the power dissipation over the loop is to provide a variable voltage source, instead of the traditional fixed voltage. This can be accomplished by using a DC to DC converter, or a switching regulator. The feedback circuit of the switching regulator can be used to ensure that the regulator provides just enough voltage to maintain a pre-defined feed current down any length of line. The DC to DC converter can have a built in threshold detector, which would be used to release the line in case excessive currents are being drawn.

In the event of mains power loss, it is often required to maintain a minimal voice service powered off the line. Figure 3 shows the block diagram of a digital telephone, illustrating the necessary components required to maintain a voice service.

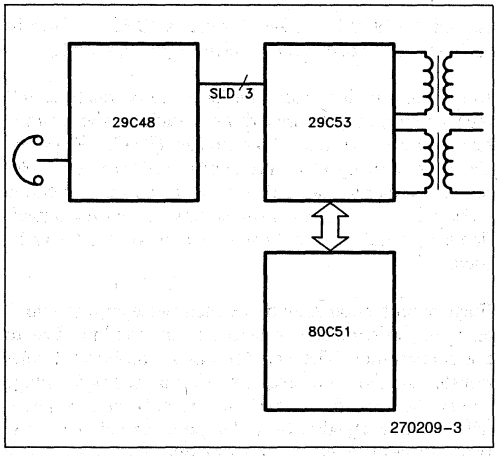


Figure 3. Digital Telephone

The 80C51 is a low power microcontroller while the 29C48 is an SLD compatible combo (codec and filter). The gains through the 29C48 can be set externally or programmed by the microcontroller via the SLD interface. The 29C48 is designed to allow insertion of sidetone and DTMF (dual tone multi-frequency); both these features are presently used to provide feedback to the user.

PROTECTION

Next, let us examine the question of protection. A telecommunication system comprises subscribers linked together through the cable plant and a switching network. The cable plant consists of multiple pairs of transmission lines, either suspended on poles, or buried in the earth. In either case, transient energy can be coupled from lightning (or other electromagnetic events) and conducted to the switch or the terminal. The other major source of transient energy is the commercial AC power system, where high currents that accompany faults can induce overvoltage in the lines, or the power lines can fall and make contact with the telephone lines. The latter is sometimes referred to as a mains or power cross.

It is generally agreed, as shown in Figure 4, that two or more levels of protection are required. The primary protector is usually placed on the line at a distance greater than 25m from the line card. The impedance of the line will ensure that the primary protector will operate first and the secondary protector will not be exposed to the full surge. If the primary protector is to be placed closer to the secondary, then a small resistor can be inserted in series with the line between the primary and the secondary protector (1). A 5Ω 3W resistor or a positive temperature coefficient resistor may be used. During a surge, the voltage drop across the resistor will increase allowing the voltage across the primary protector to build up thus driving it to conduction.

The primary protection can be a gas discharge tube, such as the General Instrument three terminal PMT3-(310). These devices consist of spaced metallic gaps enclosed in a combination of gases at low pressure. In the event of a surge, the gap breaks down, diverting the transient and thus rerouting the energy. These devices can be operated a number of times and present a capacitance of less than 5 pF. Since the templates in Figures 10 and 11 of I.430 specify a low output capacitance for the terminal and the network terminator, the low output capacitance feature of the gas discharge tube makes it ideal for ISDN i.e. it will have a minimal effect on the line drivers.

The secondary protection can be provided by Schottky diodes chosen for the low voltage drop and capacitance across them. The diodes are placed between the power supplies and the loop interface pins on the 29C53, thus forming a diode bridge across the line. This will ensure that the voltage on these pins does not exceed the power supplies by more than approximately 300 mV, thus fulfilling the specification that the voltage on any pin

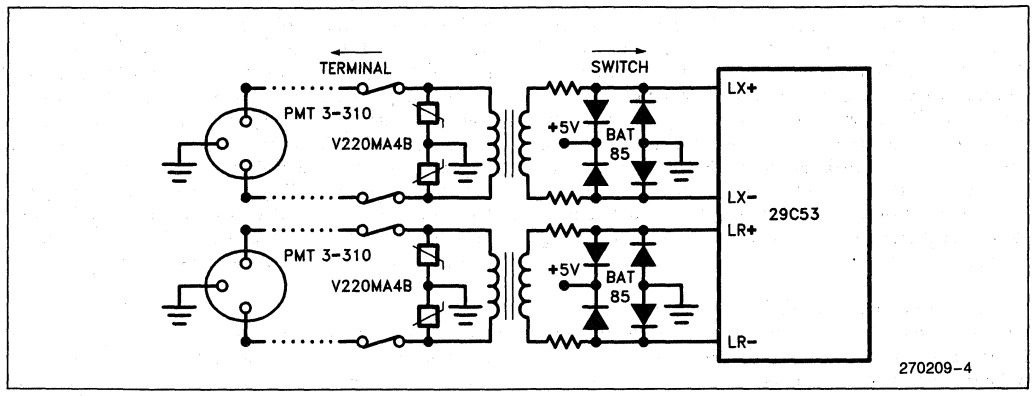


Figure 4. Protection

may not exceed the power supply by more than 500 mV. The 5V and ground connections to the diodes should be as close as possible to the 29C53 power supply pins, which in turn should be decoupled by a 0.1 μF capacitor. The capacitor serves a secondary function of bypassing surge currents. The particular diodes chosen are dependent on the expected surge current, however, BAT85 from Philips used in this application can withstand 200 mA forward current while presenting a maximum of 10 pF capacitance across it. The maximum current through the diodes can be limited by placing a resistor in series with the diodes and the transformer. The value of this resistance can be extracted from the transformer design discussion. To further limit the current to the 29C53, the series resistance can be split, with part of it on the 29C53 side of the diodes, and part of it on the transformer side of the diodes. For the receive direction it is possible to replace the diode bridge by placing a resistance in series with the 29C53 receive pins. This series resistance will limit the surge current that the 29C53 is exposed to. The value of this resistance is limited by the input impedance presented by the 29C53 and the loss that can be tolerated in the received signal. The receive differential input impedance of the 29C53 is 100 KΩ, hence a 10 KΩ resistor in each arm will reduce the received signal by 17%.

In case of a mains cross, the loop can be made to self recover by using thermal devices such as the positive temperature coefficient thermister (PTC). Keystone Carbon Company has a range of PTCs specific to telephone line applications that they refer to as resettable fuses. Economic considerations may make this unjustifiable in which case a fusible resistor or link may be used.

Further protection may be deemed necessary, in which case two varistors can be placed across the line close to the transformer. The varistor has a volt-current relationship similar to a diode i.e. after a specified voltage across the varistor is reached, the current through it will rise dramatically; thus clamping the voltage to the specified level. A typical varistor that may be used as a back-up protection is the GE V220MA4B. This device typically presents a 21 pF capacitance.

The ideas discussed thus far are encompassed in Figure 5 for a minimal component count protection scheme.

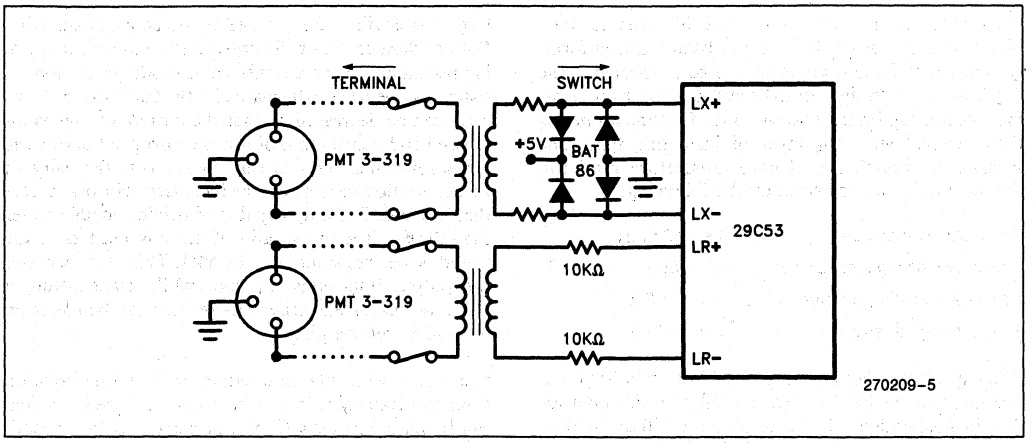


Figure 5. Protection with Minimal Components

LINE TRANSFORMER

A transformer is used at both the terminal and the line card to provide isolation from the line. A well balanced I.430 transformer resolves the issue of DC currents since they induce self-cancelling fluxes. Generally speaking, a pulse transformer with minimum leakage inductance and self capacitance is required. The impedance templates in I.430 specify the minimum value of the inductance required at the line side. This value can be calculated to be 20 mH. A further requirement is to minimize the winding resistance, so that a minimal voltage is dropped across it. A 2.5:1 ratio transformer can be used with the 29C53 to produce the proper pulse amplitude. The transformer design discussed below can be used with the 29C53 at either the line card or the terminal. Alternatively it can be used for example purposes to aid designs.

The RM series of ferrite cores are chosen to facilitate easy winding and PCB mounting, additionally the RM series is available internationally from various vendors–Ferroxcube in the U.S. and Mullard in Europe, to name two. The RM6 core was selected to be the smallest size that accommodates wiring which does not exceed the maximum allowable DC resistance. The core material has to have a high enough permeability to allow the 20 mH inductance with a minimum number of turns hence, the Ferroxcube core material 3E2A was selected. This material has a very high inductance factor, A_L. This is given by the manufacturer as the inductance (in mH) per 1000 turns.

For the core RM6PL00-3E2A

$A_L = 6710 \pm 25\%$

Therefore minimum

$A_L = 5032 \cong 5000$

The number of turns, Ns, required for 20 mH is given by:

$Ns = 10^3 \sqrt{L/A_L}$ L - required inductance in mH

Ns = 70 turns - assume 25 mH is required

The 29C53 side winding will require 2.5 times this number of turns.

Np = 175 turns

The transformer is now ready to be wound, the 32 gauge wire will just fill the RM6PCB1 bobbin. The bobbin is started by bifilar winding the 175 turns. Bifilar winding is accomplished by taking two separate pieces of wire and winding them simultaneously. The finish of one winding is then soldered to the start of the other and often, as is the case in this implementation, the point of connection of the two wires (center tap) is brought out to a pin of the transformer. The remaining ends (start and finish) now comprise the winding. The transformer is now followed by 1¼ layers of insulating tape. The insulating tape used was the Permacel P-256 which forms a dielectric capable of withstanding 5 KV, this serves to protect the line card and the subscribers

from lightning induced surges. The 70 turns are then bifilar wound; this results in a well balanced transformer. The start of one winding should be connected to the finish of the other and brought out to a pin, thus creating a center tap on the line winding. The transformer is then finished with 1½ layer of insulating tape. The transformer thus designed gave satisfactory results in the lab and is characterized by the following:

Secondary inductance	Ls = 26 mH
Secondary leakage inductance	ls = 20 μH
Secondary winding resistance	Rs = 1.5Ω
Primary winding resistance	Rp = 2.7Ω

The capacitance between the two bifilar windings was measured to be 100 pF and this may be too high for certain applications. For this case the bifilar winding can be replaced by the cross winding technique shown in Figure 6a. The two windings are now wound in opposite directions, one wire is on top on the top side while the other is on top on the bottom side. This technique reduced the above mentioned capacitance to less than 50 pF.

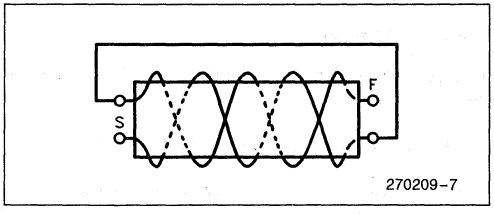


Figure 6a. Crosswinding

The 29C53 has been designed to drive voltages as specified in the I.430, since the transformer presents a series resistance, some of this voltage will be dropped across it. For the transformer designed above, the overall series resistance is (2.7 + 1.5*6.25) = 12Ω which will result in a 3.8% error over the allowed peak transmit signal in I.430. This is acceptable as I.430 allows a 10% error for the peak voltage. If series resistors are required to protect the Schottky diodes, their value may be calculated by having the maximum allowed peak voltage error. Note that equal value resistors should be placed on both arms of the line. If larger values of protection resistors are required, the above procedure may be repeated with a larger core. This will allow the same inductance to be achieved with a fewer turns and the larger core will make it possible to use a thicker wire. Both of these factors will contribute to reduce the winding resistance, hence a larger value diode protection resistor may be used. Alternatively, the transformer turns ratio can be decreased so that the output voltage is increased and hence more of it can be dropped across the series resistance. This in turn means that the value of this protection resistor can be increased. However, note that the 29C53 is only capable of driving loads greater than 200Ω. If a turns ratio of 1.8:1 is used then the overall series resistance can be 64Ω. This also increases the output impedance to 20Ω while transmitting a pulse. As discussed earlier this resistor can be larger on the 29C53 receive pins.

Some establishments may require further line isolation from the transformer in which case a Faraday shield can be placed in between the primary and the secondary windings. The Faraday shield can be made by wrapping 1¼ layers of a copper tape (such as the permacel P-389) between the two windings. The copper tape should be insulated from the windings and should be brought out to the local ground. As well as isolating, the Faraday shield also serves to reduce the interwinding capacitance.

The transformer designed was connected up as shown in Figure 6b to measure its longitudinal balance.

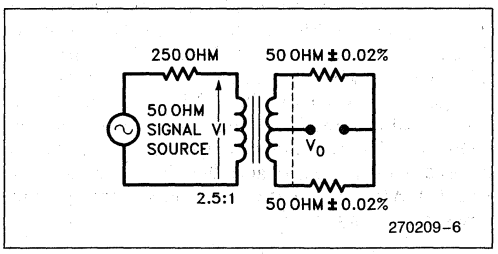


Figure 6b. Longitudinal Balance

Let v = vi/2.5
Then longitudinal balance is given by: 20 Log V/Vo

Measurements conducted showed this figure to be better than 70 dB for the frequency range of 10 KHz to 1 MHz.

The center tap on the primary (29C53 side) is coupled to ground via a 10 nF capacitor. In this manner longitudinal signals on the primary are bypassed to ground. Measurements produced greater than 70 dB of longitudinal signal rejection.

When designing the System board, special care should be paid to the layout. The transformer and the 29C53 should both be placed on a ground plane. The connecting tracks from the 29C53 to the transformer should be as short as possible. The two devices should be placed close to the edge where the transmission lines interface, while the high frequency logic should be placed on the opposite edge. The analog ground wiring should follow a star configuration and should have a separate isolated lead originating from the system ground where it enters the board.

Though the analysis of pulse transformers is beyond the scope of this brief (2), one should be aware of the pertinent parameters affecting the good reproduction of the pulse. The pulse transformer is generally analyzed by different equivalent circuits, depicting the varying phases of the pulse.

Figure 7 shows these circuits. The pulse shape is then optimized by considering the transient response of the equivalent circuits.

The pulse response of the transformer is characterized by a finite rise time, a decaying top period and finite fall time as depicted in Figure 7d. The fastest rise time that can be obtained without overshoot is for the critically damped case and is given by:

$$tr = 3.35\sqrt{\alpha Lc} \quad \text{where } \alpha = R_L / (Rg + R_L)$$

For the top period, there will be some decay leading to a fractional droop, this is given by:

$$D \cong \tau \frac{Lp}{R} \quad \text{where } \tau = \text{pulse width}$$

$$R = R_L \text{ and } Rg \text{ in parallel}$$

The fall period is characterized by the second order circuit of Figure 7c; the primary concern here to prevent severe undershoot or backswing when the 29C53 transmitter is in the high impedance mode. This can best be achieved by having an overdamped system, which is the case when:

$$Lp > 4CR_L^2$$

Commercially available pulse transformers exist which are compatible with the 29C53. Some examples are given in Table 1. Most manufacturers will modify their design to meet the requirements of a particular application.

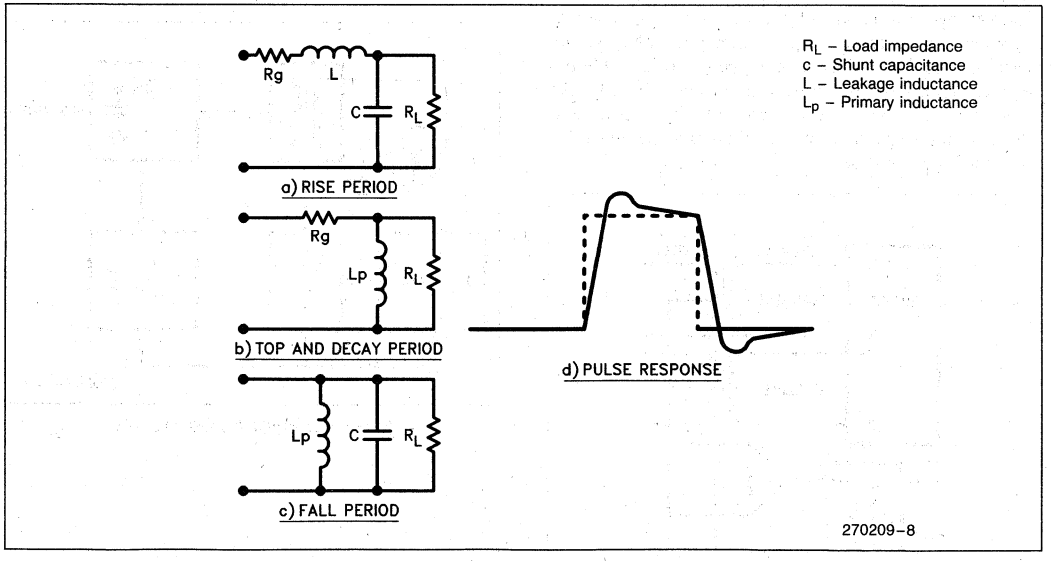


Figure 7. (a) Equivalent Circuits for Rise Period (b) Top and Decay Period (c) Fall Period (d) The Pulse Response

TABLE 1. Manufacturers of Pulse Transformers

Manufacturer	Location	Winding Ratio	Part No.
AIE Magnetics	St. Petersburg, FL (813) 347-2181	1.8:1 2.5:1	325-0228 325-0172
Schott Corporation	Nashville, TN (615) 889-8800	1.8:1 2.5:1	11207 11124
CTM Magnetics	Tempe, AZ (602) 967-9447	1.8:1 2.5:1	22087 25585
Pulse Engineering	San Diego, CA (619) 268-2400	1.8:1 2.5:1	64994 64996

POWER EXTRACTION

The same transformer can be used at both the line card and the terminal, and the same protection scheme can be used at both ends of the loop. The need now arises to provide power to the terminal. There are a number of ways of providing power to the terminal, for instance a secondary cell can be used as battery back-up in conjunction with a main supply. There is also some scope for trickle charging secondary cells from the line or from a small solar cell array, but the drawback with secondary cells tends to be their short life span. This disadvantage can be offset by using special purpose primary cells as a back-up supply, these do not need any charging circuitry and can be expected to have life expectancy twice that of the secondary cells. Finally, the power can be fed from the switch, in which case a regulator is required at the terminal to extract the power off the line. Figure 8 illustrates this approach.

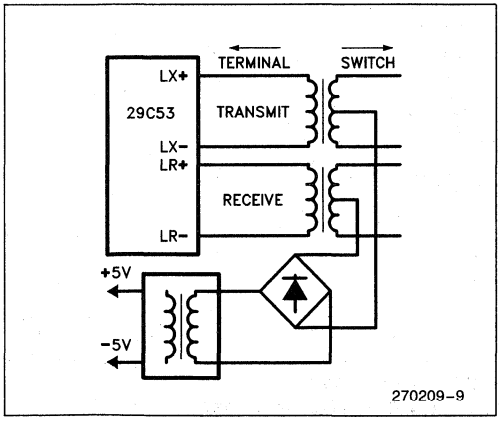


Figure 8. Power Extraction

A DC to DC converter is required to convert the line voltage to 5V for the local circuitry. In order to obtain the lowest losses in the conversion process, it is necessary to use a high efficiency regulator, specifically, a switched mode regulator. Basically, there are three types of switched mode power supplies, the forward, the push pull and the flyback converter (3). This section is devoted to the flyback implementation of a DC to DC converter. The flyback is the most suitable converter for this application, as it provides the highest achievable efficiency and the simplest drive circuitry. Figure 9 shows a block diagram of a flyback converter.

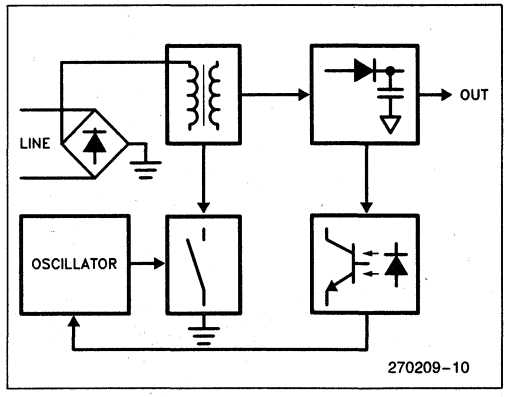


Figure 9. Flyback Converter

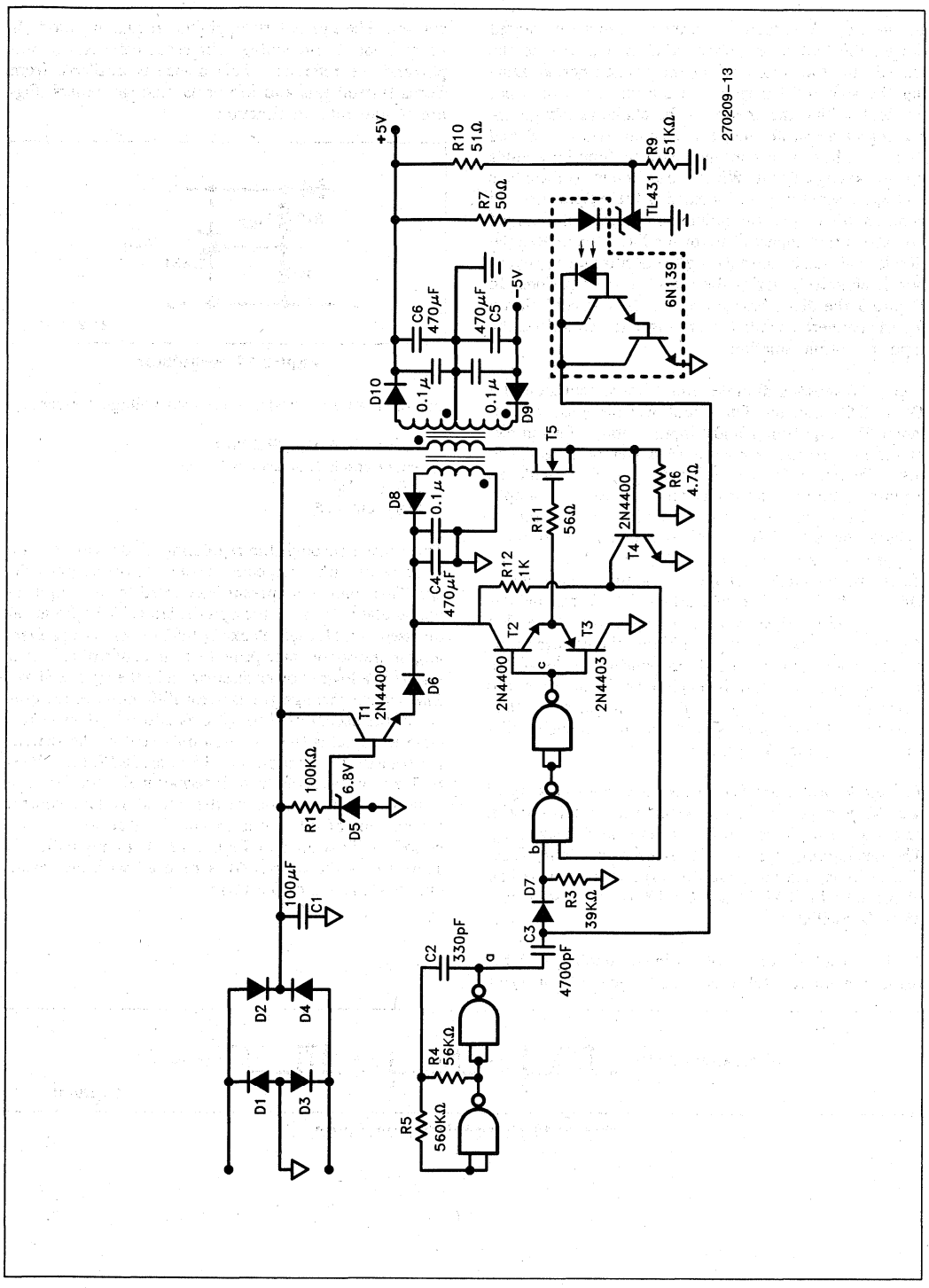


Figure 10. DC to DC Converter

In the flyback inductor, energy is inductively stored during the switch on period, and then passed to the load during the switch off, or the flyback period. During the switch on period, the output diode does not conduct so that the energy in the choke (although appearing as a transformer, this element will be referred to as the choke in accordance with its function) builds up with rising current. While the switch is off the choke voltage reverses in polarity causing the output diode to conduct whereupon the inductive energy is discharged into the output capacitor to form a DC voltage. Regulation is achieved by modulating the oscillator duty cycle, which effectively varies the switch on/off periods. In Figure 9 the diode bridge ensures the correct polarity for the converter while the opto-isolator completes the input to output isolation.

Figure 10 shows a discrete circuit implementation of a DC to DC converter. This circuit was designed to regulate a 5V output for 20-60V input voltage. This implementation provides a maximum power of at least 450 mW. The DC to DC converter consists of an oscillator, a pulse width modulator incorporating an error amplifier and isolating stage, the start up circuitry and the flyback converter. When T5 is on, the choke stores energy and reverse biases diodes D8, D9 and D10. While T5 is off, the choke voltage is negative, hence diodes D8, 9 and 10 are all forward biased and thus build a DC voltage on their respective capacitors. Note that due to the reverse winding technique, the voltage in the output windings are opposite in polarity to the switch winding. The 5V output is regulated by comparing it to a reference voltage, the error in the comparison is then used to modify the transistor T5 on time in such a way so as to keep the 5V output constant.

The diode bridge D1-D4 ensures a certain polarity of the DC voltage for the converter, this is necessary in case the network uses polarity reversal for signaling. The decoupling capacitor Cl serves a secondary function of bypassing any induced surge current. One half of the Schmitt NAND gate CD4093 is used to form a 25 KHz oscillator.

At the output, the opto-isolator in conjunction with the regulating diode TL431 is used to generate an error current. The current through the regulating diode is proportional to the voltage difference between the output and the reference. This device is available from Texas Instruments and Motorola amongst others. Figure 11 illustrates its function.

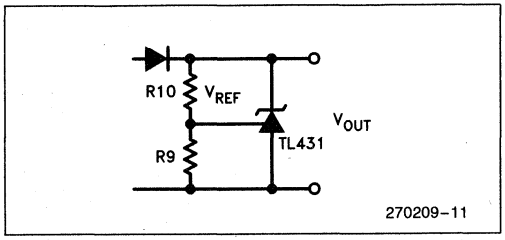


Figure 11. Regulator

For the regulator diode, the output voltage is given by:

$$V_{out} = (1 + R_{10}/R_9)\, V_{ref}$$

where Vref is typically 2.5V.

If $R_{10} = R_9$

then Vout = 5V

The current through the regulating diode will increase or decrease with a respective change in the output voltage. This change in current is coupled to the output of the oscillator through the opto-isolator. The opto-isolator used is a Hewlett Packard 6N139, which has Darlington transistor stage providing high current gain that results in a lower power dissipation in the opto-isolator. The current through the isolator differentiates the output of the oscillator through capacitor C3. This differentiated signal is then squared off to define the switching transistor T5 on period. T5 is an IRFD110 MOSFET and is available from International Rectifier. The isolator current and hence the output voltage control the amount of differentiation or the transistor T5 on period as illustrated in Figure 12. Thus regulation is achieved, as the on period is reduced with increasing output voltage and vice versa.

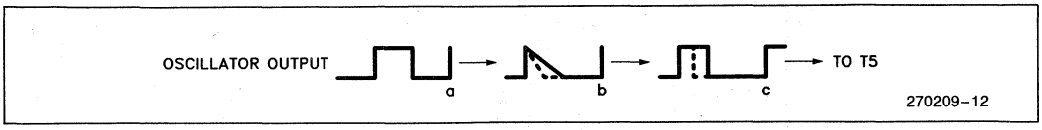


Figure 12. Pulse Width Modulation

The two transistors T2 and T3 provide a low source impedance driving stage for the switching transistor. The fast current sinking and sourcing will ensure fast switching of transistor T5.

The input capacitance of the MOSFET IRFD110 is a maximum of 200 pF. Without the buffer stage the MOSFET will stay in the linear region longer before saturating, thus resulting in a slower switching speed. The slow switching in turn will result in a lower overall efficiency for the converter.

The resistor R6 and transistor T4 provide current overload protection. Transistor T4 will conduct when the voltage across R6 exceed 0.6V or conversely, the current through it is greater than 150 mA. With T4 conducting, the drive to the MOSFET is nulled by the associated NAND gate.

The transformer choke is a three winding transformer consisting of the switching winding, the output winding, which is split for the +5V and −5V and the self-bias winding. The transformer is designed for complete energy transfer under no load conditions and incomplete energy transfer under full load conditions. Figure 13 shows the wave forms of the two modes.

At full load, the incomplete energy transfer mode exhibits a lower peak switching transistor current, while the complete mode at lower power assures a smaller core. The inductance required to achieve this is 6.5 mH for the switch winding. The core used was an RM6CA400-3B7. The number of turns required to achieve this inductance is 130 and for a 20-60V line voltage, 50 turns are required for a +5V output, hence use 50 turns for the −5V too. The self bias winding uses 70 turns. The transformer was wound with 130 turns of 34AWG, followed by 50 bifilar turns of 32AWG and finished off with 70 turns of 32AWG. The dot scheme in Figure 10 should be adhered to. The bobbin is then immersed in varnish such as the Dolph's BC356 to dispel any moisture and to provide a protective coating. Alternatively, a commercially available DC to DC converter transformer such as the 326-0533 can be purchased from AIE Magnetics.

At start up, the converter is powered by the linear regulator D5, R1 and T1, which sets the power supply at 5.3V. After start up the self bias winding forces the voltage on C4 to be between 7 and 15 volts, which will back bias diode D6, thus turning off the linear regulator. Under this condition the power supply provides a self bias voltage to keep it running, while little power is

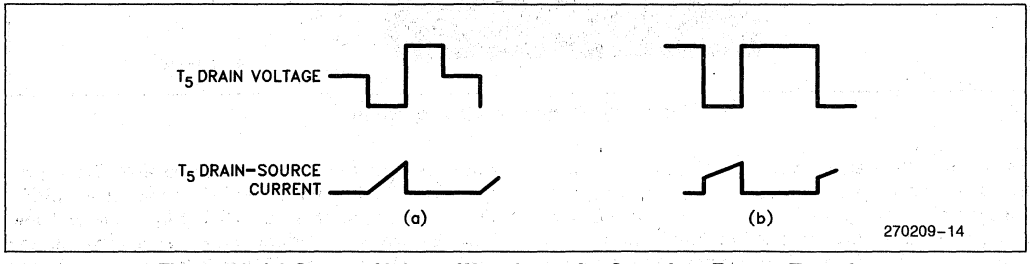


Figure 13. (a) Current Voltage Waveforms for Complete Energy Transfer (b) Waveforms for Incomplete Energy Transfer

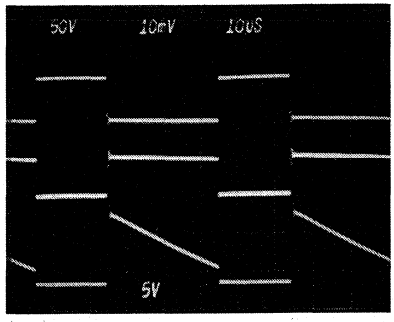


Figure 14. Converter Oscillograms

dissipated in the start up regulator. Transistor T1 is selected so that the base-collector can sustain the high voltage stress when it is off. The −5V supply will only be regulated if the load on that winding is the same as that on the +5V winding. If this is not possible, it may be necessary to use a linear post regulator to obtain a regulated −5V supply.

Figure 14 shows the volt-current oscillograms for a 30V line voltage and 400 mW output power. This shows the flyback converter working in the incomplete energy transfer mode. The results obtained in the lab gave an overall efficiency of better than 67% and a power supply ripple of less than 25 mV. The no load power consumption was less than 50 mW. Regulation of the output voltage was better than 150 mV.

The design was wire wrapped to illustrate the concept of power extraction and can of course, be optimized for better performance. Special care should be paid to the layout; Figure 15 shows good layout principles. Use star ground connections to avoid current loops in the ground.

All lead lengths going to the switching MOSFET should be minimized and in particular the gate lead. The resistor in series with the MOSFET should be placed as close to the gate lead as possible. These precautions will avoid undesired oscillations in the MOSFET. The output stage uses Schottky diode and low ESR capacitors to reduce power dissipation. In the event of any undesired EMI radiation the transformer can be placed in an electromagnetic container and the converter can be enclosed in a copper container.

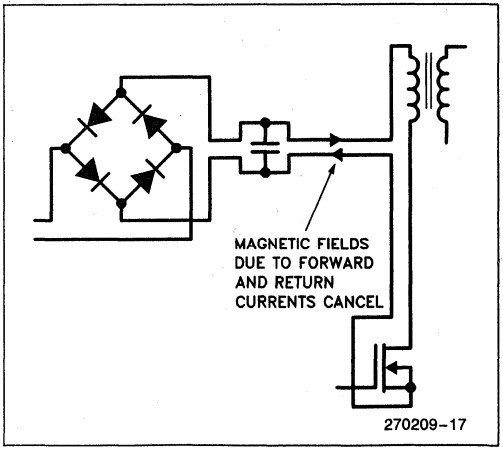


Figure 15. Good Layout Principles

POWER FAILURE CONSIDERATIONS

Without power the line interface pins of the 29C53 appear as diode drops across the line. This means that the transmitter of the Network Terminator and the powered on terminals in a multidrop configuration will be terminated by a diode instead of the usual 50Ω. In the event of a failure, it therefore becomes necessary to isolate the offending terminal from the line. This can be done by providing a switch in the transmit path that is normally closed and opens when no power is applied.

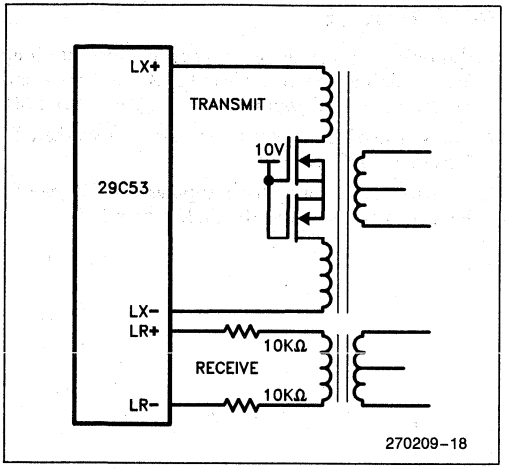


Figure 16. Isolation of Equipment in Case of Power Failure

In the receive path, it is only necessary to increase the impedance seen by the line. One way to implement this principle is to use a MOSFET bilateral switch in the transmit path and to place series resistors in the receive path, such that the impedance seen by the line is greater than 2500Ω. Figure 16 illustrates this approach. A noteworthy point is that the series resistors in the receive path not only provide terminal isolation in case of failure but also protect the terminal from current surges.

When there is power, the two MOSFETS will be on and appear as a small on resistance, which has to be included in the line transformer design analysis. When there is no power, the MOSFETS appear as back to back diodes, thus stopping any AC flow. The VN0300 MOSFETS manufactured by Siliconix may be used, when on they present a 1.2Ω resistance each. Note that in order to ensure that the MOSFETS conduct it is necessary to have a 10V supply in the system. If this is not possible the MOSFETS can be replaced by a Reed relay which presents a lower on resistance and capacitance but has the disadvantage of consuming more power. A low power relay could not be located hence a vendor was requested to customize one. Figure 17 shows the isolation technique using the Wabash 1992-2-1 25 mW relay which will operate at 3.8V and release at 0.5V.

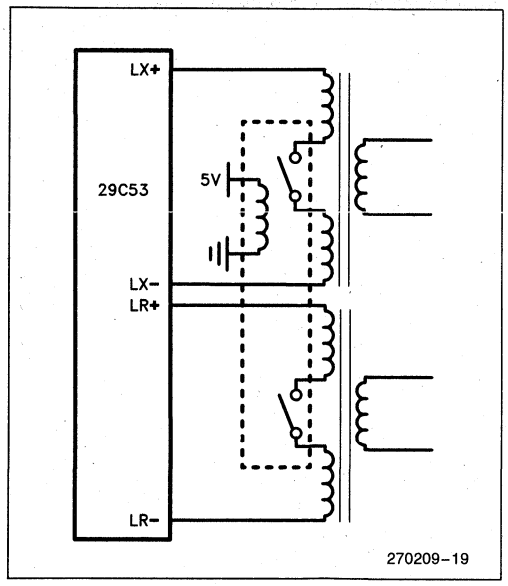


Figure 17. Power Failure Isolation

CONCLUSION

Specific implementations have been provided for the general aspects of line interfacing at both the line card and the terminal end. These solutions can be taken as they are and placed in the particular application or used to aid a system design.

The fixed voltage or constant current feed are both simpler and more economical to realize in discrete form; however the constant current variable voltage scheme may be more suitable in an integrated form. The power converter discussed was based on a low cost simple implementation and it is certainly possible to optimize it to obtain conversion efficiencies in the 75% range. As an alternative to discrete implementations, a low power switch mode power supply is commercially available from Fairchild and Motorola, to name two.

The protection circuits and the transformer, however, can only be provided in discrete form at the present time. The concepts presented in the protection section emphasized low capacitance and maximum protection. The section took an overkill approach and as such a subset of the discussed ideas should suffice most applications. The transformer designed pointed out the relevant parameters to consider and can be used as it is or modified to the particular application. Of course the ISDN transformer is also commercially available.

REFERENCES

1. Protecting against surge voltages in short and long branch circuits. By Shanawaz M. Khan, Communications Systems Equipment Design, December 1984
2. Transformers for electronic circuits. By Nathan R. Grossner, McGraw Hill
3. Design of solid state power supplies. By Eugene R. Hnatek, Von Nostrand Reinhold Company

APPLICATION BRIEF

AP-400

September 1989

ISDN Applications with 29C53 and 80188

HERBERT WEBER
TELECOM OPERATIONS

Order Number: 270247-004

TERMINAL ADAPTOR (TA)

A terminal adaptor, or "TA", is the link between existing non-ISDN equipment like terminals, facsimile, printers and the ISDN network. The function of this application is to effectively replace equipment such as a modem. Usually provided as a separate box, it processes RS232 or X.21 data and places it on the 4 wire 'S' loop. No change at the terminal is required to make it ISDN compatible.

The design is based on a 29C53 transceiver for the ISDN connection and an 80188 microprocessor in combination with an 82530 communications controller for the data connection. Benefits of the application are:

- Data rates up to 19.2 Kb/s using an RS232 interface or up to 48 Kb/s using an X.21 interface.
- Compact design and low cost.
- Virtually error free transmission.

Link Setup

The user sets up a call in the same manner as a Hayes* modem user does, i.e. a command is transferred to the adaptor via the RS232 interface. The command takes the form of an ASCII string in which the first 2 characters are "AT".

The 80188 accepts the command and begins the call setup procedure by communicating the call's destination to the NT (or CO). This is achieved by passing call setup messages to a link level protocol, which is passed to the NT over the physical level (S bus). The partitioning of the tasks is as follows:

82530

Full duplex, dual channel serial communications controller capable of working in asynchronous, bit or byte synchronous modes. The 82530 receives commands from the terminal's RS232 or X.21 interface and passes them on to the 80188.

80188

After having received the dialing information from the keyboard, the 80188 sets up the call via the 29C53 D-channel by sending the appropriate CCITT message up the link.

- Call setup message generation (CCITT I.451).
- Upper portion of link access procedure (CCITT I.440) handling:
  - — Multiple logic channels
  - — Sequence control
  - — Error correction (retransmission)
  - — Flow control

EPLD

- Interface conversion, serial to/from SLD
- B-channel assignment

29C53

- Physical level interface (CCITT I.430)
- Lower portion of the link access procedure:
  - — Zero insertion/deletion
  - — CRC generation and checking
  - — Flag appending and detection
- D-channel message buffering

The 80188 passes the information for the D-channel messages via the parallel bus into the FIFO's of the 29C53.

The NT grants a B-channel (if available) to the TA and the channel is now ready for data transfer.

Data Transfer

An indication is given to the user's terminal via the RS232 or X.21 port that communication may commence. Any subsequent data, from the terminal, is treated as follows:

Data from the terminal passes via the 82530 to RAM via one of the 80188 DMA channels.

The 80188 fetches the data from RAM, depacketizes and packetizes it before sending it back to the 82530 where a protective HDLC protocol is added.

From the 82530 the data reaches the EPLD to be inserted into the B1 or B2 channel on the SLD bus. The 29C53 sends it out over the "S" interface.

*Hayes is a registered trademark of Hayes Microcomputer Products, Inc.

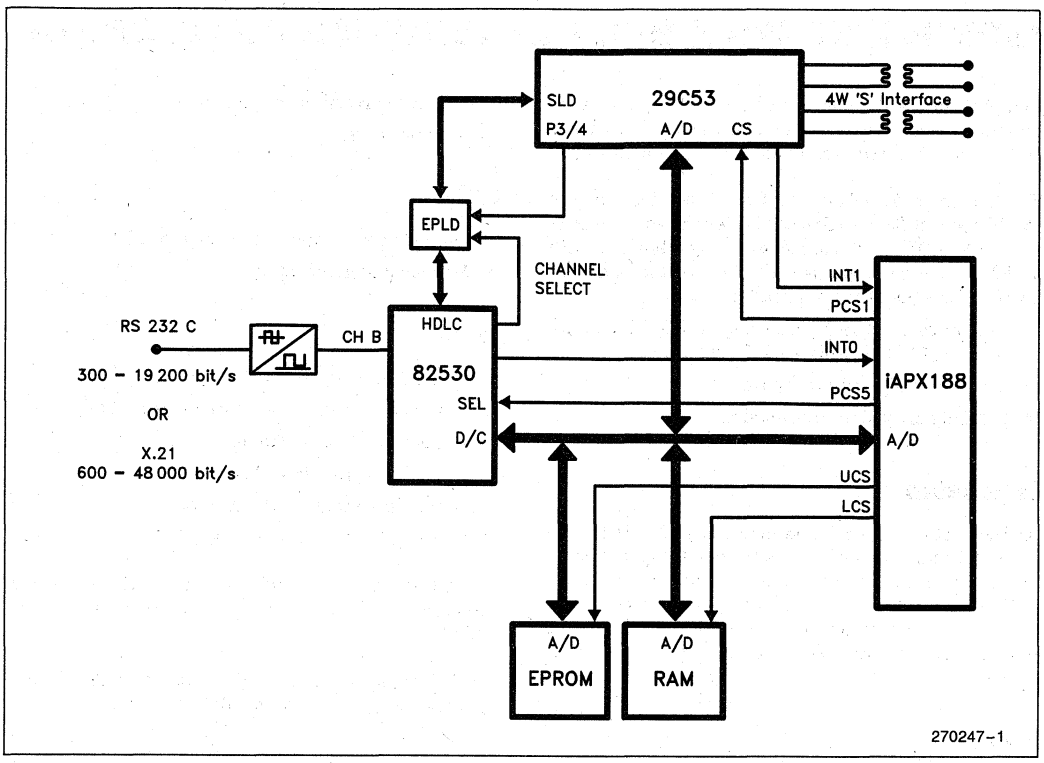


Figure 1. Terminal Adaptor

ISDN PHONE WITH BUILT IN TERMINAL ADAPTOR (TA)

Figure 2 shows the concept of an ISDN phone with hookup to standard sync/async terminals. No change at the terminal is required to make it ISDN compatible.

The design is based on a 29C53 transceiver for the ISDN connection, a 29C48 combo for the voice connection and an 80188 microprocessor in combination with an 82530 communications controller for the data connection. Benefits of the application are:

- Data rates up to 19.2 kb/s using an RS232 interface or up to 48 kb/s using an X.21 interface.
- Compact design and low cost.
- Virtually error free transmission.

Link Setup

Applies both for speech and data links. The 80188 accepts the command and begins the call setup procedure by communicating the call's destination to the NT (or CO). This is achieved by passing call setup messages to a link level protocol, which is passed to the NT over the physical level (S-bus). The partitioning of the tasks is as follows:

8279

The 8279 keyboard and display controller scans the telephone number pad and supports a small telephone display. Calls are initiated either through the terminal keyboard using an extended Hayes Smart Modem command set or via the telephone number pad.

82530

Full duplex, dual channel serial communications controller capable of working in asynchronous, bit or byte synchronous modes. The 82530 receives commands from the terminal's RS232 or X.21 interface and passes them on to the 80188.

80188

After having received the dialing information from either keyboard, the 80188 sets up the call via the 29C53 D-channel by sending the appropriate message up the link.

- Call setup message generation (CCITT I.451)
- Upper portion of link access procedure (CCITT I.440) handling:
  - — Multiple logic channels
  - — Sequence control
  - — Error correction (retransmission)
  - — Flow control

EPLD

- Interface conversion, serial to/from SLD
- B-channel assignment

29C53

- Physical level interface (CCITT I.430)
- Lower portion of the link access procedure:
  - — Zero insertion/deletion
  - — CRC generation and checking
  - — Flag appending and detection
- D-channel message buffering

The 80188 passes the information for the D-channel messages via the parallel bus into the FIFO's of the 29C53.

The NT grants a B-channel (if available) to the TA and the channel is now ready for data transfer.

Information Transfer

VOICE

The voice transfer is supported by the 29C48 which transmits the voice on either the B1 or B2 channel (controlled by the EPLD) into the 29C53 and onward to the S-bus.

DATA

An indication is given to the user's terminal via the RS232 or X.21 port that communication may commence. Any subsequent data, from the terminal, is treated as follows:

Data from the terminal passes via the 82530 to RAM via one of the 80188 DMA channels.

The 80188 fetches the data from RAM, depacketizes and packetizes it before sending it back to the 82530 where a protective HDLC protocol is added.

From the 82530, the data reaches the EPLD to be inserted into the B1 or B2 channel on the SLD bus. The 29C53 sends it out over the "S" interface.

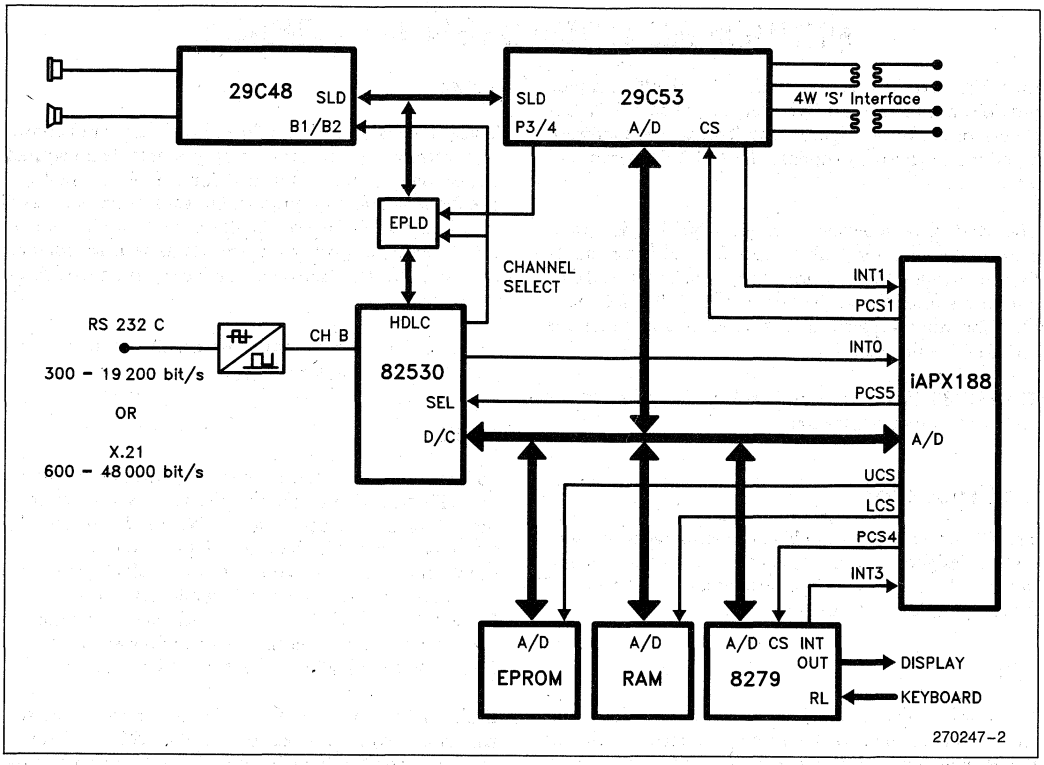


Figure 2. ISDN Phone With Built In Terminal Adaptor

PERSONAL COMPUTER INTERFACE

Like the terminal adaptor, the ISDN PC adaptor provides a link to the ISDN network. The ISDN Co-Processor shown in Figure 3 implements the hardware functions required to support the CCITT I-series "S" interface.

The ISDN Co-Processor is using the 80188 microprocessor in combination with an 82530 serial communications controller for data processing, dual port RAM as interface and buffer to the host bus, the 29C48 Codec/filter for voice support and the 29C53 transceiver for the ISDN connection.

The ISP188 ISDN Software Package is optimized for this hardware configuration.

Co-Processor

The PC adaptor is an intelligent communications subsystem designed to function as a slave processor board in the PC. This relieves the host processor of much of the communications function.

82530

Full duplex, dual channel serial communications controller. One of the two channels is attached to either of the B channels and operates at 64 kb/s. The second channel is available to external datacom equipment via an optional serial port, or for connection to the second B channel.

29C48

Voice conversion and interface to the four wire handset is performed by this software programmable integrated Codec/filter combo. Designed for ISDN terminal applications it offers programmable gain in transmit and receive direction for user loudness control and adaptation to local network requirements as well as sidetone insertion and tone injection for locally produced feedback signals.

The 29C48 can access either B1 or B2 channel by setting the B Sel pin accordingly.

29C53

"S" bus transceiver and D channel controller in a single chip. The 29C53 provides the physical level interface to the "S" bus in accordance to CCITT I.430 and the lower portion of the link access protocol. Activation, deactivation, zero insertion/deletion, CRC generation and checking and flag appending and detection are performed by the 29C53, the higher level portions of LABD are executed on the 80188 and passed on to the 29C53 via the parallel bus into the FIFO's.

B channel access is via the SLD serial port. Voice signals are directly passed on to the 29C48. Data is extracted and injected by the EPLD (Erasable, Programmable Logic Device) which performs the B channel assignment and the interface conversion to the 82530.

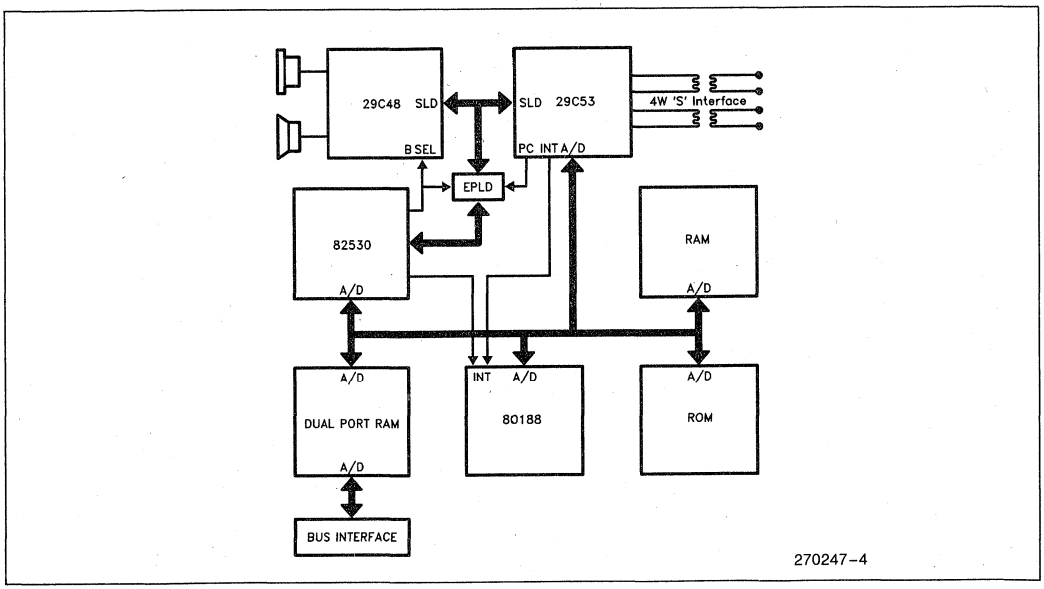


Figure 3. Personal Computer Interface

FULL FEATURE ISDN LINE CARD

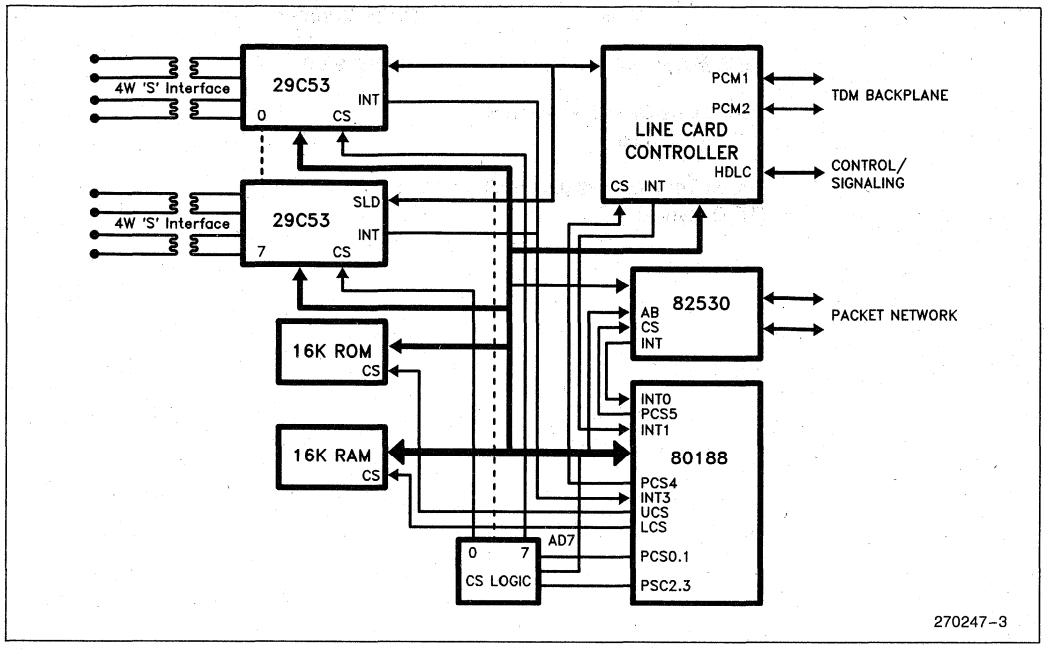


Figure 4. Full Feature ISDN Line Card

The addition of ISDN line cards to a PABX provides the user with access to the ISDN network. While the analog line card provides access for standard telephone as well as for modems, ISDN terminal adaptors, terminals and phones are connected to the ISDN line card via a 4 wire 'S' loop. The described application provides all functions to separate voice and switched data (B-channels) from signaling and packetized data (D-channel).

The 29C53 and 80188 together handle the processing of D-channel protocols and messages as follows:

1. The 29C53 executes all bit level HDLC processing, puts the "raw" messages into its FIFO and raises the interrupt signal.
2. A special status register in the 29C53s allows the 80188, through a single status read operation, to determine which of the 29C53s is requiring interrupt servicing, i.e. has D-channel messages(s) in its FIFO.
3. The 80188 accesses the FIFO concerned and the data is transferred to RAM.
4. The 80188 determines whether the data is for signaling (S-packet) or is a message to be sent out over the packet (P-packet) switched network.

   Signaling information can be processed locally or sent via the linecard controller.

   If the data is of "P" type, meant for the packet switched network, it is DMA'd into the 82530 serial communications controller which performs the necessary HDLC transmission, again without any CPU involvement.
5. The 80188 software is responsible for sending out acknowledgements for received messages from the 29C53's D-channel and can thus support large window sizes.

B-channel information is directly passed from the "S" loop over the 29C53 and line card controller to the switch backplane.

For transmission in the opposite direction, the procedure is equivalent to the one described above.

OTHER AVAILABLE TELECOM LITERATURE

Title	Order Number
29C53AA Reference Manual	296399-001
29C53 Line Card Evaluation Kit (LEK) Manual	
29C53 Terminal Evaluation Kit (TEK) Manual	

PCM Codec/Filter and Combo

6

APPLICATIONS INFORMATION
2910A/2911A/2912A

CODEC INTERFACE

The 2912A PCM Filter is designed to directly interface to the 2910A and 2911A Codecs as shown below. The transmit path is completed by connecting the VF_XO output of the 2912A to the coupling capacitor associated with the VF_X input of the 2910A and 2911A codecs. The receive path is completed by directly connecting the codec output VF_R to receive input of the 2912A VF_RI. The PDN input of the 2912A should be connected to the PDN output of the codec to allow the filter to be put in the power-down standby mode under control of the codec.

CLOCK INTERFACE

To assure proper operation, the CLK input of the 2912A should be connected to the same clock provided to receive bit clock, CLK_R of 2910A or 2911A Codec as shown below. The CLK0 input of the 2912A should be set to the proper voltage depending on the standard clock frequency chosen for the codec and filter.

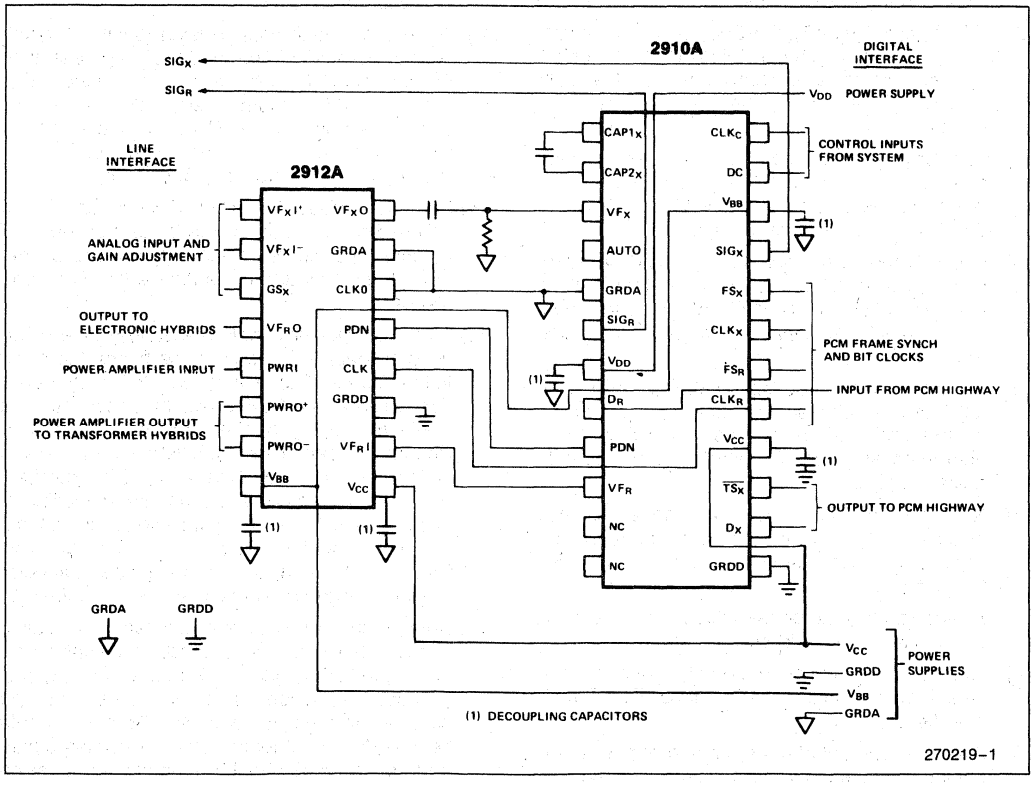


Figure 1. A Typical 2910A Codec and 2912A Filter Configuration

October 1986
Order Number: 270219-001

GROUNDING, DECOUPLING, AND LAYOUT RECOMMENDATIONS

The most important steps in designing a low noise line card are to insure that the layout of the circuit components and traces results in a minimum of cross coupling between analog and digital signals, and to provide well bypassed and clean power supplies, solid ground planes, and minimal lead lengths between components.

1) All power source leads should be bypassed to ground on each printed circuit board (PCB), on which codecs are provided. At least one electrolytic bypass capacitor (at least 50 μF) per board is recommended at the point where all power traces from the codecs and filters join prior to interfacing with the edge connector pins assigned to the power leads.
2) When using two-sided PCBs, use both corresponding pins on opposite sides of the board for the same power lead. Strap them together both on the PCB and on the back of the edge connector.
3) Lay out the traces on codec- and filter-equipped boards such that analog signal and capacitor leads are separated as widely as possible from the digital clock and data leads.
4) Connect the codec sample and hold capacitor with the shortest leads possible. Mount it as close to the codec CAP1X, CAP2X pins as possible. Shield the capacitor traces with analog ground.
5) Do not lay out any board traces (especially digital) that pass between or near the leads of the sample and hold capacitor(s) since they are in high impedance circuits which are sensitive to noise coupling.
6) Keep analog voice circuit leads paired on their layouts so that no intervening circuit leads are permitted to run parallel to them and/or between them.
7) Arrange the layout for each duplicated line, trunk or channel circuit in identical form.
8) Line circuits mounted extremely close to adjacent line circuits increase the possibility of interchannel crosstalk.
9) Avoid assignment of edge connector pins to any analog signal adjacent to any lead carrying digital (periodic) signals or power.
10) The optimum grounding configuration is to maintain separate digital and analog grounds on the circuit boards, and to carry these grounds back to the power supply with a low impedance connection. This keeps the grounds separate over the entire system except at the power supply.
11) The voltage difference between ground leads GRDA and GRDD (analog and digital ground) should not exceed two volts. One method of preventing any substantial voltage difference between leads GRDA and GRDD is to connect two diodes back to back in opposite directions across these two ground leads on each board.
12) Codec-filter pairs should be aligned so that pins 9 through 16 of the filter face pins 1 through 12 of the codec. This minimizes the distance for analog connections between devices and with no crossing analog lines.
13) No digital or high voltage level (such as ringing supply) lines should run under or in parallel with these analog VF connections. If the analog lines are on the top (component side) of the PC board, then GRDD, GRDA, or power supply leads should be directly under them, on the bottom to prevent analog/digital coupling.
14) Both the codec and filter devices should be shielded from traces on the bottom of the PCB by using ground or power supply leads on the top side directly under the device (like a ground plane).
15) Two +5V power supply leads (V_{CC}) should be used on each PCB, one to the filters, the other to the codecs. These leads should be separately decoupled at the PCB where they then join to a single 5V supply at the backplane connector. Decoupling can be accomplished with either a series resistor/parallel capacitor (RC lowpass) or a series RF choke and parallel capacitor of each 5V lead. The capacitor should be at least 10 μF in parallel with a 0.1 μF ceramic. This filters both high and low frequencies and accommodates large current spikes due to switching.
16) Both grounds and power supply leads must have low resistance and inductance. This should be accomplished by using a ground plane whenever possible. When narrower traces must be used, a minimum width of 4 millimeters should be maintained. Either multiple or extra large plated through holes should be used when passing the ground connections through the PCB.

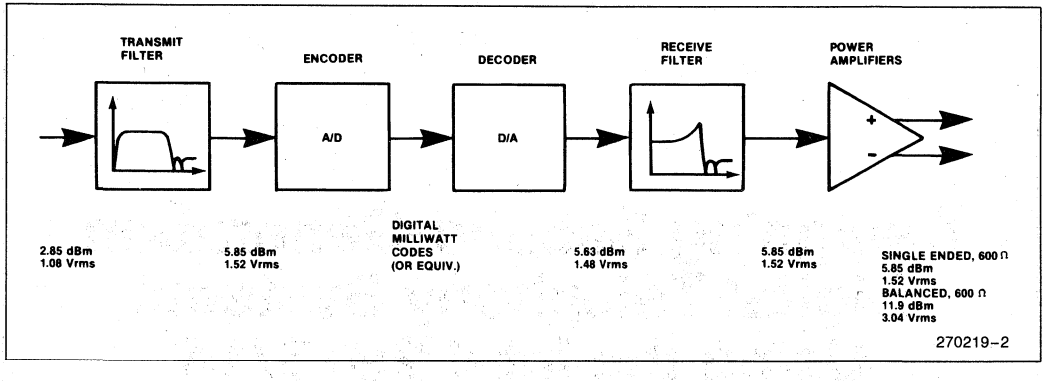
17) The 2912A PCM filter should have all power supplies bypassed to analog ground (GRDA). The 2910A/2911A Codec +5V power supplies should be bypassed to the digital ground (GRDD). This is appropriate when separate +5V power supply leads are used as suggested in item 15. The −5V and +12V supplies should be bypassed to analog ground (GRDA). Bypass capacitors at each device should be high frequency capacitors of approximately 0.1 to 1.0 μF value. Their lead lengths should be minimized by routing the capacitor leads to the appropriate ground plane under the device (either GRDA or GRDD).

18) Relay operation, ring voltage application, interruptions, and loop current surges can produce enormous transients. Leads carrying such signals must be routed well away from both analog and digital circuits on the line card and in backplanes. Lead pairs carrying current surges should be routed closely together to minimize possible inductive coupling. The microcomputer clock lead is particularly vulnerable, and should be buffered. Care should also be used in the backplane layout to prevent pickup surges. Any other latching components (relay buffers, etc.) should also be protected from surges.

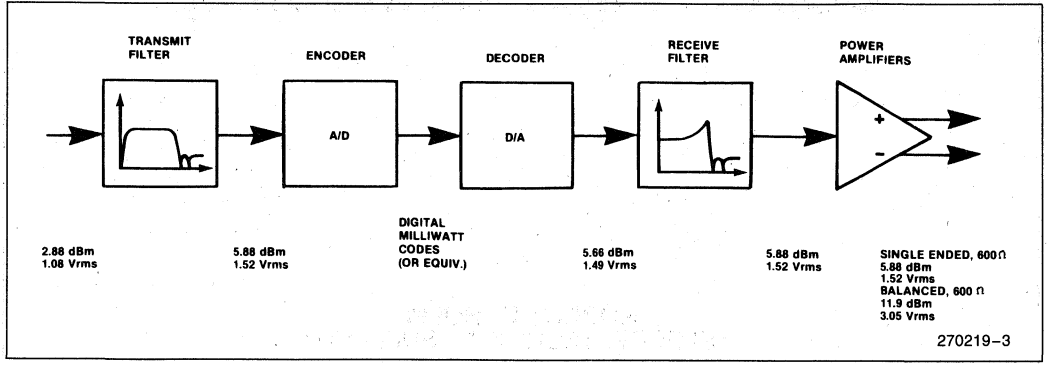
19) When not used, the AUTO pin should float with minimum PC board track area.

ZERO TRANSMISSION LEVEL POINTS

2910A/2912A 0 dBm0



2911A/2912A 0 dBm0



intel®

APPLICATION NOTE

AP-142

November 1986

Designing Second-Generation Digital Telephony Systems Using the Intel 2913/14 Codec/Filter Combochip

ROBERT E. HOLM
TELECOM TECHNICAL SUPPORT

JOHN HUGGINS
TELECOM DESIGN ENGINEERING

Order Number: 210314-002

Note: See data sheet for latest specifications. Values given in this application note are for reference only, and were considered correct at the time of publication (Feb. 1982).

1.0 INTRODUCTION

This application note describes the features and capabilities of the 2913 and 2914 codec/filter combochips, and relates these capabilities to the design and manufacturing of transmission and switching linecards.

1.1 Background

The first generation of per line codecs (Intel 2910A/11A) and filters (Intel 2912A) economically integrated the analog-digital conversion circuits and PCM formatting circuits into one chip and the filtering and gain setting circuits into another chip. These two chips helped to make possible the rapid conversion to digital switching systems that has taken place in the last few years.

The second generation of Intel LSI PCM telephony components, the 2913/14 Combochip, extends the level of integration of the linecard by combining the codec and filter functions for each line on a single LSI chip. In the process of combining both functions, circuit design improvements have also improved performance, reduced external component count, lowered power dissipation, increased reliability, added new features, and maintained architectural transparency.

The 2913 and 2914 data sheet contains a complete description of both parts, including detailed discussions of each feature and specifications for timing and performance levels. This application note, in conjunction with the data sheet, describes in more detail how the new and improved features help in the design of second-generation linecards first by comparing the two generations of components to see where the improvements have been made, and then by discussing specific design considerations.

1.2 Comparison of First- and Second-Generation Component Capabilities

The combochip represents a higher level of component integration than the devices it replaces and, because of the economics of LSI (replacing two chips with one), ultimately will cost significantly less at the component level. But comparison of the combochip block diagram with first-generation single-chip codec and filter reveals few major functional differences. Figure 1 compares the first-generation codec and filter chips to the combochip. Both provide rigidly specified PCM capabilities of voice signal bandlimiting and nonlinear companded A/D and D/A conversion. The first on-chip reference voltage was introduced in the 2910/2911 single-chip codecs and is included in the combochip. The provision of uncommitted buffer amplifiers for flexible transmission level adjustment and enhanced analog output drive was a feature of the now standard 2912 switched-capacitor PCM filter is available on the combochip. Like-

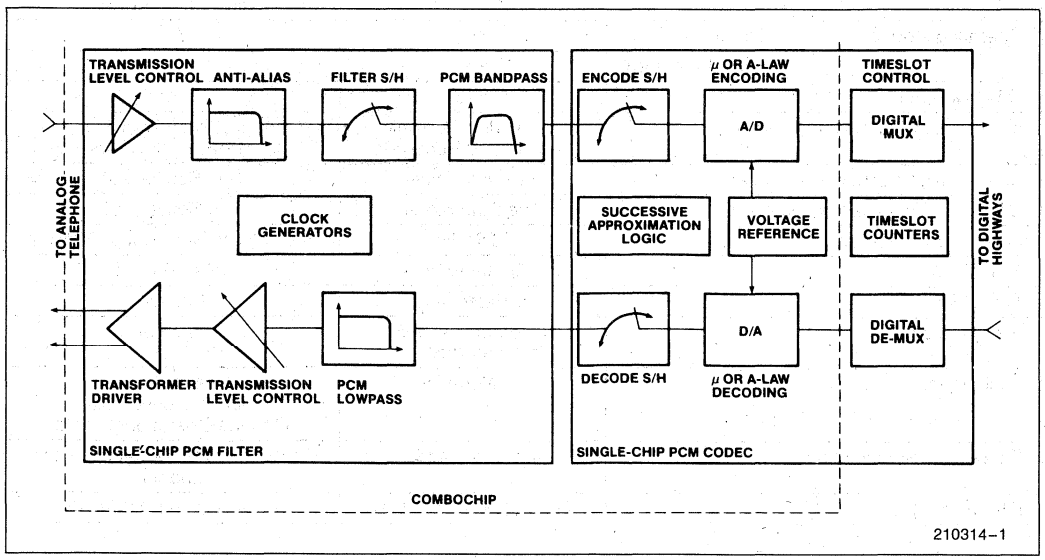


Figure 1. LSI Partitioning of Codec/Filter Functions

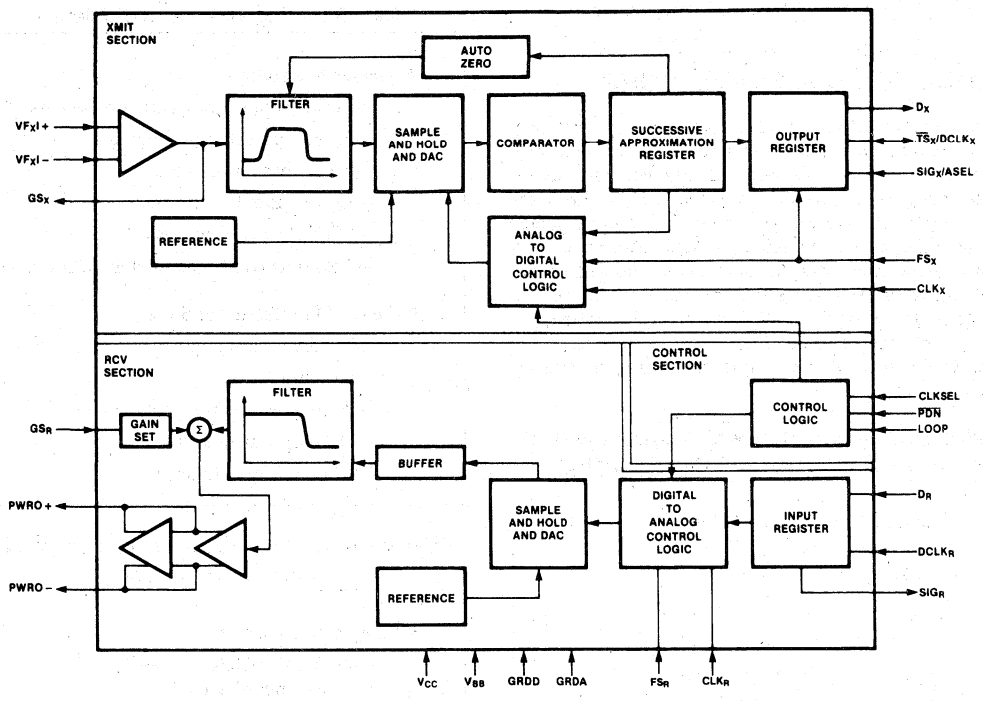
wise, independent transmit (A/D) and receive (D/A) analog voice channels which permit the two channels to be timed from independent (asynchronous) clock sources is common to the first- and second-generation devices. Finally, the ability to multiplex signalling bits on a bit-stealing basis from the digital side of the device has been duplicated on the combochip.

Data traffic-conscious systems manufacturers now provide dedicated codec, filter, and subscriber interface functions on a per-subscriber basis, which in turn puts intense cost pressures on these functions. The functional duplication of first-generation components addresses the needs of the system manufacturer who wants to cost reduce existing fixed-architecture system designs. Whereas the bulk of the system development costs (and time) are in the switching machine call processing and diagnostic software, the bulk of the production costs are in the high-volume linecards. The combochip addresses these cost pressures and defers the appetite for new integrated functions to a future generation of PCM components.

Figure 2 contains the block diagram of the 2913/14 combochip which illustrates not only the basic companding and filtering functions but also some of the changes and new features contained in the second-generation devices, such as internal auto zero, separate ADC and DAC for transmit and receive sections, respectively, precision gain setting (RCV section), and input/output registers for both fixed and variable data rates. Table 1 lists many of the features that are important to linecard design and performance. A direct comparison between first-and second-generation products

Table 1. Comparison between 2913/14 Combochip and the 2910A/11A/12A Single-Chip Codecs and Filters

Features		2910A/11A plus 2912A	2913/14
Power	Operating	280–310 mW	140 mW
	Standby	33 mW	5 mW
Pins		38–40	20–24
Board Area Including Interconnects		Normalized = 1.0	0.33
Data Rates	—Fixed	1.536, 1.544, 2.048 Mbps	Same
	—Variable	None	64 Kbps → 2.048 Mbps
Companding Law	—μ-Law	2910 + 2912	Strap Selectable
	—A-Law	2911 + 2912	
PSRR	1 KHz	30 dB	>35 dB
	> 10 KHz	Not Spec'd	> 35 dB
Gain Setting		Trim Using Pot Necessary	Precision Resistors Eliminate Trim Req.
Operating Modes	Direct	Yes	Yes
	Timeslot Assign	Yes	No
On-Chip V_{REF}		Yes	Yes
ICN — Half Channel Improvement		15 dBrnc0 Transmit 11 dBrnc0 Receive	15 dBrnc0 Transmit 11 dBrnc0 Receive
S/D — Half Channel Improvement		See Data Sheet	See Section 2.0
GT — Half Channel Improvement		See Data Sheet	See Section 2.0
Power Down (Standby)		PDN Pin	Frame Sync Removal or PDN Pin
Signalling		2910-8th Bit	2914-8th Bit
Auto Zero		External	Internal
S & H Caps		External Transmit Internal Receive	Internal
Test Modes		None	Design Tests Manufacturing Test On-Line Operational Tests
Encoder Implementation		Resistive Ladder	Capacitive Charge Redistribution Ladder
Filter/Gain Trim		Fuse Blowing ±0.2 dB	Fuse Blowing ±0.04 dB



(a) Combochip Block Diagram

V_{BB}	Power (−5V)	GS_X	Transmit Gain Control
PWRO+, PWRO−	Power Amplifier Outputs	VF_XI-, VF_XI+	Analog Inputs
GS_R	Receive Gain Control	GRDA	Analog Ground
$\overline{PDN}$	Power Down Select	NC	No Connect
CLKSEL	Master Clock Frequency Select	SIG_X	Transmit Signaling Input
LOOP	Analog Loop Back	ASEL	μ- or A-law Select
SIG_R	Receive Signaling Bit Output	$\overline{TS}_X$	Timeslot Strobe/Buffer Enable
$DCLK_R$	Receive Variable Data Clock	$DCLK_X$	Transmit Variable Data Clock
D_R	Receive PCM Input	D_X	Transmit PCM Output
FS_R	Receive Frame Synchronization Clock	FS_X	Transmit Frame Synchronization Clock
GRDD	Digital Ground	CLK_X	Transmit Master Clock
V_{CC}	Power (+5V)	CLK_R	Receive Master Clock

(b) Combochip Pin Names

Figure 2. Block Diagram of 2913/14 Combochip

shows the significant improvement in the combochip both in performance levels and system flexibility.

2.0 DESIGN CONSIDERATIONS

The key point with the 2913/14 is that it will result in a linecard that performs better and costs less than any two-chip codec/filter solution. The lower cost results from many factors, as seen in Table 2. Both direct replacement costs and less tangible design and manufacturing time savings combine to yield lower recurring and nonrecurring costs. As an example, the wider margins to transmission specs and the higher power supply rejection ratios of the 2913/14 will both shorten the design time needed to build and test the linecard prototype and reduce the reject rate on the manufacturing line.

Table 2. 2913/14 Factors which Lower the Cost of Linecard Design and Manufacturing

- Lower LSI Cost (2914 vs. 2910/11 + 2912)
- Fewer External Components
- Less Board Area
- Shorter Design/Prototype Cycle
- Better Yields/Higher Reliability
- Lower Power/Higher Density

Part of the recurring cost of linecard production is the efficiency of the manufacturing line in turning out each board. This is measured in both parts cost and time. Average manufacturing time is strongly effected by the line yield, i.e., the reject rate reliability. A linecard using the 2913/14 has many labor-saving features, which also increases the *reliability* of the manufacturing process. Some of these features are detailed in Table 3.

The combination of fewer parameters to trim (gain, reference voltage, etc.), tolerance to wider power supply variations, and on-chip test modes make the linecard very manufacturable compared to first-generation designs.

Probably the most obvious improvement in linecard design based around the 2913/14 is the reduction in linecard PCB area needed compared to two-chip designs. The combination of the codec and filter into a single package alone reduced the LSI area by one-third. Table 4 shows many of the other ways in which board area is conserved. In general, it reduces to fewer components, more on-chip features, and layout of the chip resulting in an efficient board layout which neatly separates the analog and digital signals both inside the chip and on the board.

Table 3. 2914 Factors which Increase Linecard Manufacturing Yields and Efficiency

- Higher Reliability
  - —Fewer connections and components
  - —More integrated packaging
  - —More margin to specs
  - —Lower power
  - —NMOS proven process
  - —Less sensitive to parameter variations
- Fewer Manufacturing Steps
  - —No gain trimming
  - —On chip V_{REF}
  - —Wide power supply tolerance
  - —On chip test modes
  - —Wide margins to spec

Table 4. Design Factors for 2914 which Reduce Linecard PCB Area

- Integrated Packaging
  - —2914 vs. 2910/11 + 2912 = 1/3 board area
  - —2913 takes even less space
- Fewer Interconnects/Components
  - —Codec/filter combined
  - —On-chip reference voltage
  - —On-chip auto zero
  - —On-chip capacitors
  - —No gain trim components
  - —No voltage regulators
- Efficient Layout (Facilitates Auto Insertion)
  - —Analog/digital sections separated on chip
  - —Digital traces can cross under chip
  - —Two power supplies only
  - —Low power/high density

Table 5. 2913/14 Operating Mode Options Add Flexibility to Linecard Design

Option	Mode Control Pins	Results of Mode Selection	
		2914 (24 Pin)	2913 (20 Pin)
Companding Law	SIGX/ASEL	A-Law or μ-Law + Signalling	A-Law/μ-Law, no Signalling
Power Down	$\overline{PDN}$	Transmit & Receive Side Go To Standby Power (5 mW)	
	FS_X & FS_R Removed	Same (12 mW)	
	FS_X Removed	Transmit Side Goes to Standby (110 mW)	
	FS_R Removed	Receive Side Goes to Standby (70 mW)	
Data Rate	= V_{CC}/GRDD/V_{BB} $DCLK_R$ = V_{BB}	1.536/1.544/2.048 Mbps in Fixed Data Rate Mode	
	= V_{CC}/GRDD/V_{BB} $DCLK_B$ = Clock	Variable Data Rate Mode from 64 Kbps to 2.048 Mbps, No Signalling	
Test Modes	LOOP = V_{CC}	Implements Analog Loopback	No Loopback Capability
	$\overline{PDN}$ = V_{BB}	Provides Access to Transmit Codec Through ASEL and $\overline{TSX}$ Pins	
	D_R = V_{BB}	Provides Access to RCV Filter Input at $DCLK_R$ and Transmit Filter Outputs at ASEL and $\overline{TSX}$ Pins	

Many of the factors discussed—which result in efficient, cost-effective linecard designs—are discussed in more detail both in the 2913/14 data sheet and in the following sections of this note.

2.1 Operating and Test Mode Selection

A key to designing with the 2913/14 combo is the wide range of options available in configuring, either with strap options or in real time, the different modes of operation. The 2913 combochip (20 pins) is specifically aimed at synchronous switching systems (remote concentrators, PABXs, central offices) where small package size is especially desirable. The 2914 combochip (24 pins) has additional features which are most suitable for applications requiring 8th-bit signalling, asynchronous operation, and remote testing of transmission paths (e.g., channel banks). Once the specific device is selected, there is a wide range of operating modes to use in the card design, as seen in Table 5. This table lists the optional parameters and the pins which control the operating mode. The result of selecting a mode is listed for both the 2913 and 2914.

The purpose of offering these options is to ensure that the 2913/14 combo will accommodate any existing linecard design with architectural transparency. At the same time, features were designed in to facilitate design and manufacturing testing to reduce overall cost of development and production.

2.2 Data Rate Modes

Any rapid conversion scenario presumes that the combochip will fit existing system architectures (retrofit) without significant system timing, control, or software modifications. To this end, two distinct user-selectable timing modes are possible with the combochip. For purposes of discussion, these are designated (a) fixed data rate timing (FDRT) and (b) variable data rate timing (VDRT).

FDRT is identical to the 2910/2911 codec timing in which a single high-speed clock serves both as master clock for the codec/filter internal conversion/filtering functions and as PCM bit clock for the high-speed serial PCM data bus over which the combochip transmits and receives its digitized voice code words. In this mode, PCM bit rates are necessarily confined to one of three distinct frequencies (1.536 MHz, 1.544 MHz, or 2.048 MHz). Many recently designed systems employ this type of timing which is sometimes referred to as burst-mode timing because of the low duty cycle of each timeslot (i.e., channel) on the time division multiplexed PCM bus. It is possible for up to 32 active combochips to share the same serial PCM bus with FDRT.

VDRT (sometimes referred to as shift register timing), by comparison, utilizes one high-speed master clock for the combochip internal conversion/filtering functions and a separate, variable frequency, clock as the PCM bit clock for the serial PCM data bus. Because the serial PCM data rate is independent of internal conversion timing, there is considerable flexibility in the choice of PCM data rate. In this mode the master clock is permitted to be 1.536 MHz, 1.544 MHz, or 2.048 MHz, while the bit clock can be any rate between 64 KHz and 2.048 MHz. In this mode it is possible to have a dedicated serial bus for each combochip or to share a single serial PCM bus among as many as 32 active combochips.

Thus, the two predominant timing configurations of present system architectures are served by the same device, allowing, in many cases, linecard redesign without modification of any common system hardware or software. Additional details relating to the design of systems using either mode are found in section 3.0.

2.3 Margin to Performance Specifications

The combochip benefits from design, manufacturing, and test experience with first-generation PCM products on the part of the system manufacturer, component suppliers, and test equipment suppliers. The sub-millivolt PCM measurement levels and tens of microvolts accuracy requirements on the lowest signal measurements often result in tester correlation problems, yield losses, and excess costs for system and PCM component manufacturers alike. Thus additional performance margin built into the PCM components themselves will have its effect on line circuit costs even though the system transmission specifications may not reflect the improved performance margin.

Half channel measurements have been made of the transmission parameters—gain tracking (GT), signal to distortion ratio (S/D), and idle channel noise (ICN).

Gain Tracking—Figure 3 shows the gain tracking data for both the transmit and receive sides of the combo using both sine wave testing (CCITT G712.11 Method 2) and white noise testing (CCITT G712.11 Method 1). The data shows a performance very nearly equal to the theoretically best achievable using both test techniques. End to end measurements, although not spec'd, also show a corresponding good performance with errors less than or equal to the sum of the half channel values.

Signal to Distortion Ratio—This is a measure of the system linearity and the accuracy in implementing the companding codes. Figure 4 shows the excellent perfor-

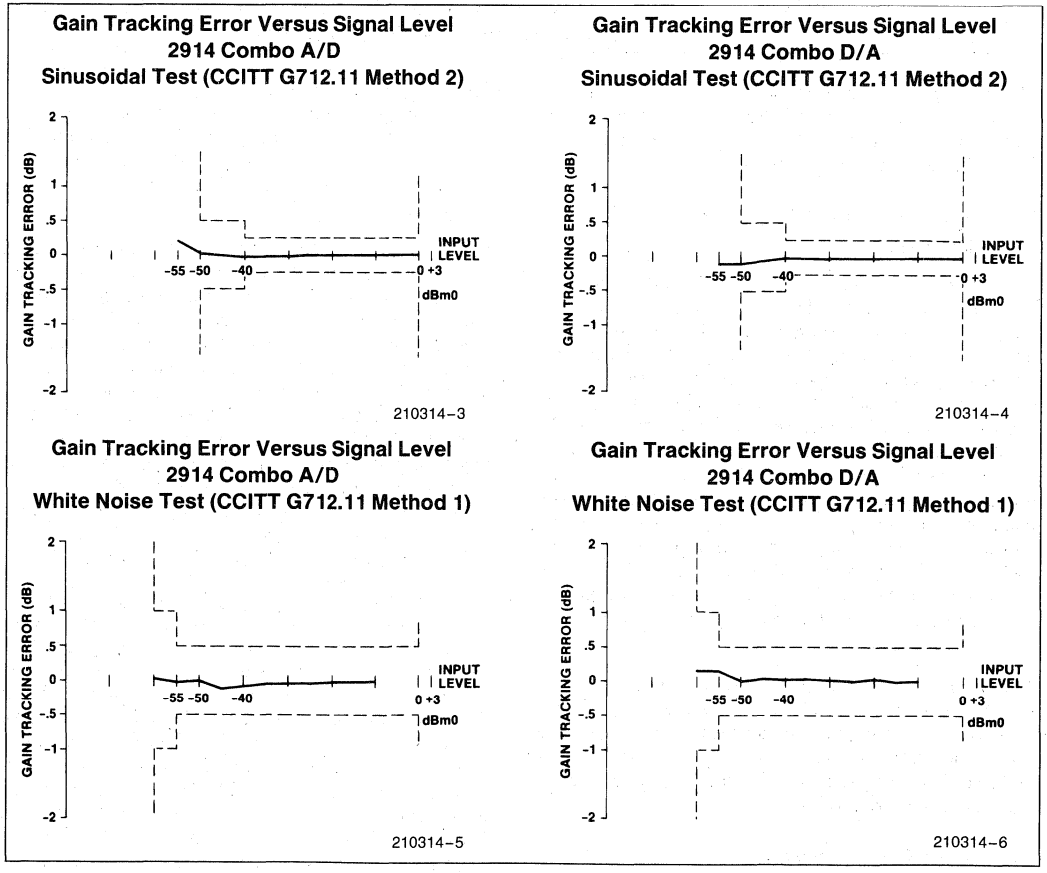


Figure 3. 2914 Half Channel Gain Tracking Performance Measurements for Both Sine and Noise Testing

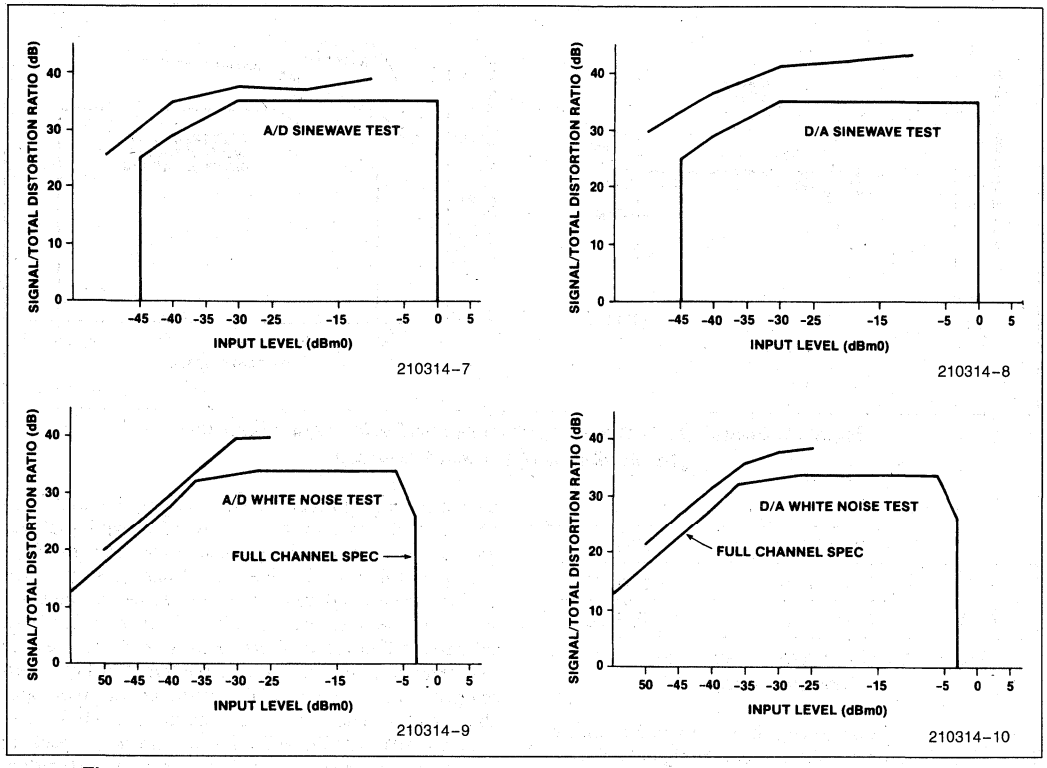


Figure 4. 2914 Half Channel Signal to Distortion Ratio (S/D) Performance Measurements for Both Sine and Noise Testing

mance of the 2914 for both the transmit (A/D) and receive (D/A) channels using sine wave and noise testing. The margin is greater than 3 dB above the half channel spec which means that a larger error budget is available to the rest of the channel.

Statistical Analysis—A statistical analysis of G.T. and S/D measurements over many devices shows a very tight distribution, as seen in Figure 5. There are several consequences resulting from this highly desirable distribution: (1) the device performance is controllable, resulting in high yields, (2) the device circuit design is tolerant of normal process variations, thereby ensuring predictable production yields and high reliability, and (3) understanding of the circuit design and process fundamentals is clearly demonstrated—largely as a result of previous telephony experience with the Intel NMOS process.

Idle Channel Noise—The third transmission parameter is idle channel noise (ICN). Figure 6 gives half channel ICN measurements which show a substantial margin to specification.

Power Supply Rejection—Circuit innovation in the internal combochip design has resulted in significant improvements in power supply rejection in the 5 to 50 KHz range (Figure 7), and it is this frequency band which usually contains the bulk of the switching regulator noise. These higher frequencies, outside the audio range as they are, are not objectionable or even detectable in the transmit direction except to the extent that they alias into the audio range as a result of internal sampling processes in the transmit filter and A/D converter. Sampling techniques in the combochip minimize this aliasing. In the receive direction, excess high frequency noise which propagates onto the subscriber loop can interfere with signals in adjacent wires and is thus objectionable even without aliasing. The symmetrical true differential analog outputs of the combochip are an improvement from earlier designs which failed to maintain true power supply symmetry through the output amplifiers. Not only does the differential design improve transmission performance, but it also reduces the need for power supply bypass capacitors, thereby saving component cost on the linecard.

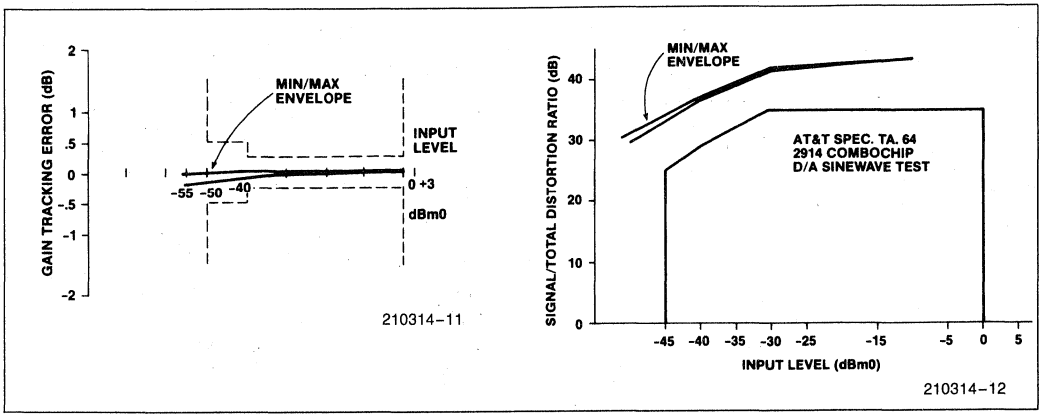


Figure 5. Statistical Analysis of Transmission Performance Showing Tight Distribution Over Many Devices

	Weighting	ICN
A/D	C Message	15 dBrnC0
D/A	C Message	11 dBrnC0

Figure 6. 2914 Idle Channel Noise (ICN) Measurements

Autozero—The autozero circuit is contained completely on-chip. It automatically centers the signal/noise distribution at the encoder input. This ensures minimal ICN due to bit toggling and also maintains maximum sensitivity to the AC signals of interest.

2.4 Power Conservation

Figure 8 illustrates typical power consumption and office equipment dissipation for a resistive line biasing arrangement (with no loop current limiting) and for the per-line PCM components. It can be seen that overall line circuit power consumption and dissipation are strong functions of subscriber loop resistance, and are dominated by line biasing current regardless of loop length. It can also be seen that the combochip achieves significant reductions in PCM component contributions relative to both the 2910A/2912A and 2910/2912. Present residential traffic characteristics are such that the PCM components are active less than 10% of the time, and in its low-power standby state, the combochip power dissipation drops to typically 5 mW as the line current (and dissipation) goes to its background on-hook leakage level of typically a few milliwatts (but for very leaky lines, as much as 50 mW–500 mW).

The concern for linecard power consumption and dissipation is related both to the cost of providing power and to the system density problem involving convection heat removal from the linecards. Consequently, much recent line circuit development activity centers on elimination of the inefficient resistive line current feed both by current limiting in short loops and by more exotic and expensive per-line dc-dc converters. For both present-generation designs and cost-reduction redesigns, the typical combochip dissipation of 140 mW active/5 mW standby will allow system board packing density improvements and power supply cost reductions.

A closer look at the effect of loading (duty cycle) on the average power dissipation of a combochip is given in Table 6. Typical loading percents run as low as 5% for very large switching systems (thousands of lines) up to 100% in nonswitching applications such as channel banks. Clearly, the average power dissipation in a typical switching system is below 35 mW which facilitates board packing density and cost of power considerations.

Table 6. Typical Power Dissipation Per Line Using 2914 Combochip

	Duty Cycle	Power Dissipation
Central Office	5%	12 mW
PABX	15%	25 mW
Peak Hour C.O.	50%	73 mW
Channel Bank	100%	140 mW

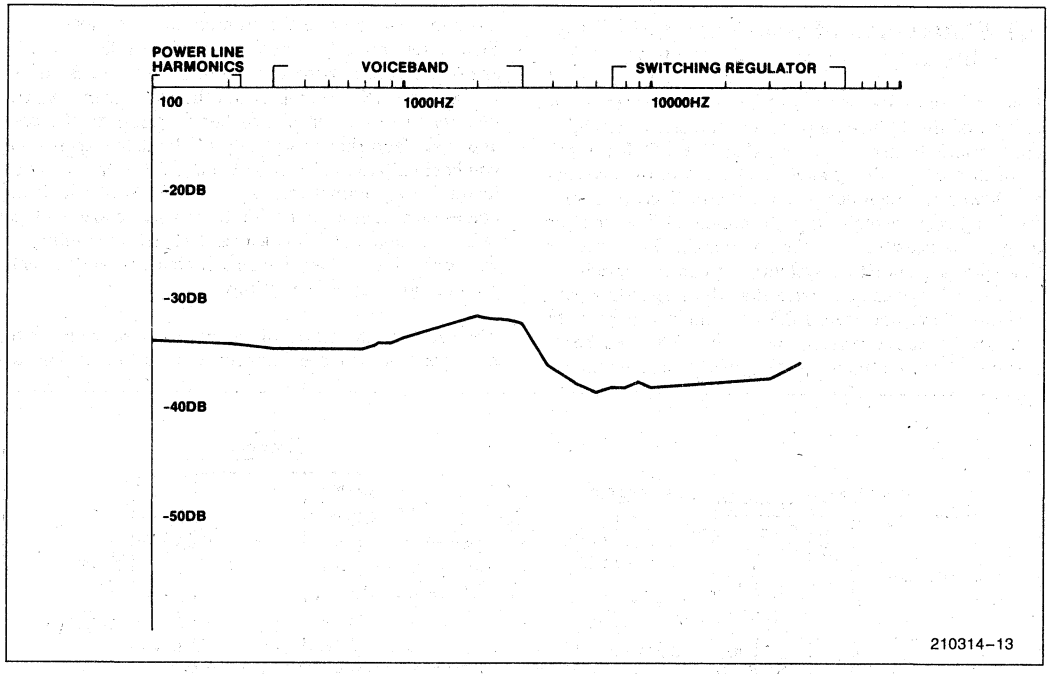


Figure 7. Wideband 2914 Power Supply Rejection Ratio (PSRR)

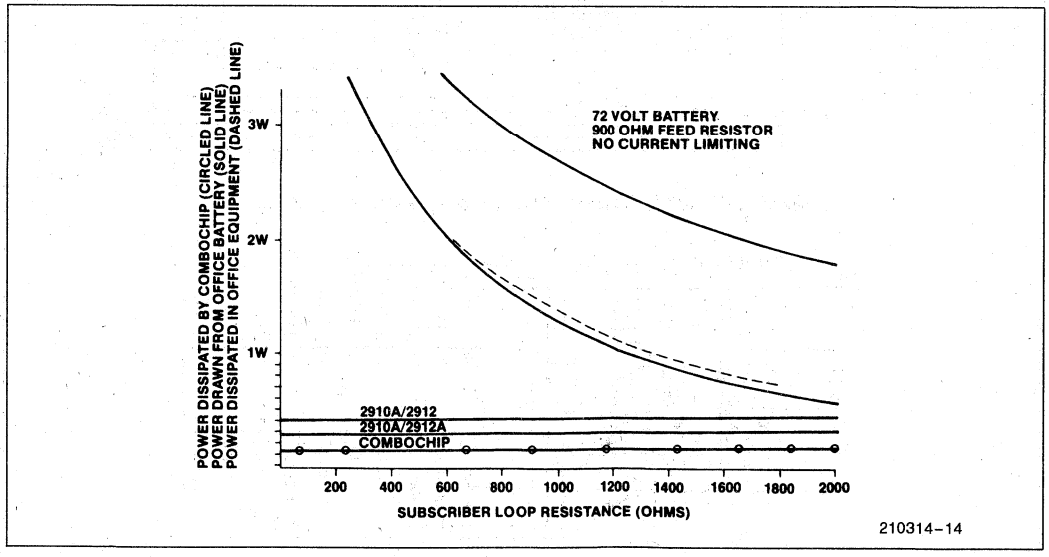


Figure 8. Line Circuit Power Consumption and Dissipation Curves

2.5 Elimination of Gain Trim in the Line Circuit

Four resistors—R1–R4 of Figure 9—on the transformer side of the PCM components are used to establish appropriate transmission levels at the PCM components and are, at first glance, equivalent in the two cases. However, a significant reduction in linecard manufacturing costs associated with individual line trim (or mop-up) is possible with the combochip. The need for this trim is dictated by system gain contrast specifications which typically require that the line-to-line gain variation shall not exceed 0.5 dB, which translates to 0.25 dB for each (transmit and receive) channel. Table 7 shows that the major portion of this gain variation has previously been in the nominal insertion loss of the PCM filter and in the uncertainty of the reference voltage of the codec. With this cumulative 0.15 dB uncertainty in the PCM components themselves, the system manufacturer had no choice but to resort to the cost and manufacturing complexity of the active trim. The combochip, however, can be trimmed during its manufacture to a nominal tolerance of ±0.04 dB which includes uncertainties in both the filter and codec voltage reference functions. This leaves 0.21 dB uncertainty to variations in the other line circuit elements and to temperature and supply variations.

The variation in combochip gain with supply and temperature has also been improved to allow as low as

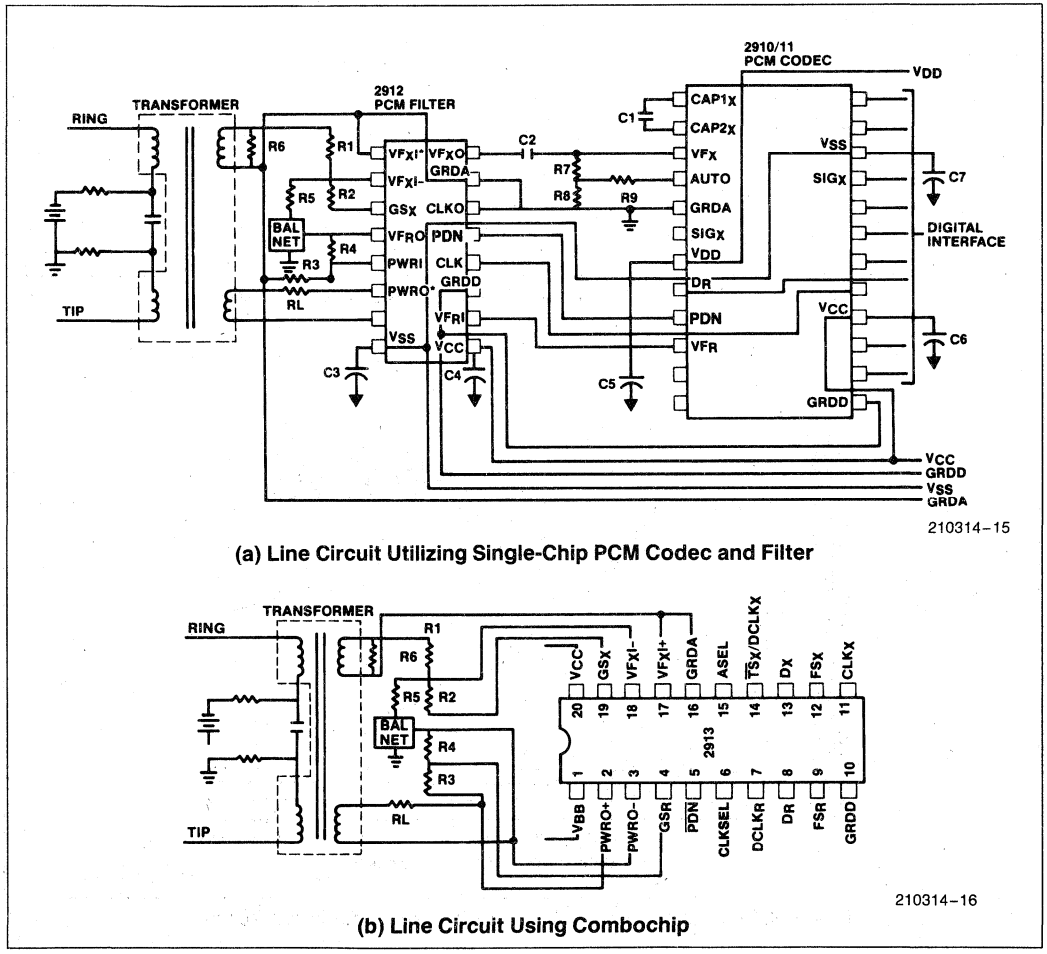


Figure 9. Schematics of the Codec/Filter Function and the 2/4 Wire Hybrid Transformers

Table 7. Gain Trim Budget for Codec/Filter Functions

Device	Manufacturing Uncertainty (Initial)	ΔT ΔSupplies	Total	Variation* Budget for Other Components
2910	±0.1	±0.1		
2912	±0.05	±0.05		
	±0.15	±0.15	±0.3 dB	0 dB
2914	±0.04	±0.08	±0.12 dB	±0.13 dB

*Assumes 0.5 dB end to end gain contrast specifications.

0.08 dB variation over supplies and temperature so that more than half the system specification could be reserved for transformer, wiring, and resistor uncertainties. This possibility of using fixed precision gain trim components and abandoning the active trim holds the potential for simplification and cost reduction of the line board manufacturing process.

2.6 Power Up/Down Considerations

Power Supply Sequence—There are no requirements for a particular sequence of powering up the combochip. All discussions of power up or power down timing assume that both V_{CC} and V_{BB} are present.

Power Up Delay—Upon application of power supplies, or coming out of the standby power down mode, three circuit time constants must be observed: (1) digital signal timing, (2) autozero timing, and (3) filter settling. An internal timing circuit activates SIF_r, D_X, and TS_X approximately two or three frames after power up. Until this time, SIG_r is held low and the other two signals are in a tri-state mode. During this time, SIG_X will have no effect on the PCM output.

Power Down Modes—These modes are described in detail in Table 3 of the 2913/14 data sheet except for a fail-safe mode in case CLK_X is interrupted. If this should happen, both D_X and TS_X go into the tri-state mode until the clock is restored. This ensures the safety of the PCM highway should the interrupted clock be a local problem.

3.0 OPERATING MODES

There are three basic operating modes that are supported by the 2913/14: fixed data rate timing (FDRT), variable data rate timing (VDRT), and on-line testing.

3.1 Fixed Data Rate Mode

The FDRT mode is described in some detail in both section 2.2 of this note and in the 2913/14 data sheet. In addition, Intel Application Note AP-64 (Data Conversion, Switching, and Transmission using the Intel 2910A/2911A codec and 2912 PCM filter) also describes the basics of using the fixed data rate mode for first-generation codecs and filters which is essentially the same as for the 2913/14 second-generation combochip.

3.2 Variable Data Rate Mode

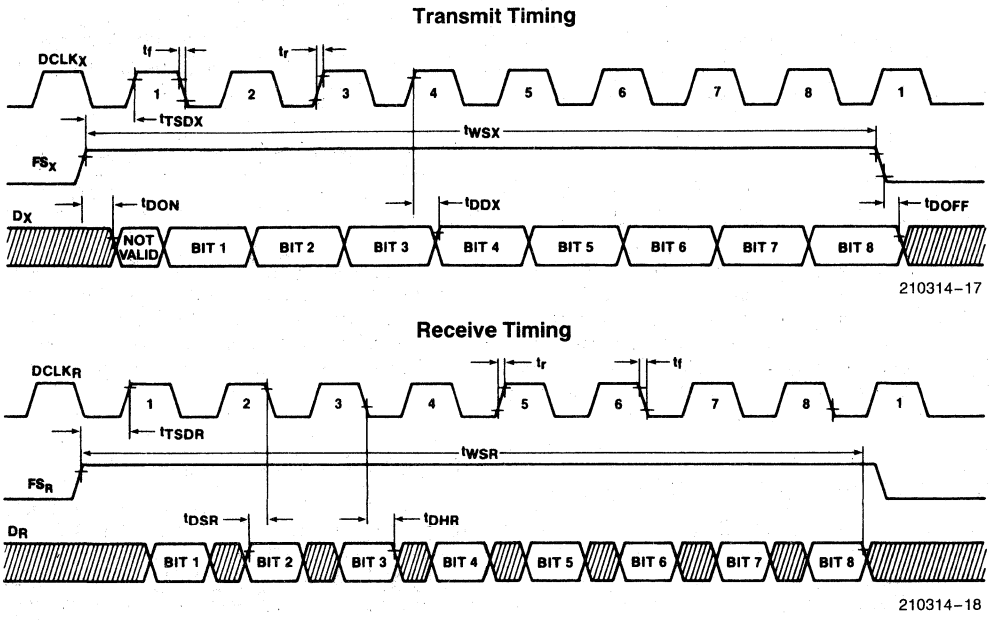
The VDRT mode is described in some detail both in section 2.2 and in the 2913/14 data sheet. This section focuses on two design aspects: (1) the advantage of clocking data on the rising edges of the clock for transmit and receive data, respectively, and (2) making the 2913/14 transparent in previously designed systems (a retrofit, cost reduction redesign).

Clock Timing—The 2913/14 is ideally set up to transmit and receive data, using the same clock, with no race conditions or other marginal timing requirements. This is accomplished by transmitting data on the rising edge of the first clock pulse following the data enable pulse FS_X and receiving data on the falling edge of the clock which is directly in the middle of the D_X data pulse. Several manufacturers use leading edge timing for both transmit and receive requiring an inversion of the receive clock.

Figure 10 shows the transmit and receive clock and data timing for an entire time slot of data. A closer look at the timing functions is given in Figure 11 which looks specifically at the first clock cycle after the transmit data enable FS_X.

According to the 2913/14 data sheet, the frame sync/data enable FS_X must precede the clock ($DCLK_X$) by at least T_{tsdx} or nominally 15 ns for that clock pulse to be recognized as the first clock pulse in the time slot. In actuality, the 2914 will allow FS_X to lag up to 80 ns the $DCLK_X$ rising edge and recognize it as the first clock pulse in a 2.048 MHz system.

Once FS_X has reached V_{IH} of about 2V, the D_X output will remain in the tri-state high-impedance mode for



NOTE:
All timing parameters referenced to V_{IH} and V_{IL} except t_{DON} and t_{OFF} which reference a high impedance state.

Figure 10. Variable Data Rate Timing for an Entire Time Slot

T_{don} or about 34 ns longer. It then comes out of tristate and will represent some data which is invalid until the valid data is available T_{DDX} or about 75 ns (100 ns worst case) after the clock rising edge. This means there is about 90 ns of invalid data after the tri-state mode. At this point there is valid data on the D_X highway that lasts for approximately one full clock cycle.

Since the D_X highway is tied directly to the D_r highway in digital loopback, the valid data above is now available to the receive channel with some propagation delay. The receiver is only interested in the data for about a 50 ns (110 ns worst case) window centered about the falling edge of the $DCLK_r$ clock which occurs about half a clock cycle from the FS_r rising edge. The window width is equal to the data set-up time. T_{dsr}, plus the clock fall time, T_f, plus the data hold time, T_{dhr}. Information at any other time on the D_r highway falls into the DON'T CARE category.

Retrofitting the 2913/14—Several switching/transmission systems have been designed using first-generation codecs which operate at data rates from 64 Kbps to 2.048 MBps. In addition, they may have been designed using the rising clock edges for both transmit and receive data.

Other aspects of these older designs could be relative skewing between the sync pulses (Data Enable) and the clock pulses in such a way that the sync pulse occurs after (Lags) the first clock pulse rising edge. All of these conditions can be easily handled using the variable data rate timing mode of the 2913/14 plus some simple external logic. By the addition of this logic, the 2913/14 becomes transparent to the older design thereby allowing an upgrade in performance while having no impact on backplane wiring or on system control hardware/software. In addition, many of the features of the 2913/14 may be incorporated, such as the test modes, which provide additional capabilities beyond those available in the original design and at a lower cost.

The circuit diagram in Figure 12 shows the maximum amount of additional random logic that could be necessary to make the 2913 or 2914 completely transparent at the linecard level (no impact on backplane wiring or timing). The inverter on $DCLK_R$ inverts all the receive clocks for each linecard. This inverter is only needed if (1) the transmit and receive clocks are inverted at the system/backplane level (as opposed to the linecard level) and (2) the previous design used only rising (or falling) edges to clock the transmit/receive data.

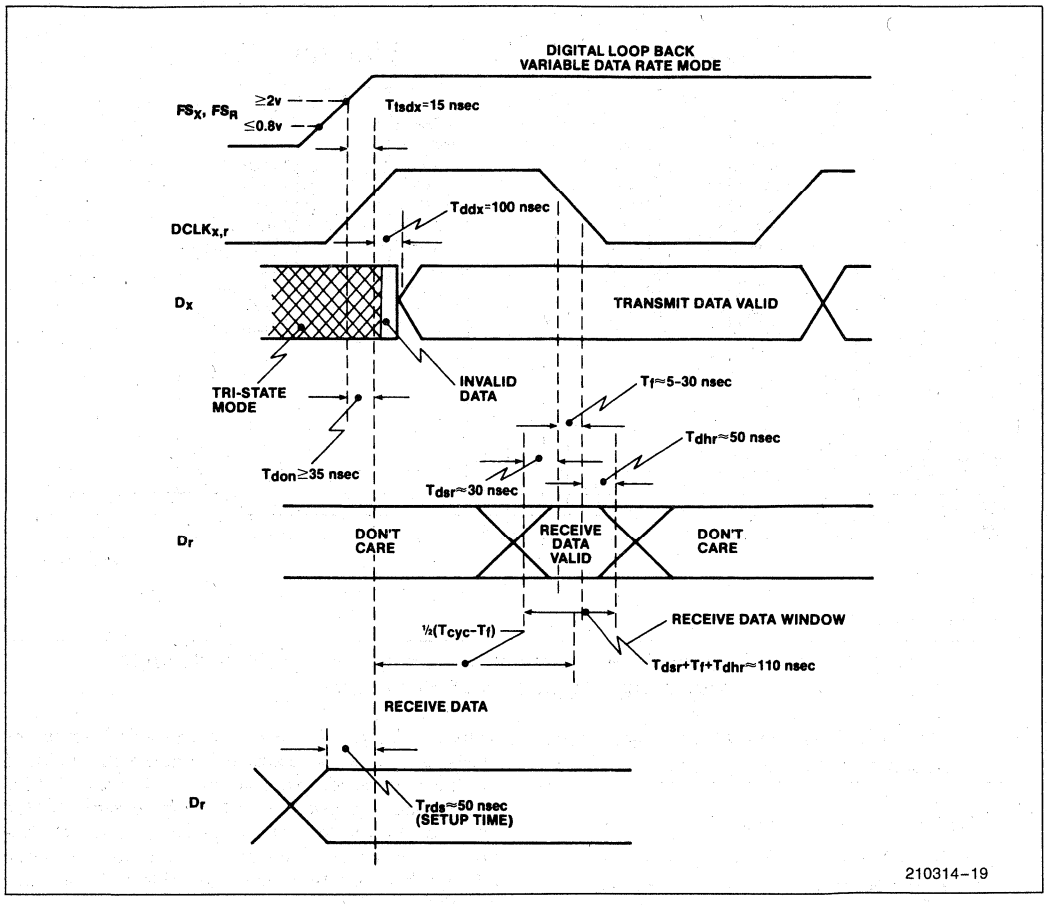


Figure 11. Waveform Timing Diagrams for the 2913/14

3.3 On-Line Test Modes

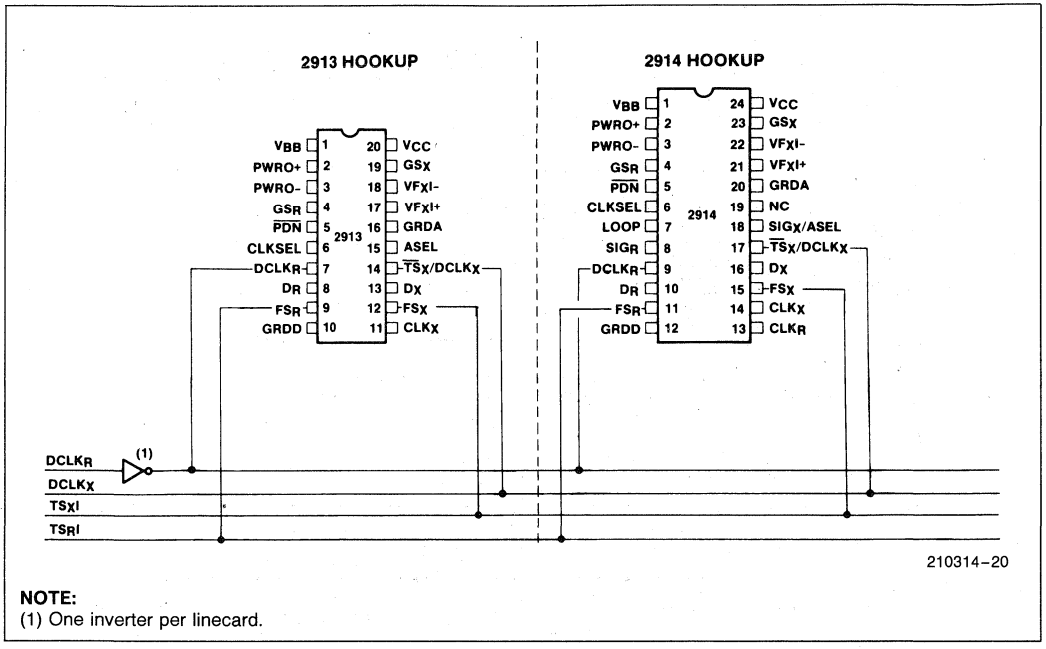
Two modes are available which permit maintenance checking of the linecard up to the SLIC/combochip interface, including the PCM highways and time slot interchanges. Tests include time slot-dependent error checking. The two test modes are called "redundancy testing" and "analog loopback." These test modes are described in detail in Section 4.3.

4.0 MULTIMODE TEST CAPABILITIES

The 2913/14 was designed with every phase of design, manufacturing, and operation taken into consideration. In particular, several test modes have been implemented within the device with essentially no increase in the package size or pin count. These test modes fall into three categories: design/prototype tests, manufacturing tests, and on-line operation tests; see Table 8.

4.1 Design/Prototype Testing

In the design of a linecard prototype or in the qualification of a device, it is often helpful to have direct access to the internal nodes at key points in the LSI system. Some manufacturers even dedicate pins specifically for this function. The Intel 2913/14 approach was to reduce cost by using multifunction pins and smaller packages to achieve this goal. Measurements through these multipurpose pins will typically yield full device capability against performance specifications, however *these measurements are not included in the device specifications.* This is done for two reasons: first, to save manufacturing cost by eliminating unnecessary tests and specifications, and, second, more cost effective manufacturing test techniques are available, as discussed in section 4.2.



NOTE:
(1) One inverter per linecard.

Figure 12. Circuit Diagram Showing Connections Needed to Retrofit the 2913/14 into Existing Variable Data Rate Systems

Table 8. Multimode Testing for Each Level from Design to On-Line Operation

- Design/Prototype Testing
  - — Direct access to transmit codec inputs
  - — Direct access to the receive filter input and the transmit filter differential outputs
- Manufacturing Tests
  - — Standard half channel tests for combined codec/filters
  - — Filter response half channel measurements
- Operation On-Line Tests
  - — Analog loopback for testing PCM and codec analog highways
  - — Redundancy checks with repeatable D_X outputs

Table 9 gives the input control pin values and the corresponding functions assigned to the key test pins on the 2914 for the design test modes.

Transmit Coded (Encoder)—The transmit filter can be bypassed by directly accessing the differential input of the transmit encoder with an analog differential drive signal. Table 9 shows the control pin voltages and the input pins for this test. This test mode permits DC testing of the encoder which is otherwise blocked by the AC coupling (low frequency reject filter) of the transmit filter.

Transmit and Receiver Filter—Table 9 shows the control values that permit access to the differential outputs of the transmit filter and the single-ended input to the receive filter. The voltage difference between the transmit filter outputs represents the filtered output that will be encoded. By driving VF_XI (single ended or differentially), the transmit filter response is obtained as a differential output. The final stage is the 60 Hz reject filter which is a switched capacitor filter sampled at an 8 KHz rate. When measured *digitally* (after the encoder), the filter characteristic is obtained directly: however, when measured in analog, a sin $(\omega T/2)/\omega T/2$ correction factor must be included.

Table 9. 2914 Test Functions and Control Inputs for the Design Test Modes

Input		Pin Function (24-Pin)			Test Function
$\overline{PDN}$	DR	Pin 9 $DCLK_R$	Pin 17 $TS_X/DCLK_X$	Pin 18 $SIG_X/ASEL$	
$0-V_{CC}$	$0-V_{CC}$	$DCLK_R$	$TS_X/DCLK_X$	$SIG_X/ASEL$	Normal Operation
V_{BB}	$0-V_{CC}$	—	+VFX	−VFX	Encoder
$0-V_{CC}$	V_{BB}	VFRI	+VFX0	−VFX0	RCV, XMIT Filter

NOTES:
The terms used above are defined as:
±VFX = Encoder Input
±VFX0 = XMIT Filter Output
VFRI = RCV Filter Input

The input to the receive filter first passes through a sample and hold. This is necessary to simulate the sin $(\omega T/2)/\omega T/2$ characteristic that results from the decoder D/A output. The net result is a filter characteristic that can be compared directly to the specifications.

Start-up Procedure for Test Modes—To place the 2913/14 in the test mode it is first necessary to operate the device for a few ms in normal operation. Then V_{BB} can be applied to the control pins to select the desired test access.

4.2 Production Testing

While it may be convenient for the designer to have access to both the filter and the codec inputs and outputs during the design or evaluation phase the final product will always use the filter and codec circuits together with all signals passing through both on the way to or from the PCM highways. It therefore makes sense to perform all manufacturing measurements with the device configured in its normal operating mode, i.e., all measurements should be complete filter/codec half channel measurements. This approach not only tests the combo as it will actually be used, but also saves time and money by eliminating separate measurements and correlation exercises to determine the full half channel performance.

Since the transmission specifications of S/D, gain tracking, and ICN all require measurements which are "in-band" or "filter independent," the codec functions can be easily tested using conventional half channel measurement equipment. The apparent difficulty arises in trying to fully measure the filter characteristics beyond the half sampling frequency of 4 KHz. In fact, this is not really a problem with today's computer-based testing plus an understanding of the sampled data process which is discussed under "Filter Testing".

ENCODER/DECODER TESTING

Transmission specifications are AC-coupled in-band measurements when using either CCITT G.712.11 methods 1 & 2 (white noise testing and sinusoidal testing, respectively) or AT&T Pub 43801 (Sinusoidal Testing). The noise testing uses a narrowband of flat noise from 300 to 500 Hz to drive the filter/codec (either in analog or the equivalent digital sequence for the transmit/receive channels, respectively). The resulting harmonic products are used to determine S/D. Likewise, gain tracking is also determined from this signal input. Sinusoidal testing uses a tone at 1.020 KHz for S/D measurements and gain tracking measurements. Idle channel noise measurements require the combined filter/codec since it has long been shown that separate measurements of filters and codecs are difficult to relate to the combined measurement (usually there is no specific relationship because of the non-linear properties of the encoder/decoder operations). Typically the frequency response of ICN measurements is primarily determined by the weighting filter (either C message or psophometric, which are both AC-coupled, bandpass type filters).

The conclusion is that combined filter/codec testing in no way limits the measurement of half channel transmission parameters of S/D, G.T., or ICN.

FILTER TESTING

Testing the filter response, of the transmit and receive channels presents two separate test situations which, in some ways, are mirror images of one another. With the transmit side, signals may be introduced at any frequency to test the filter response. At the output of the filter, the resulting signals are sampled at 8 KHz and digitized resulting in a sequence of PCM words representing the samples of filtered input signal. On the receive side, a digital PCM sequence of samples representing the driving signal is converted to an analog signal by the decoder and can be measured at the filter output in analog form.

Sampling Process—In both cases of testing the filter, the signal eventually is in a sampled form. Since the sampling rate is fixed at 8 KHz, all signals must be represented below 4 KHz (half the sampling frequency). This means that the PCM bit stream can only represent signals at frequencies below 4 KHz. If a signal above 4 KHz is sampled, those samples appear exactly as if the signal was at a frequency mirror imaged about 4 KHz. Two examples include signals at 5 KHz and 7 KHz which will result in samples that look like signals of 5–8 KHz = 3 KHz and 7–8 KHz = 1 KHz, respectively.

Conversely, the sampling process produces replicas (aliasing) of the sampled signal around multiples of the sampling frequency. Therefore, if two signals are introduced digitally representing 1 KHz and 2 KHz, there will also be frequency components located at 8 KHz = ±1 KHz and 8 KHz = ±2 KHz, and so on for all multiples of 8 KHz. Thus it is possible to generate frequencies at arbitrary values after sampling by controlling the frequency of each signal within the 4 KHz input band regardless of whether it is in analog or PCM.

When an analog signal is sampled, the frequency components generated are all of the same amplitude as the corresponding input spectral components. Therefore, on the transmit side, measurements made from the PCM data will have a throughput gain of unity except where components are superimposed (e.g., a 4 KHz input signal will have an alias component at 4 KHz which may double the amplitude at 4 KHz when the two components are combined).

When an analog signal is reconstructed from digital samples, it goes through a sample and hold stage which has the effect of imposing a weighting function on the resulting spectral components that is represented by

$$\text{Sinc}\left[\frac{\omega T}{2}\right] = \frac{\text{Sin}\left(\frac{\omega T}{2}\right)}{\frac{\omega T}{2}}$$

where ω is the actual spectral component frequency going into the filter, and T is the width of the hold pulse at the decoder output. For the 2913/14, the analog output is held the full sample period of 125 μs (1/8000 Hz) so that a frequency component at f_t will have a weighting of

$$W = \left(\frac{8000}{\pi f_t}\right) \text{Sin}\left[\frac{\pi f_t}{8000}\right]$$

Transmit Filter Test Approach—Two approaches can be used for half channel testing of the transmit filter characteristic: (1) input analog test frequencies and perform an FFT on the corresponding PCM samples that are generated to determine spectral frequencies and amplitudes at the codec output, or (2) use an "ideal" D/A converter on the PCM samples to convert the digital data back to analog so that the spectral amplitudes and frequencies can be determied using analog circuits such as spectrum analyzers or filter banks. In either case, the effects of sampling will be the same. Figure 13 shows two spectral diagrams of amplitude versus frequency. The top diagram represents the locations of nine test frequencies corresponding to the seven specified frequencies in the 2913/14 data sheet plus a component at 7 KHz and one at 10 KHz. The bottom figure shows the "equivalent" spectral component locations when carried in the PCM bit stream. As an example, frequency #8 is located at 7 KHz. The corresponding PCM frequency is seen in the lower figure at 1 KHz. Note also that the analog component at 9 KHz (see #8*) would also generate the 1 KHz component in the PCM data.

To test the filter, the desired test frequencies are introduced in analog to the filter input in such a way that there is no confusion as to where the resulting component will be after sampling (i.e., don't simultaneously put in 1 KHz and 7 KHz since both of these inputs result in a 1 KHz component in the PCM data). Then, using either technique (FFT or analog) mentioned above, measure the amplitude of the corresponding

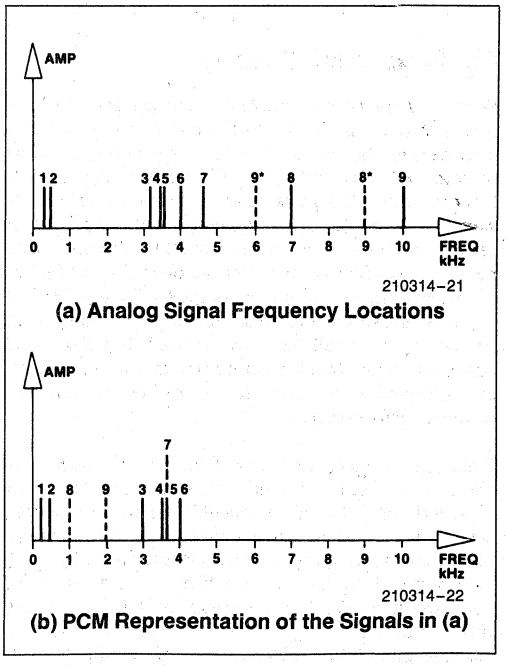


Figure 13. Spectral Properties of the Filter Test Frequencies in Analog and PCM

sampled component. The difference between that amplitude and the input amplitude represents the filter attenuation *at the frequency of the input signal.* So, if the signal was at 7 KHz, the FFT will determine the amplitude of the corresponding 1 KHz signal. The amplitude change relative to the input will represent the filter attenuation at 7 KHz.

Receive Filter Test Approach—In this case, the PCM test signals can be generated directly from digital circuits or by going through an "ideal" A/D (companded) to generate the PCM samples. Since these samples represent frequencies below the half sampling rate, Figure 12(b) now represents the input signals and 12(a) the output, but with one significant difference—a $\text{Sinc}[\pi\ f_I/8000]$ weighting function is imposed on all the frequency components because of the decoder sample and hold output. At the filter output, the spectral component amplitudes will include the effect of the filter response *and* the weighting function measured at the actual test frequency. The receive filter includes a compensation network for the weighting function in its passband. Therefore, inside the passband (300 Hz to 3.4 KHz) the measured amplitudes should be compared directly to the data sheet specifications. Frequencies outside the passband must be compensated for the weighting function first to determine the true filter response.

Summary of Filter Testing—Table 10 lists the nine test frequencies shown in Figure 12 for both the transmit and receive filter testing. For each filter test, the input frequency (analog or PCM), measurement frequency, and test circuit gain is tabulated corresponding to the desired test frequency. The various weighting values are easily handled by computer-based test equipment since the inverse weighting function can be stored in the computer and applied to each measured amplitude as appropriate.

4.3 Operational On-Line Testing

Two test modes are available which facilitate on-line testing to verify operation of both the combochip and the entire switching highway network. The first is simply the capability to duplicate the same D_X transmission in multiple PCM time slots (redundancy checking), and the second is the analog loopback capability which allows the testing of a call completion through the entire PCM voice path including the time slot interchange network.

Redundancy Checking—A feature of the 2913/14 is that the same 8-bit PCM word can be put on the D_X highway in multiple time slots simply by holding the frame sync/data enable (FS_X) high and continuing to supply clock pulses (CLK_X or $DCLK_X$). If the data enable was held high for multiple time slots, each time slot would have identical data in it. By routing this data through the PCM highways, time slot interchanges, etc., and then correlating the data between time slots, it would be possible to detect time slot-dependent data errors. When this test mode is used, no other data will be generated for the transmit highway until the frame sync returns low for at least one full clock cycle.

Table 10. Filter Response Testing Input/Output Frequencies and Amplitude Gain Schedule

	Test Freq.	Transmit			Receive		
		Input Freq.	Measured Freq.	Amp Weighting	Input Freq.	Measured Freq.	Amp Weighting
1	200	200	200	1	200	200	1
2	300	300	300	1	300	300	1
3	3000	3000	3000	1	3000	3000	1
4	3300	3300	3300	1	3300	3300	1
5	3400	3400	3400	1	3400	3400	1
6	4000	4000	4000	0 to 2	4000	4000	0 to 2
7	4600	4600	3400	1	3400	4600	$\text{Sinc}\left[\frac{4600\,\pi}{8000}\right]$
8	7000	7000	1000	1	1000	7000	$\text{Sinc}\left[\frac{7000\,\pi}{8000}\right]$
9	10000	10000	2000	1	2000	10000	$\text{Sinc}\left[\frac{10000\,\pi}{8000}\right]$

Analog Loopback—The 2914 (2913 does not have this feature) has the capability to be remotely programmed to disconnect the outside telephone lines and tie the transmit input directly to the receive output to effect analog loopback within the combo chip. This is accomplished by setting the LOOP input to V_{CC} (TTL high). The result is to disconnect VF_XI+ and VF_XI- from the external circuitry and to connect internally PWRO+ to VF_XI+, GS_r to PWRO−, and VF_XI- to GS_X (see Figure 14).

With this test set up, the entire PCM and analog transmission path up to the SLIC can be tested remotely by assigning a PCM word to a time slot that is read by the combo being tested. This data is converted to analog and passed out of the receive channel. It is taken as input by the transmit channel where it is filtered and redigitized (encoded) back to PCM. The PCM word can now be put on the transmit highway and sent back to the remote test facility. By comparing the PCM data (individually or as a series of codes) the health of that particular connection can be verified.

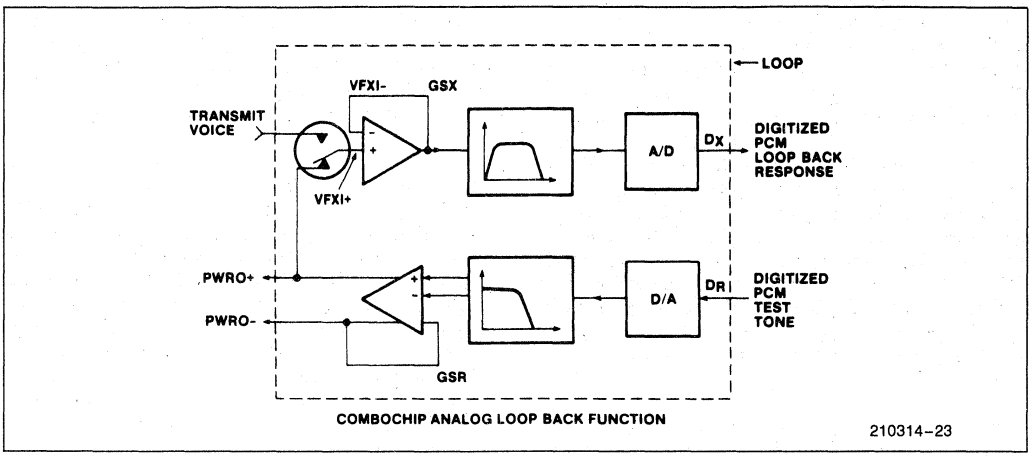


Figure 14. Simplified Block Diagram of 2914 Combochip in the Analog Loopback Configuration

UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison, BP 303
78054 Saint-Quentin-en-Yvelines Cedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8

Microcommunications

Volume II - Applications

Microcommunications: The application of microelectronic or silicon technology to the field of communications. Intel, long recognized as an established leader in the microcommunications market, has a broad range of components covering local area networks (LANs), modems and ISDN.

This handbook gives you information on how to design Intel's microcommunications components into useful system applications. Application articles explore Intel's entire line of microcommunications products, including LANs, access method evaluation tools, wide area networks and telecommunications components. Charts are included to show parameters and test conditions, layout principles and much more. For further information on microcommunication components, see Microcommunications, Volume 1.

Printed in U.S.A./0190/43K/RRD LK
Microcommunications Components

ISBN 1-55512-090-3