intel

Development Systems Handbook

Order Number: 210940-003

LITERATURE

In addition to the product line handbooks listed below, the INTEL PRODUCT GUIDE (no charge, Order No. 210846-003) provides an overview of Intel's complete product lines and customer services.

Consult the INTEL LITERATURE GUIDE (Order No. 210620) for a listing of Intel literature. TO ORDER literature in the U.S., write or call the INTEL LITERATURE DEPARTMENT, 3065 Bowers Avenue, Santa Clara, CA 95051, (800) 538-1876, or (800) 672-1833 (California only). TO ORDER literature from international locations, contact the nearest Intel sales office or distributor (see listings in the back of most any Intel literature).

Use the order blank on the facing page or call our TOLL FREE number listed above to order literature. Remember to add your local sales tax.

1985 HANDBOOKS

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

	*U.S. PRICE
QUALITY/RELIABILITY HANDBOOK (Order No. 210997-001) Contains technical details of both quality and reliability programs and principles.	**$15.00**
CHMOS HANDBOOK (Order No. 290005-001) Contains data sheets only on all microprocessor, peripheral, microcontroller and memory CHMOS components.	**$12.00**
MEMORY COMPONENTS HANDBOOK (Order No. 210830-004)	**$18.00**
TELECOMMUNICATION PRODUCTS HANDBOOK (Order No. 230730-003)	**$12.00**
MICROCONTROLLER HANDBOOK (Order No. 210918-003)	**$18.00**
MICROSYSTEM COMPONENTS HANDBOOK (Order No. 230843-002) Microprocessors and peripherals—2 Volume Set	**$25.00**
DEVELOPMENT SYSTEMS HANDBOOK (Order No. 210940-003)	**$15.00**
OEM SYSTEMS HANDBOOK (Order No. 210941-003)	**$18.00**
SOFTWARE HANDBOOK (Order No. 230786-002)	**$12.00**
MILITARY HANDBOOK (Order No. 210461-003) Not available until June.	**$15.00**
COMPLETE SET OF HANDBOOKS (Order No. 231003-002) Get a 25% discount off the retail price of $160.	**$120.00**

***U.S. Price Only**

intel®

U.S. LITERATURE ORDER FORM

NAME: _____ TITLE: _____

COMPANY: _____

ADDRESS: _____

CITY: _____ STATE: _____ ZIP: _____

COUNTRY: _____

PHONE NO.: (____) _____

ORDER NO.	TITLE	QTY.	PRICE	TOTAL
☐☐☐☐☐☐-☐☐☐		____	× ____	= ____
☐☐☐☐☐☐-☐☐☐		____	× ____	= ____
☐☐☐☐☐☐-☐☐☐		____	× ____	= ____
☐☐☐☐☐☐-☐☐☐		____	× ____	= ____
☐☐☐☐☐☐-☐☐☐		____	× ____	= ____
☐☐☐☐☐☐-☐☐☐		____	× ____	= ____

POSTAGE AND HANDLING:
Add appropriate postage and handling to subtotal
10% U.S.
20% Canada

Subtotal _____

Your Local Sales Tax _____

_____

Allow 4-6 weeks for delivery

Total _____

Pay by Visa, MasterCard, Check or Money Order, payable to Intel Literature. Purchase Orders have a $50.00 minimum.

☐ Visa ☐ MasterCard Account No. _____ Expiration Date _____

Signature: _____

Mail To: Intel Literature Distribution
Mail Stop SC6-714
3065 Bowers Avenue
Santa Clara, CA 95051.

Customers outside the U.S. and Canada should contact the local Intel Sales Office or Distributor listed in the back of this book.

For information on quantity discounts, call the 800 number below:
TOLL-FREE NUMBER: (800) 548-4725
Prices good until 12/31/85.
Source HB

Cut Along Dotted Line

Mail To: Intel Literature Distribution
Mail Stop SC6-714
3065 Bowers Avenue
Santa Clara, CA 95051.

intel®

DEVELOPMENT SYSTEMS HANDBOOK

JANUARY 1985

About Our Cover:
The design on our front cover is an abstract portrayal of the systems creation or connection point. From this central sunburst of technology, peripheral design applications and Ethernet networking cables combine in an orchestrated manner symbolically indicating a development process that is in tune with the technological explosion. The design engineer can count on Intel's complete set of integrated tools for every facet of systems development.

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:

BITBUS, COMMputer, CREDIT, Data Pipeline, GENIUS, i, $\hat{i}$, ICE, iCS, iDBP, iDIS, I^2ICE, iLBX, i_m, iMDDX, iMMX, Insite, Intel, int_el, int_eIBOS, Intelevision, $int_eligent$ Identifier, $int_eligent$ Programming, Intellec, Intellink, iOSP, iPDS, iRMX, iSBC, iSBX, iSDM, iSXM, KEPROM, Library Manager, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, *MULTICHANNEL, MULTIMODULE, OpeNET, Plug-A-Bubble, PROMPT, Promware,* QUEST, QueX, Ripplemode, RMX/80, RUPI, Seamless, SLD, SYSTEM 2000, and UPI, and the combination of ICE, iCS, iRMX, iSBC, iSBX, 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.

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

Intel Corporation
Literature Department
3065 Bowers Avenue
Santa Clara, CA 95051

Table of Contents

ALPHANUMERIC INDEX vi

INTRODUCTION viii

CHAPTER 1
Microcomputer Development Systems
Data Sheets
iMDX 430/431/440/441 Intellec® Series IV Microcomputer Development System 1-1
iPDS™ Personal Development System 1-6
iPDS™-130 Optional Flexible External Disk Drive 1-17
iPDS™-Proto Kit 1-21
iMDX 557 iAPX Resident Processor Card Package 1-23
iMDX 511 Enhanced Human Interface 1-26
Model iMDX 750 Intellec® Series II/III/IV Winchester Subsystem 1-28
Application Note
AP-156 Designing Modules for iPDS™ and iUP Systems 1-31

CHAPTER 2
Network Development Systems
Data Sheets
Network Development System II (NDS-II) 2-1
iMDX-580/581 ISIS Cluster Board Packages 2-9
Asynchronous Communication Link 2-13
Mainframe Link for Distributed Development 2-16
iNA 955 iRMX™ NDS-II Link 2-19
Article Reprint
AR-204 Smart Link Comes to the Rescue of Software Development Managers 2-23

CHAPTER 3
Microcomputer Software Development Tools
Data Sheets
Program Management Tools 3-1
PSCOPE High-Level Program Debugger 3-4
iRMX™ PSCOPE 86 High-Level Program Debugger 3-9
NDS-II Electronic Mail 3-20
8086 Software Toolbox 3-22
AEDIT Text Editor 3-24
ISIS-II Software Toolbox 3-26
Insite 3-28
Application Note
AP-162 PMT Tutorial 3-30
Article Reprints
AR-225 Debugging Catches up with High-Level Programming 3-77
AR-319 Software Development 3-83
AR-352 Integrated Environment Speeds System Development 3-88

CHAPTER 4
Microcomputer Development Languages
Data Sheets
iAPX 286 Software Development Packages 4-1
Pascal 286 Software Package 4-6
PL/M 286 Software Package 4-9
iAPX 86, 88 Software Development Packages for Series II/PDS 4-13
86/88/186/188 Software Packages 4-23

Table of Contents

Fortran 80 8080/8085 ANS Fortran 77 4-39
Pascal 80 S/W Package 4-43
PL/M 80 High Level Programming Language 4-48
8087 Software Support Package 4-51
8087 Support Library 4-54
80287 Support Library 4-58
8089 IOP Software Support Package 4-61
8051 Software Packages 4-64
MCS®-48 Diskette-Based Software Support Package 4-73
MCS®-96 Software Development Packages 4-75
VAX*/VMS* Resident iAPX-86/88/186 Software Development Packages 4-83
VAX*/VMS* Resident Software Development Packages for iAPX 286 4-90
2920 Software Support Package 4-96

Article Reprints

AR-59 Modular Programming in PL/M 4-107
AR-136 PL/M-86 Combines Hardware Access with High-Level Language Features 4-114
AR-200 Compiler Optimization Techniques 4-121
AR-239 PL/M-51: A High-Level Language for the 8051 Microcontroller Family 4-127

CHAPTER 5

In-Circuit Emulators

Data Sheets

EMV-51A 8051A Emulation Vehicle 5-1
EMV-44 Con 8044 Emulation Vehicle Conversion Package 5-9
EMV-88 iAPX 8088 Emulation Vehicle 5-17
iSBE-96 Single Board Emulator 5-28
I^2ICE™ Integrated Instrumentation and In-Circuit Emulation System 5-35
iLTA Logic Timing Analyzer 5-52
ICE™-42 8042 In-Circuit Emulator 5-58
ICE™-44 Module 8044 In-Circuit Emulator 5-66
ICE™-49A MCS®-48 In-Circuit Emulator 5-74
ICE™-51 8051 In-Circuit Emulator 5-80
ICE-85B™ MCS-85 In-Circuit Emulator with Multi-ICE™ 5-88
ICE™-86A iAPX 86 In-Circuit Emulator 5-95
ICE™-88A iAPX 88 In-Circuit Emulator 5-103

CHAPTER 6

PROM Programming

Data Sheets

iUP-200A/iUP-201A Universal PROM Programmers 6-1
PROM Programming Personality Modules 6-12

Application Note

AP-179 PROM Programming with the Intel Personal Development System (iPDS™) 6-22

CHAPTER 7

System Design Kits

SDK-85 MCS-85™ System Design Kit 7-1
SDK-86 MCS-86™ System Design Kit 7-7
SDK-C86 MCS-86™ System Design Kit Software and Cable Interface 7-13
SDK-51 MCS®-51 System Design Kit 7-15
SDK-2920 2920 System Design Kit 7-20

Table of Contents

CHAPTER 8

Third Party Software

Data Sheets

Microsoft* Inc., BASIC-80 Interpreter Software Package 8-1
Microsoft* Inc., BASIC-80 Compiler Software Package 8-4
Digital Research Inc., CP/M* 2.2 Operating System 8-6
Wordstar* Word Processing Software 8-9
Microsoft* Multiplan* Spreadsheet 8-16

*Please note:
VAX and VMS are trademarks of Digital Equipment Corporation
Microsoft and Multiplan are trademarks of Microsoft Corporation
CP/M is a trademark of Digital Research Corporation
Wordstar is a trademark of Micropro International

2920 Software 4-96
8051 Software 4-64
8086 Software Toolbox 3-22
86/88/186/188 Software 4-23
8087 Software Support Library 4-51
8087 Support Library 4-54
8089 IOP Software 4-61
80287 Support Library 4-58
AEDIT Text Editor 3-24
Asynchronous Communications Link 2-13
BASIC-80 Compiler 8-4
BASIC-80 Interpreter 8-1
CP/M 2.2 Operating System 8-6
EMV-44 5-9
EMV-51A 5-1
EMV-88 5-17
FORTRAN 80 4-39
FORTRAN 86/88 4-24
iAPX 86, 88 Software for Series II/PDS 4-13
iAPX 286 Software 4-1
iC-86 C Compiler 4-35
I^2ICE 5-35
ICE™-42 5-58
ICE™-44 5-66
ICE™-49A 5-74
ICE™-51 5-80
ICE™-85B 5-88
ICE™-86A 5-95
ICE™-88A 5-103
iLTA 5-52
iMDX 430/431/440/441 1-1
iMDX 511 1-26
iMDX 557 1-23
iMDX 580/581 2-9
iMDX 750 1-28
iNA 955 2-19
Insite 3-28
iPDS™ 1-6
iPDS™-Proto 1-21
iPDS™-130 1-17
iRMX™ PSCOPE 86 3-9
iSBE 5-28
ISIS-II Software Toolbox 3-26
iUP-200A/iUP-201A 6-1
Mainframe Link 2-16
MCS®-48 Diskette-Based Software 4-73
MCS®-96 Software 4-75
Multiplan Spreadsheet 8-16
NDS-II 2-1
NDS-II Electronic Mail 3-20
Pascal 80 4-43
Pascal 86/88 4-28
Pascal 286 4-6
PL/M-80 4-48
PL/M 86/88/186/188 4-31
PL/M 286 4-9
Program Management Tools 3-1
PROM 6-12
PSCOPE 3-4

SDK-2920 7-20
SDK-51 7-15
SDK-85 7-1
SDK-86 7-7
SDK-C86 7-13
VAX*/VMX* Resident iAPX 86/88/186 Software 4-83
VAX*/VMX* Resident Software for iAPX 286 4-90
Wordstar 8-9

*Please note:
VAX and VMS are trademarks of Digital Equipment Corporation
Microsoft and Multiplan are trademarks of Microsoft Corporation
CP/M is a trademark of Digital Research Corporation
Wordstar is a trademark of Micropro International

INTEL'S DEVELOPMENT ENVIRONMENT—THE COMPLETE SOLUTION

The emergence of high performance, low cost microprocessors has revolutionized the computer industry in the past decade. Many of today's advanced "chips" literally contain the computing power of mainframe computers that were commonly used just ten short years ago. The availability of these advanced processors has spawned many new products and vastly improved existing ones.

The rapid advances in microprocessor technology have also revolutionized the microprocessor system design process. The days of a single engineer developing all software and hardware for a given system are over. Most projects today consist of many engineers working in a team. And most projects are software intensive: a ratio of five to ten software engineers for every hardware designer is common.

Developing software intensive systems with large teams typically creates headaches for team members and project leaders alike. Large team development necessitates numerous team meetings, large numbers of software modules containing many interfaces, significant amounts of formal system documentation, and a long and tedious debug process. In addition, team members must share all types of project information: source code, object code, test suite results, etc. Project leaders in this environment must continually strive to meet project deadlines and contain costs.

Intel offers a complete line of microcomputer development tools that help developers maintain tight project deadlines and minimize costs. For example, Intel's advanced software development tools and programming languages can boost an individual programmer's productivity and also simplify team management. Our powerful In Circuit Emulators (ICEs™) minimize the risks associated with integrating system software with target hardware and thus ensure your projects are not delayed during this critical development phase. And Intel's state-of-the-art dedicated workstations, such as the Series IV Microcomputer Development System, are ideal development hosts for providing these remarkable tools at your fingertips. Moreover, as your team grows you can link team members together in a powerful network with NDS-II—the Network Development System. Because of our special knowledge of the processors we design, you can be assured that our tools are the most powerful available for the task at hand.

Software Tools Boost Programmer Productivity

Intel offers a wide range of software development tools that boost programmer productivity and minimize the costly administrative overhead that typically accompanies large software development projects.

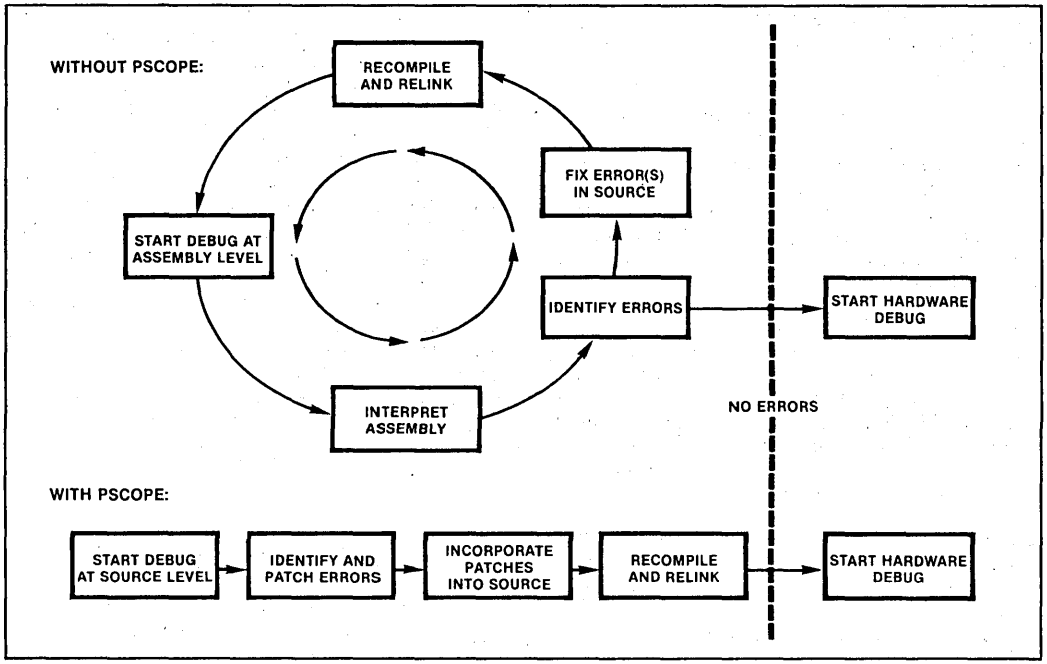
- PSCOPE Simplifies and Speeds Program Debugging

  PSCOPE is a source-level symbolic debugger that allows the high-level language program-

This chart illustrates the broad range of tools that simplify the development of products using Intel microprocessors and microcontrollers.

Tools Available	8-Bit μ Controller	8051	8085	8096	8088	8086	80188	80186	80286
LANGUAGES									
• Pascal			—		—	—	—	—	—
• FORTRAN			—		—	—	—	—	—
• PL/M		—	—	—	—	—	—	—	—
• C					—	—	—	—	--
• MACRO ASSEMBLERS	—	—	—	—	—	—	—	—	—
TOOLS									
• PSCOPE					—	—	—	—	—
• SVCS; MAKE	—	—	—	—	—	—	—	—	—
INSTRUMENTATION									
• IN-CIRCUIT EMULATORS	—	—	—	--	—	—			
• I²ICE™ SYSTEM					—	—	—	—	—

NOTE: AVAILABLE NOW —
AVAILABLE SOON --



PSCOPE is an advanced, high-level language debugger that cuts many time consuming steps out of the software debugging process. PSCOPE enables users to correct software errors with "patches," eliminating many unnecessary edits, compiles and links.

mer to completely debug his code at the same level at which it was written. Breakpointing, tracing, and patching are all done in a faster and less error-prone manner than through obsolete machine-level debuggers. Since software testing and maintenance consume a greater portion of development life-cycle time and cost, PSCOPE debugging can significantly improve programming efficiency.

- Program Management Tools Save Development and Administration Time

  Intel Program Management Tools (PMTs) tighten your control over program changes, documentation, software maintenance, system generation, and program libraries—reducing your administrative workload and time. Intel PMTs currently consist of two powerful utilities: the Software Version Control System (SVCS) and MAKE.

  SVCS is a system database manager that simplifies software development and maintenance. You never have to waste time trying to manually keep track of software changes. SVCS tracks each change (including who made the change, what changed, when, and why) so that you can always be sure you are working with the current version of a given module. SVCS enforces individual discipline to enhance team cooperation and project control.

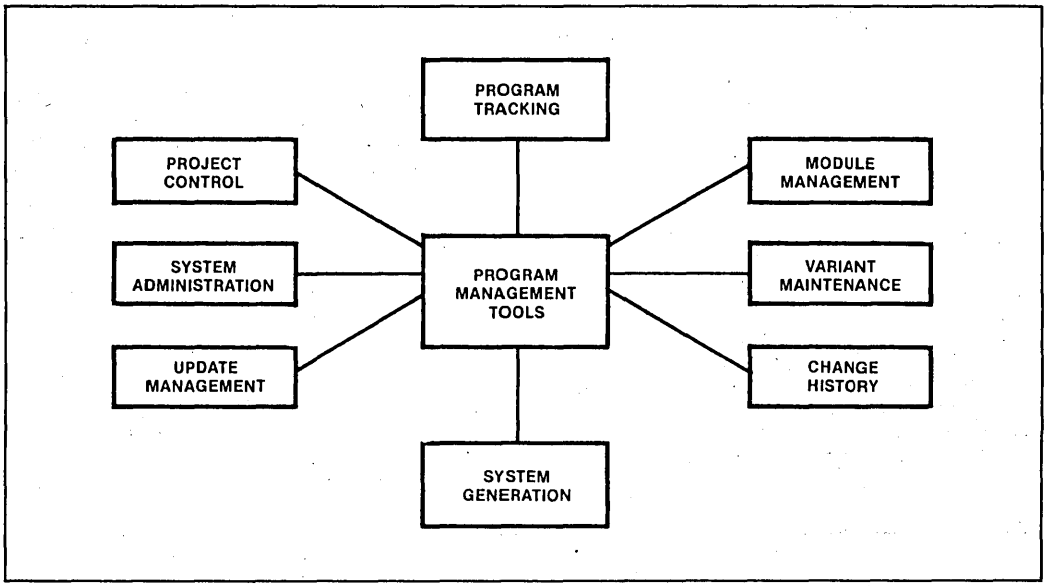
  MAKE is an automatic system generation tool that can save hours and days of software generation time. It automatically finds the most recent modules, recompiles only the ones that need it, and generates a new system. All you do is create a specification of your system. MAKE does the rest.

  MAKE works with SVCS to maintain the most current version of software—including up-to-the-minute source code changes from project software engineers. So you never waste time recompiling outdated modules or generating entire systems from outdated modules.

  Together, Intel PMTs dramatically boost productivity by eliminating redundant steps. They save weeks—sometimes months—of software development time for any given project.

- Software Toolboxes Provide Programmers With an Arsenal of Productivity Aids

  Intel's software toolboxes are collections of utilities that perform a variety of productivity-



Program Management Tools enable you to manage and easily integrate the efforts of many development team members.

oriented functions. The ISIS-II Software Toolbox offers conditional submit file control tools, source management tools, and other tools that operate at the ISIS-II command level. The 8086 Software Toolbox is a collection of 16-bit software tools that are valuable for text formatting and preparation, software testing and performance analysis, 286/287 software development, and a multitude of other applications.

- AEDIT Enables Programmers to Easily Enter Source Code and Text Files

  Intel also offers AEDIT, an advanced editor that significantly improves programmer productivity. AEDIT was designed with the programmer in mind, and offers full screen editing, the ability to edit two files at once, features for manipulating large blocks of text, and a powerful macro command facility.

High-Level Languages Improve Software Quality and Programming Efficiency

High-level languages make programming easier. And Intel's efficient high-level language compilers do not sacrifice code quality for ease of use. Moreover, different languages can be used for the various modules which make up a software system. This allows programmers to choose the optimal language for each given module, and then link the modules together into high quality software systems.

Intel provides complete high-level language support for all 8-bit and 16-bit Intel microprocessors. All Intel languages produce linkable and locatable object codes. In addition, Intel compilers pass information to debuggers to optimize the debug cycle. Standard languages supported include PL/M, Pascal, Fortran, Basic and C.

In-Circuit Emulators Simplify System Integration

Intel's In Circuit Emulators (ICEs™) accelerate system integration to the fastest possible speed. By emulating the prototype to check out the hardware design, each processor specific ICE reduces system integration and hardware debug times to a minimum.

The premier ICE for all iAPX microprocessors is the I²ICE™ (Integrated Instrumentation and In-Circuit Emulation) Emulation System.

The I²ICE system is a revolutionary tool which integrates in-circuit emulation, high-level software debugging and optional logic timing analysis into one system. With the I²ICE system you shorten debugging time and obtain easier to understand debug data—while reducing risks and product development time.

The I²ICE system gives you full speed, real-time support for all Intel iAPX microprocessors. It also offers an arsenal of break and trace points so that your design team can set up complex, multi-nested test conditions.

The I²ICE system provides a single human interface across the spectrum of debugging and system integration tasks. This eliminates the slow-down problems inherent in multiple interfaces. And there is never the need to leave the high level language environment because the I²ICE system incorporates PSCOPE to ease the transition from software debug to hardware debug and hardware/software integration.

Powerful Workstations Put Development Tools Within Your Reach

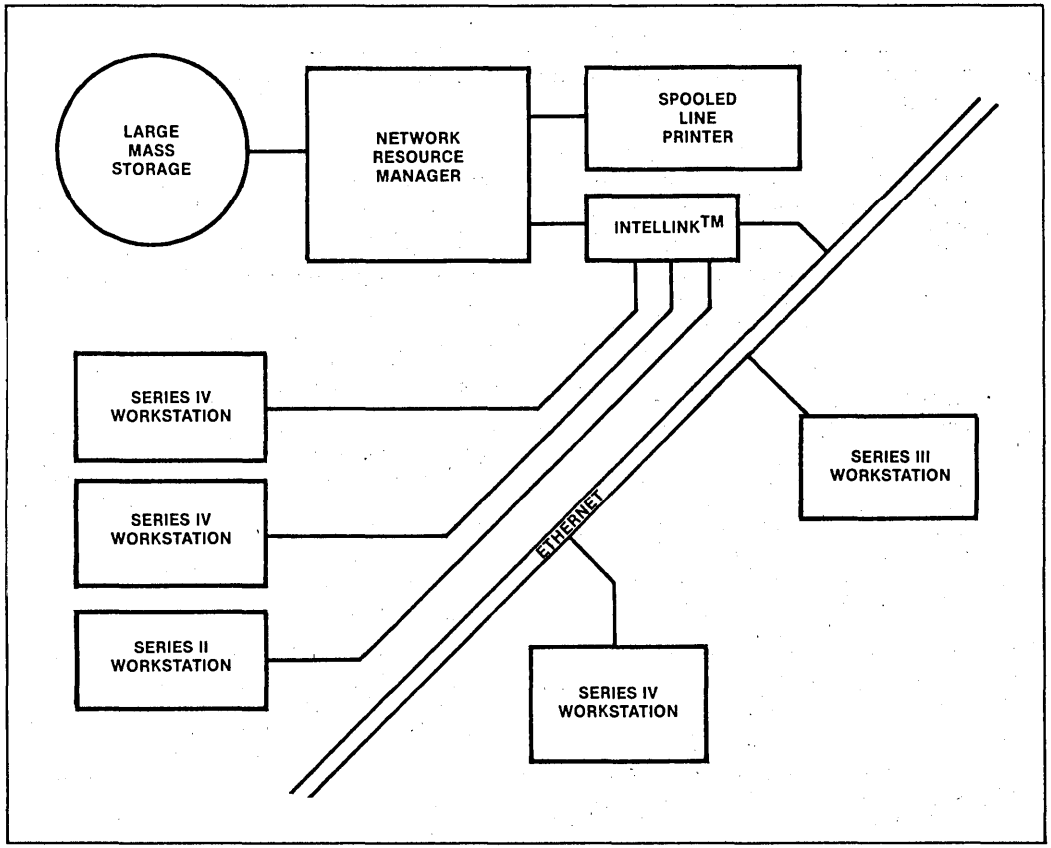
Intel workstations, such as the Series IV, are your gateway to the Intel development methodology. Their computing power puts Intel development tools literally at your fingertips. Intel offers a variety of hosts, such as those mentioned below, that can match the size, budget, and complexity of your development projects.

- Series IV Microcomputer Development System Provides Sophisticated Capabilities Not Found on Other Development Systems

  Intel's Series IV is a 16-bit workstation offering many advanced features. Like foreground/background processing that allows you to do two things at once—dramatically increasing your productivity. A hierarchical file system that enables your file structure to be set up according to project requirements. And a friendly human interface speeds learning, eliminates the need to plow through manuals, and significantly improves ease of use.

- Personal Development System Enables Low-Cost Product Development and Support

  The Personal Development System (iPDS™) is a portable, low-cost system that provides total development support for smaller 8-bit



The NDS-II Network Development System is a comprehensive Local Area Network that increases the productivity of engineers as well as equipment. Team members can easily share information and common hardware such as line printers and mass storage.

applications. It also supports the CP/M operating system, so it can double as a personal computer for your engineers.

Networking Links Project Engineers Into A Powerful Team

Intel's Networked Development System—NDS-II—is an Ethernet-based local area network that ties together Intel's development systems. It can increase the productivity of engineers as well as equipment.

For example, an engineer can use the Distributed Job Control service to send time-consuming tasks such as compiling to the NDS-II Network Resource Manager (NRM) for reassignment to a workstation that is not presently in use. The engineer can move on to other work at his/her terminal while the NRM completes the previous job.

NDS-II helps you get the most use out of your development dollars. Team members can share common hardware such as line printers and mass storage devices. In addition, NDS-II's modular approach provides a structure for orderly growth. A consistent upgrade path protects your investment in Intel development products.

Intel Offers Complete Development Support

Intel's support for your development project goes beyond the superior hardware and software tools described in this handbook—we deliver complete development solutions.

TRAINING

Intel offers training courses throughout the year at Intel Training Centers around the world. Courses can be tailored for either technical or management personnel.

FIELD SUPPORT

A worldwide network of Intel Field Service and Field Applications Engineers is available to assist you.

FIELD UPDATES

Intel customers are kept up-to-date on development system changes and enhancements to software via regular update mailings. "Hot-line" telephone support is also available.

DOCUMENTATION

Comprehensive product documentation is available—including manuals, application notes and detailed data sheets. (A complete literature guide is available upon request.)

USER'S PROGRAM LIBRARY

Through INSITE (Intel's Software Index and Technology Exchange), Intel makes available a broad collection of programs that may substantially cut development time for you.

Intel offers complete advanced microsystem solutions. Solutions in hardware, Solutions in software, Solutions in customer support, Solutions that work together for you.

Find Out More

Contact the Intel sales or distributor office nearest you (see the back of this handbook) for more information today.

Microcomputer Development Systems

1

iMDX 430/431/440/441
INTELLEC® SERIES IV
MICROCOMPUTER DEVELOPMENT SYSTEM

- **Complete Microcomputer Development System for the iAPX 86/87/88/186/188/286, the MCS® -80/85 and the MCS® -48/51/96 family microprocessors**
- **Advanced, friendly human interface with menu-driven function keys, HELP, and syntax builder/checker capabilities for increased user productivity**
- **Foreground/Background multiprocessing for simultaneous execution of two jobs by a single user; increasing system throughput**
- **Multi-user capability for simultaneous operation by two users, significantly reducing system cost per user**
- **Hierarchical file system provides file sharing and protection for large software projects**
- **Software compatible with both Series IIE and Series IIIE development systems**
- **Supports PL/M, Pascal, C, and FORTRAN, and Basic high-level languages as well as assemblers**
- **Provides Program Management Tools (PMTs), advanced AEDIT text editor and supports powerful PSCOPE symbolic, source level debugger**
- **Can be fully integrated into the NDS-II Network Development System**

The Intellec® Series IV is a new generation development system specifically designed for supporting the iAPX family of advanced microprocessors. It also supports the MCS-80/85 and the MCS-48/51 families.

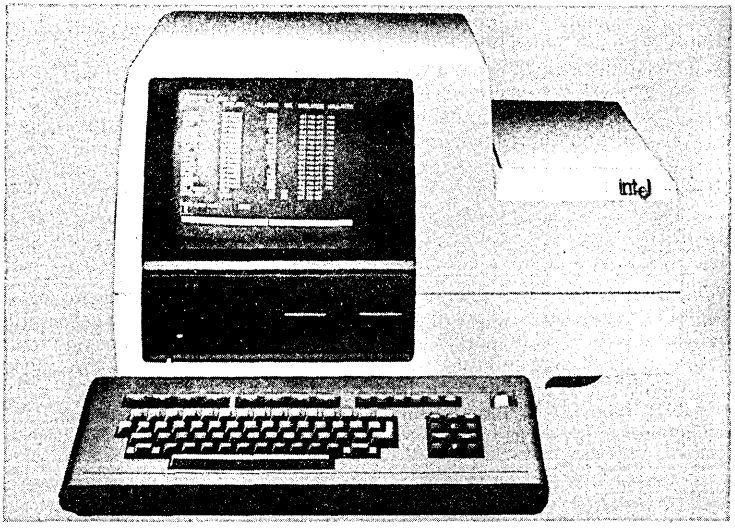


Figure 1. Intellec® Series IV Microcomputer Development System

MAY 1984

ORDER NUMBER: 230625-003

Series IV provides a state-of-the-art, easy-to-use, high performance host environment for running a wide variety of hardware and software development tools. A unique combination of tools provides an integrated microcomputer system design that results in highly improved designer productivity and considerable shortening of time to market. The length of the compile-link-load-debug-edit cycle is minimized by the friendly human interface, powerful and easy-to-use editors, a wide selection of language translators, source level debuggers, program management tools. The advanced operating system features a hierarchical file system, foreground/background multitasking, and multi-user capability. Furthermore, the Series IV can serve as a powerful workstation on the NDS-II distributed processing network for high performance multi-user software development. The networking architecture supports a distributed co-operative processing environment. Tasks like compilations can be executed in the background mode or exported to an idle workstation while the user is in the middle of an interactive edit session. The key benefit of this approach is a much higher system throughput and programmer productivity than, for instance, a system designed for raw-performance and fast compilations only.

The Series IV is offered in four different versions, providing a range of storage and performance options so that the user may select the configuration to suit his/her stand-alone or networking development station needs. The four versions are not only compatible with one another, but are also software compatible with the current generation enhanced Series IIE/IIIE systems. Existing ISIS-compatible software can run directly on the Series IV under the ISIS operation system. Finally, the NDS-II network provides an ideal means for the various hosts, e.g., Series II/III/IV to work with each other, protecting the user's past, and present, and future investment.

FUNCTIONAL DESCRIPTION

Systems Components

The Intellec Series IV model 430/431 Microcomputer Development System is an easy-to-use high-performance system in one package. It includes a CPU board for each of the iAPX 88 and MCS® 85 processors and 640K bytes of system RAM. The system has eight function keys included in its detachable standard ASCII keyboard that also has cursor controls and uppercase/lowercase capability. These function keys are menu driven and, with the use of the syntax builder/checker, greatly reduce user keystrokes. Peripheral configurations include: Model iMDX 430WD, 440WD—two floppy disks, one 35MB Winchester; and Model iMDX 431, 441—one floppy disk, one 10MB Winchester.

The 5.25" drives, a green phosphor screen, and a detachable keyboard are all integrated into the system. The main chassis has ten MULTIBUS® slots (three 12" X 12", seven 6¾" X 12") power supplies, fans and cables.

Operating System Environments/Features

The Series IV provides both an 8086/8088-based development environment and an 8080/8085 based development environment. The host execution mode is the 8086/8088, which runs under the iNDX operating system. To execute an 8080/8085 program, the ISIS-IV utility is invoked; entering the 8085 execution mode. All ISIS-compatible 8-bit software can thus be run directly on the Series IV, through a user interface that is compatible with ISIS-based development systems such as the Series II and the Series III.

HIERARCHICAL FILE SYSTEM

The iNDX operating system employs a hierarchical file system, providing file sharing and protection features. The hierarchical structure allows logical grouping of data. The structure resembles an inverted tree. The root of the system is called the logical system root. The system root logically "connects" the volumes within the file system. Each volume corresponds to a physical mass storage device. Volumes are further divided into files. Files can be either directory files or data files. Directory files contain references to further directory or data files. Data files contain only data.

It is not necessary to know the physical location of files to address them. Each file can be addressed by a path name, which is a character string recognized by the operating system.

The iNDX file system provides file protection features in the form of access rights. The owners of a file may set their access rights to their own files and separately set the WORLD's access rights (everyone else) to their files. File may thus be shared and also protected from accidental or deliberate addressing or destruction.

SINGLE-USER FOREGROUND/ BACKGROUND PROCESSING

Foreground/background processing capability allows the simultaneous execution of two jobs, resulting in improved system throughput. While a program is executing in the background, another program could be run in the foreground. For example, an interactive editor could be executing in the foreground while a compilation is taking place in the background.

A toggle key on the Series IV keyboard can be used to instantaneously move from one region to the other, allowing interactive operations in both foreground and background regions. For example, while a software debug session is taking place in the foreground, listing files can be displayed from the background.

MULTI-USER CAPABILITY

A low cost terminal can be attached to serial port 1. This terminal operates as an independent system, accessing one region, while the console and keyboard access the other region. In this mode two users will be able to perform software development tasks simultaneously at a significantly reduced cost per user.

The Human Interface

The Series IV is one of the easiest systems to learn and to use, as its human interface is designed to be friendly to both novice and expert users.

It offers eight softkeys that cut the number of keystrokes required to perform a function. On-line HELP provides instantaneous access to command definition. the menu-driven screen interface allows the user to see where he/she is at and to select the next operation. In conjunction with the soft function keys, it allows single key command invocation. The syntax builder and checker completes commands and insures proper command syntax before execution. Features such as type-ahead, auto-repeat keys, and quick view file facility are some of the many other human interface factors that improve programmer productivity.

The AEDIT Text Editor

The AEDIT text editor is one of the most powerful and easy-to-use editors available. It runs under the iNDX operating system and offers features such as:

- Display and scroll text on the screen
- Move to any character position in the text file or to any point on the screen instantly
- Correct typing mistakes as you type
- Rewrite text by typing new characters over old ones
- Make insertions and deletions easily at any point in a file
- Find any string of characters and substitute another string, querying the operator if desired
- Move or copy sections of text within a file or to/from another file
- Create macros to execute several commands at once, thereby simplifying repetitive editing tasks
- Edit two files simultaneously
- Indent text and delimit long lines automatically
- View lines over 80 characters long

Languages and Utilities

The Series IV supports popular high-level languages such as PL/M, Pascal, FORTRAN, and C, as well as powerful "high-level" macro assemblers such as ASM86. In addition, iRMX™ utilities such as ICU-86, PATCH utility, Files Utility, Crash analyzer and SDM 86 System Debug Monitor are supported by the Series IV.

The high-level language compilers produce code for the target processors. They also contain run-time floating-point arithmetic support for the 8087 Numeric Data Processor.

PSCOPE, the High-Level Language Debugger

The Series IV supports the PSCOPE debugger, an interactive, symbolic debugger for FORTRAN, Pascal, and PL/M programs. Operations are performed on source statements, procedure entry points, labels, and variables, as opposed to machine instructions memory addresses. PSCOPE improves productivity in the debug phase of development and produces more reliable software. It allows the user to perform extensive tests and consistency checks on the programs, and it automates much of the testing.

In-Circuit Emulators

The Series IV supports a host of ICE modules including the powerful I^2ICE™ for iAPX family-

based development. These tools allow the debugging of microcomputer system hardware and software concurrently, saving considerable development cost and time.

Network Capability

The Series IV may be used as a high-performance workstation for use on the NDS-II Network Development System. It has complete access to all the network resources and facilities on the NDS-II. A stand-alone Series IV can be upgraded to an NDS-II workstation with the addition of an Ethernet* Communication Board Set. The background partition of the Series IV may be made available as a network resource.

When configured as an NDS-II workstation, the Series IV can also serve as a host for up to four iMDX-580 ISIS cluster boards, providing a cost effective means for supporting incremental 8-bit software workstations on the network.

System Configurations

Series IV Systems are available in 110v, 60Hz; 220v and 100v, 50Hz models.

STAND-ALONE

iMDX 430WD Kit
A two floppy stand-alone system including detachable keyboard and integral green CRT that comes complete with a 30MB Winchester in a separate chassis. The CPU's are the 8088 and the 8085A-2.

iMDX 431
Stand-alone Intellec Development system with detachable keyboard and integral green CRT. Included in the main chassis is one 5.25" floppy and one 5.25" 10MB Winchester drive.

iMDX 440WD Kit
The same configuration as the iMDX 430WD, this model has an additional higher performance 8086 CPU.

iMDX 441 Kit
The same configuration as the iMDX 431, this model has an additional higher performance 8086 CPU.

NETWORK

iMDX 430WS Kit
A two floppy workstation that includes Ethernet NDS-II boards for network operation.

iMDX 440WS Kit
The same configuration as the iMDX 430WS, this system includes a high-performance option for resident 8086 execution and faster performance.

iMDX 430 TO 440 UPGRADE

iMDX 434
High-performance add-on option. Converts a model iMDX 430 or iMDX 431 to a model iMDX 440 or iMDX 441.

NETWORK UPGRADE

iMDX 456
Communication board set converts any Series IV stand-alone system to an NDS II workstation.

SECOND-USER TERMINALS

The following terminals have been tested and found to be interface-compatible with the Series IV CPIO board and can be used as second-user terminals.

LEAR SEIGLER, Model ADM 3A
TELEVIDEO, Model 910+

The following terminals have been successfully tested for interface-compatibility, however they do not meet Intel environmental specifications: adverse electrostatic conditions may produce unpredictable screen output, requiring terminal or system reset.

Televideo, Model 925, 950
Adds Viewpoint 3A+
Qume 102
Hazeltine 1510

PHYSICAL CHARACTERISTICS

Chassis		Keyboard
Width	26.5" (67.3 cm)	20.0" (50.8 cm)
Height	16.5" (41.9 cm)	3.0" (7.6 cm)
Depth	18.5" (47.0 cm)	8.0" (20.3 cm)
Weight	52 lb. (23.4 kg)	7 lb. (3.1 kg)

ELECTRICAL CHARACTERISTICS

DC Power Supplies

Volts Supplied	Amps Supplied
+5.1 ± 1%	45.0
+12 ± 5%	3.0
-12 ± 5%	2.0
-10 ± 5%	0.5
+12 ± 5%	5.0

*Ethernet is a trademark of Xerox Corp.

AC Requirements

110v, 60Hz
220v, 50Hz

Environmental Characteristics

Operating Temperature — 10°C to 35°C (50°F to 95°F)
Humidity — 10% — 95% (non-condensing)

Equipment Supplied

Series IV System

Series II/III to Series IV link software diskettes and cable

Series IV Software

- — iNDX OS
- — ISIS IV OS
- — AEDIT
- — Macroassemblers and utilities
- — ICE™ software
- — Prom Programmer Software
- — Debug 88
- — Program Management Tools (MAKE, SVCS)
- — Diagnostics

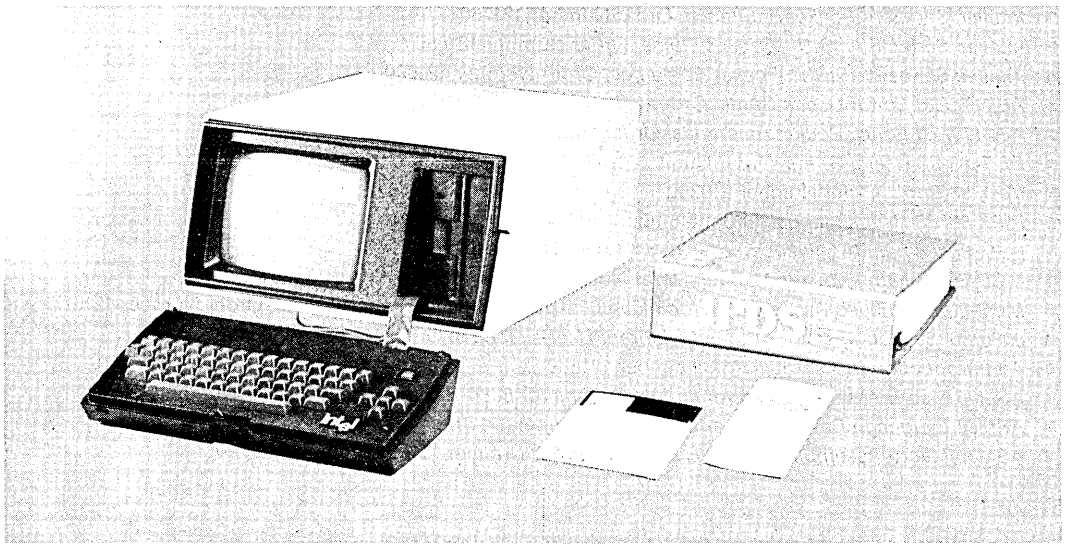
Documentation Supplied

- *Intellec Series IV Microcomputer Development System Overview,* Order Number 121752
- *Intellec Series IV Microcomputer Development System Installation and Checkout Manual,* Order Number 121757
- *Intellec Series IV Operating and Programming Guide,* Order Number 121753
- *Intellec Series IV Pocket Reference,* Order Number 121760
- *Intellec Series IVC ISIS-IV User's Guide,* Order Number 121880
- *Intellec Series IV ISIS-IV Pocket Reference,* Order Number 121890
- *AEDIT Text Editor User's Guide,* Order Number 121756
- *AEDIT Text Editor Pocket Reference,* Order Number 121767
- *DEBUG-88 User's Guide,* Order Number 121758
- *iAPX 88 Book,* Order Number 210200
- *iAPX 86, 88 User's Manual,* Order Number 210201
- *iAPX 86, 88 Family Utilities User's Guide,* Order Number 121616
- *MCS-80/85 Family User's Manual,* Order Number 121506
- *MCS-80/85 Utilities User's Guide for 8080/8085-Based Development Systems,* Order Number 121617
- *8080/8085 Floating-Point Arithmetic Library User's Manual,* Order Number 9800452
- *An Introduction to ASM86,* Order Number 121689
- *ASM86 Macro Assembler Operating Instructions for 8086-Based Systems,* Order Number 121628
- *ASM86 Language Reference Manual,* Order Number 121703
- *ASM86 Macro Assembler Pocket Reference,* Order Number 121674

iPDS™
PERSONAL DEVELOPMENT SYSTEM

- **Completely integrated computer system packaged in a compact rugged enclosure for portability**
- **Comprehensive design tool for 8 bit Intel microprocessors**
- **Microprocessor Emulator (EMV) functions**
- **Dual processing capability**
- **Expandable using standard Multimodule™ cards**
- **Desk top computer for CP/M* based applications**
- **640 K byte Integral flexible disk drive; expandable to 1.28 million bytes**
- **Powerful ISIS-PDS disk operating system with relocating macro-assembler, and CRT-based editor**
- **Optional high level languages Fortran 80,PL/M 80, PL/M 88/86 and Basic**
- **Software compatible with previous Intellec systems**
- **PROM programming functions**
- **Bubble Memory option.**

The iPDS Development System is a completely integrated computer system supporting the development of products incorporating Intel 8 bit microprocessors or microcontrollers. Used with its optional emulation vehicles (EMVs) and iUP PROM Programming Personality Modules, the iPDS system provides comprehensive support for integrated hardware and software development, product testing during manufacture, and customer support after the product is in the field. The unit is designed with portability in mind permitting the iPDS Development System to be conveniently transported around the laboratory and into the field. Extensive software is available thereby simplifying and speeding up product development. The software is designed to make the iPDS system easy to use for the novice as well as satisfying the needs of the experienced user. Used with the optional CP/M operating system, the iPDS system becomes a desk top computer that can execute CP/M compatible application programs.



Order Number: 210390-002

FUNCTIONAL DESCRIPTION

Hardware Components

The iPDS case comprises two high impact, shock resistant, poly-carbonate plastic enclosures, that when fitted together, provide a compact and fully enclosed unit. The main enclosure houses a CRT, flexible disk drive, power supply, and base processor printed board assembly. The second enclosure houses the keyboard. On the right side of the unit a spring loaded door allows insertion of an emulator module or an iUP PROM programming module. On the top, a hinged panel covers the storage space for cables and plug-in modules. The carrying handle is attached to the front of the main enclosure and folds away when the system is in use. In the closed position, the iPDS system is 8.15" high, 16" wide, 20" long, and conveniently fits under an airline seat. The basic unit weighs 27 pounds.

BASE PROCESSOR PRINTED BOARD ASSEMBLY-BPB

The Base Processor Board (BPB) contains the powerful 8085A microprocessor, 64K bytes of RAM, CRT/keyboard controller, floppy disk controller, serial I/O port, and parallel I/O port. There are interfaces for connection to the Optional Processor Board, Multimodule Adaptor Board, and the EMV/PROM Programming Adaptor Board.

INTEGRAL CRT

The CRT is a 9 inch green phosphor (P42) unit that displays 24 lines of 80 characters/line with a nominal 15.6 KHz vertical sweep rate. The CRT controller, based on an Intel 8085 and 8275 Programmable Controller Chip is located on the BPB. A single cable containing the signals, power, and ground connect it to the CRT. The contrast adjustment is accessible at the rear of the unit. A pull out bail allows the CRT to be placed in a comfortable operating position of 24 degrees to the horizontal. The standard ASCII set of 94 printable characters is displayable, including upper and lower case alpha characters, and the digits 0 through 9. Another 31 characters for character graphics are defined. If the Optional Processor Board is installed, the second processor shares the CRT with the base processor. The bottom part of the screen is assigned to the processor communicating with the keyboard. The top part of the screen displayed in reverse video is assigned to the other processor. The number of lines appearing on the screen for each processor can be completely controlled by the user via special function keys.

KEYBOARD

The keyboard is housed in a separate enclosure and a flat shielded cable connects it directly to the keyboard controller on the BPB. This 5" cable provides the flexibility to place the keyboard in a comfortable operating position relative to the main enclosure. A total of 61 keys include a typewriter keyset, cursor control keys, and function keys. Auto repeat is available for all keys and is implemented by the keyboard controller. If the Optional Processor Board is installed, it shares the keyboard with the base processor. Initially, the keyboard is assigned to the base processor. It can be assigned to the optional processor by pressing the special function key, FUNC-HOME. Subsequent use of the FUNC-HOME key alternates the keyboard assignment between the two processors.

INTEGRAL FLOPPY DISK DRIVE

The integral floppy disk drive is a 5 1/4", double-sided, 96 tracks-per-inch drive. Diskettes are written double-sided, double density and provide 640 K bytes of formatted storage in the built-in drive. The floppy disk controller located on the BPB is based on the INTEL 8272 floppy disk controller chip, and can control one additional drive. The ISIS-PDS operating system supports the disk drives. If the Optional Processor Board is installed, the integral disk drive is shared by the two processors or it can be exclusively assigned to one of the processors. When shared, only one processor can access a drive at a time. However, the disk drive sharing is transparent to the user since the ISIS-PDS operating system controls the accessing of the drive and automatically resolves file contention.

INPUT/OUTPUT

The iPDS Development System contains two I/O channels located at the rear of the base enclosure and wired to the I/O ports on the Base Processor Board. The serial channel is an EIA RS-232-C interface for asynchronous and synchronous data transfer and is based on the Intel 8251 USART and 8253 timer. The interface can be software configured using the SERIAL command. Full duplex asynchronous operation from 110 to 19.2K baud is selectable.

The parallel I/O interface is an 8 bit parallel I/O port supporting a Centronics type printer. The interface is implemented with an Intel 8255 Programmable Parallel Interface chip. A maximum transfer rate of 600 cps is supported.

Software Components

ISIS-PDS OPERATING SYSTEM

The ISIS-PDS operating system included with the basic iPDS system is designed with a major emphasis on ease of use and simplification of microcomputer development. It is based on the proven ISIS II operating system available on all Intellec Microcomputer Development Systems.

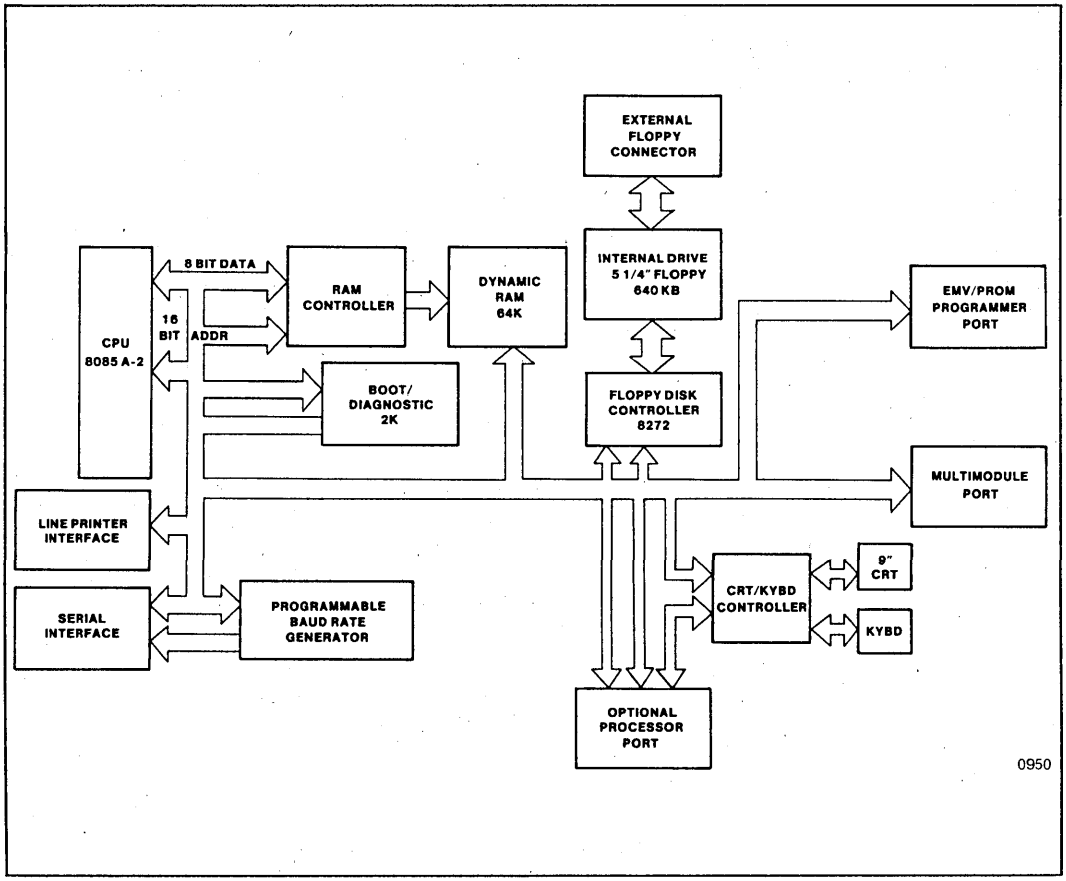


Figure 1. iPDS™ Block Diagram

ISIS-PDS has a comprehensive set of commands to control system operation. These commands can be divided into five functional groups.

- System Management Commands
- Device Management Commands
- File Management Commands
- Program Development Commands
- Program Execution Commands

Table 1 summarizes these commands. The HELP commands are especially useful, providing the user with on-line assistance, eliminating frequent referencing of the manual.

SYSTEM MANAGEMENT COMMANDS

HELP	displays help information for operating system commands.
?	displays the version number of the current Command Line Interpreter.
FUNC-R	software resets the processor to which the keyboard is currently assigned.
FUNC-S	switches the CRT display speed between a slow and fast speed.
FUNC-T	switches the keyboard between typewriter mode and locked upper case mode.
FUNC-HOME	switches the current foreground and background processors.
FUNC-↑	increases the display for the foreground processor by one line and decreases the background processor display by one line.
FUNC-↓	decreases the display for the foreground processor by one line and increases the background processor display by one line.

DEVICE MANAGEMENT COMMANDS

IDISK	initially prepares disks and bubble memory for use with the operating system.
ASSIGN	displays or assigns the mapping of physical to logical devices.
#	re-assigns the system output to the CRT display screen.
FUNC <n>	changes the system input from the keyboard to the file named JOB<n>.CSD where <n> is a one-digit number from 0 to 9.
/	changes the system input from the keyboard to a file or device which is specified by the user.
SERIAL	initializes the serial I/O port.
ATTACH	assigns a row of multimodules to a processor.
DETACH	releases a row of multimodules from a processor.

FILE MANAGEMENT COMMANDS

DIR	displays a list of the files stored on a disk or on bubble memory.
ATTRIB	displays and modifies the attributes of a file.
COPY	transfers files and appends files.
DELETE	removes files from the disk.
RENAME	changes the filename and/or extension of a file.
@	displays the contents of a file on the screen.

PROGRAM DEVELOPMENT COMMANDS

LIB	allows the user to manage a library of MCS-80/85 program modules.
LINK	combines a number of object modules into a single object module in an output file.
LOCATE	converts relocatable object programs into absolute object programs by supplying memory addresses throughout the program.
HEXOBJ	converts a program from hexadecimal file format to absolute object format.
OBJHEX	converts a program from absolute object format to hexadecimal file format.
DEBUG	provides a minimum set of 8080/8085 debugging commands.

PROGRAM EXECUTION COMMANDS

<filename>	loads and executes the object program named <filename>.
SUBMIT	reads an input SUBMIT file, creates a command file containing ISIS commands, and executes commands in sequence from the file created.
•	is a fast form of the SUBMIT command. One command line is read from the SUBMIT file, transformed into an ISIS command in memory, and executed. No intermediate file is created.
/	reads ISIS commands from a disk job file and executes them in sequence. The / command is also considered a device management command.
JOB	stores a sequence of frequently used ISIS commands in a job file as they are entered from the keyboard without executing them until the sequence is completely entered. Two job files, ABOOT.CSD and BBOOT.CSD, deserve special mention. If either of these files is present (ABOOT.CSD for Processor A and BBOOT.CSD for Processor B) when the system is initialized, commands are automatically executed from the file. This feature can be used to configure a system.
ENDJOB	stops the automatic execution of commands from a JOB file and returns control to the keyboard.
ESC	edits the previously entered or the current command line and allows the new command line to be executed.

Table 1. . Functional Summary of ISIS-PDS Commands

ISIS-PDS CREDIT™ TEXT EDITOR

Included with iPDS is the INTEL CRT-based text editor, CREDIT. It is used to create and edit ASCII text files on the Intel Personal Development System. Once the text has been edited, it can be directed to the appropriate language processor for compilation, assembly, or interpretation. CREDIT features, shown in Table 2, are easy to use and simplify the editing and manipulation of text files.

The two editing modes in CREDIT are screen mode and line mode. In screen mode the text being edited is displayed on the CRT and corrected by either typing the new text or using the single stroke character control keys. Single character control keys are used for changing, deleting, inserting, paging forward, and paging backwards.

In command line mode, high level commands are used for complex editing. Examples of the functions available in the command line mode are searching, block moves, copying, macro definitions, and manipulating external files.

8080/85 MACRO ASSEMBLER

The iPDS also includes the INTEL 8080/85 Macro Assembler. This macro assembler translates programs written in 8080/8085 assembly language to the machine language of the microprocessor. It also produces debug data. The Debug utility can be used to troubleshoot the assembler-produced machine language using features such as software breakpoints, single step execution, register display, disassembly, and I/O port access. This reduces the time spent troubleshooting the software and supports modular program development.

UTILITIES

Utility programs included with iPDS are: DEBUG, LIBRARY, LINK and LOCATE. These programs aid in software development and make it possible to combine programs and prepare them for execution from any memory location.

DIAGNOSTICS

The iPDS includes system diagnostic routines executed during system initialization. These routines verify the correct operation of the system and aid the user in fault isolation. Any failures in the basic system components, base processor, CRT/Keyboard, optional processor, or the power supply are indicated by four diagnostic LED indicators mounted on the base processor boards. These LED's are viewed through the spring loaded door on the right side of the unit. When basic system components are operational, additional errors are indicated by messages to the CRT display screen.

CREDIT™ Editor features two editing modes: cursor-driven screen editing and command line context editing

CRT Editing Includes:

- Displays full page of text
- Single control key commands for insertion, deletion, page forward and backward
- Type-over correction and replacement
- Immediate feedback of the results of each operation
- The current state of the text is always represented on the display

Command Line Editing Includes:

- String search and substitute
- String delete, change, or insert
- Block move
- Block copy
- User-defined macros
- External file handling
- Change CREDIT features with ALTER command
- Conditional iteration
- User-defined tab settings
- Symbolic tag positions
- Automatic disk full warning
- Runs under ISIS-II SUBMIT facility
- Option to exit at any time with original file intact
- HELP command

Table 2. Summary of CREDIT™ Editor Features

After ISIS-PDS is loaded and started, additional confidence tests are available to verify correct system operation. These tests included on the system disk, run as utilities under the operating system and can be selectively executed to verify individual functions on the main processor board, optional processor board, bubble memory Multimodules and EMV/PROM Programmer Adaptor.

iPDS™ HARDWARE OPTIONS

Add-On Mass Storage

Mass storage can be increased by adding one external flexible disk drive. This adds 640 K bytes of formatted mass storage. The maximum disk storage available on iPDS is 1.28 M Bytes. The optional drive is vertically mounted and housed in a plastic enclosure with its own power supply. A 20" cable connects the optional floppy drive to the external disk drive connector on the rear of the iPDS system.

The iPDS system also supports Intel's iSBX-251 Bubble Memory Multimodule. A maximum of two bubble multimodules can be added. Each contain 128 K bytes of non-volatile memory. Bubble memory Multimodules can only be added to a system containing the Multimodule Adaptor Board. The bubble memory is treated by the ISIS-PDS and CP/M operating system as an additional disk drive with the same file structure and directory structure as a diskette. The bootstrap ROM is programmed to boot the operating system from the bubble. The iSBX-251 has no moving parts, making it ideal for applications where ruggedness is an important consideration. The bubble memory is also recommended for systems requiring portability, since it is completely enclosed in the iPDS main unit.

Optional Processor Board

The Optional Processor Board provides dual processing capabilities and increases the processor power of the iPDS system. A different program can be run on each of the processors at the same time, providing a greater processing throughput. Each processor operates under ISIS-PDS control. The Optional Processor Board also provides a convenience feature for accessing directories, file displays, and HELP without interrupting the main processor task.

The Optional Processor Board contains functions identical to the base processor. There is an 8085A CPU with 64 K bytes of dynamic RAM and an additional 2 K bytes of bootstrap ROM.

Both processors share the keyboard, the CRT display unit, the disk drives, and the multimodules. Serial or parallel I/O ports can be added to the optional processor through iSBX multimodules. Each processor runs the ISIS-PDS operating system and applications programs in its own 64 K byte memory space, independent of the other processor. Special hardware function keys are provided to facilitate procedures necessary in the dual processing environment. These procedures include independent initialization of each processor, sharing of the CRT display, and assignment of the keyboard. The ISIS-PDS commands facilitate sharing of disk drives, multimodules, and files.

Emulation Vehicles (EMVs)

Emulation vehicles (EMVs) for use with the iPDS Devlopment System, are available for debugging a variety of Intel microprocessor families. Emulators consist of hardware and software. The EMV hardware is inserted into the EMV/iUP Personality Module port of the iPDS. The optional EMV/Prom Programming Adaptor Board is required to install the EMV's. The emulator software runs under the ISIS-PDS operating system and provides the user's interface to the emulator.

An EMV contains features used to debug microprocessor designs quickly and efficiently. It provides a controlled environment for exercising a user design and monitoring the results. It exactly duplicates the behavior of a target microprocessor/microcontroller in the user's prototype system while providing information to the user to aid in integrated hardware and software development. EMV's provide features for real time full speed emulation as well as single step execution of a user's design. Breakpoint features allow the user to specify a portion of the program to execute and then stop for interrogation. During execution, the EMV automatically collects execution history in the trace buffer. Once stopped at the breakpoint, the emulator acts as a window to the internal registers and logic signals inaccessible from the connector pins. This provides for examination and alteration of the internal state of the microprocessor.

The emulator accepts symbolic debug data, such as symbol tables produced by the language translators. Therefore, when debugging, the programmer can reference locations in the program elements with the symbol names used in

the source program, rather than absolute memory addresses.

Another advantage of using an emulator is functional prototype hardware is not required to begin software debugging. The emulator duplicates the behavior of the target microprocessor and provides some resources, such as memory, that can be used until the hardware prototype is closer to completion.

The software controlling the emulator comprises a set of commands the user enters to directly control interactive debugging sessions. The command families are listed in Table 3. Also, sequences of emulator commands can be executed automatically from a file, providing a basis for manufacturing and field test routines.

iUP Personality Modules

The iPDS accepts most Intel PROM Programming Personality Modules from our new iUP-200/201 product line. These modules provide all the hardware and firmware needed for programming entire families of Intel EPROMS, E^2PROMS, and micro controllers containing on-chip EPROM. The optional EMV/PROM Programming Adaptor Board is required to use the iUP Personality Modules. Intel Prom Programming Software (IPPS) runs under the ISIS-PDS operating system and is included with the EMV/PROM Programming Adaptor Module. This software provides a set of commands to control the programming and verification of the devices.

Emulation Commands

BR - Display breakpoint menu
BR0, 1, 2, 3 - Change/display breakpoint register for execution address
BRR - Change/display breakpoint register for execution range
BRB - Change/display break on branch
BV - Change/display break on value
BC - Clear all breaks
TB0, 1, 2, 3 - Enable/disable display by bit value
TR0, 1, 2, 3 - Enable/disable display by execution address
TV - Enable/disable display by register value
TR - Enable/disable display of registers
TS - Enable/disable display of PSW
TD - Enable/disable display of code disassembly
STEP - Enter slow down emulation mode
GO - Enter real-time emulation mode

Advanced Commands

MACRO - define, and display macro
IF THEN
COUNT
REPEAT } CONTROL CONSTRUCTS
WHILE
UNTIL
FUNCTION KEY - invoke macro assigned to function key

Utility Commands

HELP - Displays command syntax
LOAD - Loads object file in mapped memory
LIST - Generates copy of emulation work session
DEFINE - Defines symbol or macro
SYMBOL - Displays symbols
REMOVE - Deletes symbol or macro
ENABLE/DISABLE - Control for expanded display
EVALUATE - Evaluate any expression
SUFFIX/BASE - Sets input and display numeric base
SAVE - Save code memory to file
RESET - Resets emulation processor
EXIT - Terminate emulation session

Display/Modify Commands

REGISTER - Menu for change/display registers
MEMORY - Menu for change/display memory
DUMP - Display memory as ASCII and Hexadecimal
ASM/DASM - change/display code memory as assembly language mnemonics

Table 3. Summary of Typical Emulator Commands

Figure 2. iPDS™ With Optional Modules Installed

EMV/PROM Programming Adaptor Board

The EMV/PROM Programming Adaptor Board provides an interface between the Base Processor Board and EMV or PROM programming modules. This option is required before either of these modules can be operated with the iPDS.

Multimodules

The iPDS is expanded by utilizing a variety of Intel iSBX multimodule boards. The Multimodule Adaptor Board allows a maximum of four multimodule boards to be added. Multimodule boards are small, special function boards using the iSBX bus to interface to the CPU. The available iSBX multimodule boards include:

- iSBX 251 Bubble Memory Multimodule Board
- iSBX 350 Parallel Port Multimodule Board
- iSBX 351 Serial Port Multimodule Board
- iSBX 488 IEEE-488 Interface Multimodule Board

The INSITE Software Library contains many software routines for these multimodules. The iPDS user manual contains technical information for writing custom I/O driver routines.

Multimodule Adapter Board

The Multimodule Adapter Board provides an interface between the Base Processor Board and the Multimodule options. It is required before any Multimodule options can operate with the iPDS system.

iPDS™ SOFTWARE OPTIONS

High Level Languages

High level languages help reduce system design

effort and maintenance cost by allowing the programmer to design software at a more abstract level. A block structured language, PL/M 80, is available for the 8085, along with Fortran 80, Pascal 80 and Basic 80.

Software Support for Additional Microprocessors

Assemblers and high level languages for different target microprocessors are available to aid the software development effort. These include ASM-51, PL/M 88/86, ASM 88/86. and ASM 8048/49.

General Purpose Computing Software

The iPDS can also be used as a general purpose desk top computer. The widely used CP/M microcomputer operating system is available for the iPDS from Intel. It supports iPDS systems with single or multiple disk drives, and iPDS systems using bubble memory for mass storage. CP/M compatible software will come from three sources; vendors of CP/M based software programs, independent software makers, and Intel. The software programs available from Intel include high level languages, wordprocessing software and an electronic spreadsheet. New applications packages are also planned.

File Transfer Package

Transferring files between the iPDS system and any of Intel's Intellec Development Systems is accomplished using the iPDS-FTRANS option. This product uploads/downloads files via the RS232C serial link and under control of software running on both the iPDS and the Intellec system. Data transmission is monitored and any errors are displayed. Transfer rates up to 19.bk Baud can be selected. FTRANS can also be used to transfer files between remote systems using telephone modems.

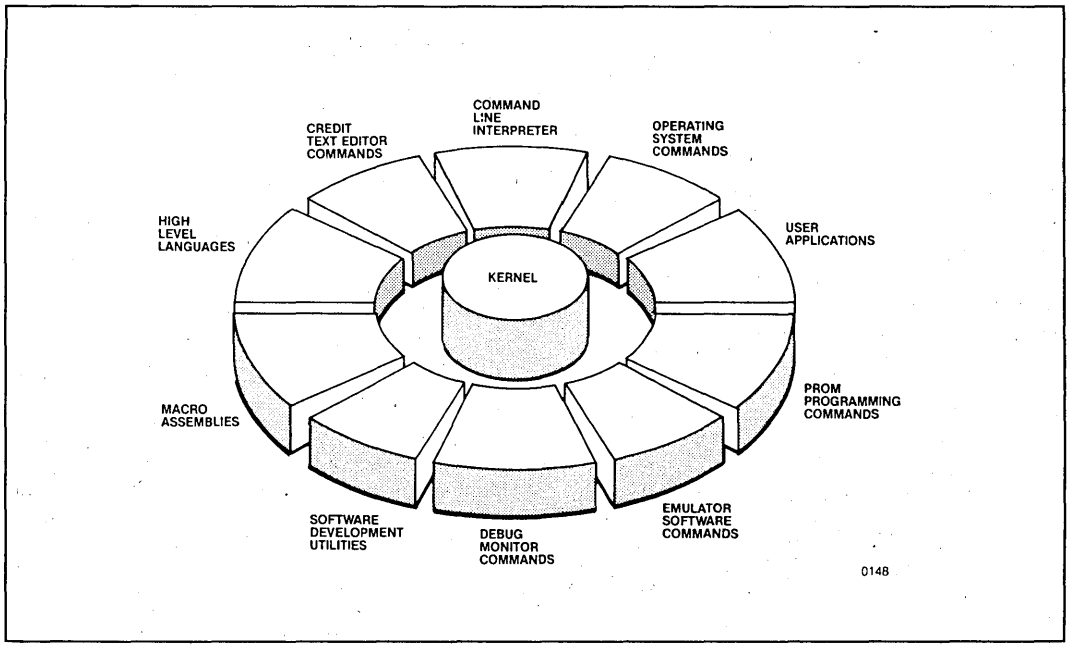


Figure 3. Overview of iPDS™ Software Environment

SPECIFICATIONS

Host Processor

8085A-2 based, operating at 5.0 MHz

Memory

RAM - 64K of User Memory on BPB
ROM - 2K bytes (Boot/diagnostic)

I/O Interfaces

I/O Serial Channel; RS-232 at 110-19.2K baud (asynchronous) or 150-56K baud (synchronous). Baud rate and serial format software controllable.

I/O Parallel Channel; 8 bit parallel supporting Centronics type printer. Transfer rate up to 600 characters per second.

Memory Access Time

RAM - 450 ns.

Option Electrical Requirements

Option Electrical Requirements (Max. in Amperes)									
Power Supply Voltage	Optional Processor	EMV/PROM Adaptor	Multimodule Adaptor	iSBX 350	iSBX 351	iSBX 251	iSBX 488	EMVs	iUP
+5 volts	1.0	0.3	0.6	0.62	0.53	0.37	0.6	2.5	0.7
+12 volts	–	0.18	–	–	0.03	0.4	–	–	0.85
-12 volts	–	0.05	–	–	0.03	–	–	–	0.4

Maximum option power requirements must not exceed 33.6 watts for any configuration.

Integral Flexible Disk Drive

System Storage Capacity
DS/DD - 640K bytes (formatted)
Data Transfer Rate
250K bits/sec.
System Access Time
Track to Track: 6 msec.
Rotational Speed: 300 rpm
Motor Start Time: 0.4 sec. max.
Media
5 1/4" disk with 1 index hole

Physical Characteristics

Closed Unit (without options)

Height	8.15 in
Width	16 in.
Depth	20 in.
Weight	27 lbs.

Power Requirement

Input Voltage:
115/220 VAC Selectable Single Phase
115 VAC (90 VAC-132 VAC) 47-63Hz, 1 amp
220 VAC (180 VAC-264 VAC) 47-63Hz, 0.5 amp

ENVIRONMENTAL CHARACTERISTICS

Operating

Temperature 10° C to 30° C
Relative Humidity 20% to 80%
Maximum wet bulb - 25.6° C

Non-Operating

Temperature -40° C to 62° C
Relative Humidity 5% to 95%
(non-condensing)

Operating Vibration

0 to 0.004 inches peak to peak excursion from 10 to 55 Hz.

Non-Operating Shock

15 G with shock wave of 20 ms duration, I/2 sine wave.

Equipment Supplied

iPDS Enclosure including:

- Base Processor Board (BPB)
- CRT/Keyboard
- Integral Floppy Disk Drive
- System Diskette with ISIS-PDS operating system
- MCS-80/MCS-85 Macro Assembler
- Debug-85, Link, Locate and Library Utilities
- CREDIT CRT-based text editor
- System confidence tests.

iPDS Literature Kit including:

- Intel Personal Development System User's Guide 162606
- Intel Personal Development System Pocket Reference 162607
- 8080/8085 Assembly Language Programming Manual 9800301
- 8080/8085 Assembly Language Reference Card 9800438
- MCS-8085 Utilities User's Guide for 8080/8085 Based Development System 121671
- ISIS II 8080/8085 Macro Assembly Operating Manual 9800292

Reference Manuals

- A Guide to INTELLEC Microcomputer Development System 9800558
- ISIS-II System User's Guide 9800306

Ordering Information

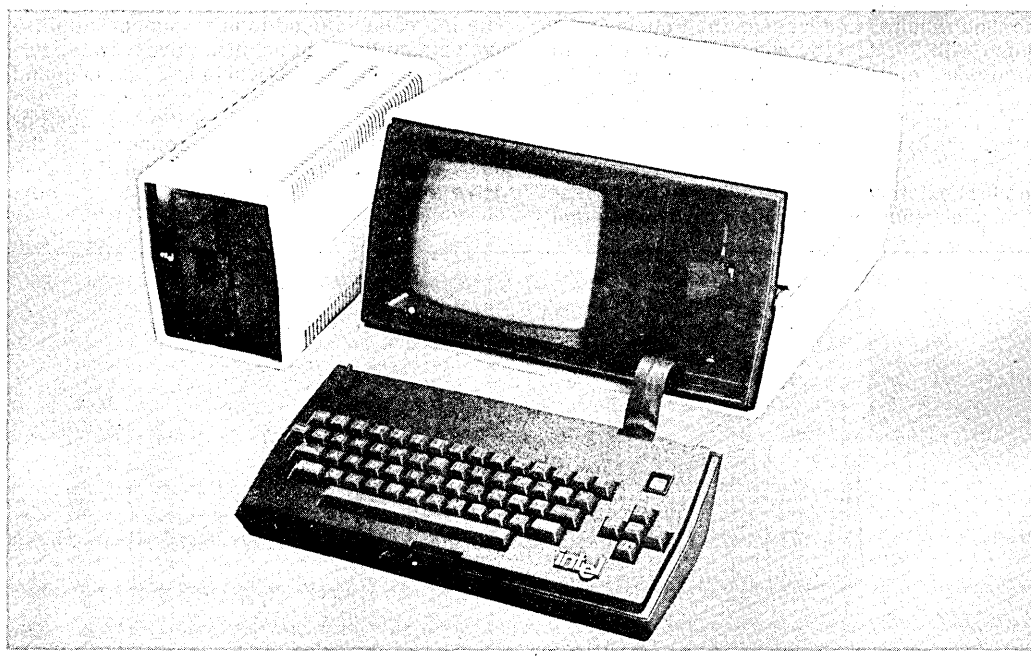
Part Number	Description
iPDS-100	iPDS System
iPDS-110	Optional Processor Board
iPDS-120	Multimodule Adapter Board
iPDS-130	Add-On Disk Drive
iPDS-140	EMV/PROM Programming Adaptor Board
iPDS-FTRANS	iPDS/iMDX File Transfer Package

* Registered Trademark of Digital Research Inc.

THE iPDS™-130 OPTIONAL FLEXIBLE EXTERNAL DISK DRIVE FOR THE iPDS™ PERSONAL DEVELOPMENT SYSTEM

- **Each disk drive provides 640K bytes of formatted mass storage.**
- **Daisy-chaining up to 3 disk drives provides a total of 2.56M bytes storage capacity.**
- **Each disk drive has its own power supply.**
- **Disk drives use industry-standard 5-¼ inch flexible diskettes as the storage medium.**
- **Disk drive has transfer rate of 4 microsec/bit, a recording density of 5922 bpi, and dual heads.**
- **Use of external disk drive eliminates disk swapping when making duplicate disks.**

When using the iPDS™ personal development system, applications may be developed that require more storage capacity than is provided by the integral disk drive of the system. The iPDS-130 optional external flexible disk drive provides the needed additional mass storage. Up to three disk drives may be added to the iPDS system, with each additional disk drive providing 640K bytes of (formatted) capacity. This means that a maximum disk storage of 2.56M bytes is available. The photograph below shows the iPDS-130 external disk drive with the iPDS system. Figure 1 shows some features of the iPDS-130 disk drive.



February 1984
Order Number: 231020-001

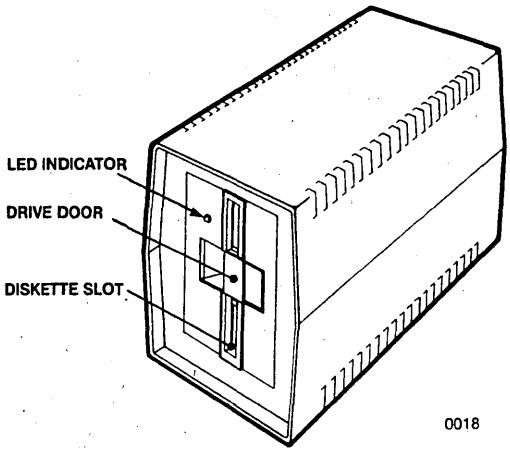


Figure 1. iPDS™-130 Flexible Disk Drive

Creating back-up diskettes is good programming practice and the iPDS-130 disk drive provides the means to create these back-ups. It shortens the time required and lessens the trouble associated with this task by eliminating the need to swap disks during the duplication process. The master diskette can be inserted in the iPDS system's integral disk drive and the duplicate diskette in the external disk drive.

The first external disk drive attaches to the rear of the main enclosure, and the other two external drives are connected to the rear of the previous external drive. Each additional drive has its own power supply and is mounted in its own housing. Figure 2 shows the iPDS unit with all three external drives.

HARDWARE

Each drive is 7.3 in. high and weighs approximately 11 lbs. The front of each disk drive contains a door, a door release mechanism, and a drive indicator that is lit during disk I/O operations. The drive is mounted in the vertical position. Different ac voltage ranges may be selected. The rear panel of the drive contains the ac power connector, the power ON/OFF switch, a fuse holder, a voltage selector card, and two I/O cable connectors. Figure 3 shows the disk drive's rear panel.

I/O Cable

The I/O cable is used to interconnect the iPDS system and the external disk drives. The external portion of the input cable is 30 in. long and connects to the flexible disk connector on the rear of either the iPDS unit or the previous optional disk drive. The output connector of the daisy-chain mounts on the rear panel of the disk drive and provides the connector to the next disk drive.

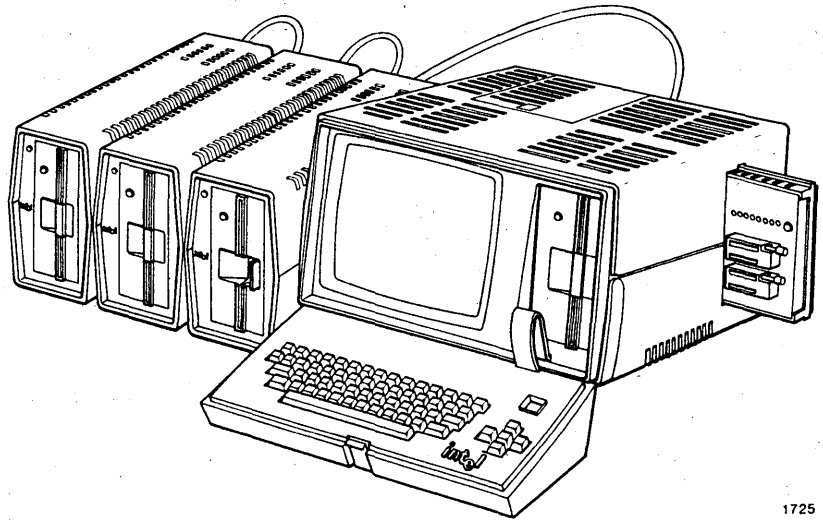


Figure 2. iPDS™ System with External Flexible Disk Drives

Order Number: 231020-001

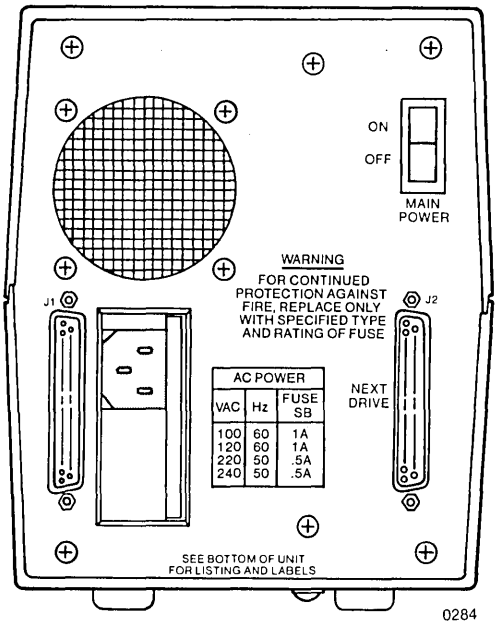


Figure 3. iPDS™-130 Optional Flexible External Disk Drive Rear Panel

Power Supply

The flexible disk drive unit contains a linear power supply with a maximum power input of 40 watts. The output consists of two regulated dc voltages (5v and 12v).

I/O SPECIFICATIONS

Floppy Disk Interface

The floppy disk interface controls up to four 5-¼ in. double-sided 96 tpi floppy disk drives.

The floppy disk is a 5-¼ in., 96 tpi, dual-headed unit. With a total of 80 tracks of sixteen 256-byte sectors per side, the formatted capacity of the unit is 640K bytes. The interface is the industry standard for 5-¼ in. drives.

OPTIONAL FLEXIBLE EXTERNAL DISK DRIVE SPECIFICATIONS

The specifications for the optional flexible external disk drive are given in Tables 1 through 4.

Table 1. Environmental Characteristics

Temperature	
Operating	10°C to 35°C
Non-operating	−40°C to 62°C
Humidity	
Operating	20% to 80% non-condensing
Non-operating	5% to 95% non-condensing
Cooling	Up to 60 watts are dissipated by fan cooling

Table 2. Physical Characteristics

Width	6.1 in (155.4mm)
Height	7.3 in (174.2mm)
Depth	13.8 in (350.6mm)
Weight	11.0 lbs. (5.0kg)

Table 3. Electrical Characteristics

Input power	90 VAC to 132 VAC, 47 Hz to 63 Hz; or 198 VAC to 264 VAC, 47Hz to 63Hz
Drive power	12 VDC ± 1%
Logic power	5 VDC ± 1%
Adjustable range	± 5%, drive and logic
Power dissipation	25 watts average, 34 watts maximum

Table 4. Functional Specifications

Transfer rate	4 μsec/bit
Rotational speed	300 rpm ± 1.5%
Track density	96 tpi
Number of cylinders	80
Number of sides	2
Recording density	5922 bpi
Encoding method	MFM
Unformatted capacity	6.25K bytes/track
Formatted capacity	640K bytes
Motor start time	0.4 sec maximum
Track-to-track step rate	6 msec maximum
Side-to-side delay time	0.2 msec maximum
Head loading time	35 msec maximum
Head setting time	15 msec maximum
Medium	Industry standard 5-¼ in. with single hole

Order Number: 231020-001

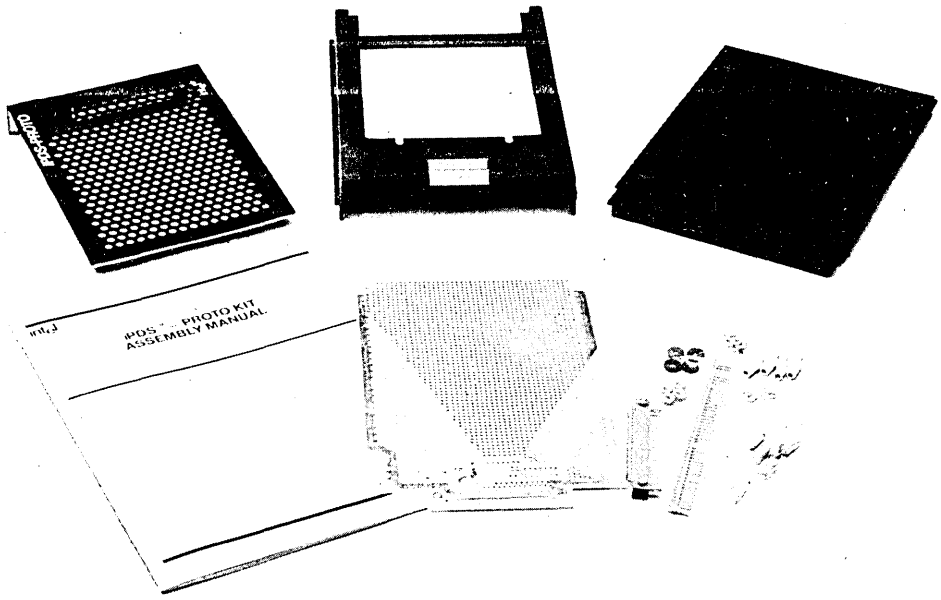
ORDERING INFORMATION

Part Number	Description
iPDS-130	Optional external flexible disk drive

iPDS™-PROTO KIT

- **Design aid for developing your own specialized plug-in modules for the iPDS™ development system and for the iUP-200/201 system, such as:**
  - **Emulation vehicle (EMV) modules**
  - **PROM or Programmed Logic Array (PLA) programming modules**
  - **Instrumentation modules (logic or signature analyzers)**
  - **Specialized communications modules**
  - **Analog interface modules**
  - **Program storage modules**
- **iPDX bus interface**
- **Easy-to-follow assembly instructions**

The iPDS™-PROTO Kit is a complete kit for engineers who want to enhance the iPDS development system and the iUP-200/201 Universal Programmer system by developing their own specialized plug-in modules such as those noted above. The module case and PROTO board are specifically designed to plug into both the iPDS system and the iUP-200/201 system.



AUGUST 1984
ORDER NUMBER: 280046-001

KIT COMPONENTS

The iPDS-PROTO Kit comprises the module case, the PROTO board, and a hardware kit. The hardware kit includes one iPDS bus connector, five isolation capacitors, wire-wrap pins, screws, washers, and lock nuts. Also included are the *iPDS™-PROTO Kit Assembly Manual* and the application note, *Designing Modules for the iPDS™ and iUP Systems* (order number 230682).

The PROTO board can accept up to 30 integrated circuits and associated discrete components.

iPDX BUS INTERFACE

The iPDX bus is a byte-wide, parallel interface between the plug-in module and the iPDS development system or the iUP-200/201 system. For further information on the iPDX bus, refer to the *Designing Modules for the iPDS™ and iUP Systems.*

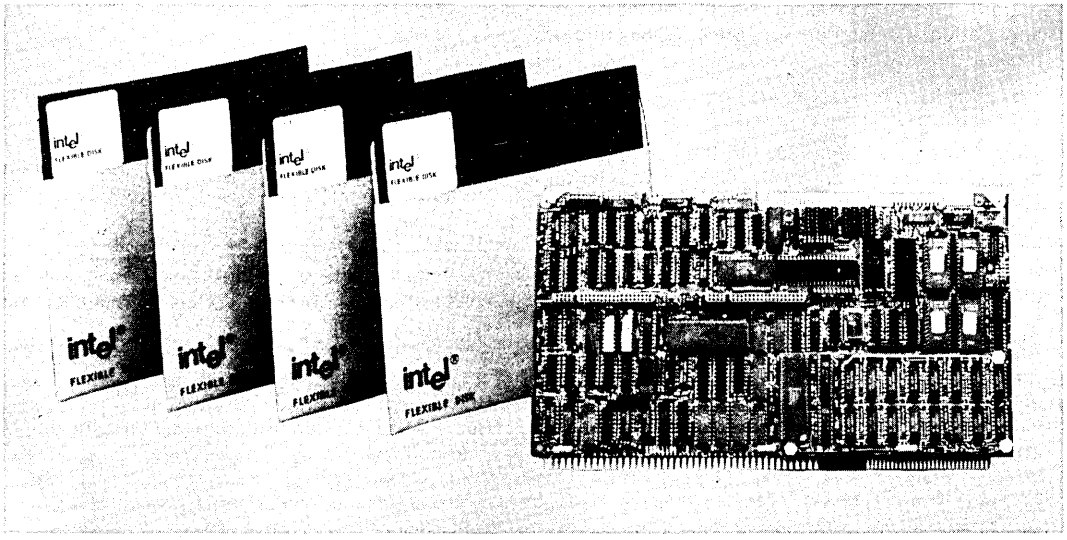
ORDERING INFORMATION

Part Number	Description
iPDS-PROTO Kit	iPDS-PROTO board, module cover, hardware kit, and assembly manual.

iMDX 557
iAPX Resident Processor Card Package

- **High-Performance 8086-Based CPU Board for Increased Intellec® Development System Performance and Improved iAPX 86/88 Development Environment**
- **Upgrades Intellec® Series II and Model 800 Microcomputer Development Systems to the Functionality of Series III Systems**
- **224K Bytes of User Program RAM Memory Available for iAPX 86/88 User Programs**
- **Software Applications Debugger for iAPX 86/88 User Programs**
- **Supports Full Range of iAPX 86/88-Resident, High-Level Languages: PL/M-86/88, PASCAL-86/88, and FORTRAN-86/88**
- **Includes iAPX 86/88-Resident Relocating Macro Assembler, Linker, Locator, and Librarian**
- **Dual Processor Disk Operating System Software with 16-bit AEDIT Editor**
- **Supports PSCOPE™ Advanced 86/88 Software Debugger**

The iMDX 557 is a performance enhancement package for Intellec® Series II and Model 800 Development Systems, specifically designed for iAPX 86/88 microprocessor development. The iMDX 557 includes an iAPX-based CPU board with 256K RAM memory, CRT-based menu-driven editor, iAPX 86/88-Resident Relocating Macro Assembler, Linker, Locator and Librarian, software applications debugger for iAPX 86/88 user programs, and complete user documentation. The DX-557I kit includes an iMDX 557 plus an IPC-85.



MAY 1983
ORDER NUMBER: 210534-001

FUNCTIONAL DESCRIPTION

Hardware Components

Resident Processor Card RPC-86 - The heart of the RPC-86 is an Intel 8086-2 16-bit HMOS microprocessor, running at 8.0 MHz. There are 128K bytes of RAM memory provided on the board, with transparent refresh from the Intel 8203 dynamic RAM controller. There are 16K bytes of ROM on the board, preprogrammed with an iAPX 86/88 applications debugger. The debugger provides features necessary to debug and control execution of applications software for the iAPX 86/88 microprocessors. The 8086 and the host processor use interrupts for interprocessor communications.

RAM Multi-Module - The module contains an additional 128K bytes of read/write RAM memory. Refresh hardware is provided on-board for all the dynamic memory elements. Data buffering occurs for all data written to or read from the memory array. The RPC-86 board with the RAM multi-module occupies two slots in the Intellec cardcage.

SYSTEM FEATURES

The iMDX 557 offers many key advantages for iAPX 86/88 applications and Intellec Development Systems: enhanced system performance through a dual-host CPU environment, a full spectrum of iAPX 86/88-resident high-level languages, expanded user program space for iAPX 86/88 programs, and a powerful high-level software applications debugger for iAPX 86/88 microprocessor software.

Dual-Host CPU

The addition of a 16-bit 8086 to the existing 8-bit host CPU increases iAPX 86/88 compilation speeds and provides for iAPX 86/88 code execution. When the 8086 is executing a program, the 8-bit CPU off-loads all I/O activity and operates as an intelligent I/O controller to double buffer data to and from the 8086. The 8086 also provides an execution vehicle for 8086 and 8088 object code. An added benefit of two-host microprocessors is that 8-bit translations and applications are available in the same system. This feature provides complete compatibility for current systems and means that software running on current Intellec Development Systems will run on the new system.

High-Level Languages for iAPX 86/88

The iMDX 557 allows the current Intellec system user to take advantage of a breadth of new resident iAPX 86/88 high-level languages: PL/M 86/88, PASCAL 86/88, and FORTRAN 86/88. The iAPX 86/88 resident Macro assembler and these high-level language compilers execute on the host CPU, thereby increasing system performance.

Extended Program Memory

By adding an iMDX 557 to an existing Intellec Development System, 224K bytes of user program RAM memory are made available for iAPX 86/88 programs. System memory can be expanded by adding RAM memory boards. This, combined with the dual-host CPU system architecture, dramatically increases the processing power of the system.

Software Applications Debugger

The RPC-86 contains the applications debugger which allows iAPX 86/88 programs to be developed, tested, and debugged within the Intellec system. The debugger provides a subset of in-circuit emulator commands such as symbolic debugging, control structures and compound commands specifically oriented toward software debugging needs.

ALTER™ Editor

This 16-bit based, menu-driven, full-screen editor is included with the iMDX 557. Designed for the programmer, it has features that allow easy code generation and fast, convenient program alteration.

SPECIFICATIONS

Resident Processor Card (RPC-86):
8086-based, operating at 8.0 MHz with 128K RAM memory module
RAM - 256K bytes on the CPU board including the 128K RAM multi-module
ROM - 16K bytes (applications debugger)
Bus - MULTIBUS architecture; 8.0 MHz maximum transfer rate

Electrical Characteristics
DC Power Supply

Voltage Requirements	Current Requirements (Amperes Max.)
+5±5% Volts	5.6 A
+12±5% Volts	25 mA
−12±5% Volts	23 mA

Environmental Characteristics (constrained by Series II mainframe)

Operating Temperature: 10° to 35° C (50° to 95° F)
Relative Humidity: To 20% to 80% (non-condensing)

Equipment Supplied

iAPX 86 Resident Processor Card (RPC-86) with 128K Byte RAM Multi-module
Self-test Diagnostics
iAPX 86/88 Applications Debugger
iAPX 86/88 Resident Macro Assembler and Utilities
AEDIT Text Editor

DOCUMENTS SUPPLIED

A Guide to Intellec Series III Microcomputer Development Systems, 121632

Intellec Series III Microcomputer Developmen' System Product Overview, 121575

Intellec Series III Microcomputer Development System Console Operating Instructions, 121609

Intellec Series III Microcomputer Development System Pocket Reference, 121610

Intellec Series III Microcomputer Development System Programmers Reference, 121618

iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems, 121616

An Introduction to ASM86, 121689

ASM86 Reference Manual for 8086-Based Development Systems, 121703

8086/8087/8088/80186 Macro Assembler Language Pocket Reference, 121674

8086/8087/8088/80186 Macro Assembler Operating Instructions for 8086-Based Development Systems, 121628

MCS-86 Assembly Language Converter Operator Instructions, 9800642

Model 557 Installation Manual, 122015

MCS-80/85 Utilities User's Guide, 121617

iAPX 86,88 Family Utilities Pocket Reference, 121669

iAPX 86,88 User's Manual, 210201

iAPX 88 Book, 210200

AEDIT (CRT-Based Text Editor) User's Guide

AEDIT (CRT-Based Text Editor) Pocket Reference

Additional manuals may be ordered from any Intel sales representative or distributor office, or from Intel Literature Department, 3056 Bowers Avenue, Santa Clara, California 95051.

ORDERING INFORMATION

Part Number	Description
iMDX 557	Performance upgrade package for Intellec Series II/85 and Model 800 Microcomputer Development Systems (110V/60 Hz or 220V/50Hz). Specifically designed for iAPX 86/88 microprocessor development. Upgrades Intellec Series II models to Intellec Series III Development Systems.
DX-557I Kit	Performance upgrade package for Intellec Series II/80 Microcomputer Development Systems. Specifically designed for iAPX 86/88 microprocessor development. The 557I package consists of the iMDX 557 software and hardware performance package and the integrated 8085 processor board (IPC-85). This upgrade package is only for Intellec Series II/80 Development Systems (110V/60 Hz or 220V/50 Hz) and upgrades these models to the full performance and functionality of an Intellec Series III Development System.

AFN-00646B

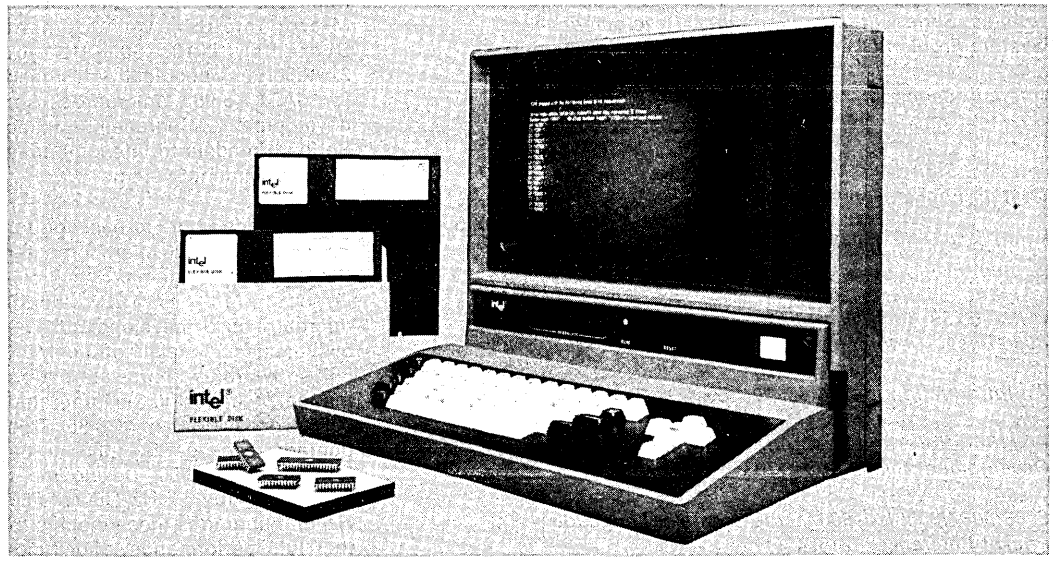
iMDX-511
ENHANCED HUMAN INTERFACE

- **Command Line Recall/Command Line Edit permits rapid entry of repetitive command sequences and aids in the correction of errors.**
- **Auto Repeat Function provides single key character repeat.**
- **Scrolling and Paging improve file viewing, making information easy to find.**
- **Batch Job Capabilities allow more efficient system utilization.**
- **Soft Keys for frequently used ISIS commands reduce keystrokes and increase productivity.**
- **Customizable soft keys invoke user-programmed command sequences.**
- **Keyboard Help reminds the user of commands and the associated soft keys.**
- **Enhanced CRT Interface provides direct cursor addressing and block write to the screen.**

User productivity is a major element in the development of microprocessor designs. These enhancements provide a more useable system requiring fewer keystrokes to complete the usual development sequence. The system guides the user to the correct command, helps minimize the typing and allows for easy correction/changes to repetitive command sequences.

KEYBOARD FEATURES

The keyboard interface has been improved to allow fewer keystrokes and easier key manipulation. The basic function of the repeat key (RPT) has been changed. A given character will continually repeat one half second after its key is depressed and held. The new "RPT" key (FUNC) allows automatic command printing in "soft-key fashion." For example, the Intel ISIS operating system requires reference to a physical device (:Fn:). The "FUNC" key plus a given number key will produce ":Fn:"; where n is the number depressed. Common commands such as DIR, COPY, and command syntax elements, such as space TO space, can be selected by pressing the "FUNC" key and a specified character. Ten customizable keys invoke user-programmed JOB files, allowing single keystroke access to complex command sequences.



MARCH 1984
ORDER NUMBER: 210504-002

KEYS DEPRESSED				RESULT
FUNC	+	"1"	=	:F1:
FUNC	+	"A"	=	AEDIT
FUNC	+	"C"	=	COPY
FUNC	+	"D"	=	DIR
FUNC	+	"E"	=	CREDIT
FUNC	+	"H"	=	HELP MENU
FUNC	+	"I"	=	ATTRIB
FUNC	+	"J"	=	JOB
FUNC	+	"K"	=	DELETE
FUNC	+	"L"	=	:LP:
FUNC	+	"M"	=	LOGON
FUNC	+	"N"	=	ASSIGN
FUNC	+	"O"	=	LOGOFF
FUNC	+	"P"	=	:SP:
FUNC	+	"R"	=	RUN
FUNC	+	"S"	=	SUBMIT
FUNC	+	"T"	=	TO
FUNC	+	"U"	=	ACCESS
FUNC	+	"X"	=	EXPORT
FUNC	+ SHIFT +	"1"	=	/JOB1 "CR"

KEYBOARD HELP

A HELP facility is available that indicates the appropriate soft key to get a particular command element.

COMMAND LINE INTERPRETER (CLI)

The CLI has been enhanced to provide edit capability. Typed command strings do not have to be cancelled and/or retyped. Depressing "ESC" and repositioning the cursor allows user to add or delete command line characters as necessary. Command line recall (depressing "ESC") allows the user to redisplay the last entered command, for correction or update. This eliminates the need to reenter commands with similar content.

Viewing the contents of files is more flexible. Entire screens of text will be displayed for the user to review. A keystroke will cause the next complete page of text to be printed on the screen. File contents can be scrolled on the screen as well. The scrolling speed can be selected, fast, slow, or line by line, by a keystroke.

Several commands can be staged and executed in "BATCH" fashion. That is, all the commands can be entered before any is executed.

Frequently used single line command strings can be stored and passed a parameter for execution.

Example:
.L MYJOB

Executes the pseudo SUBMIT file L.CSD which contains:
RUN LINK86 %O.PAS,CEL.LIB,P86RNO.LIB etc.

Output from program execution can be directed to a file for future reference.

VIDEO DISPLAY

The new CRT screen handler which provides direct cursor addressing makes available many graphic capabilities of the 8275. User-written procedures allow the use of reverse video, blinking and underscore of characters on the screen.

SPECIFICATIONS

Hardware Provided

4 ea. 2716
1 ea. 8741A
1 ea. Single Density Diskette
1 ea. Double Density Diskette
1 Key Cap

Hardware Required

Intellec Series II/80 or Series II/85 Development System (Model 220, 225, 230, 235, or 286)

Software Provided

ISIS-II

Software Supported

ISIS-II (W)
ISIS-III (N)

ORDERING INFORMATION

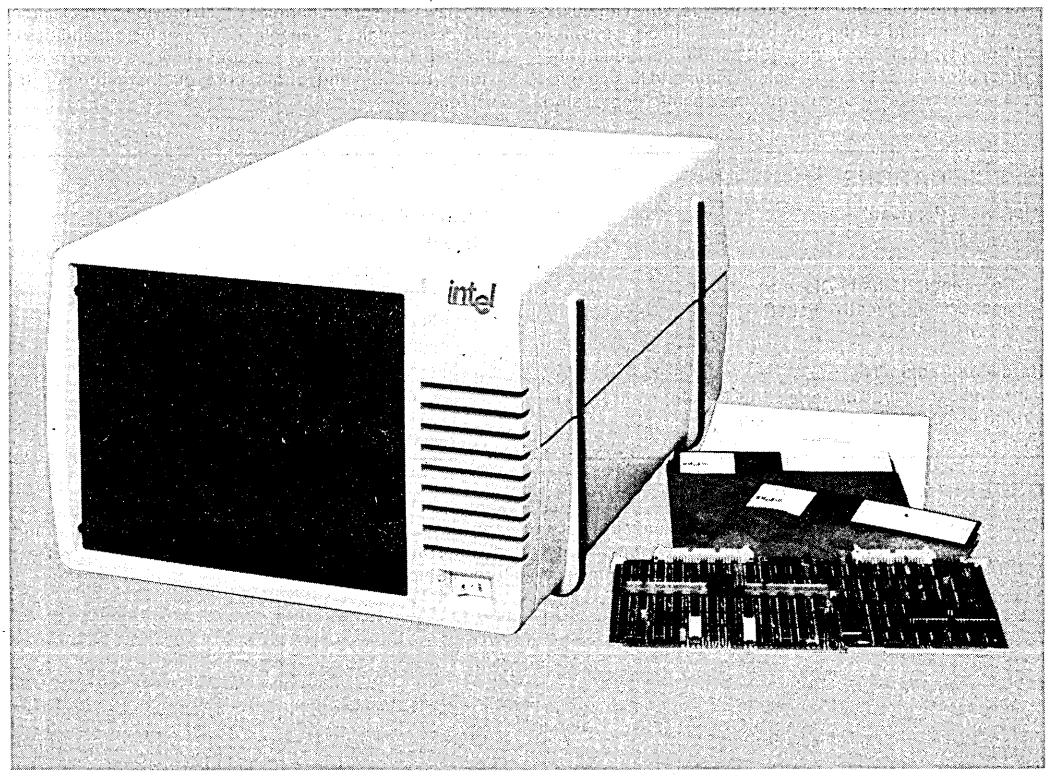
IMDX 511 Series II Human Interface Enhancement package. This package contains 4 EPROMs, 1 microcontroller, and software to provide the Intellec Development System with soft keys, command line edit/recall, enhanced text viewing and other improved human interface features. (Shipping weight 3 lbs.)

Model iMDX-750
Intellec® Series II/III/IV Winchester Subsystem

- **22 Mbyte formatted capacity**
- **High performance speeds program development**
- **Reliable 8-inch Winchester technology increases disk integrity**
- **Upgrades Series II, Series III, and Series IV development systems**
- **Freestanding chassis with power supply and cables**
- **Provides intelligent archival utility**
- **Upgradable to the NDS-II, distributed multi-user network**

Intel's Model 750 Winchester Disk subsystem provides on-line, high-capacity storage to improve system throughput and reduce development time.

Both disk access speed and data transfer rate of the iMDX-750 are faster than Intel's Model 740 cartridge disk system, enabling the disk to provide 10–50% better system throughput for various development functions. The 750 can be integrated into the NDS-II, providing easy upgrade from stand-alone to multi-user environment, while protecting the user's investment in the 750.



ORDER NUMBER:210351-003

FUNCTIONAL DESCRIPTION

Hardware Components

Intel's Winchester disk subsystem consists of a disk drive and power supply enclosed in a freestanding peripheral chassis, plus an intelligent disk controller, interconnecting cables and documentation.

The Winchester disk provides 22 Mbytes of formatted storage at data transfer rate of 6.4 Mbits/second.

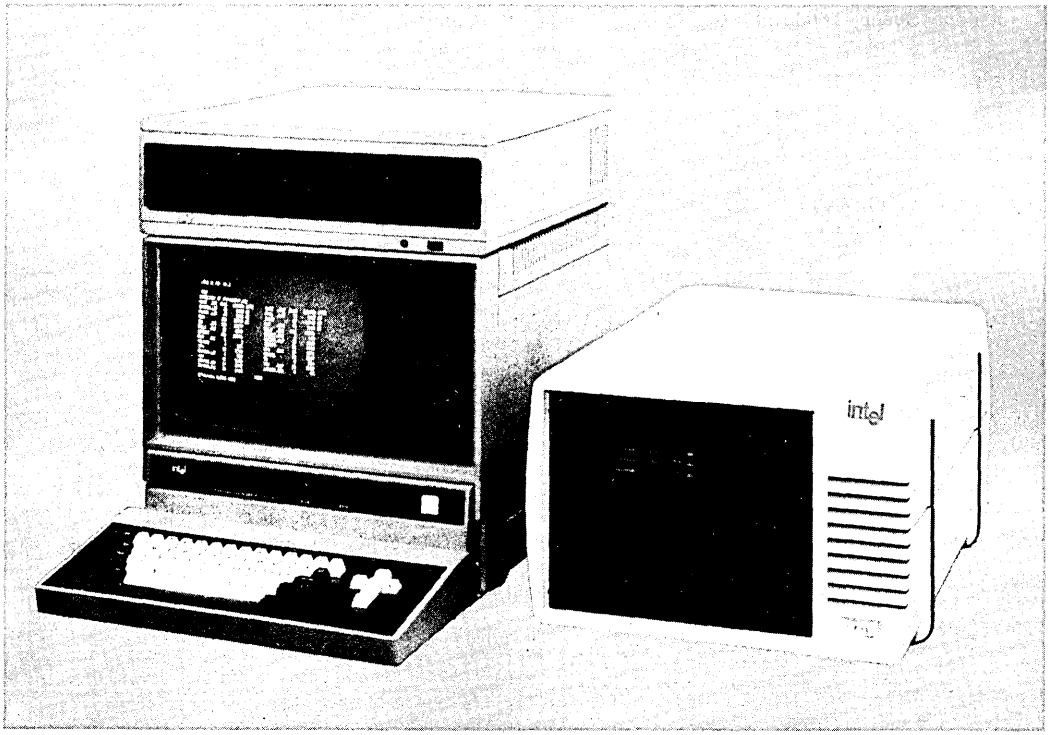
The disk controller provides the interface to Intel's system bus. This single-board controller resides in one slot of an Intellec® system card cage.

Associated Software—Intel Systems Implementation Supervisor (ISIS-II)

The Winchester subsystem is to be used in conjunction with the ISIS-II (W) Operating System. ISIS-II (W) provides total file management capabilities, file editing, library management, run-time support, and utility management.

Equipment Supplied

—Peripheral chassis with Winchester drive and power supply
—Interconnecting cables
—Winchester disk controller (1 board)
—ISIS-II (W) System Diskette
—ISIS-II (W) System User's Guide
—Modified backpanel for Intellec System



AFN-02168C

SPECIFICATIONS

Hardware

Disk Drive

Type—Winchester sealed disk
Tracks per Inch—480
Mechanical Sectors per Track—70
Recording Technique—70 MFM
Tracks per Surface—525
Density—6,670 bits/inch
Bytes per Track—13,440
Recording Surfaces—5

Disk System Capacity

Disk Transfer Rate—6.4 Mbits/sec
Disk System Access Time—
  Track to Track: 10 ms max.
  Full Stroke: 90 ms
  Rotational Speed: 3,600 rpm

Physical Characteristics

Disk Drive in Peripheral Chassis
Width—16.9 in. (42.9 cm)
Height—11.3 in. (28.7 cm)
Depth—24.3 in. (61.7 cm)
Weight—55 lb. (25 kg)

Electrical Characteristics

Chassis
  DC Power Supplies—Internal to Cabinet
  AC Power Requirements
    110 VAC: 60 Hz; 5A (max)
    220 VAC: 50 Hz; 3A (max)
Controller Boards
  5V @ 2.5A (typ), 3.25 (max)

Environmental Characteristics

Media, Drive and Chassis
  Temperature:
    Operating: 15°C to 35°C
    Non-operating: −9°C to 60°C
  Humidity:
    10% to 90% non-condensing
Controller Boards
  Temperature:
    Operating: 0°C to 55°C
    Non-operating: −55°C to 85°C
  Humidity:
    Up to 90% non-condensing

ORDERING INFORMATION

Part Number	Description
iMDX 750A 110V, 60 Hz iMDX 750B 220V, 50 Hz	Series II/III Winchester subsystem. Provides Winchester disk, chassis, power supply, cables, disk controller, software and documentation for Series II/85 and Series III Intellec® systems.
iMDX 750IA 110V, 60 Hz iMDX 750IB 220V, 50 Hz	Series II/80 Winchester subsystem with IPC-85 computer board. Provides Winchester disk, chassis, power supply, cables, disk controller, IPC-85 computer board, software and documentation for Series II/80 Intellec® systems.

intel

APPLICATION NOTE

AP-156

July, 1984

Designing Modules for iPDS™ and iUP Systems

DALE OLLILA
DSHO TECHNICAL PUBLICATIONS

Order Number 230682-001

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:

BXP, CREDIT, i, ICE, I^2ICE, ICS, iDBP, iDIS, iLBX, i_m, iMMX, Insite, INTEL, int$_e$l, Intelevision, Intellec, int$_e$ligent Identifier™, int$_e$llBOS, int$_e$ligent Programming™, Intellink, iOSP, iPDS, iRMS, iSBC, iSBX, iSDM, iSXM, Library Manager, MCS, Megachassis, Micromainframe, MULTIBUS, Multichannel™ Plug-A-Bubble, MULTIMODULE, PROMPT, Ripplemode, RMX/80, RUPI, System 2000, and UPI, and the combination of ICE, iCS, iRMX, iSBC, 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.

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

Intel Corporation
Literature Department
3065 Bowers Avenue
Santa Clara, CA 95051

DESIGNING MODULES FOR iPDS™ AND iUP SYSTEMS

CONTENTS

INTRODUCTION 4

The iPDX BUS 4
iPDX Bus Features 4
Advantages and Limitations of iPDX Bus Implementaton 4
iPDX Bus Functional Description ... 5

IPDX BUS SPECIFICATIONS 8

iPDX Bus Signal Descriptions 8
Power Specifications 10
Electrical (DC) Specifications 11
Timing (AC) Specifications 11
Mechanical Specifications 13

HARDWARE DESIGN CONSIDERATIONS 13

Mechanical Considerations 14
Power Considerations 14

PROGRAMMING CONSIDERATIONS 14

iPPS Software Protocol 14
Switched Voltage Programming 22
User-Written iPDX Bus Drivers 23

INTRODUCTION

The Intel Personal Development System (iPDS™) is a new development tool concept. It provides a subset of the capability of an Intellec® Series II/III development system, in a portable, and less expensive package. One of the features offered by the iPDS system is the expansion capability designed into the product. The basic iPDS system can be expanded to include a parallel processor, a wide range of serial (RS232C interface) and parallel (Centronics interface) devices, numerous MULTIMODULE™ (iSBX™ interface) devices, additional flexible disk drives, and a growing line of plug-in emulator and PROM programming modules.

The plug-in modules for the iPDS system communicate over an interface referred to as the Intel Personal Development Expansion bus (iPDX bus). The iPDX bus is also used in another Intel product, the iUP-200/201 Universal Programmer (iUP). There are some differences in iPDX bus implementation between the iUP and iPDS systems, but the basic interface is the same. Intel PROM programming modules can be used in either system.

THE iPDX BUS

The iPDX bus is a byte-wide, parallel interface between a plug-in module and the iPDS or the iUP system. The iPDX bus allows a variety of plug-in modules to be added to the iPDS system. (The iUP system normally is used with PROM programming modules.) Some of the possible types of plug-in modules are:

- PROM programming modules
- Emulator (EMV) modules for various microprocessor or microcontroller families
- Test instrumentation modules (e.g., logic or signature analyzers)
- Analog interface modules (e.g., analog/digital or digital/analog converters)
- Serial communication modules (e.g., modem or cassette controller modules)
- Parallel communication modules (e.g., direct interface to other CPU buses)
- Program storage modules (e.g., modules storing alternate operating systems, diagnostic programs, or games)

Intel Corporation produces plug-in modules that allow PROM programming and emulation for a variety of Intel chips. The special needs of individual users may not be satisfied by the plug-in modules that are available. This application note presents the specifications and design criteria for user-designed plug-in modules using the iPDX bus. User-designed plug-in modules can expand the usefulness of the iPDS system in the design lab, on the production floor, and in field applications.

iPDX Bus Features

The iPDX bus's capabilities are nearly equal to the capabilities of the iSBX™ bus. In some respects the iPDX bus is more powerful than the iSBX bus, due to the variable and switched supply voltages included on the bus. The features of the iPDX bus are:

- The controlling (iPDS or iUP) system supplies +5VDC and ground to the iPDX bus.
- The controlling (iPDS or iUP) system supplies switched voltages of +5.7VDC, -12VDC, and +8VDC to +27VDC to the plug-in modules. In addition, the iUP system controls a variable switched voltage (+8 to +15 VDC) and the iPDS system controls a +12 VDC switched voltage to the plug-in modules. The switched voltages are turned on and off under program control.
- A number of options are available for controlling iPDX bus transactions. These options include:
  1) Using iPPS software to supervise the uploading and execution of firmware from the plug-in module.
  2) Using a user-written driver program to supervise the uploading and execution of firmware from the plug-in module.
  3) Using a user-written driver program to control all iPDX bus activity.
  4) Using a user-written monitor program to allow control of iPDX bus activity from the system console.
- The plug-in modules that interface with the iPDX bus enable easy and fast changes of entire I/O subsystems.
- A prototyping tool (product code iPDS-PROTO) allows users to quickly design and build custom plug-in modules.
- The resources of a powerful, general-purpose development system (the iPDS system) are available to plug-in modules that use the iPDX bus.

Advantages And Limitations of iPDX Bus Implementation

The system (iUP or iPDS) that the iPDX bus is implemented on offers advantages for and imposes limitations on plug-in module use. The user's design requirements may dictate that the plug-in module be used with only one of the available systems. Plug-in modules that are universal must be designed to avoid the limitations of both systems.

iUP/iPDX BUS ADVANTAGES AND LIMITATIONS

Plug-in modules used with the iUP system are normally restricted to PROM-type programming functions. Table 1 lists the advantages and limitations of the iUP/iPDX Bus.

iPDS™/iPDX BUS ADVANTAGES AND LIMITATIONS

Plug-in modules used with the iPDS system can make use of all the features listed in the iPDX Bus Features section on page 4. The limitations for an iPDS/iPDX bus plug-in module are in the amount of power available from some of the voltage supply lines. Table 2 lists the advantages and limitations of the iPDS/iPDX Bus.

iPDX Bus Functional Description

The iPDX bus is an extension to the CPU bus of the iUP or iPDS system. The iPDX bus is active in the I/O address range 10H - 1FH of the controlling CPU. Figure 1 is a functional block diagram of the iPDX bus as implemented on the iUP system. Figure 2 is a functional block diagram of the iPDX bus as implemented on the iPDS system.

iUP/iPDX BUS IMPLEMENTATION

The iPDX bus is the only I/O interface for the iUP-200/201 Universal Programmer, other than the serial interface of the iUP system. The iUP system normally performs one function, the programming of PROM-type devices. Intel PROM-type devices include EPROMs, E^2PROMs, and the EPROM portion

Table 1. iUP/iPDX Bus Implementations

Advantages	Limitations
The iUP system provides ample power for programming any type of PROM device.	Direct control of CPU operation is only possible using uploaded plug-in module firmware.
Two variable supply voltages are available for plug-in module use.	The V_{cc} line supplies a maximum of 1.0 A to the plug-in module.
The I/O space of the iUP system is mostly unused, so operation in unused I/O space is possible.	

Table 2. iPDS™/iPDX Bus Implementations

Advantages	Limitations
The resources of the iPDS system (RAM, console, mass storage, etc.) are available to the plug-in.	Only one of the variable supply voltages (+VHSW) is available on the iPDS bus. The other variable line (+VLSW) has a fixed output of +12VDC.
The user has the option of using iPPS software or user-written programs to control the plug-in module.	Power supplied to the iPDX bus is not adequate for gang programming modules.
Any PROM programming module that works with the iPDS system and iPPS software also works with the iUP system. The V_{cc} supply line can handle up to a 2.5 A draw. This draw is adequate for most user applications.	

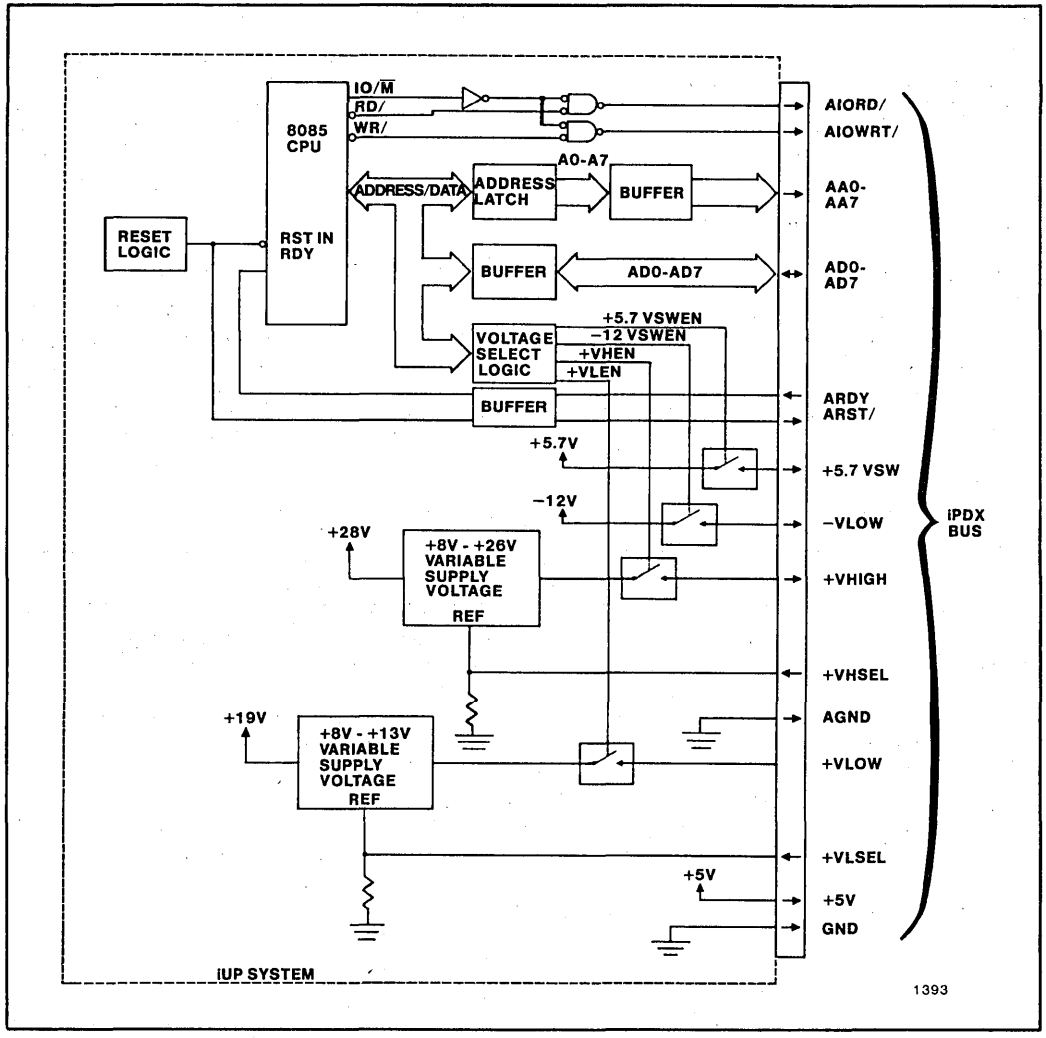


Figure 1. iUP/iPDX Bus, Functional Block Diagram

of various microcontrollers. The iUP system can program non-Intel PROM-type devices, but in most cases a personality plug-in module for the non-Intel device must be designed by the user. Note, however,that the Intel iUP-Fast 27/K PROM programming module (with firmware change) can program any 28-pin JEDEC device.

The iPDX bus implementation on the iUP system is optimized for maximum programming power capabilities. Each of the switched voltage supply lines from the iUP system provides at least twice the power of the corresponding line from an iPDS system. Refer to the Power Specifications (page 10) section for specific power capabilities.

The switched voltage lines are turned on and off under program control by the controlling CPU. The switched voltages are:

- +5.7VSW
- +VHIGH
- +VLOW
- −VLOW

Two of the switched voltages (+VHIGH and +VLOW) are variable. The +VLOW line provides +8V to +15V at 700 ma as determined by a precision resistance on the +VLSEL line. The +VHIGH line provides +8V to +27V at 300 ma as determined by a precision resistance on the +VHSEL line.

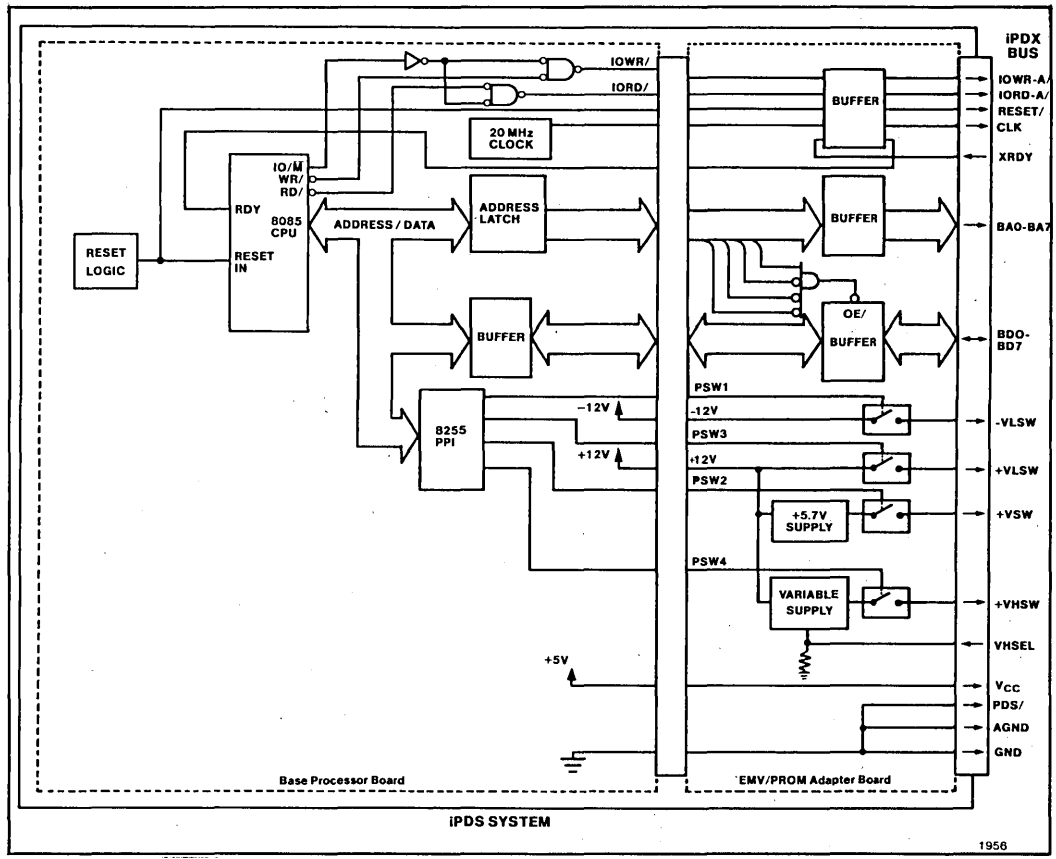


Figure 2. iPDS™/iPDX Bus, Functional Block Diagram

Refer to the Power Considerations (page 14) section for details on the control of the variable supply voltages.

The iUP/iPDX bus implementation provides not only program control of the switched voltage lines. It also allows monitoring of the on/off condition of these lines. The I/O ports used to control and monitor the switched voltages are discussed in the Switched Voltage Programming section (page 22).

Buffered data (AD0 - AD7) is placed on the iPDX bus each time address line 4 (A4) is '1' during I/O accesses by the controlling CPU. This ensures that the data lines will be active for I/O addresses of 10H to 1FH. It also places data on the bus for addresses of 3XH, 5XH, 7XH, 9XH, BXH, DXH, and FXH. The iPPS software only uses I/O addresses of 1XH when initially contacting the plug-in module, so there is no problem with this I/O addressing.

The address, read, write, reset, and ready lines feed directly from the iUP system to the plug-in module on the iPDX bus. Figure 1 is a functional block diagram of the iUP system that shows the iPDX signals, their direction of flow, and the controlling circuitry in the iUP system. Refer to other sections of this application note for specific details on iUP/iPDX bus implementation.

iPDS™/iPDX BUS IMPLEMENTATION

The iPDS system implementation of the iPDX bus is a powerful, general-purpose interface to plug-in modules. The iPDS interface has less power handling capabilities than the iUP interface, but it has additional system resources.

The iPDS/iPDX bus interface uses a separate board in the iPDS system. The iPDS-140 option for the iPDS system is an interface between the iPDX bus

and the base processor board of the iPDS system. The iPDS-140 option buffers all address, data, and control signals that go to the iPDX bus. The top address nibble is decoded on the iPDS-140 option to enable data transfers during reads or writes to I/O addresses 10H to 1FH.

The switched voltages for the iPDX bus are developed on the iPDS-140 option. The iPDS-140 option uses +12VDC and −12VDC from the iPDS system to generate the switched voltages. Refer to the Power Specifications and Power Considerations sections (pages 10 and 14) for details on the power available for the iPDX bus.

The switched voltages are under program control of the CPU in the iPDS system. These control signals are sent through an 8255 PPI chip to the iPDS-140 option. Refer to the Programming Switched Voltages section (page 22) for details on switched voltage control.

The V_{cc} (+5VDC) and ground lines from the base processor board are fed directly to the iPDX bus. The PDS/ and AGND lines of the iPDX bus are connected to the ground line within the iPDS-140 option. The PDS/ line is used by PROM programming plug-in modules to indicate the controlling system to iPPS software. All PROM programming plug-in modules feed the PDS/ line (J1-20) back so iPPS software can read its '1' or '0' status. Refer to the iPPS Software Protocol section (page 14) for details on the module status byte.

The address, read, write, reset, clock, and ready lines are buffered on the iPDS-140 option, but they are not modified by the iPDS system. Figure 2 is a functional block diagram of the iPDS system that shows the iPDX signals, their direction of flow, and the controlling circuitry in the iPDS system. Refer to other sections of this application note for specific details on iPDS/iPDX bus implementation.

iPDX BUS SPECIFICATIONS

The specifications for the iPDX bus are divided into four catagories:

- Signal listings and descriptions.
- Detailed power (DC) specifications.
- Detailed timing (AC) specifications.
- Outline drawings and detailed mechanical specifications.

iPDX Bus Signal Descriptions

Table 3 presents the pinout of the iPDX bus and gives the associated signal names for both the iPDS and iUP systems.

Table 4 lists the signal names (iPDS and iUP systems) of the iPDX bus and gives a short description of each group of signals.

Table 3. iPDX Bus Pinout

Pin	iPDS™ Mnemonic	iUP Mnemonic	Input/ Output	Pin	iPDS™ Mnemonic	iUP Mnemonic	Input Output
1	GND	GND	O	22	GND	GND	O
2	GND	GND	O	23	Reserved	Reserved	N/A
3	BA0	AA0	O	24	BD0	AD0	I/O
4	BA1	AA1	O	25	BD1	AD1	I/O
5	BA2	AA2	O	26	BD2	AD2	I/O
6	BA3	AA3	O	27	BD3	AD3	I/O
7	BA4	AA4	O	28	BD4	AD4	I/O
8	BA5	AA5	O	29	BD5	AD5	I/O
9	BA6	AA6	O	30	BD6	AD6	I/O
10	BA7	AA7	O	31	BD7	AD7	I/O
11	V_{cc}	+5V	O	32	Reserved	Reserved	N/A
12	V_{cc}	+5V	O	33	+VHSW	+VHIGH	O
13	+VSW	+5.7VSW	O	34	+VLSW	+VLOW	O
14	+VSW	+5.7VSW	O	35	Reserved	Reserved	N/A
15	CLK	Not Used	O	36	−VLSW	−VLOW	O
16	IOWR-A/	AIOWRT/	O	37	AGND	AGND	O
17	IORD-A/	AIORD/	O	38	+VHSEL	+VHSEL	I
18	RESET/	ARST/	O	39	Not Used	+VLSEL	I
19	XRDY	ARDY	I	40	GND	GND	O
20	PDS/	PDS/	O(iPDS)	41	GND	GND	O
21	GND	GND	O				

Table 4. iPDX Bus Signal Descriptions

Signal Name(s)		Description
iPDS™	iUP	
GND	GND	Reference potential for all signals and supply voltages.
AGND	AGND	Analog ground. Reference potential for the programmable high voltage signal (+VHSW or +VHIGH).
BA0-BA7	AA0-AA7	Address lines from the iPDS system or the iUP system that define the I/O register to be accessed.
BD0-BD7	AD0-AD7	Bi-directional, parallel data lines between the plug-in module, and the iPDS or the iUP system.
V_{cc}	+5V	Supply voltage for plug-in module circuitry.
CLK	Not Used	Clock signal (20 MHz) from the iPDS system.
IOWR-A/	AIOWRT/	I/O write signal from the iPDS or the iUP system. An active low indicates that output data from the iPDS or the iUP system is on the data lines. Data is sampled on the trailing edge of this signal.
IORD-A/	AIORD/	I/O read signal from the iPDS or the iUP system. An active low indicates that input data from the plug-in module should be placed on the data lines. Data is sampled on the trailing edge of this signal.
RESET/	ARST/	Reset signal from the iPDS or the iUP system.
XRDY	ARDY	Asynchronous ready signal from the plug-in module. An active high indicates that the plug-in module has accepted write data from, or presented valid read data to, the iPDS or the iUP system. A low level causes the iPDS or the iUP system to enter a wait state after either the IORD-A/ (AIORD/) or IOWR-A/ (AIOWRT/) line is activated.
PDS/	Not connected	A ground from the iPDS system. This signal is sampled by iPPS software and indicates that a PROM programming module is installed in an iPDS system.
+VSW	+5.7VSW	Switched +5.7VDC that can be turned on or off by the iPDS or the iUP system under program control.
+VHSW	+VHIGH	Switched +8VDC to +26VDC that can be turned on or off by the iPDS or the iUP system under program control. The actual voltage is determined by the +VHSEL signal from the plug-in module.
+VLSW	+VLOW	Switched +8VDC to +13VDC that can be turned on or off by the iPDS or the iUP system under program control. For the iUP system the actual voltage is determined by the +VLSEL signal from the plug-in module. The iPDS system outputs only a fixed voltage of +12VDC on the +VLSW line.
-VLSW	-VLOW	Switched -12VDC that can be turned on or off by the iPDS or the iUP system under program control.
+VHSEL	+VHSEL	High plus programming voltage select (iPDS and iUP systems). A precision resistance in the plug-in module determines the voltage on the +VHSW (+VHIGH) line.
Not Used	+VLSEL	Low plus programming voltage select (iUP system only). A precision resistance in the plug-in module determines the voltage on the +VLOW line.

Power Specifications

The +5VDC line is always active on the iPDX bus. This line normally powers plug-in module circuitry. Switched voltages are also available to power plug-in module circuitry. The user must first set up appropriate driver routines and programming voltages before the switched voltage lines become active.

Table 5 lists the supply signals available at the iPDX bus and the specifications for each signal.

Figure 3 shows the power available on the iPDS +VHSW signal line for the programmable voltages. The other power supply signals give rated power over their full range.

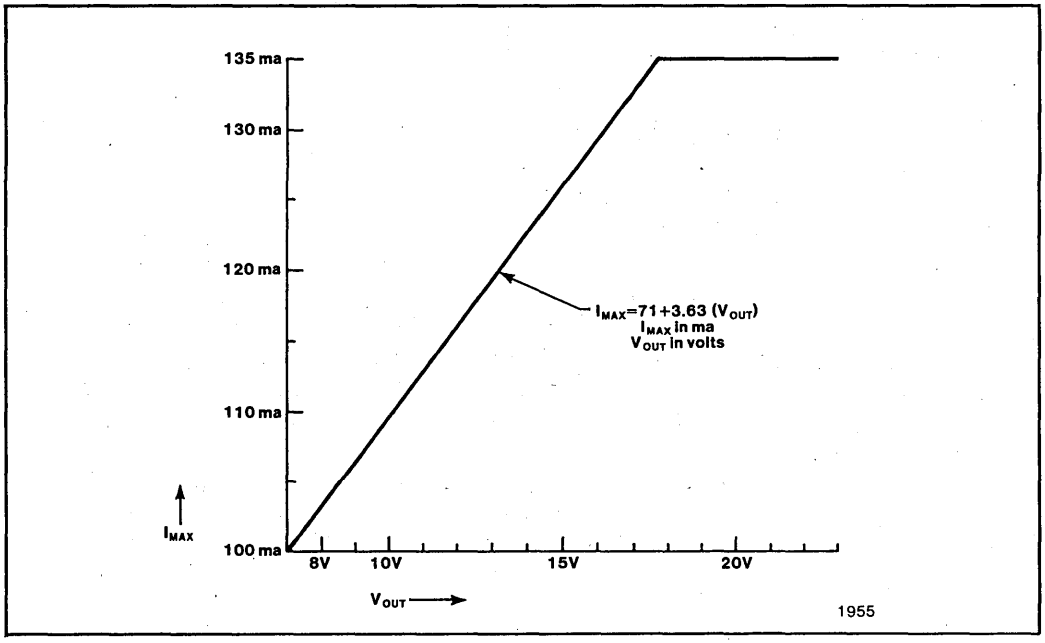


Figure 3. Power Available (iPDS™ +VHSW Signal)

Table 5. iPDX Bus Power Specifications

Signal Name		Supply Voltage and Tolerance		Maximum Current		Notes
iPDS™	iUP	iPDS™	iUP	iPDS™	iUP	
V_{cc}	+5V	+5VDC ±2.5%		2.5 amps	1.0 amp	
+VSW	+5.7VSW	+5.7VDC ±50mv		250 ma	1.5 amps	1
+VHSW	+VHIGH	+8VDC to +27VDC ±2%		135 ma	300 ma	1, 2
+VLSW	+VLOW	+12VDC ±1.0v	+8VDC to +15VDC ±2%	200 ma	700 ma	1, 3
-VLSW	-VLOW	-12VDC ±0.5v		50 ma	100 ma	1

NOTES: 1. This voltage is switched and is under program control of the iPDS or the iUP system.
2. The voltage is controlled by the +VHSEL signal. Figure 3 shows the derating required for each selected voltage of +VHSW.
3. The voltage is controlled by the +VLSEL signal (iUP system only).

Electrical (DC) Specifications

The signal names for the iPDX bus indicate whether or not the signals are active high or active low. If the name ends with a slash (/), the signal is active low. If the name has no slash following it, the signal is active high. Table 6 shows the electrical specifications for the iPDX bus.

The electrical characteristics for the iPDX bus signals are shown in Table 7. The voltage and current specifications refer to the TTL high or TTL low state of the iPDX bus signal. The signal type (input or output) is the signal direction when viewed from the iPDS or the iUP system side of the iPDX bus. Positive currents are defined as currents entering the interface.

Timing (AC) Specifications

Figure 4 shows the timing specifications for the iPDX bus. Table 8 lists definitions of the timing parameters used for the iPDX bus. Refer to the *MCS®-80/85 Family User's Manual* or the *8085A-2* data sheet for specific details on the timing specifications for the iPDX bus.

The +VHSEL/+VLSEL signals, the data bus signals, and the ready (XRDY or ARDY) signal originate in the plug-in module. The voltage select and data bus signals have staightforward timing requirements, but the timing requirements for the ready signal need explanation.

CAUTION

Whenever the ready signal (XRDY or ARDY) goes low, the CPU generates wait-states until the ready signal returns high. The ready signal should not be driven low for more than a few bus cycles unless complete suspension of all CPU bus activity is allowable in the user's application.

The ready signal is normally high for all read/write transfers over the iPDX bus. The ready signal can be driven low to insert one or more wait-states in the CPU bus cycle, in cases where the plug-in module uses slow memory devices or slow peripheral devices.

Table 6. iPDX Bus Electrical Specifications

Active State	Logical State	Electrical Signal Level	At Receiver	At Driver
LOW	0	H=TTL High State	5.25V ≥ H ≥ 2.0V	5.25V ≥ H ≥ 2.4V
	1	L=TTL Low State	0.8V ≥ L ≥ -0.5V	0.5V ≥ L ≥ 0.0V
HIGH	0	L=TTL Low State	0.8V ≥ L ≥ -0.5V	0.5V ≥ L ≥ 0.0V
	1	H=TTL High State	5.25V ≥ H ≥ 2.0V	5.25V ≥ H ≥ 2.4V

Table 7. Electrical Characteristics of iPDX Bus Signals

Signal Type	I_{OL} Max	I_{IL} Max	I_{OH} Max	I_{IH} Max	V_{OL} Max	V_{IL} Max	V_{OH} Min	V_{IH} Min
All Outputs	24 ma		-5 ma		0.5V		2.4V	
Inputs (except RDY signal)		-12.8 ma		50 μa		0.8V		2.0V
ARDY (input)		-4 ma		50 μa		0.8V		2.0V

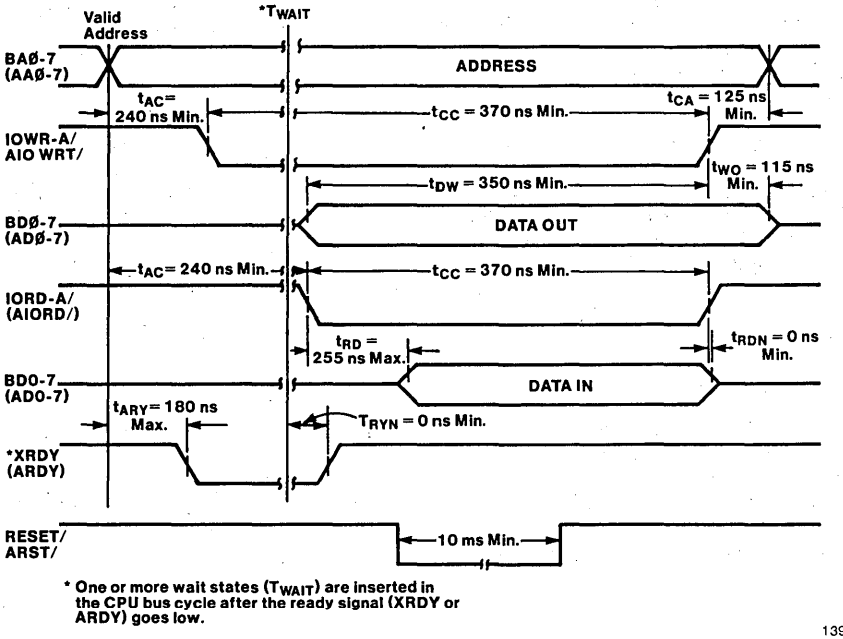


Figure 4. iPDX Bus Read/Write Timing

Table 8. iPDX AC Timing Definitions

Symbol	Description
t_{AC}	The time between valid address (A0 - A7) and the leading edge of the control signal.
t_{ARY}	The time between valid address (A0 - A7) and the trailing edge of the ready signal.
t_{CA}	The time between the trailing edge of the control signal and the end of valid address.
t_{CC}	The width of the control signal.
t_{DW}	The time between the start of valid data (D0 - D7) and the trailing edge of the write control signal.
t_{RD}	The time between the leading edge of the read control signal and the start of valid data (D0 - D7).
t_{RDH}	The time between the trailing edge of the read control signal and the end of valid data (D0 - D7).
t_{RYH}	The time between the end of T_{WAIT} and the leading edge of the ready signal.
t_{WD}	The time between the trailing edge of the write control signal and the end of valid data (D0 - D7).

Mechanical Specifications

The mechanical specifications define the connector requirements and the outline and mounting dimensions for plug-in modules using the iPDX bus. Figure 5 is an outline drawing of a plug-in module for the iPDX bus. All plug-in modules for the iPDX bus must comply with the dimensions specified in Figure 5.

HARDWARE DESIGN CONSIDERATIONS

Plug-in modules designed around the iPDX bus must follow certain design rules. These design rules are:

- The first four inches (measured from the connector end) of the plug-in module must meet the mechanical and outline specifications shown in Figure 5.

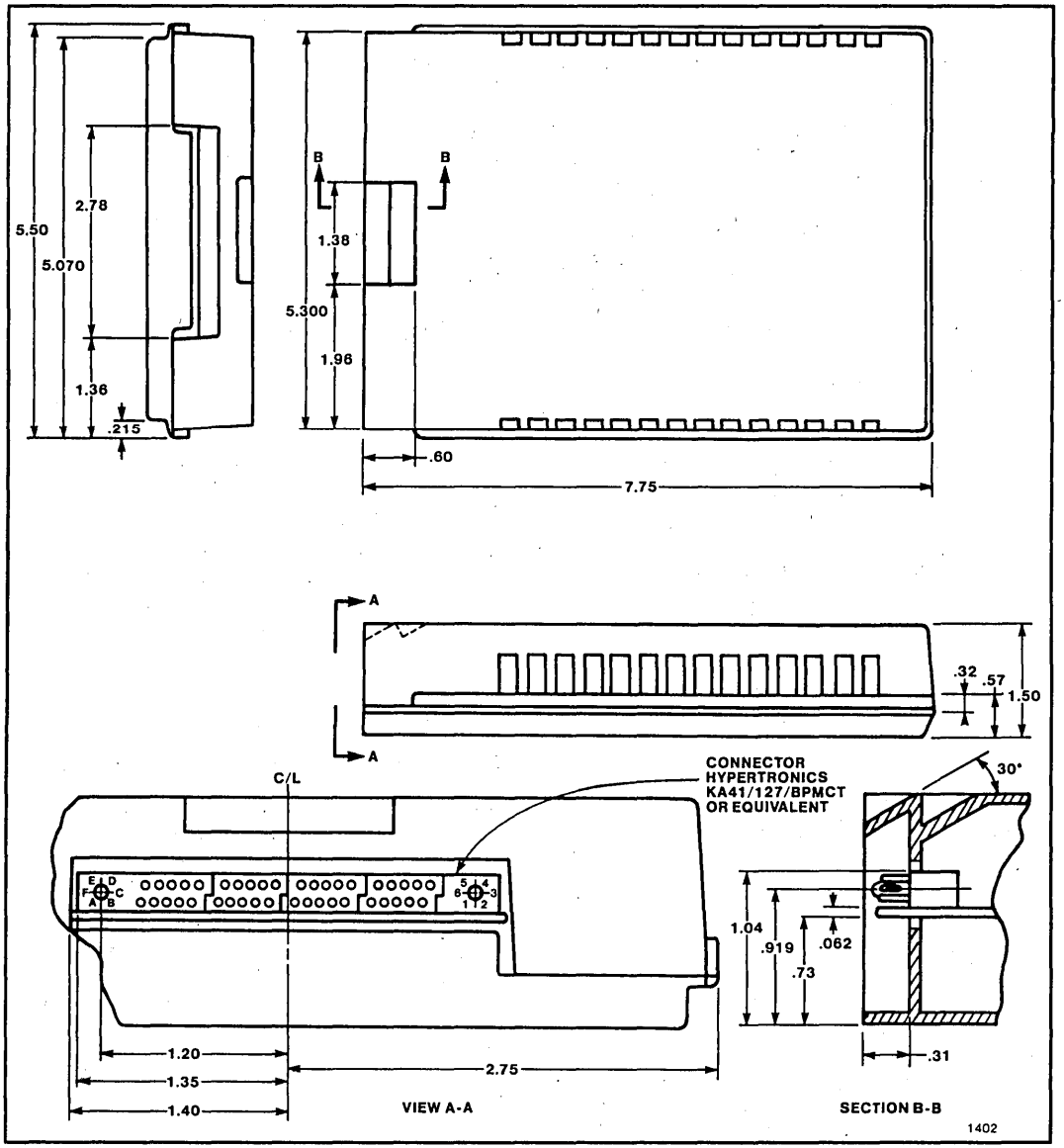


Figure 5. Plug-In Module Mechanical Specifications

- The maximum V_{cc} (+5VDC) current available is 2.5 amps for iPDS plug-in modules or 1.0 amps for iUP plug-in modules.
- Switched voltages of +5.7VDC, +8VDC to +27VDC, +12VDC, and -12VDC are available to circuitry on a plug-in module under program control. Table 5 lists power specifications for the iPDX bus.
- If a programmed voltage (positive only) is required by the plug-in module, an appropriate precision resistor must be installed in the plug-in module.
- All signals (except +VHSEL and +VLSEL) returned by the plug-in module must be TTL levels.
- Provisions must be made to sample the PDS/ signal on PROM programming plug-in modules that use iPPS software while connected to an iPDS system. (The PDS/ signal is low when the module is connected to the iPDS system and floating when connected to the iUP system. Firmware can use the signal 1) to specify whether a power supply status port is available, 2) to specify whether E3H (iPDS) or 03H (iUP) is the correct port for turning on power supplies, and 3) to compensate for differences in timing between the two systems.)
- Direct memory access (DMA) transactions are not supported on the iPDX bus.

Mechanical Considerations

Plug-in modules for the iPDX bus must have an enclosure that meets the mechanical specifications shown in Figure 5 for the first four inches (measured from the connector end) of the module. Intel has developed a prototyping kit (product code iPDS-PROTO) to simplify the mechanical and hardware portions of the design. This prototyping kit consists of the plug-in module enclosure, a prototyping board, iPDX bus connector, a hardware kit, isolation capacitors, and wire-wrap pins. The iPDS-PROTO kit can accept up to 30 ICs and associated discrete components in the available board space. If a plug-in module designed around the iPDX bus goes to a production phase, use of the module tooling can be licensed through Intel.

Power Considerations

The maximum power dissipation for an iPDS plug-in module is 20.5 watts with a maximum draw of 12.5 watts from the V_{cc} line. The maximum power dissipation for an iUP plug-in module is 32.5 watts with a maximum draw of 5.13 watts from the V_{cc} line and 8.625 watts from the +5.7VSW line. A maximum of 7.5 watts can be dissipated within a plastic plugin module (more power can be dissipated at the PROM socket).

V_{cc} (+5VDC) is the only voltage present at all times on the iPDX bus. If the plug-in module circuitry requires other voltage levels for operation, the switched voltages must be turned on first by software. The Programming Considerations section shows the iPDX bus set-up requirements for turning on/off each of the switched voltage signals.

The variable switched voltages (+VHSW on an iPDS plug-in module, and +VHIGH and +VLOW on an iUP plug-in module) use one or more precision resistors on the plug-in module to determine their line voltage. The precision resistor on the plug-in module must be connected between the AGND line and the +VHSEL line of the iPDX bus. Plug-in modules for an iUP system can also program the +VLOW line by connecting a precision resistor between the AGND line and the +VLSEL line of the iPDX bus. Figure 6 shows a chart and two equations that indicate the precision resistor values corresponding to programmable voltages. Figure 7 shows three kinds of circuits that allow the plug-in module to select more than one programming voltage level.

PROGRAMMING CONSIDERATIONS

PROM programming modules are normally controlled by iPPS software residing in either the iPDS or the iUP system. User-designed plug-in modules (other than programming plug-in modules) are controlled by user-supplied driver programs. The iPPS Software Protocol section explains the iPPS - iPDX bus interface. The Switched Voltage Programming section gives programming requirements for accessing switched voltages in the iPDS/iPDX bus interface. The User-Written iPDX Bus Drivers section presents the programming requirements for user-supplied driver programs.

iPPS Software Protocol

PROM programming plug-in modules that run under control of iPPS software must contain firmware. The firmware in the PROM programming module is a program that has routines for programming the device(s) that the plug-in module is designed to program. This firmware is uploaded into RAM in the controlling (iPDS or iUP) system the first time the TYPE command in the iPPS command language is executed. After the module firmware is uploaded, the iPDS or the iUP system controls the programming operation. The iPPS software communicates with the plug-in module over 7 of the 16 I/O ports allocated for iPDX bus communication. Table 9 lists the I/O port assignments recognized by iPPS software.

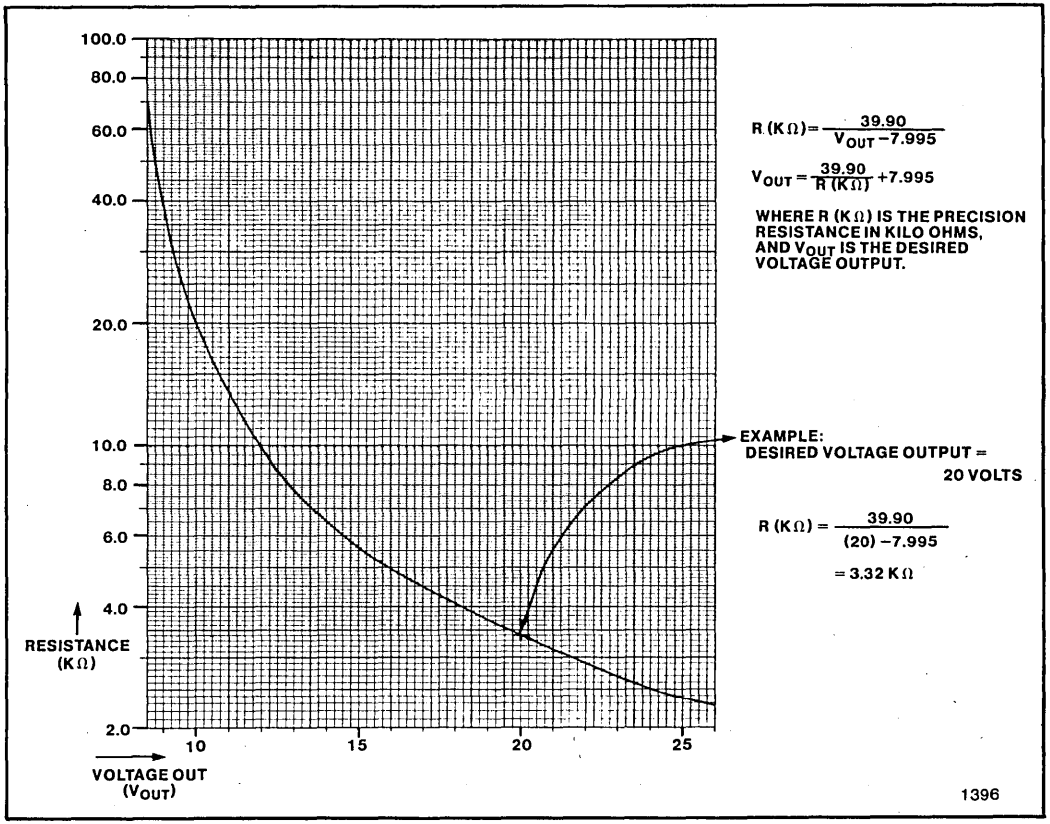


Figure 6. Programmable Voltage Resistor Values

The control words, corresponding to an I/O write to port addresses 10H, 11H, and 12H, control various functions on the plug-in module. These functions may include voltage select and routing for the target PROM socket, the programming pulse, or chip selects, and set/clear the upload flag. The bit definitions for the control words are shown in Figure 8.

The status word, corresponding to an I/O read of port address 10H, contains information about the current state of monitored functions on the plug-in module. The bit definitions for the status word are shown in Figure 9.

The plug-in module firmware is read when the iPPS TYPE command is first executed. The iPPS software uploads plug-in module firmware by writing the plug-in module PROM location to I/O ports 13H (A0-A7) and 14H (A8-A15), respectively, and then reading the data at I/O port 11H. The plug-in module firmware uploads to absolute address 7020H in the iPDS or iUP system. After the plug-in module firmware is uploaded to the iPDS or the iUP system, the upload flag (bit 1 of control word 0) is set by the controlling system. Setting the upload flag causes bit 1 of the status word to indicate that additional firmware uploads are not required.

PLUG-IN MODULE FIRMWARE

The firmware (for plug-in modules running under control of iPPS software) controls all plug-in module operations, except the firmware upload operation itself. This firmware must be written in 8085 code and formatted as shown in Table 10.

The first two bytes of plug-in module firmware must contain the total number of bytes to be uploaded (including the two length bytes and the two checksum bytes). The third byte must contain the number of different devices the plug-in module can read or program.

The plug-in module firmware is divided into segments and a segment is required for each PROM type that the module can program. Each segment contains a descriptor (first 14 bytes) and a code section.

Descriptor Section

The first two descriptor bytes contain the address of the next segment of firmware. The last segment of the firmware must contain the address of the first segment. If there is only one segment, the segment must reference itself.

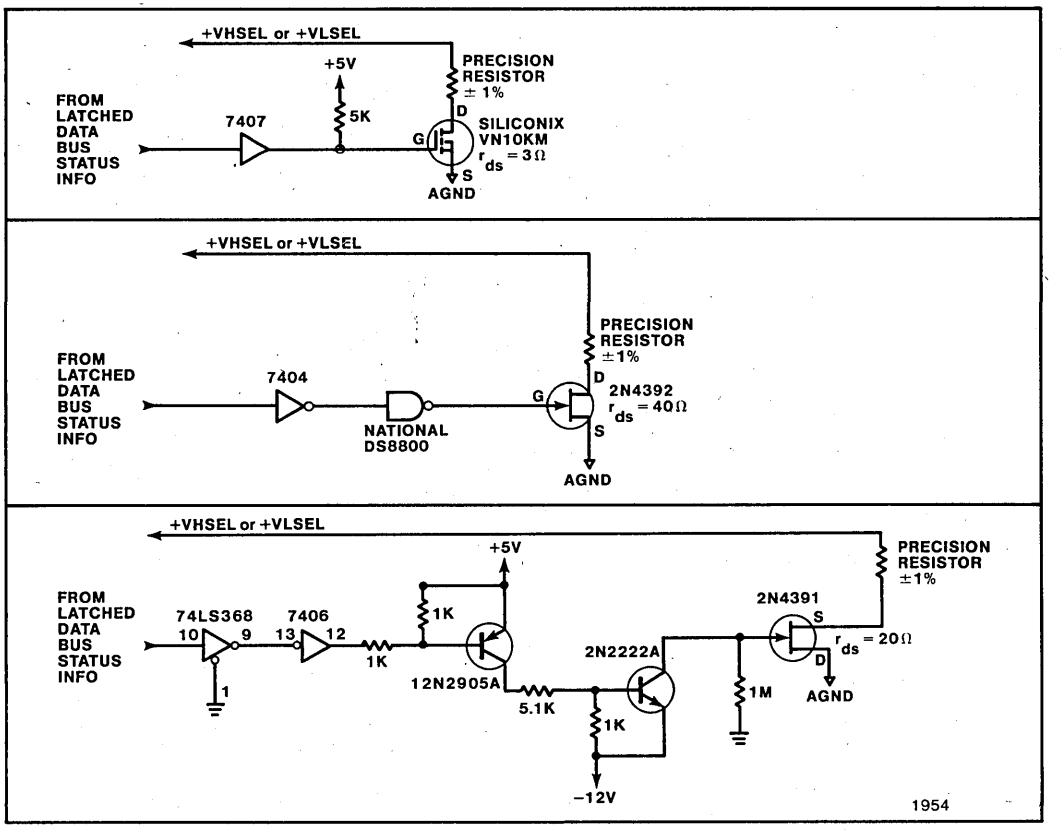


Figure 7. Three Precision Resistor Switching Circuits

Table 9. I/O Port Assignments Used by iPPS Software

I/O Port Address	I/O Write Active	I/O Read Active
10H	Write control word 0	Read module status
11H	Write control word 1	Read personality PROM data
12H	Write control word 2	Available
13H	Write address (A0-A7)	Available
14H	Write address (A8-A15)	Available
15H	Write address (A16-A19)	Available
16H	Write data (D0-D7)	Read device data

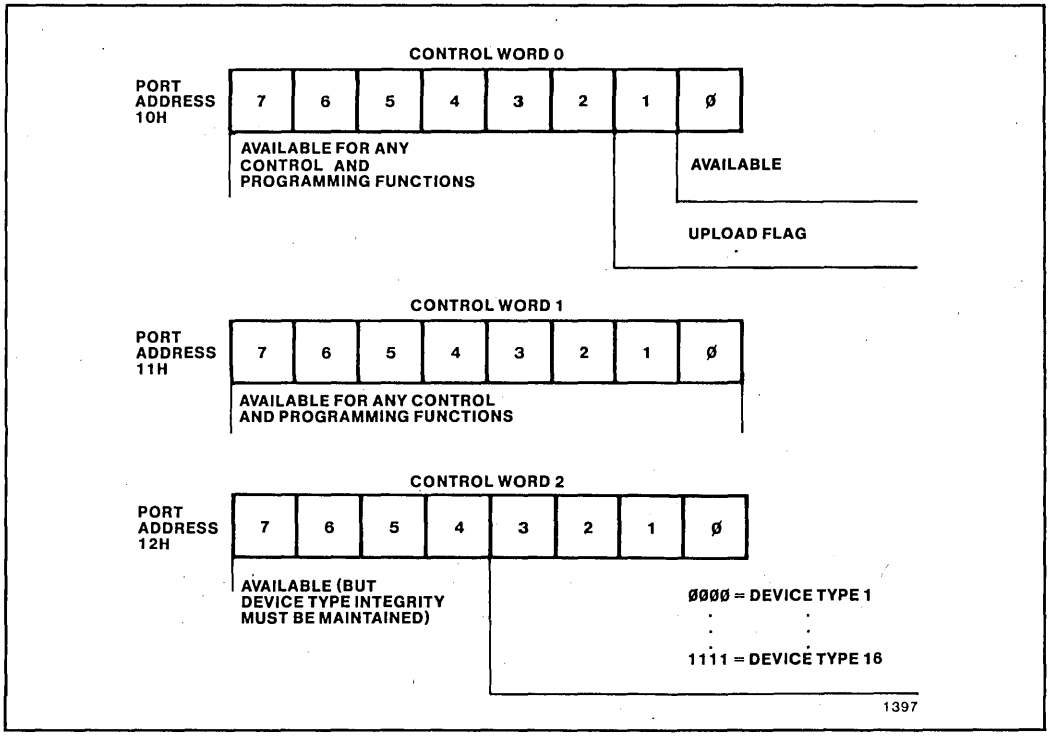


Figure 8. iPPS Control Word Bit Definitions

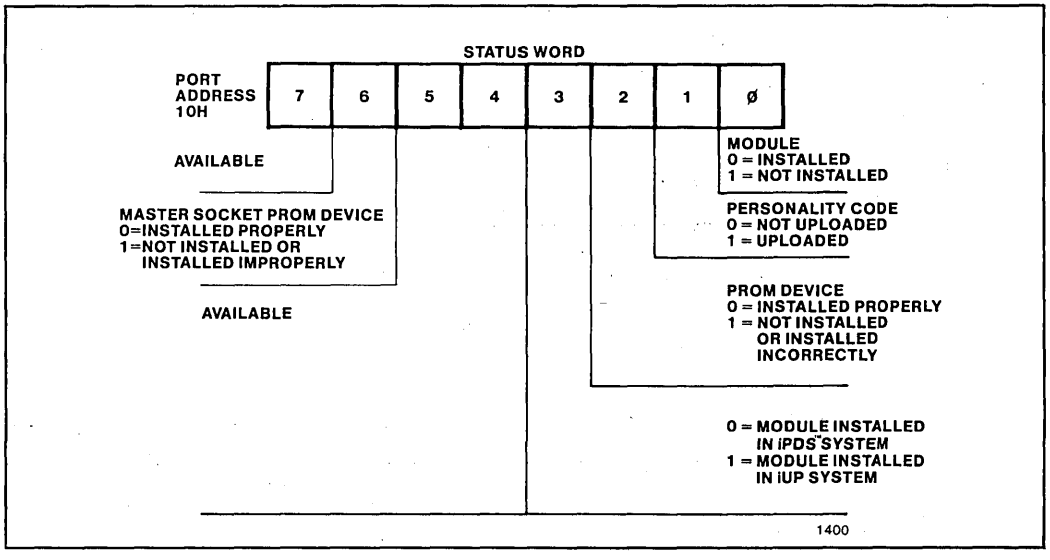


Figure 9. iPPS Status Word Bit Definitions

Table 10. Plug-In Module Firmware Format

Personality Prom Address			Contents
		0	8 LSBS of the length of the personality PROM.
		1	8 MSBS of the length of the personality PROM.
		2	Number of types the module can program.
		3	8 LSBS of the address of the next segment in the table (U).
	D	4	8 MSBS of the address of the next segment in the table (U).
	E	5	1st ASCII character of PROM type.
	S	6	2nd ASCII character of PROM type.
	C	7	3rd ASCII character of PROM type.
	R	8	4th ASCII character of PROM type.
	I	9	5th ASCII character of PROM type.
	P	10	6th ASCII character of PROM type.
	T	11	7th ASCII character of PROM type.
	O	12	8th ASCII character of PROM type.
	R	13	8 LSBS of PROM address range.
		14	8 MOBS of PROM address range.
S		15	8 MSBS of PROM address range.
		16	Bits 0-5 indicate PROM word length. Bit 6 indicates the blank state of the PROM.
E			Bit 7 is not used.
		17	Jump to blankcheck routine (V).
G		18	8 LSB of address of blankcheck routine.
		19	8 MSB of address of blankcheck routine.
M		20	Jump to program routine (W).
		21	8 LSB of address of program routine.
E		22	8 MSB of address of program routine.
		23	Jump to overlay check routine (X).
N		24	8 LSB of address of overlay check routine.
	C	25	8 MSB of address of overlay check routine.
T		26	Jump to reverse socket routine (Y).
	O	27	8 LSB of address of reverse socket routine.
		28	8 MSB of address of reverse socket routine.
	D	29	Jump to read routine (Z).
		30	8 LSB of address of read routine.
	E	31	8 MSB of address of read routine.
		V	Start blankcheck code.
		V+N	"RETURN"
		W	Start code for program routine.
		W+N	"RETURN"
		X	Start code for overlay check.
		X+N	"RETURN"
		Y	Start code for reverse socket routine.
		Y+N	"RETURN"
		Z	Start code for read routine.
		Z+N	"RETURN"
Next segment (U)			
Next two locations after last byte of last segment			Checksum (LSB) Checksum (MSB)

230682-001

The next eight descriptor bytes contain the ASCII code for the device being programmed. Spaces (ASCII code 20H) must be used to fill any unused bytes of this ASCII code.

The remaining four descriptor bytes contain specific PROM device information, with the first three bytes holding the available PROM address range and the final byte holding PROM data information. Bits 0-5 of the PROM data information byte contain the word length (binary equivalent in bits) of the selected PROM. Bit 6 of the PROM data information byte indicates the unprogrammed state of each PROM bit (i.e., a 0 in the bit 6 location means a device bit is unprogrammed in the high state and programmed in the low state). Bit 7 of the PROM data information byte is not used.

Code and Checksum Sections

The code section is subdivided into a jump op code section followed by blankcheck, program, overlay check, reverse socket detect, and read routines.

The jump op code section contains the jump op codes and addresses of each programming routine for the device covered in this segment. The programming routines referenced in this section include read, blankcheck, program, overlay check, reverse socket detect, and read. The referenced routines may actually reside in other segments.

The blankcheck, program, overlay check, reverse socket detect , and read programming routines must be in 8085 code. These routines are hardware specific instructions for checking and programming the device. The following subsections describe relevant details of these routines and provide other information needed to develop module firmware.

The final two bytes of firmware following the last segment contain the checksum for the plug-in module firmware chip. The checksum is the 2's complement of the sum of the previous bytes in the plug-in module firmware chip.

Memory Variable and Stack Locations — Memory locations 6000H to 60FFH are reserved for variables and stack. Please note that this leaves space for a very small stack. The following is a list of variables that the user needs to know to interface to iPPS software.

6000H	Lowest address for 80 bytes of input buffer.
6050H	Lowest address for 80 bytes of output buffer; space is also used for variables when PROMs greater than 32K bytes are edited.
601AH	Used to indicate on-line (00H) or off-line (01H) operation.
60A2H	Used to pass the current status of the iUP programmer to the iPPS software.
60B4H-60B5H	Both 60B4H and 60B5H are general purpose locations for passing information. See information in this section on creating firmware for displaying messages on the host.
60B6H	Used to indicate when powering down has finished, i.e., when an operation has been completed. The module firmware should set this location to 01H when power is turned on. This location is reset to 00H when the power is shut off. This information is needed by the iPDS system, since the iPDS system does not have a status port (such as 02H in the iUP programmer) to indicate whether power is on or off.
60B7H	For passing an address between module and iPPS software: contains LSB of address.
60B8H	For passing an address between module and iPPS software: contains MOB of address.
60B9H	For passing an address between module and iPPS software: contains HOB of address.
60BAH	Contains data to be programmed from the iUP programmer to PROM.
60BBH	Contains data read from PROM to iUP programmer.
60CCH	Indicates operation in process. Used in off-line keyboard interrupts. See keyboard interrupt routine below.
60CFH	Used for the lock function. The iPPS software sets this location to 00H before calling the reverse socket check. The module firmware sets this location to FFH if a lock function is available or leaves it at 00H to indicate that no function is available. (This ensures backwards compatibility with older modules.) The iPPS software then sets this location to 01 before calling the programming routine. This value indicates to the module that lock (rather than programming) is requested. (If programming is requested, the value is 00H.)
60D0H	Used in the lock function. The module firmware uses this location to indicate which parameter is being passed. On modules that just lock (like 8751AH), the lock sequence will never go above 1.

On authenticated PROMs, the sequence numbers may be greater than 1. This allows the module, iPPS software, or user to edit the parameters. The parameters should be stored in a buffer and this location is used to index the buffer. If the user responds NO to the EXECUTE query, module firmware should reset this location to the beginning (0). The buffer values (instead of the PROM's actual values) are then sent back. These locations are programmed only when the user responds YES to the EXECUTE query. Module firmware should be set to 0 when finished.

60D2H — Indicates a PROM that is greater than 32K bytes has been edited. (00H = NO; 01H = YES).

60D3H — Indicates whether the module should be using the programming socket. There is a bug in the initialization of this flag, so until iPPS-PDS software and the iUP programmer firmware are upgraded, the module firmware needs to set this location as follows:

(1) For PROMs less than 32K bytes, set to 00.

(2) For all devices when on-line, set to 00.

This covers the two conditions in which the master socket will never be accessed.

60FFH — Top of the stack.

Parameters for Major Subroutines — Unless otherwise noted, the module returns results using the following codes:

00H means "pass."
12H means "power supply failure."
07H means "abort."

Information on Code Section Routines — The following paragraphs provide information on routines included in the code section of the PROM programming firmware. Note that the meaning of "iUP programmer" in these paragraphs depends on the system being considered. "iUP programmer" can mean either iUP-200A/201A firmware or iPPS-PDS software.

Blank Check Routine — The iUP programmer passes no parameters to the module. The module firmware checks the entire PROM and passes back results in the B register. (Fail = 05H.) If the PROM fails the blank check test, the actual value of the PROM is passed back in 60BBH and the location in 60B7H, 60B8H, and 60B9H. In the off-line mode, any undefined value in B defaults to abort.

Program Routine — The iUP programmer sends the location to be programmed in 60B7H, 60B8H, and 60B9H, and sends the data to be programmed in 50BAH. It also resets 60CFH to 00H. The module returns results in the A register. (Fail = 01.) In the off-line mode, any undefined results default to abort. If the programming failed, the actual value of the PROM is passed back in 60BBH and the location in 60B7H, 60B8H, and 60B9H. The off-line error message will show the address of the failure and user data XOR PROM data. In the on-line mode, the host console will the show failure address, user data, and PROM data.

Overlay Check Routine — The iUP programmer passes no parameters. The iPPS software does not use the overlay check routine; it does its own overlay check on the portion of PROM to be programmed.

In the off-line mode, data the user wants to program is in memory starting at 8000H, and the entire PROM is checked with results sent back in the B register. (Fail = 01.) The module firmware may also send back 03H in the B register to indicate that the iUP programmer should perform the overlay check (on edited PROMs greater than 32K, the iUP programmer automatically performs the overlay check). Any undefined result defaults to abort.

The iUP programmer uses the following algorithms to determine whether the new user data can be programmed over a nonblank PROM location:

1. For PROMs with FFH as a blank state:

   IF [(user data AND PROM data) XOR user data = 0] THEN overlay is possible

2. For PROMs with 00H as a blank state:

   IF [(user data XOR PROM data) AND PROM data = 0] THEN overlay is possible

Reverse Socket Check Routine — The iUP programmer indicates in 60D3H which socket to check and initializes 60CFH to 00H. The module sends back results in the A register. (Fail = 04H.) In the off-line mode, the iUP programmer only recognizes pass, abort, and will default to fail for any other unrecognized result. On chips which support the lock function, 60CFH is set to FFH; on old modules or for chips that do not support the lock function, 60CFH is left at 00H. Addition of other initialization tests can be accomplished by adding these tests to the module reverse socket code. Then, if an error occurs, the module can send a specific error message and abort.

Read Routine — The iUP programmer passes the location to be read in 69B7H, 60B8H, and 60B9H; a code for the (master or program) socket that is to be read from is passed in SKTFLG. The module passes

the data read in 60BBH and the result in the A register. NOTE: There is no failed status, only pass, abort, or power supply failure. In the off-line mode, any undefined result defaults to power supply failure.

Lock Routine — The iUP programmer checks module installation, sets location 60CFH to 00H, and performs the reverse socket test. If 60CFH still equals 00H after the reverse socket check, then the lock function is not available for that module and/or chip. If, however, 60CFH equals 01H after a reverse socket check, then the lock function is available; 60CFH will remain at 01H until the command is finished.

Next (with 60CFH = 01 and 60D0H = 00H), the iUP programmer calls the program subroutine. The module firmware can then communicate with the user by returning (in the A register) one of the values shown in Table 11. When needed, the HL register pair points to the text to be displayed (where the first byte of the message is the length of the message). Handshaking will continue until the result returned is 00H or one of the aborts occurs. (During this process, data sent by the user is contained in location 60BAH and data from the PROM or buffer is contained in 60BBH.) If data values are required, the module stores these values in a buffer (in the module firmware) using 60D0H as an index. No programming or locking is performed until the user has answered YES to the EXECUTE query. At this point, interrupts are disallowed.

Table 11. A-Register Results

Value	Meaning
00H	Pass/done and 60D0H = 00H
02H	Continue and send message pointed to by HL registers
04H	Send execute query to user
07H	Abort (with message)
09H	Lock not available/illegal operation
0AH	Failed; send "PROM BLANK" message
0BH	Failed; send "LOCK FAILED" message
0CH	Failed, send "LOCK FAILED AT" message
0DH	Illegal parameter value
12H	Power supply failure
17H	Abort (without abort message)

Verify — On-line verification is performed by iPPS software using reads. Upon failure, the addresses, user data, and PROM data are displayed. Off-line verification is done by the iUP programmer firmware. Upon failure, the address and user data or PROM data are displayed. The user then has the option of pressing the VERIFY key again to continue verification or pressing the CLEAR key to abort.

Editing PROMS Larger than 32K Bytes — In the off-line mode, editing of PROMs greater than 32K requires a master socket and some special considerations. The iUP programmer has only 32K of image RAM; so, on PROMs greater than 32K, the iUP programmer expects a master PROM in the master socket. The iUP programmer uses this master PROM as the source for programming and overlay checks. (Note that for PROMs larger than 32K bytes, pressing the ROM-to-RAM key does not load data into the URAM. Thus, in using this method of expanding the editing features of the iUP programmer, it is no longer possible to load a 27512 into URAM and then copy URAM to a 27256.)

When the user wishes to edit (off-line) a PROM greater than 32K, data to be edited is copied in 1K blocks to the URAM. (Each 1K block copied always starts on a 1K boundary.) Up to thirty-one 1K blocks can be copied and edited; the last 1K of URAM is not available because this space is needed to manage the editing.

Power-Down Sequence — For current modules, there is an assumption that the module does not need to know when the iPPS software is going to shut off the power supplies; so, the module firmware cannot find this out. For modules that require a certain power-down sequence, there are two possibilities.

- Plan the module to correspond to the iPPS software power-down sequence:
  1. Port 11H is set to 0.
  2. 60B6H is set to 0.
  3. All bits in port 10H are set to 0 except bit 1 (the upload flag), which is not modified.
  4. All power supplies are shut off.
- Module firmware shuts off selected controls in the appropriate order until there is no danger when the iPPS software decides to shut off power supplies. The one check that may be needed is an off-line check. When off-line, the module always checks, reads, or programs the entire PROM — so that if the module is off-line and at the last address, then the iUP programmer will be powering down.

Creating Firmware for Displaying Messages on the Host — To send messages to be displayed by the host, use the following algorithm.

Check location 60A1H to determine whether the host is the iUP programmer or iPDS system.

If host is the iUP programmer
Call 7006H to blank the display
Set HL to 6050H (output buffer)
Insert a carriage return as the first character
Fill in the message in the output buffer
Increase the byte count of the message by 1 (for the carriage return) and place the count in the B register
Set HL = 0
Call 7003
If the host is on-line (i.e., if the host is the iPDS system)
Set 60B4H = 21H to indicate message to iPPS software
Fill in the message starting at 6054H (output buffer plus 3)
Insert a carriage return and linefeed at the end of message
Set B register = message length plus 6
Call 7000H

(7000H and 7003H are actually jump tables to the real address. The jump tables are generated by iPPS software so that updates to iPPS software will be backwards compatible.)

Power Supply Status — There is no status register (02H) to read to tell whether the power supplies have been turned on in the iPDS. Thus, module firmware must monitor 60B6H, if the host is an iPDS. 60B6H is set to 0 upon initialization and when power supplies are turned off. The module firmware must set it to 1 when the power supplies are turned on and set it to 0 when the power supplies are turned off.

WAIT Routine Difference — The 250 microsecond WAIT routines in the iUP programmer and iPDS firmware are inaccurate for short periods of time and do not match each other exactly. (These routines were not revised to ensure backwards compatibility.) For precise timing, the user should write a loop taking into account the differences between the iUP programmer and iPDS clocks.

Use of the E Register — The E register is reserved for use in keyboard interrupts. The module may use the E register if interrupts are first disabled and a known value is restored before re-enabling interrupts. This use of the E register will cause no key presses to be serviced. It is much safer to leave the E register alone.

Keyboard Interrupt Logic — The keyboard interrupt logic is as follows.

```
Save PSW and HL
Save the character read in 60C1H
If the iUP progr· mmer is on-line
    then if key pressed is the on-line key
            then E register = 81H
            else ignore key pressed
    else if key pressed is clear display
            then E register = 88H
      if 60CCH <> 0 /*if operation is process */
      E register = 80H        /* value key press*/
Restore PSW and HL
Return
```

Switched Voltage Programming

There are four switched voltages on the iPDX bus that are turned on or off under program control. The iPDS and iUP systems use different I/O addresses for programming the switched voltages. Under iPPS software, the plug-in module firmware controls the switched voltages. Under user-prepared driver software, separate commands must be included to turn on or off the required switched voltages.

iUP SWITCHED VOLTAGE PROGRAMMING

The iUP system switches the +5.7VSW, +VLOW, +VHIGH, and -12VSW supply lines on and off under program control. The controlling program must write twice to I/O port 03H to set/clear and then clock (high to low transition) the switched voltage flip-flops. The first write to I/O port 03H must have bit 0 (clock) high and bits 1 through 4 set for the desired program voltages. The second write to I/O port 03H keeps bits 1 through 4 at the desired program voltage level while bit 0 goes low. The on/off status of each switched voltage line can be checked by reading I/O port 02H. The iUP system turns off a switched voltage supply line whenever an overcurrent condition is sensed on that line. Figure 10 contains switched voltage control and status bit definitions for the iUP system.

iPDS™ SWITCHED VOLTAGE PROGRAMMING

The iPDS system switches the +5.7VSW, +VLOW, +VHIGH, and -12VSW supply lines on and off under program control. The controlling program (either iPPS software or a user-written driver

program) must write to I/O port E3H in order to turn on/off the required switched voltages. Figure 11 shows the bit definitions for programming the iPDS switched voltage lines.

User-Written iPDX Bus Drivers

User-written iPDX bus driver programs normally access plug-in modules designed for use with the iPDS system. A user-designed iPDX bus plug-in module can address a wide range of applications. The iPDX bus driver program for a user-designed plug-in module can range from simple (e.g., using a single I/O port to upload PROM data to the iPDS system), to complex (e.g., using nearly all the I/O ports to control a high-level instrumentation function).

The I/O ports available to the iPDX bus occupy addresses 10H through 1FH in the iPDS I/O space. Since both an I/O read and an I/O write are associated with each I/O address, the user has 32 I/O ports available for each driver program. Figure 12 is a blank chart that can be used to assign I/O addresses for a specific user driver program. Keep this chart for reference while writing the driver program.

The driver program must be written in 8085 code. Use no more than byte-wide transfers of address, data, and control information. The plug-in module can operate on information of virtually any bit length. The 8-bit width of the iPDX data bus imposes a byte-wide only requirement on all information transfers over the iPDX bus.

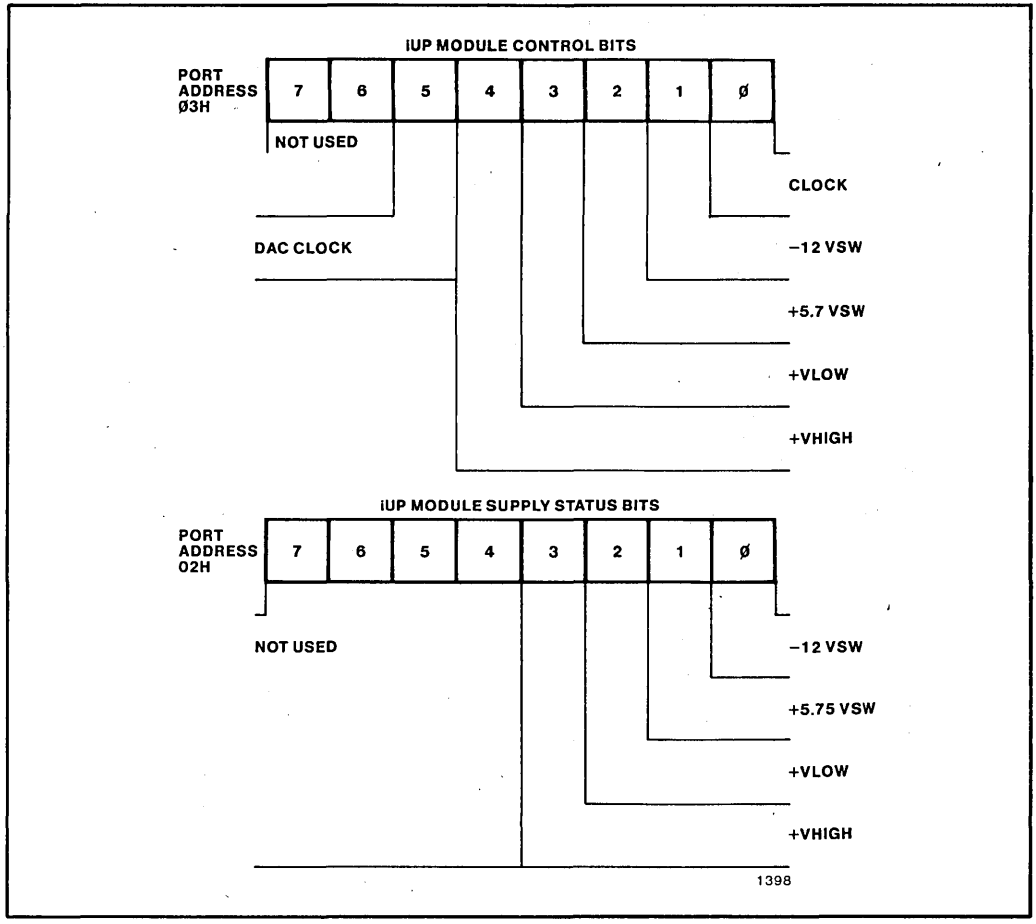


Figure 10. iUP Switched Voltage Control and Status Bit Definitions

230682-001

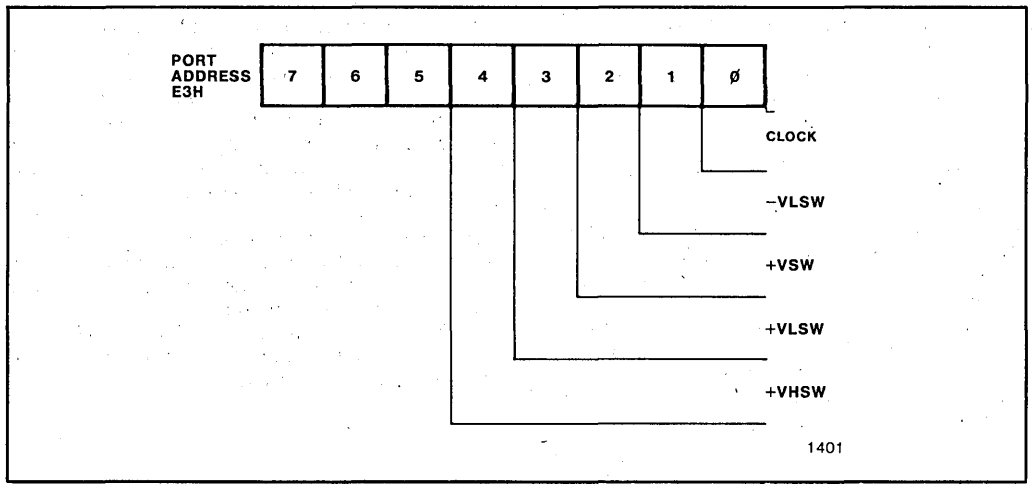


Figure 11. iPDS™ Switched Voltage Control Bit Definitions

230682-001

I/O Port Address	I/O Write Active	I/O Read Active
10H		
11H		
12H		
13H		
14H		
15H		
16H		
17H		
18H		
19H		
1AH		
1BH		
1CH		
1DH		
1EH		
1FH		

1403

Figure 12. Chart of iPDX Bus I/O Address Assignments

230682-001

Network Development Systems

2

NETWORK DEVELOPMENT SYSTEM II (NDS-II)
iMDX-450

NDS-II is a local area network (LAN) of development systems which share resources coordinated by the Network Resource Manager (NRM). The NRM and workstations are interconnected using the 10 megabit/second Ethernet technology. All existing Intel development systems can be upgraded to become NDS-II workstations. In addition, low-cost software workstations are available.

This distributed processing LAN provides:

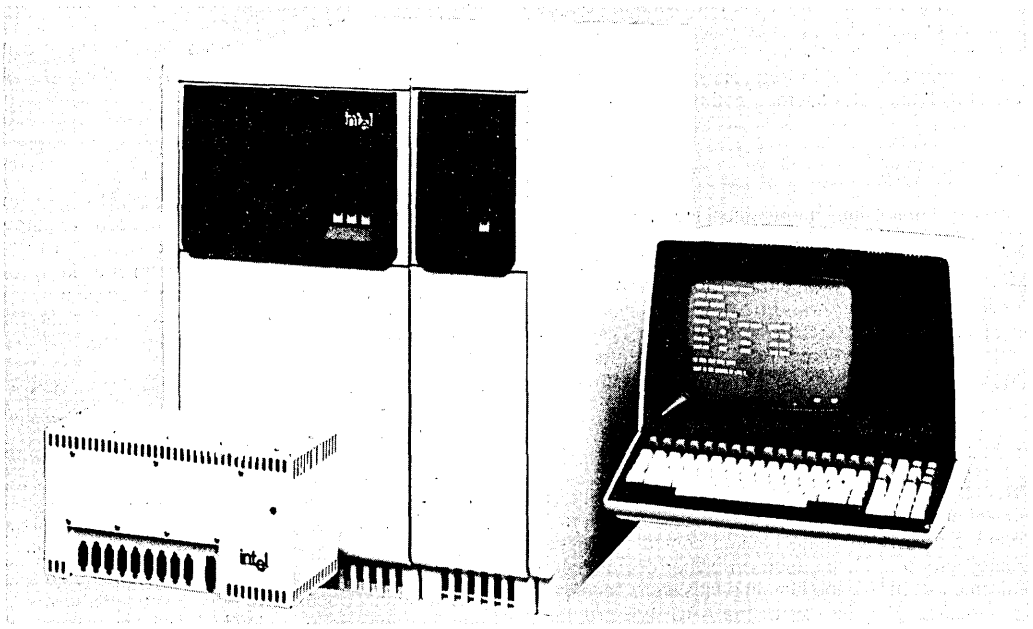
- **Central, Shared Mass Storage Using New or Existing Winchester and Hard Disk Subsystems**
- **Efficient, Intelligent Archival Facilities on Convenient Cartridge Tape Media**
- **A Spooled Line Printer Shared Among All Workstations**
- **Support for All Existing Intellec® Development Systems as Network Workstations**
- **Support for Low Cost ISIS Cluster Software Workstations**
- **A Protected Hierarchical File System**
- **Job Queues that Allow Users to Export Jobs to Other Available Network Workstations**

NDS-II provides the ideal environment for microcomputer development. Software and hardware engineering tools are used most effectively in conjunction with the project management aids provided in this networked host environment. Development equipment cost and product development time are reduced.

NDS-II hosted tools are optimized for the following tasks:

- **Project Organization**
- **Software Version Control**
- **Automated Software System Generation**
- **Electronic Mail Communication**
- **Source Code Creation and Compilation**
- **High-Level Language Debugging**
- **Hardware/Software Integration**
- **In-Target Software Debugging**

The entire spectrum of Intel microcomputer architectures is supported by complete sets of tools: programming languages, software debuggers, in-circuit emulators, PROM programmers, and system tools.



OVERVIEW

NDS-II is a distributed processing LAN optimized for development of microcomputer-based products. It addresses the needs of software engineers, hardware engineers, and engineering management by providing the base environment for development tools and management aids. NDS-II has the capacity to expand to create a network tailored to the needs of any project team.

FUNCTIONAL DESCRIPTION

The Network Resource Manager (NRM) manages all workstation requests for network resources. NRM tasks include the servicing of workstation file requests from the central hierarchical file system, spooling and printing of workstation print requests, routing of electronic mail messages, and queuing and assignment of remote job execution requests. The NRM is also used to perform network maintenance functions such as user name creation, configuration selection, back-up of the central file system, and file system maintenance.

The NRM and workstations communicate via iNA (Intel Network Architecture), which is based on Ethernet communications protocol. The physical Ethernet connections can be made through a low-cost Intellink™ module, through transceivers and Ethernet coaxial cable, or through some combination of these options.

All existing Intellec® Development Systems, Series II, III, IV or Model 800, can be used as NDS-II workstations. The workstations retain all of their standalone functions and, in addition, can access all of the NDS-II shared resources and network services.

File System

The NRM maintains a hierarchical file system, allowing a logical and systematic organization of files. This file system is shared by all network users, and the access to an individual file or directory is controlled by its creator and the Superuser (system administrator.) A centralized archival facility is available, allowing convenient backup of files stored on the shared disks.

Intelligent Archival

The NDS-II ARCHIVE program provides selective archival of files in the shared hierarchical file system. It can incrementally create backup copies of any selected group of files, based on owner or pathname information and/or the last time files were accessed or modified. Regular use of the ARCHIVE program to store files onto tape cartridges or other secondary storage devices ensures security of critical project files.

Distributed Job Control

NDS-II offers distributed processing with local workstation resources and remote network resources. Queues are created at the NRM, where batch jobs are "exported" from some workstations, and "imported" by others for execution. This allows otherwise idle workstations to be utilized, and gives 8-bit workstation users access to 16-bit capabilities. In addition, the Series IV, with its multitasking capability can "import" jobs in one partition while continuing interactive user operations in the other.

Print Spooling

A user-supplied line printer may be attached to the NRM. This allows the line printer to be used as a shared network resource, freeing up workstation processor time and eliminating the need to supply one printer per workstation. Any network user can send print jobs to the spooler queue at the NRM, where they are printed on a FIFO basis.

Tools

All current Intel development tools such as assemblers, high-level language compilers, linkers/locators, software debuggers, and in-circuit emulators can be used on NDS-II workstations. In addition, there are two new tools designed specifically to suit the needs of multi-engineer projects utilizing the NDS-II development environment.

PROGRAM MANAGEMENT TOOLS

Intel's Program Management Tools (PMTs) provide the essential services to efficiently manage large software-intensive development projects. PMTs decrease the amount of time spent on tracking program changes and manually generating software systems, thereby giving engineers more time for software design, development, and testing.

The PMTs consist of "Software Version Control System" (SVCS) and an automated software generation facility (MAKE).

SVCS controls and documents software changes for all file types. SVCS handles storage and retrieval of different versions of a given module, controls update privileges, prevents different users from making changes independently, and requires all changes to be thoroughly documented by recording who made what change, when, and why.

MAKE produces the specification of a "minimum work" job required to generate a new system. This job (i.e. a SUBMIT file) typically includes compiles and links of the latest versions of specified source and object modules. If a newer source module exists for any specified object module, MAKE will specify a compile of this module, replacing the older module in the completed program. Unnecessary links and compiles are eliminated. MAKE does the minimum work required to ensure consistent, up-to-date software thus saving many hours of compiles and links.

The close relationship between SVCS and MAKE helps simplify the overall job of software control. For example, the very latest version of a source module may not be stable enough to be included in a software generation. A less functional, but more reliable version may exist. Since SVCS keeps unique versions distinct, an SVCS module containing the more stable version may be specified for use by MAKE.

ELECTRONIC MAIL

Electronic Mail enables users to send and receive messages and files between any nodes on the NDS-II. Mail maintains a directory called the "post office" which contains user mailboxes (accessible to only a single user), group mailboxes (accessible by a selected group of users), and bulletin board mailboxes (accessible by any user). Users can send interactively created messages, or text or object files, to any mailbox type.

Users can interactively read their mail, save messages in a file, forward messages to other users, and reply to message senders. Or, if they prefer, users may request a simple mailbox summary which includes, for each message, the sender's name, date sent, urgency, and message type (text or object).

NDS-II PMTs and Mail execute on all existing NDS-II workstations, including Series II, Series III, Series IV, Model 800, and ISIS Cluster.

HARDWARE COMPONENTS

Network Resource Manager

The NRM is a free standing unit containing thirteen MULTIBUS® card slots, an integrated 5-¼" flexible disk drive, and an optional cartridge tape subsystem. Processors, memory, and the Ethernet Controller (iSBC-550) occupy six of the card slots. Some of the remaining card slots are used for the various mass storage device controllers.

The NRM is delivered with a console terminal, Intellink module, 50 foot transceiver cable, 10 foot shielded printer cable (with Centronics parallel interface), and 35Mb peripheral-box support accessories. The iNDX Operating System, Program Management Tools, and Electrical Mail software and documentation are also provided.

Intellink™ Module

The Intellink module is a communication device used to connect all NDS-II components (the NRM and workstations) within a local proximity. It serves as a Ethernet local station concentration and provides full Ethernet functionality.

Each Intellink module has nine transceiver ports for connecting workstations and the NRM (via transceiver cables only), plus one Ethernet port for connecting to Ethernet cable (via transceiver and transceiver cable) or to a second Intellink (via the adapter included and a transceiver cable.)

Upgrades

Creating a network in your development environment is accomplished by inserting communication board upgrades in your existing development systems, and by adding a Network Resource Manager. The resources of the network can be incrementally expanded as your development needs increase. Workstations can be added, mass storage increased, and new software tools integrated.

SYSTEM CAPACITY

NDS-II has been designed to efficiently handle the needs for mass storage expansion, increase in the number of users, and expansion of the development laboratory physical size. These needs are met in the following ways:

Storage

NDS-II users may add winchester disk storage capacity to the NRM using peripheral attachments. Each peripheral attachment can contain two 84 megabyte 8" winchester drives. The NRM can support up to two peripheral attachments.

In addition, up to two existing Model 740 Hard Disk Subsystems can be connected to the NRM for use as shared network storage devices. One Model 750 35Mb Winchester Drive Subsystem can be attached to the NRM (if no 35 megabyte peripherals are attached.)

An optional Cartridge Tape Subsystem can be installed in the NRM chassis to provide convenient back-up onto standard 12 megabyte tape cartridges.

Users

NDS-II allows multiple users to access the system via workstations that are attached to NRM via Ethernet technology. Up to 16 Intellec workstations can be incorporated into NDS-II. A maximum of 28 active users can be supported on the network, though the feasibility of such a configuration varies with network loading and workstation types.

Geography

NDS-II can be expanded to connect local development groups in different locations throughout a building. Within a 50-meter radius, up to eight Intellec workstations can be attached to the NRM using a single Intellink module. Within a 75-meter radius, a total of sixteen Intellec workstations can be attached using two Intellink modules. Alternatively, or in conjunction with one or more Intellink modules, Ethernet coaxial cable and transceivers can be used to connect any or all of the workstations and the NRM along a maximum of 1 kilometer of Ethernet coax.

WORKSTATIONS

Intellec® Workstations

All Intellec development systems produced since 1975 can be used as NDS-II workstations. These include: Model 800, Series II, Series III, and Series IV.

Development systems must be upgraded with the communication boards, cables, and the appropriate software. See Ordering Information for the product code of the kit that corresponds to your present Intellec System.

Low-Cost Workstations

ISIS Cluster Board Packages provide additional, inexpensive workstations on NDS-II. Each Cluster Package includes an 8085 CPU, 4K of ROM (bootstrap and dianostics), and 64K of RAM. The Cluster Board must be hosted in an Intellec workstation (Series II, Series III, Series IV, or Model 800 workstation) with which it shares the power supply and network communication boards.

When attached to the RS232C port of a user-supplied terminal, an ISIS Cluster workstation will boot onto the network and provide an ISIS environment which can run all Intellec-supported 8-bit software and EXPORT jobs to other network resources.

CONNECTION TO OTHER EQUIPMENT

iRMX™ System Interface

iRMX-based microcomputer systems (86/330, 86/380, 86/310) can be connected directly to NDS-II using the iNA-955 NDS-II/iRMX Link software package and the iSBC-550 Communication Board Package. iRMX system developers can use the development system environment of NDS-II to develop their application and then download at Ethernet speed to the iRMX target system(s). The iRMX Link also provides a programatic interface to NDS-II, which allows iRMX OEMs to develop customized network environments.

VAX* Interfaces

The iMDX-394 and iMDX-395 Asynchronous Communication Link products can be used to connect VAX/VMS and VAX/UNIX general computing environments to the NDS-II development environment. The Communication Link operates via a serial connection to any NDS-II workstation and allows files to be uploaded or downloaded between the VAX and NRM mass storage devices.

Contact your local Intel Sales Office for information about the Ethernet-based link for VAX/VMS.

iPDS™ Development System, IBM-PC, and Other Interfaces

A program available from the INSITE User's Program Library can be used to connect the IBM-PC and the iPDS Development System to NDS-II. The interface is composed of an ISIS Cluster board which is connected via RS232C to the PC or iPDS, and via Ethernet to the NRM. Source code is provided, and can be adapted to suit other systems.

SUPPORT

Site Preparation/Configuration Guide

A site preparation manual, the "NDS-II Configuration and Ordering Guide" (order # 121969) is available. The manual assists the future NDS-II owner in both configuring a network suited to his needs and in preparing the physical area for the new system.

* VAX is a registered trademark of Digital Equipment Corporation

Installation/Warranty

Within Intel service areas, on-site installation is currently included in the price of the Network Resource Manager. In addition, 90-day on-site maintenance, including parts and labor is currently included. Service contracts for periods beyond 90 days are currently available. Installation, warranty, and service contracts for locations outside normal service areas are currently available.

The NRM also currently includes 90 days of initial support, consisting of software updates/releases (if available), and subscription services (telephone hotline support, software performance report service, and technical reports.) To receive this support, the customer must mail in the software registration card. Additional software support beyond 90 days is currently available.

On-Site Training/Technical Assistance

Intel training courses, Field Application Engineers, and System Engineers are available to assist the NDS-II customers in maximizing the benefits of the system. Contact your local Intel representative regarding the services currently available.

SPECIFICATIONS

System Overview

ELECTRICAL CHARACTERISTICS

AC Power Requirement (for NRM with up to 2 peripheral attachments):
175V - 260V
50Hz or 60Hz
15A (maximum)

Control Terminal is available for:
110/120V, 50/60Hz, 2A (max)
or 220/240V, 50/60Hz, 1A (max)

ENVIRONMENTAL CHARACTERISTICS

Operating Temperature: 5 C–35 C
Humidity: 10%–80% non-condensing
Non-Operating Temperature: –10 C–55 C
Electrostatic Discharge (ESD) Tolerance: 8KV*

* Any peripherals added to the system that do not meet Intel's ESD specification will void the ESD portion of the warranty.

Network Resource Manager (NRM)

PHYSICAL CHARACTERISTICS

Width: 16″ (40 cm)
Height: 32″ (80 cm)
Depth: 31″ (78 cm)
Weight: 110 lb (50 kg)

Flexible Disk Drive (integrated in NRM)

Type: 5-¼″ mini-floppy
Density: double sided, double density
Capacity: 656 Kbyte

Cartridge Tape Subsystem (integrated in NRM)

SPECIFICATION

Type: ¼″ tape DC300XL data cartridge
Density: 6400 BPI
Capacity: 12 Megabytes
(formatted, using 4K records)
Recording Technique: CGR
Record Size: 1–16 Kbytes

PERFORMANCE

Tape Transfer Rate: 24Kb/sec
Read/Write Speed: 30″/sec
Fast Tape Motion: 70″/sec
Start/Stop: 25 ms @ 30″/sec
75 ms @ 70″/sec

WINCHESTER SUBSYSTEM ("peripheral attachment")

PHYSICAL CHARACTERISTICS

Width: 6″ (16 cm)
Height: 32″ (80 cm)
Depth: 31″ (78 cm)
Weight: 90 lb (41 kg)

DRIVE SPECIFICATION

Type: Winchester Sealed Disk
Capacity: 84 Megabytes (unformatted)
73.92 Megabytes (formatted)
Density: 9950 bit/inch
Recording Technique: MFM
Bytes/Sector: 512
Sectors/Track: 35
Tracks/Surface: 589
Recording Surfaces: 7

DRIVE PERFORMANCE

Disk Transfer Rate:	5 Mbits/sec
Disk Access Time:	
Average	20 ms
Full Stroke	40 ms
Rotational Speed:	3600 rpm

Control Terminal

PHYSICAL CHARACTERISTICS

Logic Box:	19″ W × 14″ D × 3″ H (48 cm × 36 cm × 7 cm)
Video Module:	13″ W × 14″ D × 10″ H (33 cm × 35 cm × 25 cm)
Keyboard:	19″ W × 8″ D × 3″ H (48 cm × 20 cm × 7 cm)
Total Weight:	32 lbs. (15 kg)

HOST INTERFACE

Type:	RS232C
Speed:	110–19.2K baud

CRT

Screen:	12″ diagonal tilt & swivel
Display:	phosphor, P31, green non-glare faceplate
Format:	2 pages (3840 bytes) 24 lines/page 80 characters/line
Cursor:	blinking underscore
Characters:	7 × 9 matrix ASCII character set

KEYBOARD (DETACHABLE)

# Keys:	103
Types:	alpha-numeric typewriter block numeric keypad cursor control & editing block 16 function keys

Intellink™ Module

PHYSICAL CHARACTERISTICS

Width:	14″ (36 cm)
Height:	7.5″ (19 cm)
Depth:	5.5″ (14 cm)
Weight:	5 lb (2.3 kg)

INTERFACES

Transceiver Cable Ports:	9
Ethernet Ports:	1
Adapter:	for Ethernet port (to connect to transceiver port on second Intellink module)

SOFTWARE

iNDX Operating System (including support for Series IV workstations)

ISIS III (N)/III (C) Operating System (for Series II, Series III, Model 800, and ISIS Cluster workstations)

Program Management Tools (8-bit & 16-bit versions)

Electronic Mail (8-bit & 16-bit versions)

NRM Diagnostics

DOCUMENTATION

(Installation and Checkout Manuals are also provided.)

NRM:

NDS-II Network Development System Overview (#121761)

NDS-II Network Resource Manager User's Guide (#134300)

Series II, III, Model 800 Workstations:

NDS-II ISIS-III (N) User's Guide (#121765)

Series IV Workstations:

Intellec® Series IV Operating and Programming Guide (#121753)

ISIS Cluster Workstations:

NDS-II ISIS-III (C) User's Guide Supplement (#122098)

Software:

NDS-II Electronic Mail User's Guide (#122146-001)

A User's Guide to Program Management Tools (#121958)

ORDERING INFORMATION:

Network Resource Managers (see Table 1)

iMDX-450-A000 NDS-II NETWORK RESOURCE MANAGER

Includes 220V NRM Processor Chassis, 110V System Console Terminal, Intellink, 50-meter transceiver Cable, 10-foot Printer Cable, Cabling for One Model 740 Hard Disk and/or one iMDX-750 35Mb Winchester Disk, System Software and Documentation, Program Management Tools, and Electronic Mail.

iMDX-450-B000 NDS-II NETWORK RESOURCE MANAGER

Includes 220V NRM Processor Chassis, 220V System Console Terminal, Intellink, 50-meter transceiver Cable, 10-foot Printer Cable, Cabling for One Model 740 Hard Disk and/or one iMDX-750 35Mb Winchester Disk, System Software and Documentation, Program Management Tools, and Electronic Mail.

iMDX-450-AT84 84Mb NDS-II NETWORK RESOURCE MANAGER

Includes iMDX-450-A000 Network Resource Manager plus 84Mb Winchester Subsystem and 12Mb Cartridge Tape Subsystem.

iMDX-450-BT84 84Mb NDS-II NETWORK RESOURCE MANAGER

Includes iMDX-450-B000 Network Resource Manager plus 84Mb Winchester Subsystem and 12Mb Cartridge Tape Subsystem.

Table 1. Network Resources Managers

Order Code	NRM Voltage	Terminal Voltage	Winchester Subsystem	Tape Subsystem
iMDX-450-A000	220V	110V	not included	not included
iMDX-450-B000	220V	220V	not included	not included
iMDX-450-AT84	220V	110V	84 megabytes	12 megabytes
iMDX-450-BT84	220V	220V	84 megabytes	12 megabytes

NRM Peripheral Upgrades (see Table 2)

iMDX-771-B3 1ST 84Mb PERIPHERAL ATTACHMENT FOR NRM.

Includes one 84Mb Winchester Disk Drive configured as drive 0, cabling to support drive 1, plus the 84Mb Winchester Controller.

iMDX-771-B2 2ND 84Mb PERIPHERAL ATTACHMENT FOR NRM.

Includes one 84Mb Winchester Disk Drive configured as drive 2, and cabling to support drive 3.

iMDX-772 ADD-IN 84Mb DRIVE FOR NRM.

Includes one 84Mb Winchester Disk Drive, to be used as drive 1 or 3 in an iMDX-771 Peripheral Attachment.

iMDX-452 CARTRIDGE TAPE SUBSYSTEM FOR NRM.

Includes Cartridge Tape Drive, Controller, one standard tape cartridge, and accessory kit. Requires iMDX-3008 900 Watt Power Supply for NRMs with serial numbers below 740.

Table 2. NRM Peripheral Upgrades

Order Code	Voltage	Drive Type	Drive #	Controller	Chassis
iMDX-771-B3	220V	84Mb Winchester	0	included	included
iMDX-771-B2	220V	84Mb Winchester	3	*	included
iMDX-772	**	84Mb Winchester	2 or 4	*	***
iMDX-452	**	Cartridge Tape	n/a	included	****

* controlled by existing 84Mb controller in NRM
** operates in 110V or 220V systems
*** second drive in -771-B3/-B2 (or -A1/-B1) chassis
**** fits into NRM chassis

Software Workstations

iMDX-580 ISIS CLUSTER BOARD PACKAGE FOR SERIES II, SERIES III, OR MODEL 800.

Includes processor board, cables, and documentation. Must be installed on NDS-II in a Model 800, Series II, or Series III workstation and attached to a user-supplied terminal.

iMDX-581KIT ISIS CLUSTER BOARD PACKAGE FOR SERIES IV.

Includes iMDX-580 and iMDX-582. Must be installed on NDS-II in a Series IV (or Model 800, Series II, or Series III) workstation and attached to a user-supplied terminal.

iMDX-582 ISIS CLUSTER UPGRADE KIT FOR SERIES IV.

Includes internal cable, mounting hardware, and documentation required to install an existing iMDX-580 ISIS Cluster Board in a Series IV host.

Workstation Kits

iMDX-455 NDS-II WORKSTATION UPGRADE KIT FOR SERIES II/85, SERIES III, AND MODEL 800.

Includes network communication board set, software, and documentation. Transceiver cables must be ordered separately.

iMDX-455I NDS-II WORKSTATION UPGRADE KIT FOR SERIES II/80.

Includes iMDX-455 plus 8085-based CPU board. Transceiver cables must be ordered separately.

iMDX-456 NDS-II WORKSTATION UPGRADE KIT FOR SERIES IV.

Includes network communication board set and documentation. Transceiver cables must be ordered separately.

Cables & Accessories

iMDX-457 10 METER TRANSCEIVER CABLE.

iMDX-458 50 METER TRANSCEIVER CABLE.

iMDX-3016F-1 25 METER ETHERNET COAX ASSEMBLY.

Includes 25 meter (76.8 feet) teflon coaxial cable segment, terminators, and coupler for joining additional coax segments.

iMDX-3016F-2 100 METER ETHERNET COAX ASSEMBLY.

Includes 100 meter (383.9 feet) teflon coaxial cable segment, terminators, and coupler for joining additional coax segments.

iMDX-3015F ETHERNET TRANSCEIVER KIT FOR TEFLON COAX.

iDCM-911-1 INTELLINK MODULE.

Contains 9 transceiver cable ports, plus Ethernet port for optional connection to transceiver or second Intellink (adapter included.)

MDS*-506 HARD DISK CABLE KIT FOR SECOND MODEL 740 ON NRM.

Connects second Model 740 Hard Disk Subsystem to first Model 740, to allow shared NDS-II usage of these mass storage devices. Includes internal cable and I/O cable. (Converts MDS-740 into MDS-743.)

iMDX-450-U11 110V TO 220V UPGRADE KIT FOR NRM AND ONE PERIPHERAL ATTACHMENT.

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

iMDX-580/581
ISIS CLUSTER BOARD PACKAGES

- **Converts Spare Slots in Series II, III, IV, or Model 800 Workstations into Additional Workstations.**
- **Up to Seven Additional NDS-II Workstations May Reside in One Development System Host.**
- **Utilizes the Powerful ISIS-III(C) Operating System.**
- **Supports all 8-Bit ISIS-Based Software Development Tools including the AEDIT-80, Text Editor, Program Management Tools, and NDS-II Electronic Mail.**
- **Supports 8-Bit Macroassemblers and High-Level Languages.**
- **Supports 16-Bit Development with local ASM-86 and PL/M-86, and via NDS-II Distributed Job Control.**
- **Provides Execution Environment for 8085-Based Application Programs.**
- **Compatible with a Variety of 9.6K or 19.2K Baud Terminals.**

The ISIS Cluster Board Package is an NDS-II upgrade that cost effectively supports incremental software workstations on the network. Each Cluster board provides an 8085 CPU, 4K of ROM and 64K of RAM, and must reside in a Series II, Series III, Series IV, or Model 800 development system host. When attached to a user-supplied terminal, an ISIS Cluster workstation will boot onto the NDS-II and provide an ISIS environment which allows users to log on to the network and run Intellec®-supported 8-bit software, as well as "export" jobs to other network resources.

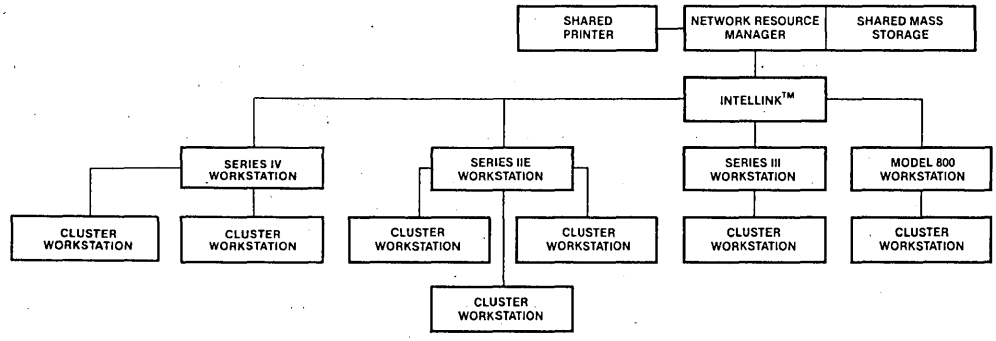


Figure 1. Example of an NDS-II Configuration

MAY 1984

ORDER NUMBER: 210938-003

FUNCTIONAL DESCRIPTION

Summary: The ISIS Cluster board is a single-board computer centered around an 8085AH-2 CPU running at 4.0 MHz. 64K bytes of dual-ported RAM are provided on-board, along with 4K of ROM preprogrammed with a bootstrap program and self-test diagnostics.

The ISIS Cluster MULTIBUS® interface provides data and address interface latches. The serial I/O interface provides a full duplex RS232C serial data communications channel that can be programmed to handle serial data transmission at 19.2K or 9.6K baud. Software reset may be accomplished using the BREAK key on the terminal.

A block diagram of the ISIS Cluster board is shown in Figure 2.

Central Processing Unit

Intel's powerful 8-bit 8085AH-2 CPU running at 4.0 MHz is the central processor for the Cluster board. It is fully software compatible with all 8-bit ISIS-based languages and utilities which run on the Intellec® Model 800, Series II/80, Series II/85, or Series IIE.

System ROM

4K bytes of non-volatile read only memory are included on the Cluster board using Intel's 2732A EPROM. Preprogrammed with the ISIS Cluster Boot program, the system ROM provides boot-up and diagnostic capabilities, and a generalized I/O system.

The Boot program communicates with the operator via an interactive console. Upon reset of the Cluster system, execution is handled by the bootstrap PROMs which overlay 4K bytes of system RAM, initialize Cluster board devices, run self-test diagnostic, and perform a communication handshake before prompting the user.

RAM

The Cluster board uses eight 2164 RAMs and a dual port RAM controller to provide 64K of dual-ported dynamic read/write memory. Slave RAM decode logic allows extended MULTIBUS addressing with a 1 Megabyte address space, so that RAM accesses may occur from either the Cluster board or from the network communication boards interacting via the MULTIBUS interface. Since on-board RAM accesses do not require MULTIBUS accesses, the bus is available for other concurrent operations. Dynamic RAM refresh is accomplished automatically by the Cluster board.

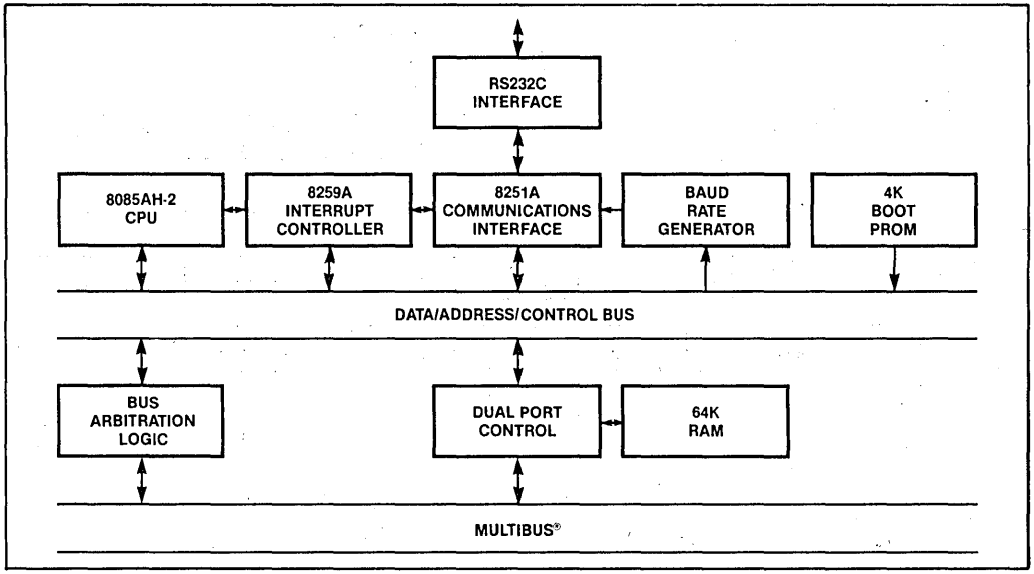


Figure 2. Block Diagram of the ISIS Cluster Board

Serial I/O

A programmable communications interface using the Intel 8251A USART (Universal Synchronous/Asynchronous Receiver/Transmitter) is on the Cluster board, and provides a full duplex RS232C serial communications channel. The transmit and receive lines are link exchangeable to enable a data set or data terminal to be used with the Cluster board. The board is pre-set for 9600 baud, but may be jumpered for 19.2K baud.

Programmable Timers

The interval timer capability is implemented with an Intel 8254 Programmable Interval Timer. The 8254 includes three 16-bit BCD or binary counters. The first two counters are not used. The output from the third counter is applied to the serial I/O interface and provides the baud rate frequency for serial communications.

Interrupt Controller

The Cluster board also includes an Intel 8295A Interrupt Controller. It is pre-configured with Interrupt 1 triggered by the BREAK key on the user-supplied terminal.

MULTIBUS® Interface

The Cluster board is a complete computer on a single board, capable of supporting a variety of 8-bit development tools. For applications requiring additional processing capacity, the Cluster board provides full MULTIBUS arbitration control logic. The bus arbitration logic operates synchronously with a MULTIBUS clock. All memory references made by the CPU refer to the on-board RAM. The Cluster board cannot access devices local to the host development system, but all of the shared network resources are accessible.

The Cluster board communicates with the Network Resource Manager via the MULTIBUS interface and the network communication board set in the host development system.

System Configuration

Each ISIS Cluster board requires one master slot in an Intellec cardcage. The host development system may be a Model 800, Series IV, Series II or IIE, or Series III or IIIE with an optional expansion chassis. A Series II or IIE with an expansion chassis will support a maximum of seven ISIS Cluster workstations, since the Integrated Processor Card and Network Communication boards occupy three of the ten cardcage slots. A Model 800 will support a maximum of 2 ISIS Cluster workstations, and Series IV workstation will support a maximum of 4 ISIS Cluster workstations. Each ISIS Cluster workstation counts as one additional network workstation, so the maximum number of Cluster workstations on a network is constrained only by the total number of users supported by the NDS-II Network Resource Manager. NDS-II iNDX Release 2.8 or later will support ISIS Cluster workstations in any Intellec development system host, including the Series IV.

Programming Capability

The Cluster workstation's ISIS environment supports all 8-bit Intellec-supported ISIS-based software, including the programmer-oriented AEDIT-80 text editor, PMT-80 Program Management Tools, NDS-II Electronic Mail, 8-bit macroassemblers, and PL/M, FORTRAN, PASCAL, and BASIC high-level 8-bit languages. 16-bit development is supported by the ASM86 cross assembler and the PL/M-86 cross compiler, or by "exporting" any 16-bit job to a 16-bit workstation for execution.

SPECIFICATIONS

CPU: 4.0 MHz 8085AH-2

MEMORY:

On-board RAM, 64K bytes, dual-ported

On-board ROM, 4K bytes preprogrammed with the ISIS Cluster Bootstrap Program

Interfaces

SERIAL I/O:	RS232C compatible, programmable interface
BUS:	MULTIBUS compatible, TTL level
TIMER:	3 programmable 16-bit BCD or binary counters, 1 used as baud rate timer
INTERRUPTS:	1 interrupt level available to user via the BREAK key on the terminal

Physical Characteristics

Two-sided printed circuit board fits into Intellec cardcage:

Length:	12 inches
Width:	6.75 inches
Depth:	0.062 inches

Internal flat ribbon cable connects ISIS Cluster board edge connector to the development system rear panel.

External 10-foot RS232C compatible cable connects the development system rear panel to a user-supplied terminal.

Electrical Characteristics

DC Power Requirements (from Mainframe)

V_{CC} = +5V, 4.5 Amps
V_{DD} = +12V, 25 mA
V_{AA} = -12V, 23 mA

Environmental Specifications

Operating Temperature: 0°C to 55°C

Humidity: up to 90%, without condensation

Documentation

ISIS Cluster Installation, Operation, and Service Manual (#122100)

Series IV iMDX-580 and iMDX-582 ISIS Cluster Board Package Installation, Operation, and Service Manual (#134650)

NDS-II ISIS-III(C) User's Guide Supplement (#122098)

Equipment Required

Recommended Terminals* (one per ISIS Cluster Board)

The following terminals meet Intel environmental specifications and are recommended for use with the ISIS Cluster Board Package:

ZENTEC, MODEL ZMS-35, COBRA

The following terminals have been tested and found to be interface compatible with the ISIS Cluster board; configuration files are provided for these terminals. However, they do not meet Intel environmental specifications: adverse electrostatic conditions may produce unpredictable screen output, requiring terminal reset.

Hazeltine, Model 1510

Televideo, Model 910+, 925, 950

Lear Seigler, Model ADM 3A

Adds Viewpoint, Model 3A+

Qume, Model 102

*All of the recommended terminals run at 9.6K or 19.2K baud.

CAUTION: Other RS232C-compatible devices may not meet Intel environmental specifications, and could degrade overall system performance.

Host Development System (requires one open 6.75 x 12 in. master slot in system cardcage per ISIS Cluster board):

Series II/85 or Series IIE*

Series III or Series IIIE*

Model 800**

Series IV

*with optional Expansion Chassis
**supports maximum of 2 ISIS Cluster Boards

Workstation Upgrade Kit (one per host system):

iMDX-455 for Series II, III, or Model 800

iMDX-456 for Series IV

NDS-II Network Resource Manager with Winchester or Hard Disk Mass Storage

Software Required

For Series II, III, or Model 800 Host:

NDS-II iNDX Operating System, Release 2.0 or later

ISIS-III(N)/III(C) Operating System, Version 2.0 or later*

For All Development System Hosts, Including Series IV:

NDS-II iNDX Operating System, Release 2.8 or later

Series IV iNDX Workstation Operating System, Release 2.8 or later**

ISIS-III(N), version 2.2 or later**

ISIS-III(C), version 2.2 or later**

*included with NDS-II Release 2.0
**included with NDS-II Release 2.8

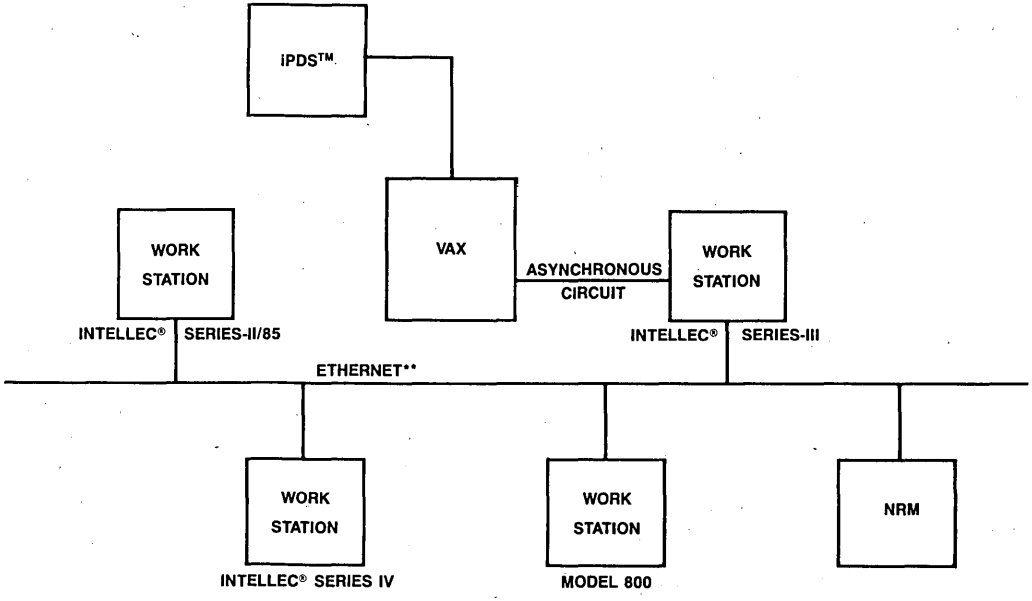
ORDERING INFORMATION

Part Number	Description
iMDX-580	ISIS Cluster Board Package for Series II, Series III, or Model 800 -includes processor board, cables, and documentation. Must be installed on NDS-II in a Model 800, Series II or Series III workstation and connected to a user-supplied terminal.
iMDX-581KIT	ISIS Cluster Board Package for Series IV - Includes iMDX-580 and iMDX-582. Must be installed on NDS-II in a Series IV (or Series II, III, or Model 800) workstation and attached to a user-supplied terminal.
iMDX-582	ISIS Cluster Upgrade Kit for Series IV - Includes internal cable, mounting hardware, and documentation required for installing an existing iMDX-580 ISIS Cluster Board in a Series IV host.

INTEL ASYNCHRONOUS COMMUNICATIONS LINK

- **Communications software for VAX* host computer and Intel microcomputer development systems**
- **Compatible with VAX/VMS* and UNIX† operating systems**
- **Supports Intel's Model 800, Intellec® Series II, Series III, Series IV and iPDS™ microcomputer development systems**
- **Supports NDS-II workstations**
- **Allows development system console to function as a host terminal**
- **Operates through direct cable connection or over telephone lines**
- **Software selectable transmission rate from 300 to 9600 baud**

Intel's Asynchronous Communications Link (ACL) enables one or more Intel microcomputer development systems to communicate with a Digital Equipment Corporation VAX family computer. The link supports Intel Model 800, Intellec Series II, Series III, Series IV or iPDS™ development systems and NDS-II workstations. Programmers can use the editing and file management tools of the host computer and then download to the Intel microcomputer development system for debugging and execution. Programmers can use their microcomputer development system as a host terminal and control the host directly without changing terminals.



NDS-II Example

*VAX and VAX/VMS are trademarks of Digital Equipment Corporation.
†UNIX is a trademark of Bell Laboratories.
**Ethernet is a trademark of Xerox Corp.

January, 1984
ORDER NUMBER: 210903-002

FUNCTIONAL DESCRIPTION

The Asynchronous Communications Link (ACL) consists of cooperating programs: one that runs on the host computer, and others that run on each microcomputer development system. The development system programs execute under the ISIS-II or ISIS-III(N), ISIS-IV, ISIS-II(W) or ISIS-PDS operating system. They invoke the companion program on the VAX-11/7XX, which runs under either the VAX/VMS or UNIX operating system.

The link provides three modes of communication: on-line transmission, single-line transmission, and file transfer. In on-line mode, the development system functions as a host terminal, enabling the programmer to develop programs using the host computer's editing, compilation, and file-management tools directly from the development system's console. Later, switching to file transfer mode, text files and object code can be downloaded from the host to the development system for debugging and execution. Alternatively, files can be sent back to the host for editing or storage. In single line mode, the programmer can send single-line commands to the host computer while remaining in the ISIS environment.

The user can select transmission rates over the link from 300 to 9600 baud. The link transmits in encapsulated blocks. The receiver program validates the transmission by checking record-number and checksum information in each block's header. In the event of a transmission error, the receiving program recognizes a bad block and requests the sender to retransmit the correct block. The result is highly reliable data communications.

SOFTWARE PACKAGE

The Asynchronous Communications Link Package contains either a VAX/VMS or UNIX compatible magnetic tape, a single 8", double 8", Series-IV 5¼", and PDS 5¼" diskette compatible with the Intellec development system, and the *Asynchronous Communications Link User's Guide* containing installation, configuration, and operation information.

HARDWARE CONNECTION

The Link sends data over an RS232C cable. The communication line from the host computer connects directly to a development system port.

TELECOMMUNICATIONS USING THE LINK

The ACL is ideal for cross-host program development using a commercial timesharing service. This configuration requires RS232C compatible modems and a telecommunications line. Depending on the anticipated level of usage, wide-area telephone service (WATS), a leased line, or a data communications network may be chosen to keep operating overhead low.

NDS-II ACCESS USING THE LINK

The ACL is ideal for interconnecting VAX host computers with NDS-II. This configuration requires that an NDS-II workstation be connected to the VAX host computer using the RS232C interface and to NDS-II using the Ethernet interface.

All three modes of communication operate identically on NDS-II. In the on-line mode, the development workstation operates as a host terminal, and concurrently, as an NDS-II workstation. It is an easy transition between the VAX and ISIS operating system environments as LOGON/LOGOFF sequences are not required to re-enter environments.

In file transfer mode, text and object files can be transferred from the VAX directly to the Winchester Disk at the NRM without first copying the files to the workstation local floppy disk. Similarly, files residing on the NDS-II Network File System (the Winchester Disk at the NRM) can be transferred directly to the VAX without using local workstation storage.

Using the EXPORT/IMPORT mechanisms of NDS-II, a network workstation which is not directly connected to the VAX can cause files to be transferred between the VAX and NRM. For example, any NDS-II workstation can "EXPORT" ACL commands to another "IM-

210903-002

PORT"ing NDS-II workstation which is physically connected to a VAX. The "IMPORT"ing workstation executes the ACL command file causing the desired action to occur.

VAX ACCESS USING THE LINK

Users who want multiple workstations concurrently operating as VAX terminals (ONLINE mode) must physically connect each workstation to the VAX. However, users who want multiple workstations to be able to upload/download files, for example, must only physically connect one workstation to the VAX. By using the EXPORT/IMPORT mechanism of NDS-II as described above, the user can have multiple workstations accessing the VAX using only one connection.

SPECIFICATIONS

Software

Asynchronous Communications Link development system programs

VAX/VMS or UNIX companion program

Media

Single- or double-density ISIS 8" and Series-IV, PDS 5¼" compatible diskette

600-ft. 1600 bpi magnetic tape, VAX/VMS or UNIX compatible

Data Transfer Speeds

All systems up to 9600 bps

Online Terminal Mode Speeds

Series II, Series III, Series IV — 2400 bps max
PDS — 9600 bps max
Model 800 — equal to or less than the Terminal speed

Manual

Asynchronous Communications Link User's Guide, Order No. 172174-001

Required Host Configuration

VAX-11/7XX running VAX/VMS (Version 3.2) or fourth Berkeley distribution of UNIX 4.1

Required Intel Development System Configuration

Model 800, Series II, Series III, Series IV, or iPDS under ISIS

Required Connection

RS232C compatible — cable 3M-3349/25 or equivalent; 25-pin connector 3M-3482-1000 or equivalent

Recommended Modems for Telecommunications

300 baud — Bell* 103 modem; VADIC† 3455 modem or equivalent

1200 baud — Bell 202 modem; VADIC 3451 modem or equivalent

9600 baud — Bell 209A (full duplex, leased line) or equivalent

Note: Since one of the two Model 800 ports uses a current loop interface, Model 800 users need a terminal or modem that is current loop compatible, or a current loop/RS232C converter.

The Model 800 might require modification by a qualified hardware technician. Intel does not repair or maintain boards with these changes.

ORDERING INFORMATION

Product Name

Asynchronous Communications Link

Ordering Code‡

iMDX 394 for VAX/VMS systems
iMDX 395 for UNIX systems

*Bell is a trademark of American Telephone and Telegraph.
†VADIC is a trademark of Racal-Vadic Inc.
‡See price book for proper suffixes for options and media selection.

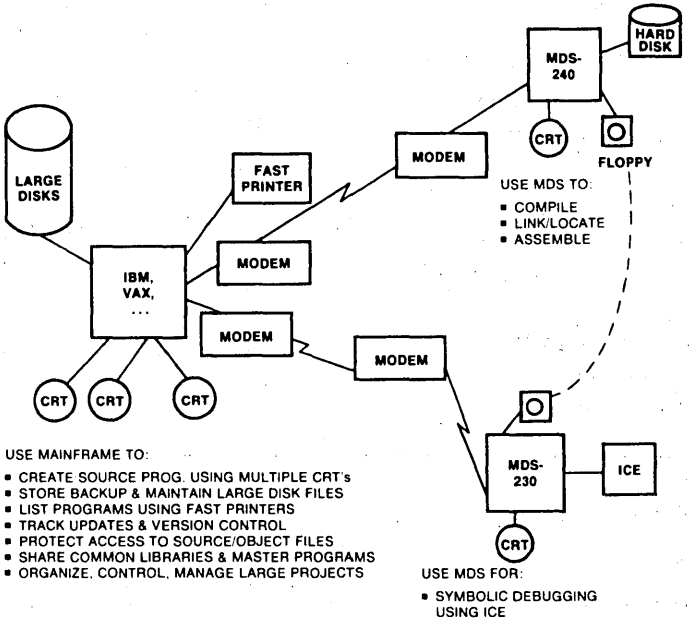
210903-002

MAINFRAME LINK FOR DISTRIBUTED DEVELOPMENT

- **Integrates user mainframe resources with Intellec® Development Systems.**
- **Uses IBM 2780/3780 standard BISYNC protocol supported by a majority of mainframes and minicomputers.**
- **Protocol supports full error detection with automatic retry.**
- **Software runs under ISIS-II on any Intellec® Development System.**
- **Communicates with remote systems on dedicated or switched (dial-up) telephone lines.**
- **Package also includes tests and a connector for loop-back self-test capability.**

The Mainframe Link consists of software, modem cable to connect the development system to the modem and a loopback connector for diagnostic testing. The software runs under ISIS-II on Intellec Development Systems. It emulates the operation of an IBM 2780 or 3780 Remote Job Entry (RJE) terminal to (1) transmit ISIS-II files to a remote system or (2) receive files from a remote system using standard BISYNC 2780/3780 protocol. The remote system can be any mainframe or minicomputer which supports the IBM 2780 or 3780 communications interface standard. Files may contain ASCII or binary data so that either program source files (ASCII) or program object files (binary) may be transmitted.

The Mainframe Link allows the user to integrate in-house mainframe resources with Intellec Microcomputer Development resources. The mainframe can be used for storage, maintenance and management of program source and object files. The program source can be downloaded to a development system for compilation, assembly, linkage, and/or location. The linked modules can be transmitted and saved on the mainframe to be shared by all programmers. The linked program can then be downloaded to a development system for debugging using ICE emulation.



MAY 1983
AFN-01549C

FEATURES

- Runs under ISIS-II on any Intellec® Microcomputer Development System.
- Communicates with a remote system using IBM 2780/3780 standard BISYNC protocol, which is supported by a majority of minicomputers and mainframes, on dedicated or switched (dial-up) telephone lines.
- The modem cable supplied with the package can be used to connect the Intellec® Development System to the modem (or modem eliminator) using the standard RS232C port.
- Supports user selectable data transmission rates of up to 9600 baud.
- Package includes diagnostic tests used to verify the operation of the Intellec® Development System using the loop-back connector supplied and data transmission up to the modem using the analog loop-back feature.
- System can be configured to match the requirements of the installation, i.e., using modem eliminators for connections up to fifty (50) feet, or by using modems and telephone lines.
- Software can be configured from several configuration options such as:

  2780, 3780 or Intel Mode

  Transparent mode for binary data

  Non-transparent mode for ASCII data

  Automatic translation from ASCII to EBCDIC and vice versa

  Receive chaining for receiving multiple files
- Intel mode is used mainly for file transfers between two Intellec® Development Systems. The files are duplicated exactly.
- Console commands support all standard features including:

  SEND data in Transparent or Non-transparent mode, with or without translation to EBCDIC

  RECEIVE in Transparent or Non-transparent mode, with or without translation to EBCDIC.

  Support for an IBM RJE console (such as HASP)
- Special utility programs are provided. STRZ strips extra binary zero's from the end of object files. CONSOL assigns system console input to an ISIS-II disk file.
- Can process commands interactively from the console or sequentially from an ISIS-II file under the SUBMIT facility for semi-automatic batch operation.
- Error detection in line transmission and error recovery by automatic retransmission.
- A special command such as DIAGNOSE, allows logging of all data activity on the line, during transmission and reception.
- When not used for communicating with the mainframe, the Intellec® Development System is available as a complete, stand-alone system.

BENEFITS

- Allows the customer to use an in-house mainframe or minicomputer for program source-preparation, editing, back-up and maintenance using inexpensive CRT's and multi-terminal access. The common files may be shared and others protected.
- Many programmers can use and share the high-performance devices normally available on large computer systems, e.g., fast printers to reduce listing time, the large capacity disks with their fast access time to store large program files.
- The source files can be downloaded using the Mainframe Link to an Intellec Development System (e.g., Model 240 or 245) for compilation, linking and locating.
- The compiled and/or linked object files may be transmitted back to the remote for storage. Updates and version numbers and dates can be tracked to ensure that the latest version is always used and back-up files are available. Binary object files can be later downloaded to an Intellec Development System for debugging using an ICE emulator.
- In short, provides a powerful and flexible tool combining the best of both micro and mainframe worlds, i.e., powerful CPU with large disk capacity, file sharing, multi-terminal access, etc., from a mainframe or minicomputer with Intel's versatile and compatible software support systems (including PL/M, PASCAL, FORTRAN, Assembler, R & L) and sophisticated debugging tools such as ICE emulators.

SPECIFICATIONS

Operating Environment

Required Hardware:

Intellec® Microcomputer Development System
Model 800
Models 220, 225, 230, 235, 240 or 245

64KB of Memory

One Diskette Drive
Single or Double Density

System Console
Intel CRT or non-Intel CRT

Recommended Hardware for Compilation:

Hard Disk (Models 240, 245, or Model 740 Upgrade)

Additional Hardware Required for Model 800
iSBC-955™, iSBC-534™

Required Software:

ISIS-II Diskette Operating System
Single or Double Density

Documentation Package

Mainframe Link User's Guide (121565-001)

Shipping Media

Flexible Diskettes
Single and Double Density

Remote System Requirements

- IBM 2780/3780 BISYNC protocol as supported by a majority of mainframes and minicomputers including: all IBM-360/370 Systems, PDP-11/70, VAX-11/780, Data General ECLIPSE.
- Users should purchase this standard software package from the remote system vendor and any additional required hardware such as a synchronous communications interface.
- The operating system at the remote must be configured (SYSGEN'ed) with correct options such as line address, 2780 or 3780, . . .

Communication Equipment Requirements

The Intellec Development System may be connected to the remote system using any one of the following methods:

- For short distances (up to 50 feet), use a synchronous modem eliminator (e.g., SPECTRON ME-81FS-2).
- For distances up to four miles, use short haul synchronous modems and telephone lines.
- For distances greater than four miles, use synchronous modems and telephone lines. The following BELL modems or their equivalents are recommended:

BELL 201C	2400 bits/second (half duplex, switched line)
BELL 208A	4800 bits/second (full duplex, leased line)
BELL 208B	4800 bits/second (half duplex, switched line)
BELL 209A	9600 bits/second (full duplex, leased line)

- Modems at either end must be compatible.

ORDERING INFORMATION

Part Number	Description
*MDS-384 Kit	Mainframe Link for Distributed Development

*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.

iNA955
iRMX™ NDS-II LINK

- Transfers files between iRMX™86-based systems and the NDS-II NRM
- Supports fast and reliable download into iRMX™86 target system
- Supports Intel's 86/310, 86/330A, 86/380 systems
- Configurable at nucleus level with iRMX™86 operating system
- Operates through Ethernet communications controller and cable connected to NDS-II
- Utilizes Ethernet technology with data transmission speeds at 10M bits per second

The iNA955/iRMX™ NDS-II LINK is a software package that allows an iRMX based system 86/310, 86/330A, or 86/380 system to be connected to an Intel Network Development System (NDS-II) network via an Ethernet coaxial cable or Intellink™ module.

iRMX system developers can use the Series II, III, IV and Model 800 for editing, compilation and debugging to develop, store, and manage software programs at the Network Resource manager. Using iNA955 these developers can download programs at Ethernet speeds from the Network Resource Manager into their target iRMX hosts for execution and system integration.

System developers can also use the iNA955 programmatic interface to develop their own application programs which run in the iRMX environment and interface with the NRM. This is a way for OEM developers to customize the operating environment to suit their own application.

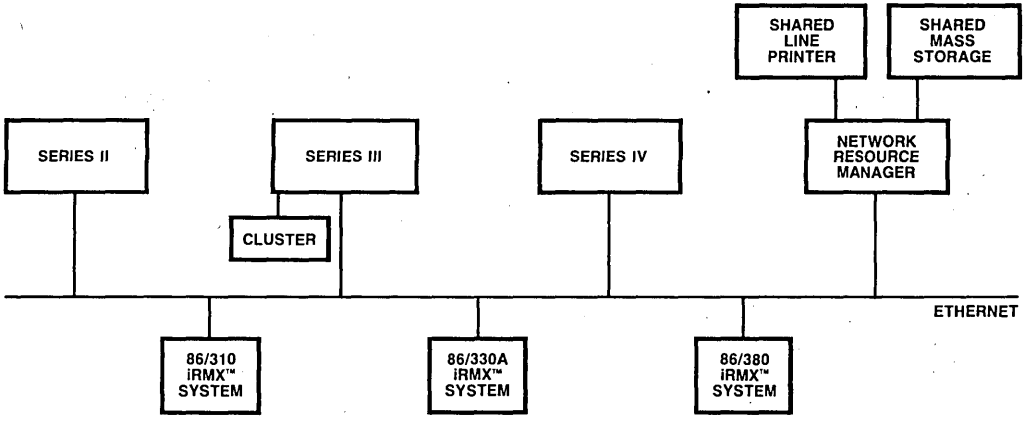


Figure 1. Example of NDS-II Configuration using the iRMX™ NDS-II LINK

MARCH 1984

Order Number 231034-001

NDS-II OVERVIEW

The NDS-II is a distributed processing local area network optimized for development of microcomputer-based products. It addresses the needs of both software and hardware engineers by providing the base environment for shared development tools plus the capacity for expansion.

An NDS-II network consists of an NRM which serves as the file server for a variety of Intel's development systems. These development systems include Series II, Series III, Series IV, and Model 800. By configuring iNA955 into an iRMX 86 system, an iRMX system can also be served by the NRM.

NDS-II's Network Resource Manager (NRM) manages all workstation requests for network resources. NRM tasks include service of workstation file requests, printer spooling, management of the distributed Hierarchical File System, the Distributed Job Control System and network maintenance functions such as user-name creation, file archival and system generation.

iNA955 provides a basic upload/download file transfer capability between an iRMX 86 system and the NRM. When used with an iSBC® 550 Ethernet controller, iNA955 allows users at iRMX 86 systems to move files between iRMX systems and the NRM, list directories at the NRM, delete or rename files at the NRM and copy files between two directories on the same NRM.

Access to files is accomplished using two interfaces:

A A CUSP interface which operates on the network file system in a manner similar to iRMX CUSPS which operate on local iRMX files under a full iRMX operating system.

B A programmatic interface which allows user programs running with a iRMX nucleus to access files at the NRM. These interfaces are similar to those present in UDI and EIOS.

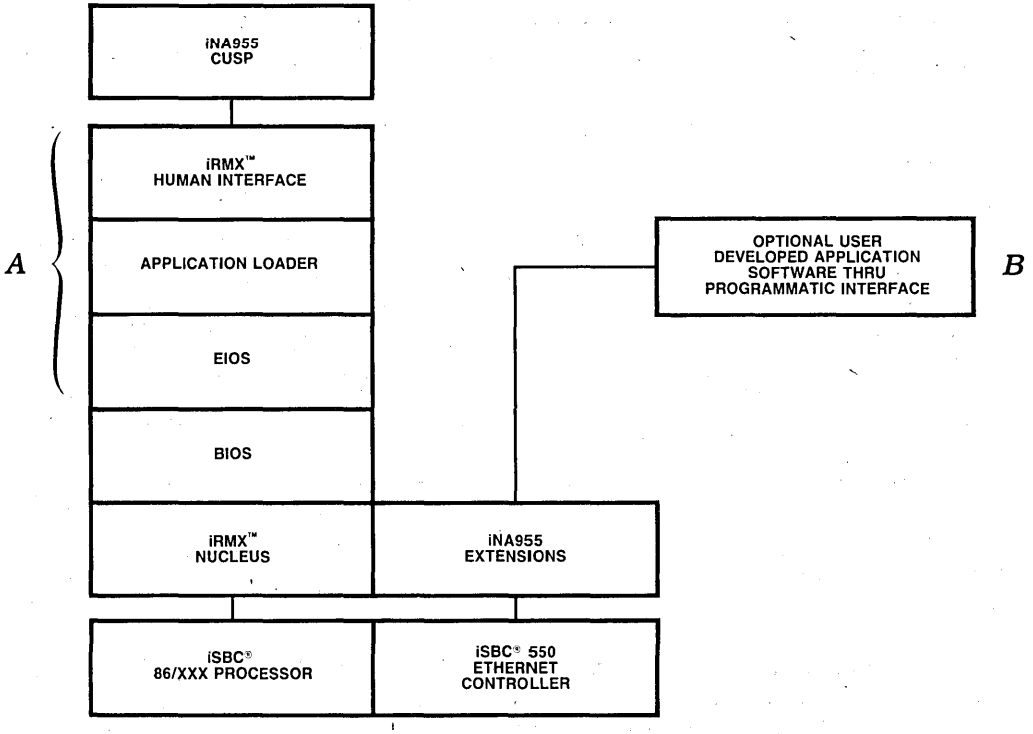


Figure 2. iNA955 Functional Diagram

231034-001

FUNCTIONAL DESCRIPTION

iNA955/iRMX NDS-II LINK consists of a program which runs on the system 86/3XX family of host computers. iNA955 executes under the iRMX 86 operating system and uses the local iRMX file system.

The iRMX-based host computers communicate with the NRM via iNA (Intel Network Architecture) which is based upon Ethernet communication protocols. These protocols are supplied by the iSBC550 board set which must be included in the iRMX host system since iNA955 uses the iSBC550 to communicate over Ethernet.

The following tables summarize the user commands and programmatic calls with their descriptions.

User Interface Commands	Function
NACCESS	examines/changes NRM file access rights
NCREATE	creates NRM directory
NDELETE	deletes NRM file
NDIR	examines NRM directory
NLOGOFF	logs off from NRM
NLOGON	logs on to NRM
NRCOPY	copies file from NRM to iRMX station
NNCOPY	copies NRM file to NRM file on the same NRM
RNCOPY	copies files from iRMX station to NRM
NRENAME	renames NRM file or directory

Programmatic Calls	Function
NQ$CHANGE$ACCESS	change access of file on the NRM
NQ$CREATE$DIR	create directory on the NRM
NQ$DELETE	delete file on the NRM
NQ$FILE$INFO	get information of file on the NRM
NQGETVIRTUAL$ROOT	get names of volumes at NRM accessible to user
NQ$LOGOFF	logoff user from the NRM
NQ$LOGON	logon user to the NRM
NQ$OPEN	open file at the NRM
NQ$READ	read contents of file at the NRM
NQ$READ$DIR$ENTRY$EXP	read expanded directory entry at the NRM
NQ$RENAME	rename file at the NRM
NQ$WRITE	write file to the NRM

Configuring iNA955

Like other iRMX systems iNA955 must be configured according to the system environment. To assist you in configuring your system, iNA955 comes with a configuration template. The file containing this template is contained on the release diskette. This template is designed to be self-explanatory.

The user has the option of integrating into his applications the iNA955 CUSPS. iNA955 CUSPS require the iRMX Human Interface to execute.

Physical Connections

The physical Ethernet connections can be made either through an "Intellink"™ module or through transceivers and the Ethernet cable. The Intellink module serves as an Ethernet local station concentrator. It allows workstations to be located up to 50 meters from the Intellink module and has 9 ports for connecting the NRM and workstations, and one port for connecting an Ethernet cable or other Intellink modules.

SPECIFICATIONS

Operating Environment

HARDWARE SUPPORTED

- System 86/310
- System 86/330A
- System 86/380

HARDWARE REQUIRED

- iSBC® 550 Ethernet Communication Controller Set

SOFTWARE REQUIRED

- iRMX™86 Operating System version 5.0
- NDS-II System software Release 2.5 or greater

Software Supplied

MEDIA

One 8 inch, single sided, double density iRMX™86 format diskette
One 5¼ inch, single sided, double density iRMX™86 format diskette

PROGRAMS

- iRMX/NDS-II LINK software linked into iRMX system library
- Examples of iRMX Integration Configuration utilities
- iSBC550 Diagnostics

DOCUMENTATION

- iNA955/iRMX NDS-II LINK Installation and User's Guide, Order Number 12256-001
- Complete NRM and Network operating manuals are included with the NDS-II systems
- iSBC550 Ethernet communications controller Hardware Reference Manual 121746.

ORDERING INFORMATION

Part Number	Description
iNA 955	iRMX/NDS-II Link
iSBC 550	Ethernet Communication Controller Set
iMDX 457	10 meter transceiver cable
iMDX 458	50 meter transceiver cable
iDCM 911-1	Intellink Module
iMDX 3015	Ethernet transceiver kit
iDMX 3016-1	25 meter Ethernet coaxial assembly
iMDX 3016-2	100 meter Ethernet coaxial assembly

Installation

On-site installation is included with the NDS-II Network Resource Manager. iNA955 is customer installable.

Technical articles

Smart link comes to the rescue of software-development managers

Resource-management hardware and software join existing development systems into an Ethernet-based network that eases software creation and control

by James P. Schwabe, *Intel Corp., Santa Clara, Calif.*

☐ A strong lifeline in a sea of complexity, the new NDS II network development system will help manage the writing of complex software for tomorrow's powerful microsystems. It builds on existing Intellec development systems and the specifications of the Ethernet protocol to create a local network for distributed software development.

Considerable intelligence is contained within the NDS II system, linking programmers' work stations and managing the interactive flow of software development that results. Communications control, via Ethernet or an even simpler alternative, is split between the central manager and the work stations.

At the heart of the system is the network resource manager, which both controls the net of work stations and lets the user configure it to suit the development task under way. The NRM will also manage a powerful system memory of Winchester-technology disk drives.

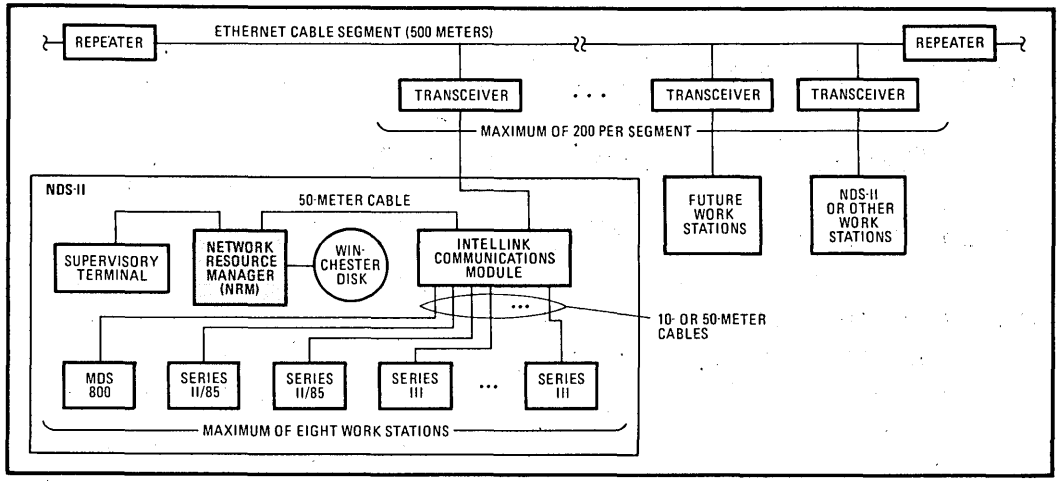
The manager itself is an example of the boons of well-thought-out and complex software, for it contains powerful system tools. Among these features are a hierarchical file structure that is also distributed and a file-protection setup that offers the maximum flexibility in access to files while guaranteeing their integrity.

Important program-management tools include a routine that oversees the rewriting of software during development and another that automates the generation of a complete program from the most current modules.

The NDS II is the second step in the evolution of Intel's network architecture, iLNA [*Electronics*, Aug. 25, 1981, p. 120]. It connects Intellec development systems together so they can share large-capacity Winchester disk drives and a line printer located at the NRM. It will also serve as the basis for a whole new line of modular development system tools such as remote emulators, logic analyzers, and more.

Both the NRM and each work station can be connected directly to the Ethernet coaxial cable by a transceiver or by the Intellink communications module (Fig. 1). By itself, the Intellink module provides nine ports for interconnection, creating a local network of nine systems (eight work stations and one NRM). To another controller, the Intellink represents a segment of Ethernet cable that has nine transceivers already in place and working.

For networks with a radius of 50 meters or less, Intellink is a simple, low-cost alternative to installing Ethernet cabling and transceivers. Any work station can



1. Developing net. The NDS II brings existing Intel development systems, or work stations, into an Ethernet. A new network resource manager and the Intellink communications manager make management of distributed software development possible.

be installed by simply plugging a 50-m transceiver cable directly into the Intellink—a 5-second operation.

For expansion beyond nine systems or to a distance greater than a 50-m radius, the Intellink provides a built-in port for connecting the local cluster to Ethernet cable by means of a transceiver. Connection to the Ethernet allows communication with other work stations, NDS II networks, or other Ethernet-compatible devices that use the iLNA network architecture.

No matter which physical setup is chosen, each work station has independent access to, and can be directly accessed from, the Ethernet and the NDS II network. Each has a unique work-station identifier, distinguishing it from every other terminal in the world and ensuring correct communication between stations on the various local networks.

For multiple-net environments, each network can have a unique network identifier to allow their coexistence on one Ethernet. In a single net, the network identifier is not used, but its assignment ensures an orderly progression to a multi-net environment.

All current Intellec development systems can be upgraded to NDS II work stations. An upgrade consists of a communication-controller board set, software, and either 10- or 50-m cables.

The communication controller, a two-board set that plugs into any Intel Multibus chassis, provides many of the data- and physical-link functions of the six-layer standard reference model for open-systems interconnection (Fig. 2). The data-link functions performed are framing, link management, and error detection. Physical-link functions include preamble generation and decoding and bit encoding and decoding.

One board contains a 5-megahertz 8086 microprocessor with local random-access and read-only memory and interval timers, as well as direct-memory-access channels for sending and receiving data at 10 megabits per second. The second board contains bit-serial send-and-receive logic, packet address-recognition logic, and error-detection logic. The boards ensure that bad packets resulting from a collision are ignored.

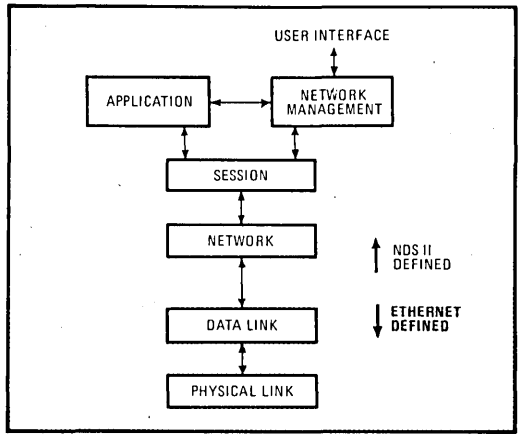
The NRM coordinates all the work stations' activities and manages file access to the shared disks. Initially, it will support one 8-inch 35-megabyte Winchester disk subsystem, as well as Intel cartridge-module disks. Multiple-disk support is in the wings, along with a larger 84-megabyte disk. It will be possible to attach six disks to one NRM, providing more than enough on-line shared storage for large program development and archiving. In addition, each work station can contain 2.5 megabytes of floppy-disk storage as a local resource.

Control contingent

The NRM (Fig. 3) comprises 13 Multibus slots, power supply, 8086-based system-processor board, input/output board based on the 8088 and 8089, 512-K-byte memory board with error checking and correction, two communication boards, and one 5¼-in. floppy-disk drive. The cabinet also has space for a cartridge-tape unit, expected to be delivered in mid-1982, which will give full intelligent archival backup for the Winchester disks housed in the attached cabinet.

To protect the integrity of the network, access to the NRM is restricted: a special supervisory terminal connected to the unit's serial port provides an interface with its commands and utilities. These facilities include system generation, intelligent archiving, and normal network maintenance such as the creation of any necessary user identifications.

The most important utility for system configuration is called Sysgen, an interactive routine designed to assist the supervisor, or project manager, in creating the NRM operating system. Sysgen makes it possible to create, modify, or delete system parameters, peripheral-devices configuration, and network configuration. It allows the project manager to tailor the network configuration on the fly in order to fit the changing needs of microprocessor development projects.



2. New layers. To the hardware layers of Ethernet, NDS II adds software layers that permit up to eight users to work together. The network layer need not be present if NDS II is not linked to the Ethernet, simplifying the operating system.

From the work-station perspective, the NRM is a remote file system. Each station functions as a stand-alone development system for all tasks not requiring NRM resources. When access to these resources is required, the user simply logs onto the network. The work station's resident operating system formats the appropriate file request, which the NRM processes interactively with other stations' demands.

The NRM operating system is multitasking, allowing a work station to access a file on the shared disk while other stations concurrently access other disk files. The interleaving of disk accesses, as well as the high-speed packet transmissions on the Ethernet, enables each work station to share equally the large file store—its being accessed by one user does not prevent other work stations from gaining access.

In an eight-station environment, the performance degradation due to network contention and the NRM operating system will be no more than 10%. This performance is one of the major reasons why distributed development systems provide a more cost-effective method for microprocessor development than time-shared systems; the former are much less susceptible to saturation under concurrent loading than are the latter.

Managing the work

To ensure efficient software development, high performance must be combined with tools to manage software complexity. For example, large software projects are often broken down into small tasks, and efficient file sharing becomes essential to project coordination. The shared-file system on NDS II is built on the RMX-86 volume-based hierarchy in which each user directory represents a node on a hierarchy of directories, commonly referred to as a hierarchial file system (Fig. 4).

Hierarchical file systems can contain a multitude of directories and data files. At the apex is the root volume, a conceptual file from which all directories emanate. The root volume contains all the volumes of the directories. Each volume can contain as many directories or files as available disk space will allow, and any directory may contain other directory files or data files. Each file (directory or data) can be traced through the hierarchy by its own path name. The NDS II hierarchical file system goes one step further by extending from the NRM to include the directories at the user's work station. When the user logs off the network, the only directories available are those on the work-station disks. When the user logs on, he or she gains access to the NRM system directories.

Thus each programmer has access to a common data base without the confusion of sifting through one massive directory. What's more, the structure keeps other users' files out of the way. In addition, it permits logically separate types of software within a user's directory. A programmer can create subdirectories to separate source files from object files, from backup files, and so on.

As a project's size increases, the number of directories and the complexity of path names in the system also increases. To simplify the task of accessing any particular directory, the user can assign a less cumbersome name—what amounts to a macroinstruction. Then, the user simply types in this macroname. Maximum flexibility is maintained, as each programmer can assign macronames to any directory.

An added benefit from macroname assignment is device transparency: the user concerns himself only with directories, irrespective of physical location. Physical devices are fixed in size and location, as opposed to directories, which can be adjusted to organize the contents in an optimal fashion.

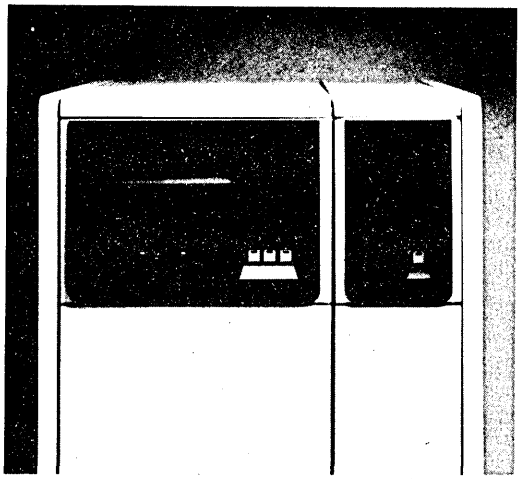
File protection

Before accessing the network, each user must be identified to the NRM through a log-on procedure. This setup establishes a unique user identification that is subsequently used to control access to files and directories in the hierarchical file system. Each directory and data file has specific "owner" and "world" access rights, which protect against accidental modification or deletion.

A file has three possible access rights for both the owner and the world: read, write, and delete. A directory also has three similar access rights for both the owner and the world: list a directory, add a directory entry, and delete a directory entry.

The access rights in file systems improve coordination during software development by allowing complete modules that have been tested and debugged in a user's work space to be converted into read status for the world. Then these modules can be integrated and tested with other independently developed software modules. Thus modules declared as read-only are guaranteed to be the most current debugged versions, and a common data base of completed modules is ensured.

Extended to multiple-project environments, the file system can provide logically separate work spaces for each project group. Specific directories can be set aside for complete modules for various projects. Each user can develop portions of the program in a private work space with guaranteed file protection and can use the public files (or directories) for integration and testing of the



3. Manager. The network resource manager (NRM) in the cabinet's left side governs access to the 35-megabyte Winchester drive on the right. Access to network-managing software is gained only through a supervisory terminal attached directly to the NRM.

module under development. Commonly used utilities and compilers can be accessible in a specific directory as public files (read-only for world access) to eliminate the necessity of redundant files at each work station. As a result, all programmers can proceed without fear of inadvertent modification of private files either by others or by themselves.

As well as managing communications between shared disks and work stations, the NRM maximizes the use of all network resources with distributed job control. DJC allows the user of any work station to export a batch job to the NRM for remote execution.

To accomplish this, the NRM classifies each work station into one of two groups—private and public. It keeps track of the public work stations and uses them to execute the queue of batch-type jobs. A user can declare any work station as public: available for use by the NRM for remote execution. Also, a programmer can send a job to a specific queue at the NRM by using the export command. The NRM executes the job on a public work station and return the results to the user directory.

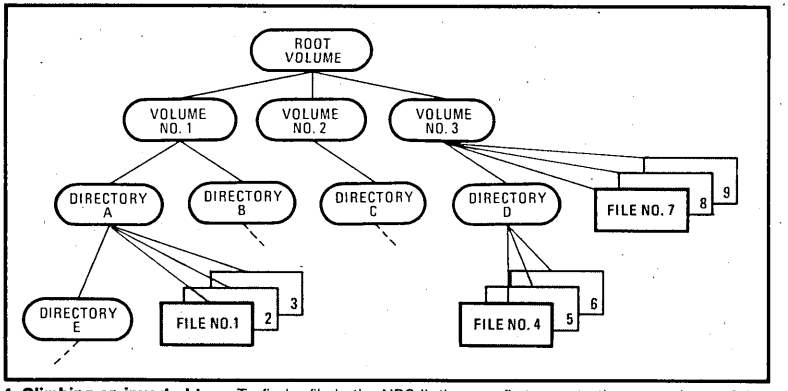
With DJC, the resources of the entire network can be shared to maximum advantage. A typical project involves program-module editing and debugging at Intellec series II or model 800 work stations, while a 8086-based Intellec series III unit can provide a host execution environment to compile completed modules quickly. DJC allows the user to export the compilation process to the high-performance series III work station, then return immediately to other tasks while the NRM oversees the compilation. At any time, the users can check on job status or queue status by typing a command from their work stations.

New work stations

Currently, Intellec development systems provide a single-task environment and therefore can be declared public to the NRM as users finish on-line work. Later this year, Intel will introduce high-performance work stations with foreground-background capability to allow a user to run a job in the foreground while making the background public so that jobs exported by other programmers can be executed through DJC. Foreground-background capability with DJC will effectively double the usefulness of the work station and substantially cut the cost of development time.

In-house benchmark tests indicate that the performance of each work station connected to the NRM is much improved. For example, a compilation executed with all file requests from the NRM hard disk is twice as fast as requesting files from the work station's floppy disk. Each station enjoys hard-disk performance during compilation, assembly, and any file manipulation—at a fraction of the cost of a dedicated disk system.

User's tools also speed program development, as well as make management easier. The most important programmer tools on NDS II are SVCS (software-version control system) and MAKE, an automatic software-generation tool. They provide a superset of the functions offered by the SVCS and MAKE found in the Unix programmers workbench.



4. Climbing an inverted tree. To find a file in the NDS II, the user first goes to the root volume of this hierarchical file structure. From that volume, he or she can go to the project volume assigned by the project manager and access other directories or files that have been declared accessible.

SVCS controls and documents changes to software products, handling both source and object files. It contains facilities for storing and retrieving different versions of a given program module, for controlling update privileges, and for recording who made what changes, when, and why.

Documentation of module status and of the levels, or versions, involved is the key factor determining the success of program development by group effort. Valu-

MAKEing it easy to revise programs

NDS-II's MAKE facility is a development tool for both generation and documentation of a software system. Suppose, for example, a software system called PGM.86 consists of three separate programs linked together, and, for simplicity, that each program consists of only one compiled source file, rather than a subsystem of multiple files. This relationship forms a dependency that would be graphed by the user as in the figure below.

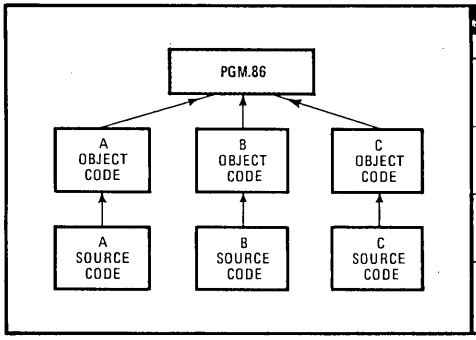
With the MAKE facility, a user can create an automated-generation procedure for the system PGM.86 that checks the currency of each subprogram. A MAKE command file that does so is illustrated in the accompanying table.

When the command file is invoked, the commands it contains are executed in top-down fashion. In step 1 of the table, the facility first checks if the PGM.86 is older (represented by the greater-than sign) than any of its dependent object-code modules. The facility checks and compares the date and time stamp of each module with that of PGM.86. Date and time stamps are updated automatically whenever a file is modified.

If any of the object modules are newer versions, then MAKE is instructed to link together the latest versions of the object modules to form the latest version of the software system. Before executing the link routine, the MAKE facility must first check to see if any of the object files are older than the related source files given in the dependency graph, as shown in steps 2, 3, and 4.

The MAKE facility goes through each step and executes the specified task only if the specified condition is true. Once the dependency graph is created, the MAKE facility can quickly and automatically generate the latest version of a software system under development even when source files change frequently.

The MAKE facility removes much of the guesswork surrounding software-system generation by ensuring the latest versions of source code is incorporated into the final software system. The dependency graph in its current form can also be printed by NDS II to document the software-system construction without having to keep an out-of-date sketch taped to the laboratory wall.



MAKE PROGRAM FOR PGM.86

Steps	Statements
1	IF PGM.86 > A.OBJ, B.OBJ, C.OBJ THEN RUN LINK86 A.OBJ, B.OBJ, C.OBJ TO PGM.86 END
2	IF A.OBJ > A.SRC THEN RUN PLM86 A.SRC END
3	IF B.OBJ > B.SRC THEN RUN PLM86 B.SRC END
4	IF C.OBJ > C.SRC THEN RUN ASM86 C.SRC END

able development time can be lost trying to work someone else's modified modules if documentation specifying what, where, when, and why changes were made is not available. In fact, as programs become more complicated, even the module writer may not exactly remember the history of the module.

Automatic documentation

SVCS provides a tool for automatic documentation of these facts. When a new module is created, it is set to level 1. All subsequent versions of the module are maintained with in a single file. Changes to the module are stored as "deltas" to the original. SVCS automatically records what changes were made and when they were made, and it requires the modifier to specify a reason for the change. The project manager may create a software checkpoint at any time by declaring the module as the next release level; subsequent deltas will then be applied to only this new release level.

Other capabilities in SVCS also increase project control. Restrictions may be placed on who is allowed to make changes to which modules and at which levels. An identification facility is also included, allowing the system to stamp modules containing object code with version information. From this information alone, a user can determine the level of source code used to generate the object module and thereby determine exactly which level of software is current and which level is being executed. To aid support groups in future maintenance of the program, any level of a software system can be regenerated from the original modules.

The second important program management tool on NDS-II is called MAKE, (see "MAKEing it easy to revise programs," above). When MAKE is invoked, a software system is automatically generated from the most current version of specific modules delineated by a dependency graph. MAKE ensures that the software generation is current and correct, while recompiling only program modules that need to be updated. To coincide with the concept of modular program development, any component of a MAKE could invoke another MAKE to generate a lower-level component such as a library. □

Microcomputer Software Development Tools

3

PROGRAM MANAGEMENT TOOLS

- **Increase Software Engineering Productivity**
- **Decrease Software Administration Overhead**
- **Allow Users to Control, Automate and Examine the Evolution of a Software Project**
- **Enhance the Capability of Networked (NDS-II) and Standalone Development Systems**
- **SVCS Simplifies Administration of Software Modules and Systems**
- **MAKE Automatically Generates New Releases of Software Systems**
- **Both Tools Easily Incorporated Into Existing Software Development Methodologies**

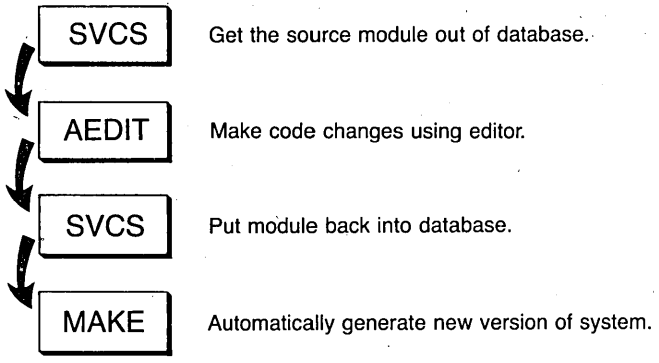
Intel's Program Management Tools (PMTs) provide the essential ingredients to manage large software development projects. PMTs decrease the time spent on tracking program changes and manually generating new systems, thereby giving engineers more time for software design, development, and testing.

PMTs consist of a "Software Version Control System" (SVCS), and an automated software generation facility (MAKE). Together these tools control, examine, and automate the management of a software system that may contain many versions consisting of numerous modules.

SVCS controls and documents software changes for all file types. SVCS handles storage and retrieval of different versions of a given module, controls update privileges, prevents different users from making changes independently, and requires all changes be thoroughly documented by recording who made what changes, when and why.

MAKE produces the specification of a "minimum-work" job required to generate a new system. This job (i.e. submit file) typically includes compiles and links of the latest versions of specified source and object modules. If a newer source module exists for any specified object module, MAKE will specify a compile of this module, replacing the older module in the completed program. Unnecessary links and compiles, however, are eliminated. MAKE does the minimum work required to ensure consistent, up-to-date software, thus saving many hours of compiles and links.

Incorporating PMTs into an existing project is easy. PMTs work with existing operating systems and software tools (editors, compilers, utilities) and require very little relearning. New users can quickly gain expertise in using PMTs by working through the examples contained in the PMT Tutorial Manual and Diskette, which are included with every PMT software package. Program Management Tools are ideal in a networked (NDS-II) environment, where multi-version software control is critical. PMTs are also extremely valuable on standalone systems (with Winchester disk) as well.



OPTIMAL CONTROL OF A SOFTWARE PROJECT.

MARCH 1984
ORDER NUMBER: 210567-003

SOFTWARE VERSION CONTROL SYSTEM (SVCS)

- **Simplifies Administration of Software Modules and Systems**
- **Maintains Change History Information on Every Module**
- **Prevents Users From Accidently Deleting System Software or Making Simultaneous Module Changes**
- **Offers an Effective Software Version Generation and Control Mechanism**

Intel's Software Version Control System (SVCS) is a utility that greatly simplifies software system housekeeping. SVCS automatically controls and documents software modules in a large project, eliminating costly manual administration by a project leader or librarian.

SVCS maintains a system database of software modules called units. Each unit is divided into four classes: Source, which contains the unit's source code; Object, which contains the unit's object code; History, which contains the unit's history file; and Composition, which can be arbitrarily used by the user.

Users interact with the database by using SVCS administrative and access commands. Project managers use administrative commands to create new system databases, add and delete database units, set unit access rights, and create and name new system variants. Programmers use SVCS access commands to check out and return database modules when making system changes. For every change made, SVCS records *what* changed, *who* changed it, *when* it was changed, and *why*.

SVCS variant generation and control enable project administrators to effectively create and identify new versions of software systems. Stable versions may be write protected and placed in the public domain, working versions may be identified and accessible only to programming personnel, and special versions may be created for customized releases. In addition, version control can minimize software archival, maintenance, and support administrative overhead.

AUTOMATED SOFTWARE GENERATION (MAKE)

- **Automatically Creates New Software Systems, Using the Latest Versions of Source Modules**
- **Automatically Determines Which Source Modules Need Recompiling**
- **Eliminates Unnecessary Compiles and Links**
- **Works Closely with SVCS for Generating Complete, Up-To-Date Systems**
- **Easily Adopted into Existing Development Methodologies**
- **Offers Many Powerful Macro Constructs**

MAKE is a utility that greatly simplifies the generation of software systems. MAKE produces a "minimum-work" submit file that can generate a complete, up-to-date system without any unnecessary compiles and links. MAKE can reduce system generation times from hours to minutes while concurrently minimizing administrative overhead.

210567-003

MAKE accepts a text input file that instructs it how to generate a new software system. The input file specifies all modules required to generate the new system and includes a description of system dependencies. It also specifies specific system operations, such as compiles, links, SVCS operations, line-printer spoolings, and other system commands. MAKE uses this input file in conjunction with the time and date stamps on each module to determine the optimum system generation procedure that eliminates all unnecessary compiles and links.

Typically a MAKE input file is created once at the start of a project. Very occasionally during the life of the project it may need modification. A powerful set of macros makes the creation and subsequent modification of a generation procedure an easy task. Overall, the management of the MAKE input file is negligible compared to maintaining numerous submit files for system generations.

The close relationship between SVCS and MAKE help simplify the overall job of software control at all levels. For example, the very latest version of a source module may not be stable enough to be included in a generation. A less functional, but more reliable version may exist. Since SVCS keeps unique versions distinct, an SVCS-module containing the more reliable version may be specified in the MAKE input file.

BENEFITS: SVCS AND MAKE

Intel's Program Management Tools eliminate common problems such as:

"We've modified module F00, which has introduced a new set of problems. Now we can't restore it back to the earlier version."

"Module F002 has been modified; no one seems to know who changed it, or why."

"We often have several programmers making changes to the same modules. Trying to avoid simultaneous changes is a lot of effort, and we waste time synthesizing two sets of changes into one module."

"To ensure that we release up-to-date, correct software, we periodically go into "release mode" for a few days. Everyone stops work completely while we find the latest versions, and then start the generation from the ground up. It literally takes days, when we could be making productive changes."

SVCS and MAKE together provide a service that fits easily into your existing design methodology, and solves administrative problems such as those described above.

SPECIFICATIONS

Networked, Multi-User Software Control

NDS-II with at least one Intellec Microcomputer Development System
iNDX, ISIS-III(N) System Software

Standalone Use

Intellec Series III with Model 750 Winchester Disk or Intellec Series IV
SVCS and MAKE will not operate on ISIS-II local floppies or Model 740 Hard Disks.
SVCS and MAKE may be exported from any workstation in an NDS-II configuration.

Documentation

"A User's Guide to Program Management Tools" (121958)

SOFTWARE SUPPORT

This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Part Number	Description
iMDX-332	Intel Program Management Tools

210567-003

PSCOPE
HIGH-LEVEL PROGRAM DEBUGGER

- **Source-Level Debugging for High Productivity**
- **Breakpoint, Single-Step and Execution Trace by Statement Numbers, Procedure Names and Labels**
- **High-Level Code Patching**
- **Compatible with Intel's I²ICE™ Integrated Instrumentation and In-Circuit Emulation System for Target System Debugging**
- **Native CPU Execution for iAPX 88 and 86 Architectures**
- **Supports PL/M, Pascal, and FORTRAN Program Debugging**

PSCOPE is an interactive, symbolic debugger for high-level language programs. It allows users to scrutinize program execution at the source level, using high-level statement numbers, procedure and variable names and labels. This is typically a more productive way of debugging high-level language (HLL) programs than at the machine level.

Source-level debugging means that traditional functions, such as setting breakpoints or tracing execution flow, are more powerful in PSCOPE. For example, tracing procedure entry (or exit) points conveys much more information than tracing machine instructions. Single-step execution is more powerful, using statements and procedures, as well.

The productivity improvement from debugging in a high-level language is analogous to programming in a high-level language, when compared to assembly-level programming and debugging.

PSCOPE users may define high-level code patches, which are "compiled" and patched into the user's program. Code patches may be stored on disk, so they may be later incorporated into the program source file.

PSCOPE is an integral part of the advanced I²ICE Integrated Instrumentation and In-Circuit Emulation System. This allows a smooth migration from *program* debugging to *target system* debugging.

PSCOPE's symbol capacity is virtually unlimited. Symbols are paged to disk when necessary.

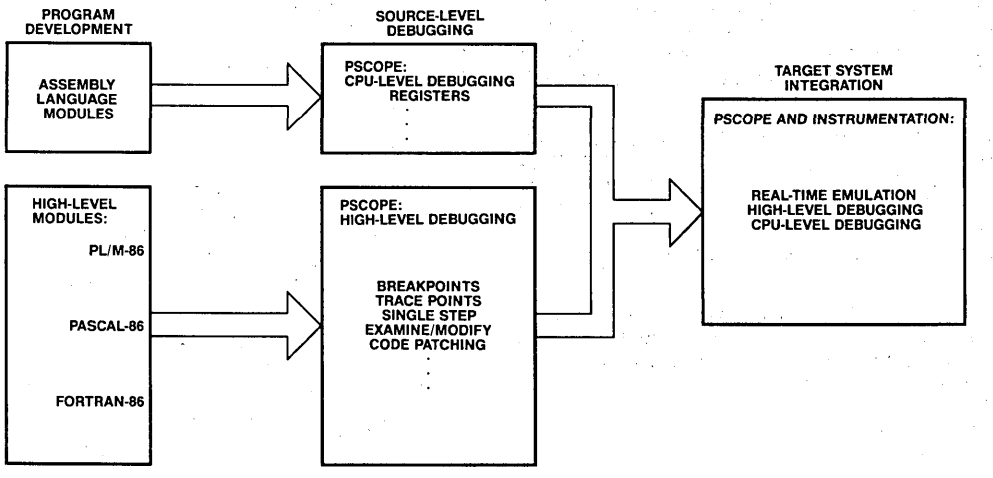


Figure 1. Debugging Methodology with PSCOPE

MAY 1983
ORDER NUMBER:210350-003

SAMPLE DEBUGGER SESSION

```
SERIES-III Pascal-86, V1.1


Source File: :F2:MAXMIN.PAS
Object File: :F2:MAXMIN.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F2:MAXMIN.PAS
   1    1  0  0         program calc(input,output);
   2    2  0  0         var a,b:integer;

   3    4  0  0          procedure sum(x,y:integer);
   4    5  1  0          var z:integer;
   5    6  1  0          begin
   5    7  1  1           z:=x*y;
   6    8  1  1           writeln('The sum is ',z);
   7    9  1  1          end;

   8   11  0  0           procedure difference(x,y:integer);
   9   12  1  0           var z:integer;
  10   13  1  0           begin
  10   14  1  1            z:=abs(x-y);
  11   15  1  1            writeln('The difference is ',z);
  12   16  1  1           end;

  13   18  0  0           procedure maxmin(x,y:integer);
  14   19  1  0           begin
  14   20  1  1            if x<y then writeln('The maximum is ',y,
                                                '   The minimum is ',x);
  16   21  1  1            if y<x then writeln('The maximum is ',x,
                                                '   The minimum is ',y);
  18   22  1  1            if x=y then writeln ('The two inputs are equivalent ');
  20   23  1  1           end;

  21   25  0  0         begin
  21   26  0  1           repeat (*forever*)
  21   27  0  2            write('Input two integers ');
  22   28  0  2            readln(a,b);

  23   30  0  2            sum(a,b);
  24   31  0  2            difference(a,b);
  25   32  0  2            maxmin(a,b);
  26   33  0  2           until 1<0
  27   34  0  2         end.
```

The program listing for the sample PSCOPE session illustrates the high-level nature of PSCOPE debugging. The program consists of the module CALC, the procedures SUM, DIFFERENCE, and MAXMIN, plus global and local variables. Users exercise and manipulate the program using these symbols. Code patches, stepping, tracing, etc. are all done on line numbers, procedures, labels, and symbolic names. To debug a program, just PSCOPE and a listing are required—no linkage maps, core dumps, locate maps, etc. are necessary. This is how high-level debugging relates to high-level programming.

BENEFITS

Shortened Development Cycle

The ability to define debugger procedures and make code patches is very useful. It actually allows users to *extend* the capability of the program under debug. After debug sessions, users typically make program changes or enhancements. This involves the use of an editor, compiler and linkage tools that create a "new" load module for debugging. Since PSCOPE allows these changes and enhancements to be made *in the debugger,* the number of Edit/Compile/Link iterations is lowered. More confidence can be placed on a program during debugging, because its capabilities have been more fully exercised.

Improved Debugging Productivity

PSCOPE provides users with the same conceptual interface to program debugging that was used in program design. This includes the high-level language constructs such as statements, procedures, labels and symbolic variables and data structures. Functions such as program trace and single-step execution are more meaningful with statements and procedures than machine instructions; therefore the improvement in debugging productivity is analogous to the programming productivity using high-level languages.

More Reliable Software

Debugger procedures may be used to automate the software testing process. The procedure may repeatedly generate test values, execute the program with the input values, and record the results. Running more comprehensive tests, plus being able to "batch" the tests, yields more reliable software.

Easy to Learn and Use

An extensive command language, which is similar to block-structured languages such as PL/M and Pascal, is very easy to use in an interactive debug session. The HELP facility makes learning to use PSCOPE extremely fast as well. The "Literally" facility and debugger procedures also allow users to extend and tailor the command language to suit individual needs.

Improved Software Management

The use of debugger procedures allows parts of a software system to be debugged independently. Procedures can be substituted for program stubs, allowing programmers to debug different pieces of the system separately. This results in improved project management.

SPECIFICATIONS

Supports Intel's standard 86/88 languages:

—PL/M 86/88
—Pascal 86/88
—FORTRAN 86/88

PSCOPE runs on an Intellec® Series III or Series IV Microcomputer Development System, either standalone or in an NDS-II network configuration. A 512K application memory space is recommended for most applications.

ORDERING INFORMATION

Order Code	Description
iMDX-333	PSCOPE Program Debugger (for Series III and Series IV)
III-951A	PSCOPE Program Debugger and I^2ICE Base Software for Series III with 8″ single density disk drive
III-951B	PSCOPE Program Debugger and I^2ICE Base Software for Series III with 8″ double density disk drive
III-951C	PSCOPE Program Debugger and I^2ICE Base Software for Series IV with 5¼″ double density disk drive

FEATURES

Unlimited Breakpoints

Breakpoints may be set on statement numbers, procedure names, or program labels. Any number of breakpoint registers may be defined.

High-Level Trace Points

Execution trace points are defined the same way as program breaks. Any number of trace points may be defined. A trace message is displayed when execution reaches a trace point.

Conditional Break and Trace

Any break or trace point may be defined to automatically call a *debugger procedure,* which will execute PSCOPE commands and/or evaluate predefined conditions. The operations will be performed, and the condition will determine if the break or trace will be done.

GO

The GO command initiates program execution from any starting point. A set of stopping points may be specified ("GO TIL"), and break/trace registers may be used ("GO USING").

Source-Level Stepping

A program may be executed, one high-level statement at a time, using the LSTEP command. Also, entire procedures may be treated as single statements during stepping (PSTEP); the procedures will be executed, but not stepped through.

Examine/Modify Data

PSCOPE allows users to symbolically examine (and change the value of) program variables and data structures. All PL/M and Pascal types are supported, including numerics, dynamic and stack variables, arrays, and fields within structures.

Virtual Symbol Table

All user-program symbols are stored in a virtual symbol table. This means symbols will be paged to disk, if necessary.

Help File

Many PSCOPE commands, facilities, and error messages have help information describing their use. The HELP command is used for learning the PSCOPE command language, for quick reference of command syntax, and for learning the cause of command errors.

Debugger Procedures

PSCOPE has the facility for defining procedures in its command language. This block-structured command language allows users to *extend* the capability of the program under debug. Like macros with parameters, these procedures may also be used for generating compound and conditional debugger commands.

Code Patching

Program patches may be written in the debugger command language to augment or replace current program statements. These high-level code patches are much closer to actual program changes than machine-level patches, and are easy to use.

Built-in Editor

A menu-driven, CRT-oriented editor is built into PSCOPE. This is used for creating and editing program patches, debugger procedures, and command lines. One key is used to invoke the editor to alter the last command entered, or any debugger definition (literally, trace register, patch, etc.) may be edited selectively.

Debugger Command Language

GO/LSTEP/PSTEP—For controlling program execution.

DEFINE/DISPLAY/MODIFY/REMOVE—For manipulating debugger objects (such as break registers, patches, and procedures), or program objects (variables and data structures).

CALL/RETURN—For executing debugger procedures.

WRITE/CI—For console input and output.

DO/END—For defining command blocks.

REPEAT/COUNT—For repetition of commands or blocks.

IF/THEN/ELSE—For conditional execution of commands or blocks.

INCLUDE/PUT/APPEND—For saving/restoring commands and definitions to and from disk.

```
-run :f1:pscope
SERIES-III PSCOPE-86, V1.0
*
*
*define literally d = 'define'
*d literally l = 'literally'
*d l br = 'brkreg'
*d l tr = 'trcreg'
*
*
*load :f1:maxmin.86
*dir
DIR of :CALC
PO_OUTPUT . . . . . TEXT (file)
PO_INPUT . . . . . TEXT (file)
B . . . . . integer
A . . . . . integer
SUM . . . . . procedure
  X . . . . . integer
  Y . . . . . integer
  Z . . . . . integer
DIFFERENCE . . . . . procedure
  X . . . . . integer
  Y . . . . . integer
  Z . . . . . integer
MAXMIN . . . . . procedure
  X . . . . . integer
  Y . . . . . integer
*
*
*pstep
[Step at :CALC#21]
*pstep
            INPUT TWO INTEGERS:
[Step at :CALC#22]
*pstep
                (input)   19  4
[Step at :CALC#23]
*pstep
                THE SUM IS   76
[Step at :CALC#24]
*pstep
          THE DIFFERENCE IS   15
[Step at :CALC#25]
*pstep
            THE MAXIMUM IS   19
            THE MINIMUM IS    4
[Step at :CALC#21]
*
*
*define patch #5 til #6 =  z=x+y
*
*go til #21
            INPUT TWO INTEGERS:
                (input)   19  4
                THE SUM IS   23
          THE DIFFERENCE IS   15
             THE MAXIMUM IS   19
             THE MINIMUM IS    4
[Break at #21]
*
*
*define proc PR1 = do
.*write 'the numbers and product are: ',a,b,a*b
.*write using ('@,>') 'break ? '
.*if CI == 'y' then return true
..*            else return false endif
.*end
*
*d br B3 = #21 call PR1
*go using b3
            INPUT TWO INTEGERS:
                (input)   23 24
                THE SUM IS   47
          THE DIFFERENCE IS    1
             THE MAXIMUM IS   24
             THE MINIMUM IS   24
the numbers and the product are:  +23 +24 +552
break ? y
[Break at #21]
*
*
*exit
PSCOPE terminated
-
```

The Literally facility allows users to abbreviate, redefine and extend the command language to suit individual needs.

Any PL/M-86, Pascal-86 or FORTRAN-86 program may be loaded. All symbolic names may be displayed, in total or by type. Symbols defined at debug time may be displayed as well. All program types are supported, including numerics, user-defined types, and records. The symbols' types are displayed by the DIR command as well.

Several flavors of stepping are offered. This example illustrates PSTEP, a line-by-line step where procedures are executed as a single step. This program contains five steps in the main body, with three being procedure calls.

There appears to be a bug in the program, as the sum is displayed incorrectly. Looking at the program, we notice that X and Y were multiplied instead of added, at line #5. A code patch is defined, and the program executes correctly.

This illustrates the facility where a *debug procedure* (PR1) is called when reaching a breakpoint at line #21. Here, some values are displayed, and a condition is evaluated (in this case, a query to the user). Had the condition been false, program execution would continue with no break. The high-level constructs in the command language make this a very powerful facility.

iRMX™ PSCOPE 86
HIGH-LEVEL PROGRAM DEBUGGER

- Debugs PL/M-86, Pascal-86, FORTRAN-86, and ASM86 programs
- Runs under the iRMX™86 operating system
- Sets breakpoints and traces program execution
- Single-steps through assembly language instructions, high-level-language statements, or procedures
- Permits creation of high-level program patches using high-level-language constructs
- Offers symbolic debugging capabilities:
  - Maintains type information about variables
  - Supports symbolic access to dynamic local variables
  - Maintains a virtual symbol table for program variables
  - Allows definition of user-defined debugging variables and procedures
  - Accesses memory locations and program variables using program-defined names
- Disassembles memory and provides a single-line assembler

PSCOPE is an interactive, symbolic debugger for high-level-language programs written in PL/M-86, Pascal-86, and FORTRAN-86 and for assembly language programs written in ASM86. PSCOPE runs under the iRMX™-86 operating system.

With PSCOPE, a user can load an application program, set breakpoints at symbolic or numeric addresses, trace program execution, and create patches. Other debugging aids include the ability to single-step a program through high-level-language statements, assembly language instructions, or procedures, to display and modify program variables, to inspect files, and to personalize the debugging environment.

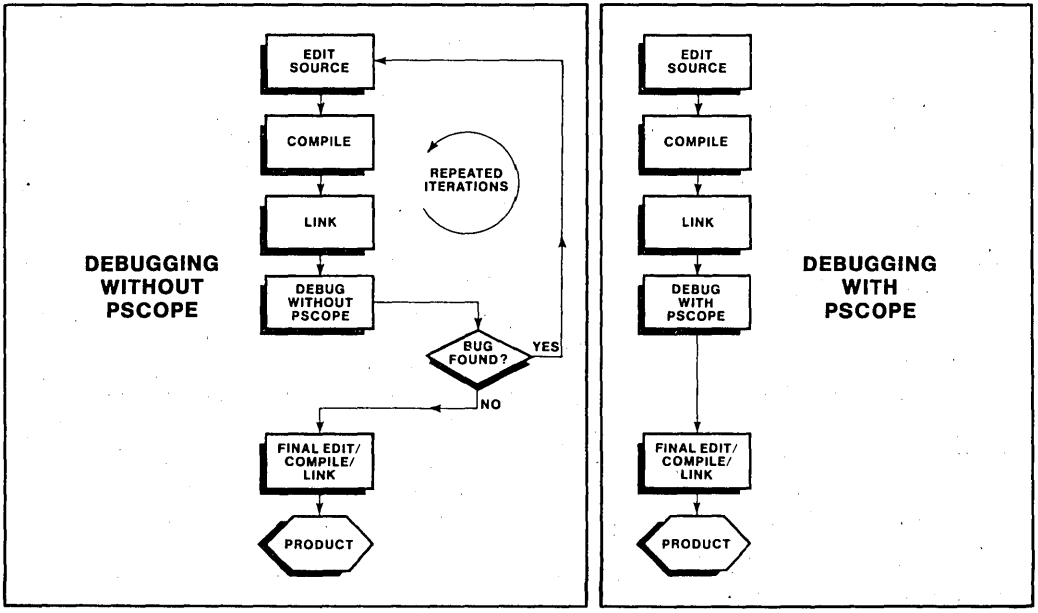


Figure 1 PSCOPE and the Program Development Process

APRIL, 1984
Order Number: 231058-001

PSCOPE OVERVIEW

PSCOPE allows testing and modifying a program without having to re-edit the source code, recompile, and relink. For example, users can refine algorithms with PSCOPE patches. The time saved not only decreases frustration of the development team, but it may also lead to faster product development.

PSCOPE can debug programs that employ system calls. The same microprocessor runs PSCOPE, an application program, and the iRMX-86 operating system. This ensures that the application program can make system calls to the iRMX-86 operating system.

MAJOR FEATURES

Symbolic Debugging

With symbolic debugging, a user can examine or modify a memory location by using its symbolic reference. A symbolic reference is a procedure name, variable name, line number, or program label that corresponds to a location in the user program's memory space. For example, to display the value of the program variable *linesend*, users need only type that variable's name. (Note that * is the PSCOPE prompt.)

```
*linesend
50
```

Notice that PSCOPE returns the variable value without the user having to indicate the variable's type. The capability to recognize a variable's type and scope is a special feature of PSCOPE's support of symbolics. Few other debuggers offer this feature. The feature provides three benefits:

- It eliminates the need for fully qualified symbolic references. Users no longer have to fully qualify the symbolic reference with a prefix that specifies the pathname to the symbol. (The pathname specifies the module and all procedures that enclose the symbol.)
- By recognizing the scope of a variable, it makes available information on dynamic (local) variables.
- It eliminates the need to remember how a variable was declared in a user's program. This can be a significant advantage. Consider an example. Suppose the user's program has an array of employee records called *emprec* that includes salary and other employee information. Using PL/M, the user might declare it like this:

```
DECLARE emprec (100) STRUCTURE
                     (name (20) BYTE,
                     ss (10) BYTE,
                     number INTEGER,
                     salary REAL);
```

With PSCOPE, to determine the salary of the nth employee, the user need only type:

```
emprec[n].salary
```

PSCOPE would then respond:

```
2.200E + 03
```

By contrast, other debuggers will not return the salary, unless the user supplies information on the structure of the array. For example, if the array had 36 bytes to each record and salaries were stored 32 bytes into the record, on other debuggers someone's request for the salary of the nth employee would have to mention these facts of array structure and indicate that salary information is of type REAL — in other words, the requester would have to type:

```
REAL emprec[n] + (n*36) + 32
```

Patch

A PSCOPE patch consists of PSCOPE commands that augment or replace current program instructions. Patching allows the user to refine a program's algorithm and test these refinements without having to edit the source code, recompile, and re-link.

For example, assume that statements #9 through #11 of a program set some program variables equal to some incorrect initial values. The following patch skips statements #9, #10, and #11. It replaces them with new variable assignment statements. Program execution resumes at statement #12. A GO command automatically makes use of any existing patches. (Note in this and other examples that PSCOPE returns the symbol "." before a prompt to indicate to the user that an END is needed to complete the command.)

```
*DEFINE PATCH :pager#9 TIL :pager#12=DO
.*leftmargin = 9
.*linesend = 45
.*double = true
.*END
*
```

A patching command can also insert new statements. The following command inserts the statement "i = 0" on the line preceding line 10.

```
*DEFINE PATCH :pager#10 = i = 0
```

The Virtual Symbol Table

PSCOPE maintains a virtual symbol table for program symbols; that is, the entire symbol table need not fit into memory at the same time. An option on the PSCOPE invocation line determines how much memory is reserved for the resident portion of the symbol table. The rest of the symbol table exists in a temporary disk file.

Breakpoints

Breakpoints permit the stopping of a program at specified locations. The user can then enter PSCOPE commands and perform such functions as constructing patches, examining or changing program variables and registers, and disassembling memory. Execution can then be resumed from where the program left off or from another point.

A breakpoint specification is the execution address just before which the user wants the program to stop. Users can represent execution addresses as symbolic addresses, absolute addresses, segment/offset pairs, or even high-level-language statement numbers.

For example, assume that the user has loaded a program with a module called *pager*. The following GO command sets a breakpoint just before statement #41.

```
*GO TIL :pager#41
```

If statement #41 begins at segment/offset 1BB7H:0302H, equivalent GO commands are

```
*GO TIL 1BB7H:0302H
*GO TIL 1BE72H
```

(The segment/offset 1BB7H:0302H is equivalent to the absolute address 1BE72H; to obtain an absolute address from a segment/offset pair, shift the left member of the pair by one place so that it becomes 1BB70H, and to it add the right member of the pair, the offset.)

Debugging Procedures

Debugging procedures are groups of PSCOPE commands that have been labelled. They can be stored on disk and recalled in later debugging sessions.

For example, here is the definition of a debugging procedure called *fixlnumber*. This debugging procedure sets the program variable *linenumber* to 50 if the variable is greater than 50 and increments *linenumber* by 1 if the variable is less than or equal to 50. To execute the debugging procedure, just type its name.

```
*DEFINE PROC fixlnumber = DO
.*IF linenumber > 50 THEN
..*linenumber = 50
..*ELSE
..*linenumber = linenumber + 1
..*ENDIF
.*END
*fixlnumber    /*Executing the debugging*/
*              /*procedure              */
```

One advantage of debugging procedures is the ability to execute often-used groups of PSCOPE commands without re-typing the commands. Another advantage is the ability to automate the software testing process. The procedure may repeatedly generate test values, execute the user program with new input values, and record the results. Debugging procedures aid the development of comprehensive "batched" tests.

Yet another advantage of debugging procedures is the ability to stub procedures in the user program (e.g., to construct dummy procedures). This ability improves software project management by allowing programmers to debug different pieces of the system separately.

Debugging Registers

Debugging registers are user-created, named software constructs that hold breakpoint and trace specifications. The two types of debugging registers are break registers and trace registers. A user can execute a program and specify one or more debugging registers. For example, here is the definition of a break register called *brk41* that specifies a breakpoint at statement #41 in the user program called *pager*:

```
*DEFINE BRKREG brk41 = :pager#41
```

To execute the user program and break just before statement #41, specify the break register *brk41* as part of the GO command:

```
*GO USING brk41
```

An advantage of using a debugging register is that the user doesn't have to reenter the debugging specification for another program

execution. He or she can store specifications in a debugging register, give them mnemonic names, write them to a disk file, and recall them in future debugging sessions.

Another advantage of using a debugging register is that it enables users to call a previously-defined debugging procedure when a specification in that register is met. For example, assume a user wants to break at two statements: statement #41 and statement #43, but the user only wants to break at statement #41 if the program variable *linenumber* is 50. First, the user defines a debugging procedure called *check_lnumber* that returns the value TRUE if *linenumber* is 50 and the value FALSE if *linenumber* is not 50. (Note that the symbol "==" indicates equivalence while "=" is the assignment symbol.)

```
*DEFINE PROC check_lnumber = do
.*IF linenumber == 50T THEN
..*RETURN TRUE        /*Break execution*/
..*ELSE
..*WRITE linenumber
..*RETURN FALSE  /*Continue execution*/
..*ENDIF
.*END
```

Second, the user defines a break register that specifies two breakpoints but qualifies the first with a call to the debugging procedure.

```
*DEFINE BRKREG brk41_43 = #41 CALL
 check_lnumber, #43
```

This definition directs PSCOPE to break at statement #41 if the procedure returns TRUE; if it returns FALSE, PSCOPE does not break at statement #41 but goes on to break at statement #43.

To run the program using this break register, the user specifies the GO command.

```
*GO USING brk41_43
```

Tracing

The PSCOPE trace feature displays a trace message on the host development system's console when the user program reaches a specified execution address. The trace message identifies the current execution point; no break occurs.

For example, assume again that the user has loaded the program called *pager* and defined a trace register that displays a trace message just before the user program executes statement #41.

```
*DEFINE TRCREG trc41 = :pager#41
*GO USING trc41
[At :pager#41]
[At :pager#41]
[At :pager#41]
[At :pager#41]
   .
   .
   .
```

The previous example assumes that statement #41 is a statement that exists within a loop and hence is repeatedly executed.

The PSCOPE Command Language

The syntax of PSCOPE commands resembles that of a high-level language. The PSCOPE command language is versatile and powerful while remaining easy to learn and use.

PSCOPE commands are often self-explanatory. For example, block-structured commands include DO-END, REPEAT-END, COUNT-END, and IF-THEN-ELSE. The command that begins the execution of the user program is GO.

The PSCOPE command language contains a number of functional categories.

- Emulation commands. These commands instruct PSCOPE to execute the user program. They consist of GO and the three stepping commands, ISTEP, LSTEP, and PSTEP.
- Debugging environment commands. These commands define patches, debugging procedures, debugging variables, LITERALLYs, break registers, and trace registers (the DEFINE command). A user can also delete these definitions (the REMOVE command).
- Block commands. These consist of DO-END, COUNT-END, REPEAT-END, and IF-THEN-ELSE constructs. They can be used alone or within debugging procedures and patches.
- String functions. These functions concatenate strings (the CONCAT function), return the string length (the STRLEN function), return a substring (the SUBSTR function), and accept console input (the CI function).

- Utility commands. These are general-purpose commands for use in a debugging environment. They consist of the following:

$	This pseudo-variable represents the current execution point.
ACTIVE	This function determines whether a specified dynamic variable is currently defined on the stack or not.
ASM	This command assembles or disassembles memory.
BASE	Sets or displays the current radix.
CALLSTACK	Displays the dynamic calling sequence stored on the stack.
DIR	Displays all objects of a specified type, such as debugging variables, program variables, line numbers, etc.
EDIT	Invokes the internal, menu-driven, text editor.
EVAL	Returns the symbolic name for memory locations. Also displays the result of an expression in binary, decimal, hexadecimal, and ASCII.
EXIT	Returns control to the host operating system.
HELP	Provides on-line help for selected topics and selected error messages.
INPUTMODE	Determines how a program handles input from the console.
NAMESCOPE	This pseudo-variable represents the current scope of a variable. Gives access to variables without need to use the fully qualified symbolic reference.
OFFSET$OF	This function returns the offset of a specified address (virtual or symbolic).
SELECTOR$OF	This function returns the selector of a specified address (virtual or symbolic).
WRITE	Writes variables and strings to the console's screen.

- File handling commands. These commands access disk files. The user can load program files to be debugged (the LOAD command), save patches, debugging procedures, debugging variables, LITERALLYs, and debugging registers in a disk file (the PUT and APPEND commands), read-in these definitions during later debugging sessions (the INCLUDE command), inspect a file during a debugging session (the VIEW command), and record a debugging session in a disk file for later analysis (the LIST and NOLIST commands).

- Register access commands. These commands provide access to the 8086/8088 and 8087 registers and flags.

  The REGS command displays the 8086/8088 registers and flags. Users can also inspect or change an individual register by specifying its menmonic. For example, CS represents the code segment register.

  The FLAG pseudo-variable represents the 8086/8088 flag word. The user can also inspect or change each flag separately as a Boolean variable. (For example, TFL represents the trap flag.)

  With the iSBC® 337 MULTIMODULE™ in place, users can inspect or change an 8087 (or 80287) register by specifying its mnemonic. For example, ST0 through ST7 represent the stack registers.

Debugging Variables

Debugging variables are user-created, named variables used with PSCOPE commands. They are distinct from program variables. For example, here is the definition of a debugging variable called *begin*. Its type is POINTER.

```
*DEFINE POINTER begin = $
```

$ is a PSCOPE pseudo-variable that represents the current execution point. Immediately after loading a program, $ contains the starting execution address. This is an address worth saving if users want to re-execute the program.

Literally Definitions

LITERALLY definitions are shorthand names for previously defined character strings. LITERALLY definitions save keystrokes or improve clarity. For example, here is the definition of a LITERALLY that saves keystrokes. This LITERALLY allows users to type DEF for DEFINE.

```
*DEFINE LITERALLY def = 'DEFINE'
```

LITERALLY definitions can also provide shorthand names for a group of statements. The following example shows a LITERALLY definition that allows substitution of "gs" for the two statements "$=begin" and "GO".

```
*DEFINE LITERALLY gs = '$=begin; GO'
```

Coprocessor Support

PSCOPE provides debugging support for programs that perform real arithmetic. A program performing real arithmetic under PSCOPE requires that the iRMX-86 microcomputer system contain the iSBC 337 MULTIMODULE board. (80286 users will need an 80287 numeric processor chip instead of the 8087 chip.)

The 8087 debugging support consists of the ability to assemble and disassemble 8087 instructions, to single step through 8087 instructions, and to recognize real data types (REAL, LONGREAL, and TEMPREAL) as well as the extended integer data type (EXTINT). With the iSBC 337 MULTIMODULE board present, users can also access 8087 registers and flags.

The Disassembler and Single-line Assembler

With the disassembler, memory can be displayed as 8086/8087 mnemonics. Users can also load memory with 8086/8087 instructions and specify those instructions in their mnemonic form. The next example shows how the ASM command displays the first assembly language instruction that makes up the high-level-language statement #41.

```
*ASM #41
1BB7:0368 A12C00    MOV    AX,LINESEND
```

To expand this example, suppose that the user finds a bug: the instruction should have set the program variable *linesend* to the value in the AX register rather than loading AX with the value of *linesend*. That is, the instruction should have specified a move from accumulator to memory rather than from memory to accumulator. The user can use the single-line assembler to change the instruction.

```
*ASM 1BB7H:0368H='MOV :PAGER.
 LINESEND,AX'
1BB7:0368 A32C00
```

The Editor and the View Command

PSCOPE contains an internal, menu-driven editor similar to Intel's AEDIT™ text editor. With this editor, users can create and modify patches, debugging procedures, and LITERALLY definitions.

An additional feature is the VIEW command; it permits examination of disk files without exiting PSCOPE. For example, while in a debugging session, a user can use the VIEW command to inspect the program's list file and ensure that the statement number for a breakpoint specification is correct.

On-line Help

PSCOPE provides an on-line help facility. In addition to obtaining help on topics from a help list, extended versions of PSCOPE error messages can be displayed. The extended error messages are marked with an asterisk enclosed in brackets: [*].

Stepping

PSCOPE commands allow users to single-step through assembly language instructions (the ISTEP command), high-level-language statements (the LSTEP command), and procedures (the PSTEP command). The ISTEP command displays the next instruction in disassembled form. The LSTEP and PSTEP commands display the statement number of the next high-level-language statement. For example, here is an LSTEP command and its result:

```
*LSTEP
[Step at :PAGER#42]
```

EXAMPLE OF A DEBUGGING SESSION

The example shown in Figures 2 and 3 illustrates some of the key capabilities of PSCOPE. The example program is written in Pascal-86. It was compiled and linked (with the BIND option). The resulting file consists of load-time-locatable code and is called *pager.86*.

The program reads an input file called TXTIN that is assumed to be in the default directory, formats the text, and writes it to a file called TXTOUT also in the default directory. The program inserts ten spaces at the beginning of each line, thus creating a left margin, inserts page breaks, writes a heading for each page, and numbers the pages. The program may also double-space lines. Note that line 41 in the example contains an error: "=linesend" should be ">=linesend".

This example is not intended to be a working program. Its purpose is to illustrate the use of some PSCOPE commands.

```
iRMX 86 Pascal-86, V3.0

Source File: PAGER.SRC
Object File: PAGER.OBJ
Controls Specified: XREF, CODE, DEBUG, OPTIMIZE, TYPE.

STMT   LINE   NESTING      SOURCE TEXT: PAGER.SRC
  1      1     0  0       PROGRAM   pager(INPUT,OUTPUT);
  2      2     0  0       CONST     blank=' ';
  3      3     0  0       VAR       textin,textout                    :TEXT;
  4      4     0  0                 ch                                :CHAR;
  5      5     0  0                 leftmargin,i,linenumber           :INTEGER;
  6      6     0  0                 linesend,pagenumber               :INTEGER;
  7      7     0  0                 double                            :BOOLEAN;
  8      8     0  0       PROCEDURE init(VAR leftmargin,linesend:INTEGER;
  8      9     1  0                      VAR double:BOOLEAN;
                                        VAR textin:TEXT);

  9     12     1  0       BEGIN (*init*)
  9     13     1  1       leftmargin:=10;
 10     14     1  1       linesend:=50;
 11     15     1  1       double:=TRUE;
 12     16     1  1       WRITELN('leftmargin =  ',leftmargin:2);
 13     17     1  1       WRITELN('lines/page =  ',linesend:2);
 14     18     1  1       WRITELN('double =  ',double);
 15     19     1  1       WRITELN
                          END  (*init*);

                          BEGIN  (*main*)
 16     24     0  1       RESET(textin,'txtin');
 17     25     0  1       REWRITE(textout,'txtout');
 18     26     0  1       pagenumber:=1;
 19     27     0  1       linenumber:=1;
 20     29     0  1       init(leftmargin,linesend,double,textin);
 21     31     0  1       WHILE EOF(textin)=FALSE DO
 22     32     0  1         BEGIN
 22     33     0  2         WRITELN(textout);
 23     34     0  2         WRITELN(textout);
 24     35     0  2         WRITELN(textout,'                                        ',
                                '                    Intel Corporation        ',pagenumber:4);
 25     37     0  2       WRITELN(textout);
```

Figure 2 Listing of the Program Used in the Debugging Session

```
26    39    0  2          REPEAT
26    40    0  3            FOR i:=leftmargin DOWNTO 1 DO
27    41    0  3                 WRITE(textout,blank);
28    42    0  3            WHILE EOLN(textin)=FALSE DO
29    43    0  3                 BEGIN
29    44    0  4                 READ(textin,ch);
30    45    0  4                 WRITE(textout,ch)
                                 END;
32    47    0  3            IF double = TRUE THEN
33    48    0  3                 BEGIN
33    49    0  4                 WRITELN(textout);
34    50    0  4                 WRITELN(textout);
35    51    0  4                 linenumber:=linenumber+2
                                 END
36    53    0  3            ELSE
37    54    0  3                 BEGIN
37    55    0  4                 WRITELN(textout);
38    56    0  4                 linenumber:=linenumber+1
                                 END;
40    58    0  3            READLN(textin)
41    59    0  2          UNTIL (linenumber=linesend) OR (EOF(textin)=TRUE);
42    61    0  2          PAGE(textout);
43    62    0  2          WRITELN('page =  ',pagenumber:4);
44    63    0  2          pagenumber:=pagenumber+1;
45    64    0  2          linenumber:=1
                          END;

47    66    0  1     WRITELN;
48    67    0  1     WRITELN('end of file on textin encountered')
                     END. (*main*)
```

Figure 2 Listing of the Program Used in the Debugging Session (continued)

```
(1)   *BASE
      DECIMAL

(2)   *LOAD pager.86

(3)   *DEFINE POINTER begin = $
      *DEFINE PROC again = DO
      .*$=begin
      .*NAMESCOPE=$
      .*END

(4)   *DIR DEBUG
      BEGIN . . pointer
      AGAIN . . proc
      *PUT pager.MAC DEBUG
```

Figure 3 Sample Debugging Session

```
(5)   *DEFINE PROC fixlnumber = DO
      .*IF linenumber > 50 THEN
      ..*WRITE 'old linenumber=   ',linenumber
      ..*linenumber = 50
      ..*WRITE 'new linenumber=   ',linenumber
      ..*RETURN TRUE                                    /*Break execution*/
      ..*ELSE RETURN FALSE                              /*Continue       */
      ..*ENDIF
      .*END
      *DEFINE BRKREG stat41 = #41 CALL fixlnumber

(6)   *GO USING stat41
      leftmargin = 10
      lines/page = 50
      double = T
      old linenumber= 51
      new linenumber= 50
      [Break at #41]

(7)   *GO
      page = 1
      old linenumber= 51
      new linenumber= 50
      [Break at #41]
      *GO
      page = 2
      old linenumber= 51
      new linenumber= 50
      [Break at #41]
      *GO
      page = 3

      end of file on textin encountered

      EXCEPTION: Program call to DQ$Exit
      [Stop at location 2178H:0030H]
      *EXIT
      PSCOPE terminated
```

COMMENTARY

(1) Checks to see that the default radix is decimal.

(2) Loads the user program.

(3) Defines a debugging variable called *begin* and a debugging procedure called *again*.

The debugging variable *begin* is of type POINTER and is set to the current execution point. At this point in the debugging session, $ is the beginning address of the user program.

The debugging procedure *again* sets $ and NAMESCOPE to *begin*, the initial values. If, after executing a program, a user executes this procedure, the user is ready to execute the program again.

Note that a LITERALLY definition could have been used here. For example, the following definition would allow the programmer to substitute "setup" for "$=begin" and "NAMESCOPE=$":

```
DEFINE LITERALLY setup = '$=begin; NAMESCOPE=$'
```

Figure 3 Sample Debugging Session (continued)

(4) Lists the currently defined debugging variables; then, saves them in a file called *pager.mac* on the current default directory.

(5) Defines a debugging procedure called *fixlnumber* and a break register called *stat41*.

The debugging procedure *fixlnumber* tests the program variable *linenumber* for a value greater than 50. If it is greater, the procedure sets *linenumber* to 50 and returns the value TRUE. Otherwise, the procedure returns the value FALSE.

The debugging register *stat41* defines a breakpoint just before statement #41. When the user program reaches this location, PSCOPE executes the debugging procedure *fixlnumber*. If the returned value is TRUE, the break occurs.

(6) Executes the user program with the break register *stat41*. The program writes *leftmargin*, *line/page*, and *double* to the screen. The debugging procedure writes old and new *linenumber* to the screen. The break occurs.

(7) Resumes execution of the user program. The program writes out the program variable *page* to the screen. The debugging procedure writes old and new *linenumber* to the screen. The break occurs.

Again resumes execution until the program encounters an end-of-file on TXTIN.

PSCOPE always traps a call to DQ$EXIT and stops program execution. This allows the user to continue debugging. Executing the procedure *again* at this time prepares PSCOPE to execute the program once more.

Figure 3 Sample Debugging Session

BENEFITS

As an interactive, symbolic, high-level language debugger, PSCOPE brings to debugging the same type of productivity enhancements that high-level-languages bring to writing software. PSCOPE's benefits are listed below:

- A shortened development cycle. Breakpoints, tracing, and patching decrease the number of edit/compile/link iterations.
- A standard command language. The PSCOPE command language is similar to that of Intel's in-circuit emulators.
- Increased software reliability. Debugging procedures can automate the software testing process.
- Improved project management. Software engineers can debug modules separately.
- A personalized debugging environment. The user can set up patches, LITERALLYs, debugging procedures, debugging registers, and debugging variables.

SPECIFICATIONS

8086/8088 Languages Supported

PL/M-86	Pascal-86
FORTRAN-86	ASM86

Documentation

PSCOPE-86 High-Level Program Debugger User's Guide, Order Number: 121790

Host System Requirements

An Intel System host microcomputer. The host system must have at least 110K bytes of application program memory available for iRMX PSCOPE 86. Additional memory may be required for the program being debugged.

- Intel System 86 microcomputer systems must run release 5 or later of the iRMX-86 operating system:

System 86/310	System 86/380
System 86/330A	

- Intel System 286 microcomputer systems must run release 6 or later of the iRMX-86 operating system. (Only release 6 or later of iRMX 86 supports an 8086 environment on 80286 microprocessor systems running in compatibility mode.)

User-Defined System Requirements

An 8086 or 80286-based user-configured iRMX-86 system that includes the following iRMX-86 subsystems:

Nucleus
Basic I/O system
Extended I/O system
Human Interface
UDI

ORDERING INFORMATION

Order Code	Description
iPSC 86 RMX	PSCOPE Program Debugger (to run under the iRMX-86 operating system, release 5 or greater)

NDS-II ELECTRONIC MAIL

- **Improves Project Coordination and Communication**
- **Minimizes "Phone Tag" and Excess Paperwork**
- **Users Can Send and Receive Text or Object Files**
- **MAIL Operates Either Interactively or in Command-Tail Format**
- **User, Group, and "Bulletin Board" Mailboxes Can Be Created**
- **Operates on any Workstation in the NDS-II Development Environment**

Electronic Mail enables users to send and receive messages and files between any nodes on the NDS-II network. In doing so, Electronic Mail improves the communication and coordination between members, reduces "phone tag" and paper generation, aids project configuration management by enabling simplified file transfers, and increases flexibility in workstation location.

The Mail system is governed by an Electronic Mail directory which contains user, group, and bulletin board mailboxes. Each NDS-II user has a mailbox which is only accessible to that user. Group mailboxes are acessible by a defined group of users, and bulletin board mailboxes are accessible by all users. Both group and bulletin board mailboxes can be easily created by any system users.

Users can send a message to any of the mailbox types listed above. Messages can consist of text generated when Mail is invoked, or a text or object file. Options available when sending mail include using a subject string to categorize a message, specifying a message expiration date and time, delaying message delivery until a specific date and time, marking the message URGENT, and maintaining a log of all messages sent.

Users can interactively read their mail and perform the following operations: print messages on their workstation console, delete messages from a mailbox, save messages in a file, forward messages to other users, and reply to message senders. In addition, users can request a mailbox summary which includes, for each message, the sender's name, date sent, subject, urgency, code type (text or object), and message number.

NDS-II Electronic Mail executes on all existing NDS-II workstations using either the iNDX or ISIS-III(N)/ISIS-III(C) operating systems.

TYPICAL MAIL USAGE
• DISTRIBUTE SUPERUSER MESSAGES
• CREATE AND SEND INTERNAL MEMOS
• COLLECT PROJECT MILESTONE DATA
• REPORT PROGRAM BUGS AND RECOMMEND SYSTEM CHANGES
• SEND SOURCE AND OBJECT FILES
• USE AS TELEPHONE MESSAGE CENTER

TYPICAL MAIL BENEFITS
• IMPROVE TEAM COMMUNICATION AND COORDINATION
• REDUCE PHONE TAG
• MINIMIZE PAPER GENERATION
• AID PROJECT CONFIGURATION MANAGEMENT
• INCREASE WORKSTATION LOCATION FLEXIBILITY
• OVERALL, BOOST DEVELOPMENT TEAM PRODUCTIVITY

NDS-II ELECTRONIC MAIL

ORDER NUMBER: 230916-002

OPERATING ENVIRONMENT

Required Hardware
NDS-II Environment with any 8- or 16-bit Microcomputer Development System Workstation

Required Software
iNDX or ISIS-III(N)ISIS-III(C) System Software

DOCUMENTATION
"NDS-II Electronic Mail User's Guide" (122146)

SOFTWARE SUPPORT
This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Product Code	Description
iMDX-337	NDS-II Electronic Mail

8086 SOFTWARE TOOLBOX

- **Collection of Tools That Speed Software Development**
- **MPL, a Standalone Macro Processor, is Ideal for Debugging Macros**
- **PSCAN reduces time spent doing software entry and editing**
- **SCRIPT and SPELL Assist Text Preparation**
- **OMC286 and E80287 Aid 80286 and 80287 Software Development**
- **Many Other Valuable 16-Bit Software Tools Are Included**
- **Runs on Series III and Series IV Microcomputer Development Systems**
- **Runs under iRMX™ Operating System**

The 8086 Software Toolbox is a collection of 16-bit software tools that can significantly improve programmer productivity. These tools are valuable for text formatting, editing, and preparation, software testing and performance analysis, 286/287 software development, and a multitude of other applications.

Text processing tools ease document formatting and preparation. PSCAN is a syntax-scanning editor for the PL/M language. It catches syntax errors in the editing stage and provides automatic formatting of PL/M code and more. SCRIPT is a text formatting program that uses commands embedded in text to do paging, centering, left and right margins, subscripts, etc. SPELL finds misspelled words in a text file and comes with a user expandable dictionary. COMP prepares two text or source files and displays their differences.

Test and performance analysis tools aid software testing and performance evaluation. PERF, a performance analysis tool for 8086 software, is ideal for isolating code "hot spots." PASSIF is a general-purpose assertion checking and reporting tool perfect for running test suites.

Software development for 286/287 components is assisted by two software tools: OMC286, an 8086 to 80286 object module convertor, and E80287, an 80287 emulator that runs on the 80286.

Additional tools are included that aid 16-bit software development efforts. All tools run on Series III and Series IV Microcomputer Development Systems.

TEXT EDITING AND PROCESSING
PSCAN
SCRIPT
MPL
SPELL
WSORT

PERFORMANCE MEASUREMENT & TESTING
PERF
GRAFIT
PASSIF

286/287 DEVELOPMENT
OMC286
E80287

MISCELLANEOUS TOOLS
COMP
FUNC
XREF
DC
HSORT
ESORT

8086 SOFTWARE TOOLBOX TOOLS

ORDER NUMBER: 230915-003

FUNCTIONAL DESCRIPTION

Text Editing and Processing

PSCAN—syntax scanning editor that supports all the functions of AEDIT-86 Release 1.0. Plus specialized functions for entering and editing PL/M source programs. PSCAN verifies correct code entry as you type, suppressing time consuming recompilations. In addition, PSCAN provides facilities to automatically format PL/M code, and can perform editor functions on statements, blocks or procedures.

SCRIPT—text formatting program that does paging, centering, left and right margins, justification, page headers and footers, underlines, boldface type, subscripts and superscripts, upper and lower case, and much more. Formatting commands are embedded in text.

MPL—standalone macro processor that processes the macro language used in 8086, 80286, 8089, and 8051 assemblers. Can be used interactively which makes it ideal for debugging macros. MPL can be used to preprocess any text file.

SPELL—finds misspelled words in a text file. Dictionary of correctly spelled words is user expandable.

WSORT—utility for creating the SPELL dictionary.

COMP—performs line-oriented text file comparison (shows source changes). Also understands 8086 object module formats for comparing 8086 object files.

Performance Measurement and Testing

PASSIF—general-purpose assertion checking, testing, and reporting tool. Helps automate the software testing process.

PERF—performance analysis tool for 8086 software. *Monitors references in the code segment; segment* monitored is user defined. Works with small or compact bound loadable modules. Ideal for isolating code "hot spots." Will only run on the Series III.

GRAFIT—graphing utility for use with PERF.

Miscellaneous Tools

OMC286—object module convetor that converts 8086 object modules into 80286 object modules.

E80287—an 80287 emulator that runs on the 80286.

FUNC—allows user to redefine the keys on a Series III keyboard and define function keys. Requires the iMDX 511 firmware.

XREF—produces cross-reference tables from translator list files. Cross-references all symbols—variables, labels, literallys, and quoted strings.

DC—floating point desk calculator program; allows variable definitions.

HSORT—in memory heap sort utility.

ESORT—*very flexible sort program.*

SPECIFICATIONS

Operating Environment

ISIS Operating System with RUN or INDX Operating System executing on Series III or Series IV Microcomputer Development Systems.

iRMX™86 Operating System executing in SYS X86/3XX environment.

Required Hardware

Series III or Series IV Microcomputer Development System

Required System Software

ISIS Operating System with RUN or iNDX Operating System

Documentation

"8086 Software Toolbox"
(122203)

Software Support

This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Product Code	Description
iMDX-364	8086 Software Toolbox

AEDIT TEXT EDITOR

- **AEDIT-80 operates on any Intellec® Series II, Model 800 or iPDS™ Development System**
- **Full Screen Editing**
- **Menu-Driven, Easy to Use**
- **Easy Handling of Large Blocks of Text**
- **Dual File Editing**
- **AEDIT-86 Operates on any Intellec® Series III, Series IV, or iRMX™ system.**
- **Powerful Macro Facility**
- **Split-Screen Windowing**
- **Designed for the Programmer and Technical Writer**

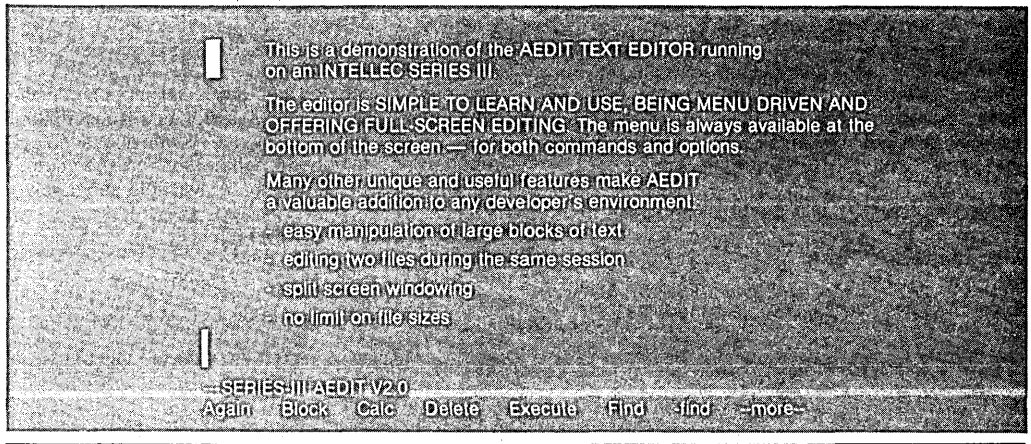
AEDIT is a full screen editor for use on any Intellec® Development or iRMX™ system. It is designed to be easy to learn and easy to use. At all times the user is guided by a menu which is used not only to select commands, but also to select options to commands. There is no need to constantly refer to or memorize detailed manuals.

AEDIT provides full screen editing capabilities and offers features to easily handle (move, copy, delete) large blocks of text. In addition to the basic editing abilities, AEDIT supports tagging positions in the text, string search and replace commands, and the option of automatic text indentation, spilling, and formatting. AEDIT is able to edit files of any length and optionally creates back-up copies of the file being edited.

With AEDIT, two files can be edited during one session. The user can easily switch between the files for quick reference, editing, or to transfer text from one file to the other. Using the windowing capabilities available with AEDIT-86, both of these files may be displayed simultaneously in a split-screen format.

AEDIT supports a powerful macro facility. AEDIT can create macros by simply keeping track of what a user is executing, "learning" the function the macro is to perform. The editor remembers the user's actions for later execution, and can store them in a file if requested. Alternatively, a user may enter a macro using AEDIT's macro language, or modify any existing macro interactively.

These and many other features combine to make AEDIT the editor of choice.



SEPTEMBER 1984
ORDER NUMBER:210996-003

MANUALS

AEDIT is supplied with a user manual documenting all the aspects of the editor, and a pocket reference card. The manual includes an introductory tutorial.

HOST SYSTEM

AEDIT-80 is an 8080/8085-based utility and can be run on any Intellec Development System, Series IIE, Series II, Model 800, or iPDS, as well as on ISIS Cluster workstations.

The higher-performance AEDIT-86 is an 8086-based utility that can be run on any Intellec Series IIIE, Series III, or Series IV Development system. Any Series IIE, Series II or Model 800 system can be upgraded to Series III functionality. AEDIT-86 is also available for the iRMX™ Operating Systems.

AEDIT can be configured to run with non-Intel terminals. Tested configurations are available for the following popular terminals:

ADDS Regent 200, Viewpoint 3A+
Beehive Mini-Bee
DEC VT52, VT100
Hazeltine 1510
Lear-Seigler ADM-3A
Zentec ZMS-35

Regent 200 is a trademark of ADDS
Mini-Bee is a trademark of Beehive
DEC designated Digital Equipment Corporation
ADM-3A is a trademark of Lear-Siegler

ORDERING INFORMATION

iMDX-335	AEDIT-80 Text Editor. Includes 8″ single and double density diskettes for Series IIE, Series II, or Model 800, and a 5¼″ diskette for iPDS.
iMDX-334	AEDIT-86 Text Editor Includes 8″ single and double density diskettes for Series III.

ISIS-II SOFTWARE TOOLBOX

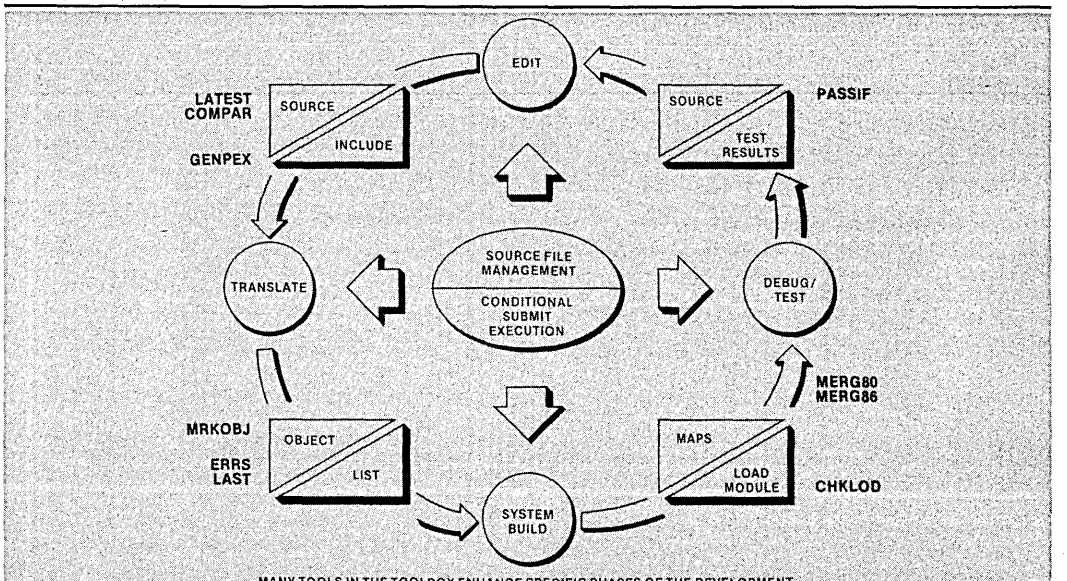
- **Significantly Improves Programmer Productivity**
- **Collection of Utilities that Speed Up Software Design**
- **Enhances Capabilities of ISIS-II Operating System**
- **Most Utilities will Operate on NDS-I Workstations, and Remote Hard Disks**
- **Provides Source File Management, Showing Source Changes, and Performing Version Control**
- **Provides Conditional Control and "Structured Programming" to Submit Files**
- **Runs on Model 800, Series II, and Series III Intellec® Development Systems**

The ISIS-II Software Toolbox is a collection of system utilities that perform a variety of "productivity-oriented" functions. There are two major subsets of Toolbox tools, in addition to numerous ad hoc utilities. These subsets provide Conditional Submit File Control and Source File Management.

The Conditional Submit File Control tools provide "structured programming" at the ISIS-II command level. Jumps, Calls, Returns, etc. are supported, as well as conditional command execution, based on assertions such as file existence, program errors, file matching, and string matching.

The Source Management Tools support version number tracking, and allow users to identify which versions of each source module were used to create a load module. There is also a tool which compares source files and reports all differences.

The tools outside of the two major subsets assist the programmer in some very specific development and debugging tasks. One tool manages all PUBLIC/EXTERNAL declarations in a system. Another merges the locate maps into a program listing, giving absolute symbolic debugging information. There's a directory sorter, a file compactor, and a tool to display just the last block of a file.



MANY TOOLS IN THE TOOLBOX ENHANCE SPECIFIC PHASES OF THE DEVELOPMENT CYCLE. OTHERS IMPROVE PRODUCTIVITY IN ALL PHASES.

FUNCTIONAL DESCRIPTION

Submit File Execution Control

IF/ELSE/ENDIF—conditional submit file execution based on file existence, program errors, pattern matching, plus several other conditions
GOTO—causes submit execution to resume at a specified label
RETURN—causes execution to return to the "submitter" (calling file)
EXIT—halts submit file execution
LOOP—forces execution to resume at the beginning of the submit file
RESCAN—allows submit execution to begin anywhere in file
NOTE—allows "progress report" notes to be placed in submit files
WAIT—displays a message, and waits for user input to continue or abort
STOPIF—halts submit file execution if specified listing contains errors

Source Management

XLATE2—submit-like tool with intelligent parameter substitution (for version control)
MRKOBJ—"marks" object modules with source version information
CHKLOD—lists source version data put in load modules by MRKOBJ
CLEAN—deletes all old versions off a specified disk
LATEST—displays latest version numbers of specified files

Operating System Functions

CONSOL—reassigns console input and console output as directed
DSORT*—alphabetically sorts floppy disk and hard disk directories
RELAB*—changes disk name to any other specified name

Program Development and Debugging

ERRS—fast display of program errors in PL/M 80, PL/M 86, and ASM 86 listings
MERG80—merges debug data from locate maps into PL/M 80 listings
MERG86—merges debug data from symbol maps into PL/M 86 and Pascal 86 listings
GENPEX—produces include file for PL/M external declarations (source level)
PASSIF—general purpose assertion checking, testing, and reporting tool

Text Processing

COMPAR—performs line-oriented text file comparison (shows source changes)
UPPER—changes all letters in an ASCII text file to uppercase
LOWER—changes all letters in an ASCII text file to lowercase
LAST—displays the last 512 bytes of a file
SORT—sophisticated line-oriented text file sorting tool

Disk Backup and File Processing

DCOPY—fast track-by-track diskette copying
HDBACK*—sophisticated hard disk to floppy disk backup program
PACK—compacts text files by removing strings of blanks
UNPACK—reconstitutes "packed" files

Disk Recovery

GANEF*—interactively reads and writes floppy or hard disk data blocks

Program Identification

WHICH—displays version number of Software Toolbox Programs

*These programs will not operate on the NDS-I remote hard disks.

ORDERING INFORMATION

Product Code	Description
MDS-363†	ISIS-II SOFTWARE TOOLBOX

Requires software license.

SUPPORT CATEGORY: Level C

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

intel®

INSITE™

USER'S PROGRAM LIBRARY

- **Programs for Intel Microprocessors**
- **Accepted Program Submittals Entitle You to a Free Membership or Free Program Package**
- **Worldwide Offices to Serve You**
- **Diskettes and Listings Available for Library Programs**
- **Program Library Catalog Offering Hundreds of Programs**
- **Updates of New Programs Sent During Subscription Period**

Insite, Intel's Software Index and Technology Exchange Library, is a varied collection of programs and routines that have been written by users of Intel microcomputers, single-board computers, and development systems. This expanding library of programs covers a broad range of software tools that includes monitors, conversion routines, peripheral drivers, translators, math packages, and even games. As a library member, you can acquire a copy of any program within the library on any of its available types of media. By taking advantage of the availability of existing library programs, numerous hours of coding and debugging time can be saved and routine or redundant programming operations can be eliminated. The Insite Program Library also serves as a learning tool for individuals unfamiliar with assembly or high-level languages associated with Intel's family of microcomputers.

Membership. Membership in Insite is available on an annual basis. Intel customers may become members through an accepted program contribution or paid membership fee.

Program Submittals. The Insite Library is built on program submittals contributed by users. Customers are encouraged to submit their programs. For each accepted program, submittors will receive a choice of three free programs, or free membership with Insite for one year. (Forms and submittal requirements are attached.)

Program Library Service. DISKETTES OR SOURCE LISTINGS are available for every program in Insite. Diskettes are available on single or double density 8″ or iPDS 5¼″. Membership is required to purchase programs.

Insite™ Program Library Catalog. Each member will be sent the Program Library Catalog consisting of an abstract for each program indicating the function of the routine, required hardware and software, and memory requirements.

Insite members will be updated with abstracts of new programs submitted to the Library during the subscription period. For catalog and yearly subscription fee please refer to the Intel OEM Price List or contact the nearest Insite or Intel Sales Office.

 ORDER NUMBER: 121707-003

SUBMITTAL REQUIREMENTS

Programs submitted for Insite review must follow the guidelines listed below:

Programs must be written in a language capable of compilation and assembly by the currently-supported version of an Intel standard compiler/assembler.

A well-documented source code furnished on an ISIS, INDEX and CP/M*-formatted 8" diskette, or PDS 5¼" diskette.

A source listing of the program must be included. This must be the output listing of a compilation or an assembly. No consideration will be given to incomplete programs or duplications of programs already in the Library.

A link and locate listing.

A demonstration program which assures the validity of the contributed program must be included. This must show the accurate operation of the program.

A complete submittal form.

Licensed software or copyrighted material must be accompanied by a written release from the appropriate, authorized person.

*CP/M is a registered trademark of Digital Research, Inc.

APPLICATION NOTE

AP-162

December 1983

PMT Tutorial

BRIAN VALENTINE/JOHN JARVE
DSSO APPLICATIONS ENGINEERING

Order Number: 230879-001

INTRODUCTION

Intel's Program Management Tools (PMT's) provide the essential ingredients for managing software development projects. Currently two productivity tools comprise the PMT's: SVCS, a Software Version Control System and MAKE, an automated software generation tool. Together they control, examine, and automate the management of a software development project, greatly decreasing the time spent on tracking program changes and the generation of new systems.

Intel's Software Version Control System controls and documents software changes for both source and object files. SVCS handles storage and retrieval of different versions of a given module, controls update privileges, prevents different users from making changes independently, and requires all changes be throughly documented by recording who made what changes, when and why.

MAKE produces the specification of a 'minimum-work' job required to generate a new system. This job (i.e. submit file) typically includes compiles and links of the latest versions of specified source and object modules. If a newer source module exists for any specified object module, MAKE will specify a compile of this module, replacing the older module in the completed program. Unnecessary links and compiles, however, are eliminated. MAKE does the minimum work required to ensure consistent, up-to-date software, thus saving many hours of compiles and links.

This tutorial covers the operation and features of Intel's Program Management Tools. By carefully working step-by-step through the examples contained herein, the user should develop the requisite skills to fully exploit the many advantages PMT's provide. We strongly suggest the user work through all examples. Each example is carefully constructed to expose the new user to a wide variety of program features and use methodologies. A tutorial diskette, which is included in the PMT Software package, supplements this manual and greatly facilitates the learning process.

The user should completely read the User's Guide to Program Management Tools before using this tutorial, and be familiar programming in the ISIS environment. In addition, he or she should be using a Series III workstation operating under the ISIS-III operating system. Series II and IV users would need to adapt the command syntax in this tutorial for correct operation.

EXAMPLE OVERVIEW

This tutorial describes the design of a software system using the unique features of Intel's Program Management Tools. The example system, REMOTE, enables an iPDS™ development workstation to communicate with an NDS-II via an ISIS Cluster board. In this configuration the iPDS essentially functions as a dumb terminal. Two special commands, SEND and RECV, transfer files between the iPDS local storage and the NDS-II remote file system. A full discussion of REMOTE, including program listings, is covered in Appendix A.

The REMOTE program consists of five separate modules whose relationship is shown in Figure 1. The program is generated using the SUBMIT file in Figure 2. This tutorial shows how to design an SVCS database for REMOTE, and gradually alter the submit file into a MAKE file. In addition, variants of REMOTE are created to enable program execution on a Series II and also run under the CP/M operating system.

The tutorial diskette contains the many files described within this manual and simplifies the execution of some examples. Before using the diskette, the user's system must by configured with the following assignments:

```
:F0: - Directory containing ISIS
system files.

:F1: - Directory containing
tutorial source, object, executable
and CSD files. (All files supplied on
tutorial disk.)

:F2: - Directory containing 8 bit
compilers, linker, locater, MAKE and
SVCS.

:F3: - Directory containing 8 bit
library files, such as PLM80.LIB and
SYSTEM.LIB.

:F4: - Directory containing SVCS
database files created during
tutorial.

:F5: - Directory containing 16 bit
software, such as 16 bit SVCS and
MAKE.
```

In addition, the tutorial disk files COMMON.LIT and ISIS.EXT **must** be copied to the 8 bit system library directory (:F3:). After the above changes have been made, the tutorial files can be utilized without further modification.

PMT TUTORIAL

CONTENTS

PAGE

INTRODUCTION 1

EXAMPLE OVERVIEW 1

DATABASE IMPLEMENTATION USING SVCS 2

A. Database Design 2

B. Database Generation 3

USING MAKE 3

A. Generating the MAKE File 3

B. Adding SVCS to the MAKE File 4

CREATING VARIANTS USING SVCS .. 4

DATABASE OVERHEADS 7

A. Initial Set-Up 7

B. Adding Variants 7

C. Total Overhead 7

USING DEFAULTS TO SIMPLIFY OPERATIONS 7

OPERATION ON A STANDALONE SYSTEM 8

A. System Differences 8

B. MAKE File Changes 8

FIGURES 9

APPENDIX A A-1

USING SVCS

Database Design

SVCS manipulates UNITS which can have up to four parts or CLASSES. The allowable CLASS categories are:

- SOURCE, which contains the UNIT's source code;
- OBJECT, which contains the UNIT's object code;
- HISTORY, which contains the UNIT's history file;
- COMPOSITION, which can be used arbitrarily by the user.

Each UNIT is given a unique name and may utilize any or all of the above CLASSES.

Referring to Figure 1, the five main modules of our example program are REMOTE, RMSEND, RMRECV, FILEIO, and SERIAL. For each of these modules we will define an SVCS UNIT having the same name. In addition, for each UNIT we will create a SOURCE CLASS, which will contain the UNIT's PLM source code, a HISTORY CLASS, which will document changes to the source code, and an OBJECT CLASS, which will contain the UNIT's compiled source. The database is depicted in Table 1.

Table 1. Preliminary Database

UNIT NAME	SOURCE	HISTORY	OBJECT	COMPOSITION
REMOTE	REMOTE.PLM	As requested	REMOTE.OBJ	?
RMSEND	RMSEND.PLM	As requested	RMSEND.OBJ	?
RMRECV	RMRECV.PLM	As requested	RMRECV.OBJ	?
FILEIO	FILEIO.PLM	As requested	FILEIO.OBJ	?
SERIAL	SERIAL.PLM	As requested	SERIAL.OBJ	?

This simple database, however, does not contain all the subsystems required to generate the executable object code REMOTE. INCLUDE files, for example, are not included. Figure 3 shows all of the modules required to generate REMOTE. Additional units must be defined for the executable object code and all include files.

The executable code, REMOTE, is placed in new unit called EXEC. REMOTE will reside in EXEC's OBJECT class. In the SOURCE class we'll place REMOTE.MKE - the MAKE file (which we will create soon) that generates REMOTE. Finally, in the COMPOSITION class we will place the user's manual USER.MAN.

Include files are added to our database as new units and also as composition classes. The include files COMMON.LIT and ISIS.EXT, used by nearly all source files, are stored in two new units: COMMON and ISIS. FILEIO.EXT and SERIAL.EXT, however, are added to the database via the composition class of the units they're most closely associated with: FILEIO and SERIAL.

Two ISIS programs, SEND and RECV, are required for complete operation of the REMOTE program. These programs are added to the database as two new units called SEND and RECV. These units will store the source, object, and executable files for the two programs.

The database is now complete and shown in Table 2.

Table 2. Complete Database

UNIT NAME	SOURCE	HISTORY	OBJECT	COMPOSITION
REMOTE	REMOTE.PLM	As requested	REMOTE.OBJ	Not used
RMSEND	RMSEND.PLM	As requested	RMSEND.OBJ	Not used
RMRECV	RMRECV.PLM	As requested	RMRECV.OBJ	Not used
FILEIO	FILEIO.PLM	As requested	FILEIO.OBJ	FILEIO.EXT
SERIAL	SERIAL.PLM	As requested	SERIAL.OBJ	SERIAL.EXT
EXEC	REMOTE.MKE	As requested	REMOTE	USER.MAN
SEND	SEND.PLM	As requested	SEND.OBJ	SEND
RECV	RECV.PLM	As requested	RECV.OBJ	RECV
COMMON	COMMON.LIT	As requested	Not Used	Not Used
ISIS	ISIS.EXT	As requested	Not Used	Not Used

Every element in the database represents one portion of the REMOTE system diagram shown in Figure 3.

230879-001

Database Generation

Having designed the database it is a mechanical process to generate and initialize it. The steps required to do so are shown in Figure 4.

Referring to Figure 4, the first SVCS command‡:

```
 RUN :F5:SVCS.
ADMIN   :F4:REMOTE.DB   CREATE
```

creates the database master file. This file contains all the information relating to variants, unit names, default accesses and file protection. The name "REMOTE.DB" is the name of the database and will be used in all SVCS command references to the database.

The second command:

```
 ACCESS :F4:REMOTE.DB WR1 WW1
```

sets the access rights for the database. This must be done if multiple users access the database. SVCS propagates these access rights to all of its database files.

The next 28 commands create and fill the database units. Note how the SOURCE class of a unit may be filled when created by including the FROM option‡:

```
 RUN :F5:SVCS. ADMIN :F4:REMOTE.DB ADD
(UNIT = REMOTE FROM & :F1:REMOTE.PLM)
```

Also note how you must GET a COMPOSITION class, with write option, before you can PUT to it.

You may either submit MAKEDB.CSD (which is on the tutorial disk) or enter the commands one-by-one to generate this database. We suggest you enter the first few commands to become familiar with the SVCS command formats. Once comfortable with the commands, you can submit MAKEDB.CSD to finish the database construction. The completed database can be viewed in a structured printout generated by the following command‡:

```
 RUN :F5:SVCS.
ADMIN   :F4:REMOTE.DB   PRINT
```

USING MAKE

Generating the MAKE File

The submit file REMOTE.CSD, shown in Figure 2, will be used as the basis for generating the MAKE file. Ensure you understand the operation of REMOTE.CSD before continuing.

Not shown in the submit file are source file dependencies on include files. These dependencies are hidden within the source files, and are displayed in Figure 3. Changing an include file is equivalent to changing a source file. For example, a change to COMMON.LIT modifies nearly all the source modules. MAKE and SVCS will be used to monitor and control include files too.

The files fall logically into three groups of executable files:

REMOTE – an executable file that runs on the remote system;
SEND – an executable file that runs on the ISIS-Cluster board to SEND a file to the Network file system;
RECV – an executable file that runs on the ISIS-Cluster board to RECEIVE a file from Network file system.

These groupings will be used within the MAKE file.

Figure 5 shows the first pass at creating a MAKE file. (Also refer to tutorial disk file REMMKE.EXM.) This MAKE file contains the least complicated MAKE constructs to keep the REMOTE project up to date.

Referring to Figure 5, the first MAKE command:

```
 $IF ALL > :f1:remote, send, recv THEN
 $END
```

is a trick used to fool MAKE into generating a dependency tree for three standalone or separate executable files. The next five commands test the time/date stamping of the five object files as compared to the source and associated include files. If the object file is older than the source or include files, then the source must be recompiled.

The last three MAKE commands test the age of the executable files REMOTE, SEND and RECV against the object modules that are linked to form these files. If any of the executable files are out of date, MAKE will add the appropriate task lines to the generated submit file to make them current.

Figure 6 shows the second pass in creating a MAKE file. (Also refer to tutorial disk file RMMKE1.EXM.) Note the macro definition:

```
 $SET work_device to ':F1:'
```

This definition substitutes the text ":F1:" for % work_device in all MAKE file commands. Macros enable the user to easily update MAKE files with future file changes.

‡Command should be typed on one line.

The SET macro command may also specify a list of items which are used in a similar fashion with an iteration command. For example, the MAKE iteration command:

```
$FOR i IN %remote_files
```

will execute the FOR loop for each item defined in the SET macro of remote_files. The above iteration command converts the five MAKE commands in Figure 5 to one command in Figure 6.

Adding SVCS Constructs To The MAKE File

Figure 7 shows the final version of the MAKE file. (Also refer to tutorial disk file REMOTE.MKE.) SVCS has now been incorporated into this version, as well as the special macros %ALL, %TARGET, and %DEPEND. All code files (source and object) now refer to units within the REMOTE.DB SVCS database. For example, the following command retrieves the file :F1:SERIAL.EXT from the database:

```
 %get (serial,,cp) to
%"work_device"serial.ext
```

In addition, the special macros %ALL, %TARGET, and %DEPEND are used to simplify MAKE file coding.

We now add the final version MAKE file to the database with the command:

```
RUN :F5:SVCS. GET :F4:REMOTE.DB (EXEC)&
TO :BB: WRITE

RUN :F5:SVCS. PUT :F4:REMOTE.DB (EXEC)&
FROM :F1: REMOTE.MKE
```

After creating REMOTE.MKE, we run MAKE with the GENALL option to create a SUBMIT file that generates everything:

```
RUN :F5:MAKE. :F1:REMOTE.MKE TARGET&
(ALL) GENALL PRINT
```

The TARGET option ALL forces MAKE to produce a submit file that includes all task lines required to generate ALL (in this case REMOTE, SEND, and RECV). The default option is the first dependency node's target. The TARGET option overrides this default.

Finally, we can submit the MAKE file:

```
SUBMIT :F1:REMOTE
```

and generate the system.

CREATING VARIANTS USING SVCS

Before creating variants, let's review what we've done so far. In Section III we designed a SVCS database to store and control all program modules that comprise the software system REMOTE. We then constructed the database using the SVCS commands listed in Figure 4, which reside on the tutorial disk in the file MAKEDB.CSD. A MAKE file was then created in Section IV to automate system generation. Starting with the SUBMIT file in Figure 2, we created several versions of a MAKE file, each increasing in complexity. The final version contained substitution macros, enumeration macros, parameter macros, special macros such as %ALL, iteration commands, header and trailer commands, and SVCS constructs. The final MAKE file version is stored on the tutorial disk as REMOTE.MKE.

At this point the most challenging work has been completed. This section will cover variant creation and database management.

Let's save the WORK variant of the database to a saved variant, which we'll call ISISPDS.

We create this variant by issuing the command:

```
RUN :F5:SVCS. ADMIN  :F4:REMOTE.DB ADD&
(VARIANT = ISISPDS FROM WORK)
```

This command DOES NOT DOUBLE the size of the database! Many auxiliary files are generated, but they only contain pointers back into the original files.

We protect the variant ISISPDS by the command:

```
RUN :F5:SVCS. ADMIN  :F4:REMOTE.DB&
WRITEACCESS (ISISPDS = FALSE)&
DEFAULTACESS (ISISPDS = ALL)&
DEFAULTACCESS (WORK = BRIAN)
```

This command:

1. Sets the ISISPDS variant to read only.
2. Gives all users who access the database the known working ISISPDS variant.
3. Gives BRIAN, a system programmer, access to the WORK variant.

BRIAN is now the only person who can write to the database. In addition, when BRIAN checks out database files, he will get WORK variant files.

We will now modify files within our SVCS database and possibly the MAKE file to generate new versions of REMOTE that execute under ISIS Series II, CPMPDS, and CPM Series II.

We must incorporate the following changes:

- To change from ISISPDS to ISIS Series II execution **(creating the variant ISISS2):**

  Change the SERIAL.PLM file (Refer to tutorial disk file SERIAL.S2).

- To change from ISIS Series II to CPM Series II execution **(creating the variant CPMS2):**

  Change the REMOTE.MKE file (Refer to tutorial disk file REMMKE.CPM);

  Change the FILEIO.PLM file (Refer to disk file FILEIO.CPM).

- To change from CPM Series II to CPM PDS execution **(creating the variant CPMPDS):**

  Change the SERIAL.PLM file (Refer to disk file SERIAL.PDS)

  Note: All changes are made to the WORK variant. Then MAKE (using REMOTE.MKE) is run on WORK. After the WORK variant is updated for the new REMOTE program, WORK is moved to a new variant in the database. Upon completion of the following steps, five variants will be in the database: ISISPDS, ISISS2, CPMS2, CPMPDS and WORK. All variants will contain appropriate files including the executable REMOTE file. Remember, when a new variant is created by WORK, only modified files are added to the database. Unchanged files are not duplicated; instead pointers are set to the original files. (Section VI discusses database overhead.)

The following commands create the three new variants ISISS2, CPMS2 and CPMPDS.

1. Follow these steps to change and save a new variant called ISISS2 for holding the ISIS Series II version:

   a. First, give "ALL" access to the work variant, then get SERIAL.PLM out of the WORK variant and place it in the file SERIAL.PLM:

   ```
   RUN :F5:SVCS. ADMIN :F4:REMOTE.DB&
   DEFAULTACESS (WORK = ALL)

   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (SERIAL) TO :F1:SERIAL.PLM WRITE
   ```

   b. Make the necessary changes to the source code (see Appendix A for the changes) with the editor and then put the file back into the database: (Note: For the purpose of this tutorial, you can ignore the changes and just put the file back into the database.)

   ```
   RUN :F5:SVCS. PUT :F4:REMOTE.DB&
   (SERIAL) from :F1:SERIAL.PLM
   ```

   c. Run MAKE to examine all files that must be recompiled, relinked and relocated. Since the module dependencies have not changed, we can use the same MAKE file. Run MAKE by entering the commands.

   ```
   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (EXEC) to :F1:REMOTE.MKE

   RUN :F5: MAKE. :F1:REMOTE.MKE TARGET&
   (ALL) PRINT
   ```

   Note that the write option was not added in the first command. We do this because we don't want to change the file, just get it out of the database. If we now tried to PUT the file back into the database, SVCS would tell us the file was not checked out for writing and cannot be PUT back.

   d. Now submit the file REMOTE.CSD that MAKE built:

   ```
   SUBMIT :F1:REMOTE
   ```

   e. The database WORK variant now contains the ISIS Series II version of REMOTE. We save the WORK variant under another name by entering:

   ```
   RUN :F5:SVCS.    ADMIN :F4:REMOTE.DB&
   ADD (VARIANT = ISISS2 FROM WORK)
   ```

   Again, the database IS NOT COPIED, only header records and changed files are added. By creating this variant, we added a new version of SERIAL.PLM to the database. By now you should realize the efficiencies PMT's provide in managing a large software project.

   f. Now write protect the ISIS Series II version of REMOTE:

   ```
   RUN :F5:SVCS. ADMIN :F4:REMOTE.DB&
   WRITEACCESS(ISISS2=FALSE)&
   DEFAULTACCESS (ISISS2=BILL, FRANK)
   ```

If Bill or Frank now access the database (under default conditions) they will get the variant ISISS2. They may also specify a variant and get it. For example, if Bill wanted a copy of the unit (FILEIO) source from ISISPDS variant, he could issue the following command:

```
RUN :F5:SVCS. GET :F4:REMOTE.DB&
(FILEIO,WORK,SO) TO :F1:FILEIO.PLM
```

PMT's allow the database administrator to assign default variants to the appropriate people. This is extremely useful in complex multifunctional projects such as REMOTE.

2. The work variant contains the ISISS2 version of REMOTE. We can change and save a new variant called CPMS2 to hold the CPM Series II version by following these steps:

   a. First, get REMOTE.MKE and FILEIO:PLM (which must be changed) out of the WORK variant:

   ```
   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (FILEIO) TO :F1:FILEIO.PLM WRITE

   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (EXEC) TO :F1:REMOTE.MKE WRITE
   ```

   b. Now make the necessary changes to :F1:REMOTE.MKE and :F1:FILEIO.PLM (See Appendix A for the changes.) using an editor and then put them back in the database:

   ```
   RUN :F5:SVCS. PUT :F4:REMOTE.DB&
   (FILEIO) FROM :F1:FILEIO.PLM

   RUN :F5:SVCS. PUT :F4:REMOTE.DB&
   (REMOTE) FROM :F1:REMOTE.MKE
   ```

   c. Next, get and run the MAKE file:

   ```
   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (EXEC) TO :F1:REMOTE.MKE

   RUN :F5:MAKE. :F1:REMOTE.MKE TARGET&
   (ALL) PRINT
   ```

   d. And submit the submit file REMOTE.CSD:

   ```
   SUBMIT :F1:REMOTE
   ```

   e. The database WORK variant now contains the CPM Series II version of REMOTE. We save the WORK variant under the name CPMS2:

   ```
   RUN :F5:SVCS. ADMIN :F4:REMOTE.DB ADD&
   (VARIANT=CPMS2 FROM WORK)
   ```

   Here again, the database IS NOT COPIED, only header records and changed files are added. In creating this variant, we only added new copies of FILEIO.PLM and REMOTE.MKE to the database.

   f. Finally write protect CPMS2:

   ```
   RUN :F5:SVCS. ADMIN :F4:REMOTE.DB&
   WRITEACCESS (CPMS2=FALSE)&
   DEFAULTACCESS (CPMS2=NONE)
   ```

   NONE used as defaultaccess allows no one default access to the CPMS2 variant.

3. The WORK variant now contains the version that runs on the SERIES II under CPM. We change the WORK variant to the PDS CPM version by following these steps:

   a. Get SERIAL.PLM out of the WORK variant:

   ```
   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (SERIAL) TO :F1:SERIAL.PLM WRITE
   ```

   b. Make the necessary changes with the editor and then PUT the file back into the database:

   ```
   RUN :F5:SVCS. PUT :F4:REMOTE.DB&
   (SERIAL) FROM :F1:SERIAL.PLM
   ```

   c. Run MAKE.

   ```
   RUN :F5:SVCS. GET :F4:REMOTE.DB&
   (EXEC) TO :F1:REMOTE.MKE

   RUN :F5:MAKE. :F1:REMOTE.MKE TARGET&
   (ALL) PRINT
   ```

   d. And Submit:

   ```
   SUBMIT :F1:REMOTE
   ```

   e. The database WORK variant now contains the version of REMOTE that runs on the PDS under CPM. We save this version under another name:

   ```
   RUN :F5:SVCS. ADMIN :F4:REMOTE.DB&
   ADD (VARIANT=CPMPDS FROM WORK)
   ```

   f. Finally write protect CPMPDS by entering:

   ```
   RUN :F5:SVCS. ADMIN :F4:REMOTE.DB&
   WRITEACCESS (CPMPDS=FALSE)&
   DEFAULTACCESS (CPMPDS=NONE)
   ```

The database now has five variants: WORK, ISISPDS, ISISS2, CPMS2, and CPMPDS.

230879-001

DATABASE OVERHEAD

We will now examine the overhead SVCS added to the database.

Initial Set-Up

We put 23 different files into the database when it was originally set-up. (See Figure 4 for the CSD file used for set-up.) These files totaled 49,689 bytes. After they were put into the database, the database expanded to 51,204 bytes. 1,515 bytes were added to the files. This represents a 3% overhead putting the files into the SVCS database. Figure 8 shows a breakdown of this overhead.

Adding Variants

By adding a variant to the database without changing any files an overhead of 10 + (length of variant name) bytes is incurred. In this scenario, the database is not copied. Only pointers are added.

If you first change a file (or files) in an old variant and then copy the unit to the new variant, the overhead incurred is 10 + (length of variant name) bytes for each unchanged class plus 54 bytes for each changed file.

Note: The **changed** file is also **added** to the database, so the database **grows** by (file length) + 54 for each class change. However, the actual overhead incurred is only 54 bytes. The file changes would have to be stored on the disk even if we weren't using SVCS. Put another way, only files that are modified in a new version are copied. These files expand the database by 54 + (file length) bytes and expand the disk spaced used by the same amount. Without SVCS the additional disk space consumed would be (file length) bytes. The SVCS overhead is the difference between these two amounts, or 54 bytes.

Total Overhead

By adding the four variants to the database we added 691 bytes of overhead. Our overhead now totals: 2,206 bytes. The total size of the database (with all four executable files, sources, objects, etc.) is 88,025 bytes. Therefore, the overhead is only 2.5% of the database - a small price to pay considering all the features included with PMT's.

USING DEFAULTS TO SIMPLIFY OPERATION

While progressing through the tutorial, we have used the DEFAULTACCESS administration command several times. We will now discuss how the DEFAULTACCESS command can simplify the database administrator's job. First using the REMOTE example, let us set-up a hypothetical work environment.

Programmer	Variant working with
Brian	WORK
Bill, Frank	ISISPDS
John, Chris	ISISS2
Tim, Howard, Mary	CPMS2
Susan, Gordon	CPMPDS

Now, the command:

```
RUN :F5:SVCS. ADMIN :F4:REMOTE.DB&
DEFAULTACCESS (CPMS2 = Tim, Howard,&
Mary)
```

will set Tim's, Mary's and Howard's defaultaccess to CPMS2. (The names used are the NDS-II logon ID's, which SVCS uses to identify users.) If Tim entered the command:

```
RUN :F5:SVCS. GET :F4:REMOTE.DB&
(FILEIO) TO :F1:FILEIO.PLM
```

SVCS would get Tim's ID name from the system and use Tim's default variant, which is CPMS2. Tim would now have the CPMS2 version of FILEIO.PLM in his directory.

What if Tim wanted the FILEIO.PLM file in the ISISPDS variant? He would issue the following command and get it:

```
RUN :F5:SVCS. GET :F4:REMOTE.DB&
(FILEIO,ISISPDS) TO :F1:FILEIO.PLM
```

Note when the variant is specified; the default isn't checked.

The DEFAULTACCESS command aids the database administrator and system users by minimizing typing and by organizing which variants programmers are using.

Note: NONE and ALL may be used to set variants to the NONE or ALL default. However, if a user's default is changed, it is deleted from the previous list. Thus, if you set 15 people to 8 different variants and then set one variant to ALL, you will relinquish the old defaults and set everyone to the new one.

STANDALONE OPERATION

This section covers the use of PMT's on a Series III standalone system running ISIS II (W).

System Differences

Two major differences exist between standalone systems and networks: standalone systems do not have user ID's nor do they have date/time stamping.

SVCS must know which user is accessing the database in order to accurately check units and default accesses. Because standalone systems do not support user ID's, the user must add the string:

ID (user name)

to all commands.

MAKE uses the 'D' attribute on standalone systems to check file modification times. The 'D' attribute on ISIS II (W) marks when a file has been modified. When you edit a file called FILE1.PLM and save the changes, the file's 'D' attribute bit is set. This bit can only be reset with the following command:

```
ATTRIB FILE1.PLM DO
```

MAKE uses the 'D' attribute on standalone systems to tell if a file has been modified.

The next section lists the necessary MAKE file changes that must be made for MAKE operation on standalone systems.

MAKE File Changes

The command:

```
$IF FILE1.OBJ > FILE1.PLM, ISIS.EXT,
$COMMON.LIT THEN
     PLM80 FILE1.PLM
$END
```

is a typical MAKE construct for a MAKE file on a Network. On a standalone system, MAKE tests the file modification times (in the IF-THEN statement above) by testing if the 'D' attribute bit is set in any of the dependency files. If the bit is set, it assumes the files have been modified.

To use MAKE on a standalone system, we must add a command to the MAKE construct that resets the 'D' bit:

```
$IF FILE1.OBJ > FILE1.PLM, ISIS.EXT,
$COMMON.LIT THEN
     PLM80 FILE1.PLM
     ATTRIB %DEPEND DO
$END
```

The additional line in the above MAKE file resets the 'D' attribute of the dependant files when the generated SUBMIT file is run. These task lines will not be included in the SUBMIT file when MAKE is run again unless one of the dependency files is modified and its 'D' attribute is reset.

FIGURES

1. Remote Source Modules
2. SUBMIT File REMCSD.EXM
3. Remote System Diagram
4. SUBMIT File MAKEDB.CSD
5. MAKE File REMMKE.EXM
6. MAKE File RMMKE1.EXM
7. MAKE File REMOTE.MKE
8. Database Overhead

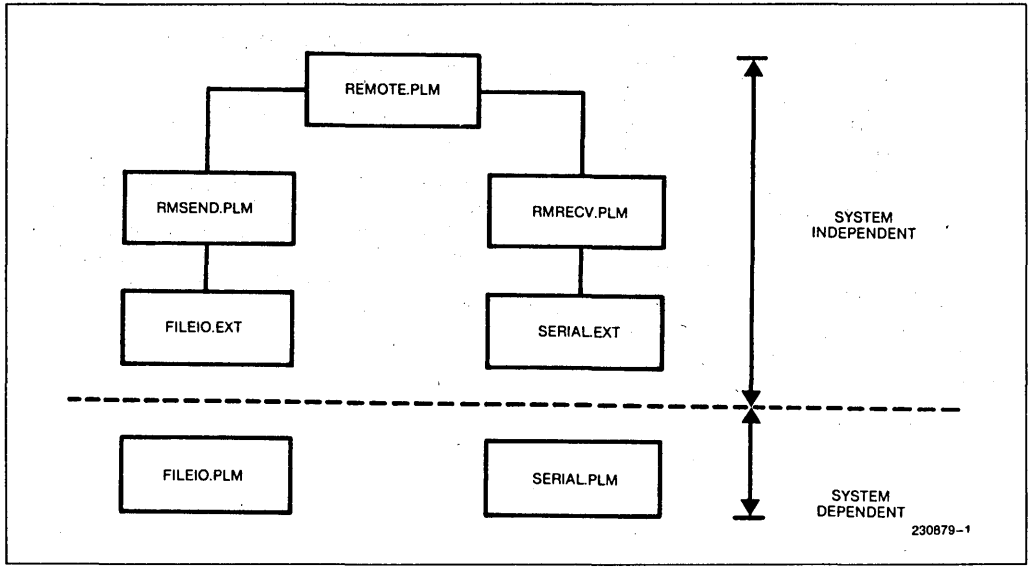


Figure 1. Shows the source modules used to create REMOTE.

REMOTE.PLM is the main module
RMSEND.PLM is a sub module used by REMOTE.PLM
RMRECV.PLM is a sub module used by REMOTE.PLM
FILEIO.PLM defines the operating system interfaces
FILEIO.EXT contains the external declarations of FILEIO.PLM
SERIAL.PLM defines the hardware interfaces of the SERIAL line
SERIAL.EXT contains the external declarations of SERIAL.PLM

It is expected that SERIAL.PLM and FILEIO.PLM will change for each system that REMOTE is configured for. The remaining modules are not expected to change.

Figure 2. Submit file used to generate REMOTE, RECV and SEND without using PMTs.

```
Page 1              :F1:REMCSD.EXM

; Submit file to generate the remote executable file that runs on the
; iPDS system under ISIS.
; Author :B. Valentine DSSO Applications Engineering 6/23/83

; Generate REMOTE file to run on the iPDS system

; First of all, compile all the source code.
:f2:plm80 :f1:remote.plm
:f2:plm80 :f1:rmsend.plm
:f2:plm80 :f1:rmrecv.plm
:f2:plm80 :f1:fileio.plm
:f2:plm80 :f1:serial.plm

; Now link them together
:f2:link :f1:remote.obj,:f1:rmsend.obj,:f1:rmrecv.obj,:f1:fileio.obj, &
:f1:serial.obj,:f3:plm80.lib,:f3:system.lib to :f1:remote.lnk

; Now locate the link file
:f2:locate :f1:remote.lnk symbols lines map print (:f1:remote.map)

; Move the file to the system directory
copy :f1:remote to remote b

; Generate SEND that runs on the network

:f2:plm80 :f1:send.plm
:f2:link :f1:send.obj,:f3:system.lib,:f3:plm80.lib to :f1:send.lnk
:f2:locate :f1:send.lnk
copy :f1:send to send b

; Generate RECV that runs on the network

:f2:plm80 :f1:recv.plm
:f2:link :f1:recv.obj,:f3:system.lib,:f3:plm80.lib to :f1:recv.lnk
:f2:locate :f1:recv.lnk
copy :f1:recv to recv b
```

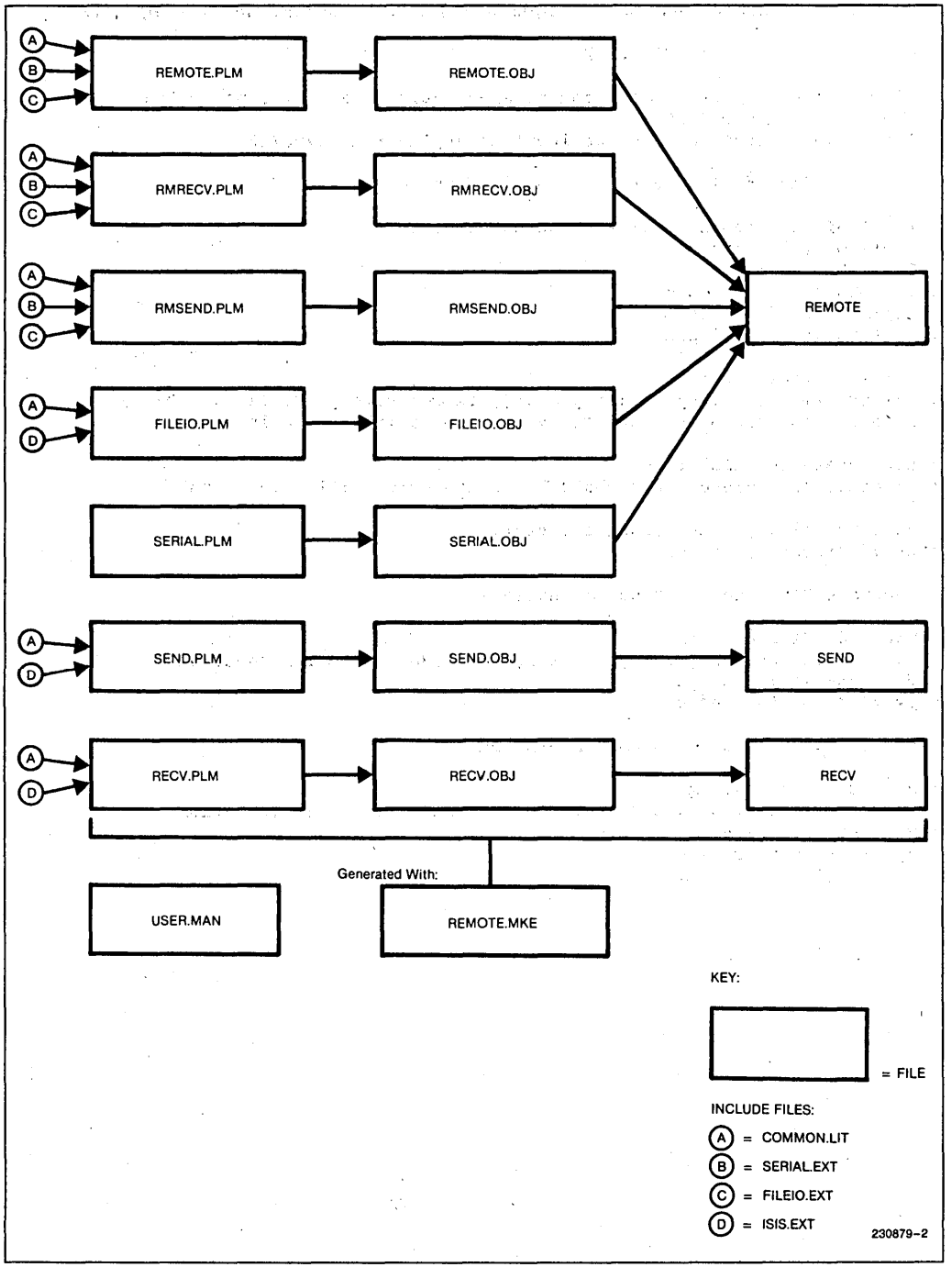


Figure 3. Remote System Diagram

230879-001

Figure 4. Submit file used to create and fill database. (Continued)

```
Page 2            :F1:MAKEDB.CSD

:f5:svcs. get    :f4:remote.db(fileio,,cp)        to :bb: write
:f5:svcs. put    :f4:remote.db(fileio,,cp)        from :fl:fileio,ext
:f5:svcs. admin :f4:remote.db add(unit = send    from :fl:send.plm)
:f5:svcs. put    :f4:remote.db(send,,oj)          from :fl:send.obj
:f5:svcs. get    :f4:remote.db(send,,cp)          to :bb: write
:f5:svcs. put    :f4:remote.db(send,,cp)          from :fl:send
:f5:svcs. admin :f4:remote.db add(unit = recv    from :fl:recv.plm)
:f5:svcs. put    :f4:remote.db(recv,,oj)          from :fl:recv.obj
:f5:svcs. get    :f4:remote.db(recv,,cp)          to :bb: write
:f5:svcs. put    :f4:remote.db(recv,,cp)          from :fl:recv
:f5:svcs. admin :f4:remote.db add(unit = common from :f3:common.lit)
:f5:svcs. admin :f4:remote.db add(unit = isis    from :f3:isis.ext)
:f5:svcs. admin :f4:remote.db add(unit = exec    from :fl:remote.mke)
:f5:svcs. put    :f4:remote.db(exec,,oj)          from :fl:remote
:f5:svcs. get    :f4:remote.db(exec,,cp)          to :bb: write
:f5:svcs. put    :f4:remote.db(exec,,cp)          from :fl:user.man

; Now the database is built, filled and saved in :f4:

; The following command creates a new variant called ISISPDS, by copying
; the WORK variant. In reality, no files are copied but pointers are set up
; pointing to the files for the ISISPDS variant; thus, disk space is conserved
; by using SVCS.

:f5:svcs. admin :f4:remote.db add(variant = isispds from work)

; Finally, the following command protects the database. Only BRIAN is given
; access to the WORK variant, everyone else is given access to database files
; in the ISISPDS variant. This SVCS feature permits the database administrator
; to assign variants to only those people needing the paticular versions.
; In addition, the writeaccess option sets the ISISPDS variant to read only;
; thus, no one can checkout the ISISPDS variant with write access.
:f5:svcs. admin :f4:remote.db writeaccess (isispds = false) &
    defaultaccess (isispds = all) defaultaccess (work = brian)

;
exit
```

230879-001

Figure 4. Submit file used to create and fill database.

```
Page 1          :F1:MAKEDB.CSD

; Submit file to create and fill the remote database from initial sources.
; It is submitted one time only. Once the database is created and filled,
; future changes are made using the SVCS ADD, PUT and GET commands.
; Device assignments are as follows:
;         :f0: - Directory containing ISIS system files
;         :f1: - Directory containing files supplied on tutorial disk
;         :f2: - Directory containing 8 bit system software files. Such
;                as PL/M-80, (8 bit) SVCS and MAKE compiler.
;         :f3: - Directory containing 8 bit libraries. Such as system.lib.
;                common.lit and isis.ext, supplied on the tutorial disk,
;                must be moved to this directory.
;         :f4: - Directory where all the database files will be created.
;         :f5: - Directory containing the 16 bit system software. Such as
;                PL/M-86, (16 bit) SVCS and MAKE.
; Author : B. Valentine - DSSO Applications Engineering 6/22/83

; Create the database
run :f5:svcs. admin :f4:remote.db create

; Since more than one person will access the database - make it shareable
; SVCS will propagate these access rights across all database files.
access :f4:remote.db wrl wwl

; Now that the database is created, fill it with the files supplied on the
; tutorial disk. The WORK variant will then contain the version of REMOTE
; for the iPDS running under ISIS..
;       ADD may be used to initialize SOURCE files, but
;       PUT must be used to initialize OBJECT,HISTORY or COMPOSITION files.
run

; Create a unit called remote and fill it with the source file remote.plm
:f5:svcs. admin :f4:remote.db add(unit = remote from :f1:remote.plm)

; Now fill the object class of the remote unit
:f5:svcs. put   :f4:remote.db(remote,,oj)       from :f1:remote.obj
; Note that if the variant name is not specified (remote,,oj), WORK is the
; default.

; Continue creating units and filling them with the files.
:f5:svcs. admin :f4:remote.db add(unit = rmsend from :f1:rmsend.plm)
:f5:svcs. put   :f4:remote.db(rmsend,,oj)       from :f1:rmsend.obj
:f5:svcs. admin :f4:remote.db add(unit = rmrecv from :f1:rmrecv.plm)
:f5:svcs. put   :f4:remote.db(rmrecv,,oj)       from :f1:rmrecv.obj
:f5:svcs. admin :f4:remote.db add(unit = serial from :f1:serial.plm)
:f5:svcs. put   :f4:remote.db(serial,,oj)       from :f1:serial.obj

; Note in the next command how you must GET (write permission) a composition
; unit before you can PUT to it. Since it has nothing in it yet, GET it to
; the byte bucket.
:f5:svcs. get   :f4:remote.db(serial,,cp)       to :bb: write
:f5:svcs. put   :f4:remote.db(serial,,cp)       from :f1:serial.ext

; Create and fill the remaining units of the database
:f5:svcs. admin :f4:remote.db add(unit = fileio from :f1:fileio.plm)
:f5:svcs. put   :f4:remote.db(fileio,,oj)       from :f1:fileio.obj
```

230879-001

Figure 5. First pass at building MAKE file. (See Figure 2 for submit file.)

```
Page 1            :F1:REMMKE.EXM

; Make file for the isis remote program
; This make program uses the least complicated make constructs to test
; the dependencies.
; Author : B. Valentine DSSO Applications Engineering 6/25/83

; Set the dependency tree for three separate executable files.
; Fool the MAKE utility into building the dependency tree for three
; unrelated (executable) programs by using ALL.
$IF all > :fI:remote, send, recv THEN
$END

; Note how in the next construct the source code include files are added.
; Also note how some of the files have the same right hand size dependencies
; accept for the changing of the file name. Pass 2 of the make file will
; show how these can be combined into an iteration loop.
$IF :fl:serial.obj > :fl:serial.plm THEN
    :f2:plm80 :fl:serial.plm
$END

$IF ;fl:fileio.obj > :fl:fileio.plm, :f3:isis.ext, :f3:common.lit THEN
    :f2:plm80 :fl:fileio.plm
$END

$IF :fl:remote.obj > :fl:remote.plm, :f3:common.lit, :fl:serial.ext,
$:fl:fileio.ext THEN
    :f2:plm80 :fl:remote.plm
$END

$IF :fl:rmsend.obj > :fl:rmsend.plm, :f3:common.lit, :fl:fileio.ext
$:fl:serial.ext THEN
    :f2:plm80 :fl:rmsend.plm
$END

$IF :fl:rmrecv.obj > :fl:rmrecv.plm, :f3:common.lit, :fl:fileio.ext,
$:fl:serial. ext THEN
    :f2:plm80 :fl:rmrecv.plm
$END

; Check the status of the remote executable file.
$IF :fl:remote > :fl:remote.obj, :fl:rmsend.obj, :fl:rmrecv.obj,
$:fl:fileio.obj, :fl:serial.obj THEN
    :f2:link :fl:remote.obj, :fl:rmsend.obj, :fl:rmrecv.obj, &
    :fl:fileio.obj, :fl:serial.obj, :f3:plm80.lib, :f3:system.lib to &
    :fl:remote.lnk
    :f2:locate :fl:remote.lnk symbols lines map print (:fl:remote.map)
$END

; Now that the remote program has been checked, check the two programs
; that run on the network.
; Check the NDS-II files RECV and SEND.!

; Check SEND
; Since there is only one module to the send program, we can test the
; executable file against the source code.
```

Figure 5. First pass at building MAKE file. (See Figure 2 for submit file.) (Continued)

```
Page 2              :F1:REMMKE.EXM

$IF send > :f1:send.plm, :f3:common.lit, :f3:isis.ext THEN
    :f2:plm80 :f1:send.plm
    :f2:link :f1:send.obj,:f3:system.lib,:f3:plm80.lib to :f1:send.lnk
    :f2:locate :f1:send.lnk
    copy :f1:send to send b
$END

; Check RECV
$IF recv > :f1:recv.plm, :f3:common.lit, :f3:isis.ext THEN
    :f2:plm80 :f1:recv.plm
    :f2:link :f1:recv.obj,:f3:system.lib, :f3:plm80.lib to :f1:recv.lnk
    :f2:locate :f1:recv.lnk
    copy :f1:recv to recv b
$END
```

Figure 6. Second pass of MAKE file. Note how macros and iteration are added.

```
Page 1              :F1:RMMKE1.EXM

; Second pass of the MAKE file for the REMOTE program.
; This pass has added the MAKE constructs of macros and iteration to
; pass one of the MAKE file.
; Author : B. Valentine DSSO Applications Engineering 6/25/83

; First of all define the macros for the MAKE file.
;  Define the substitution macros :
;    Substitution macros are used as constant defines. This way, if
;    a major change is made, such as the source code device changes
;    from :f1: to :f2:, the only update to the MAKE files is to change
;    the macro definition

$    SET work_device      to ':f1:'
$    SET 8_bit_exe        to ':f2:'
$    SET 8_bit_lib        to ':f3:'
; Note how macros may be nested and the macro is used with the %"<name>".
$     SET plm             to '%"8_bit_exe"plm80'
$     SET locate          to '%"8_bit_exe" locate'
$     SET link            to '%"8_bit_exe"link'
$     SET syslib          to '%"8_bit_lib"system.lib'
$     SET plmlib          to '%"8_bit_lib"plm80.lib'
$     SET comlit          to '%"8_bit_lib"common.lit'
$     SET isis            to '%"8_bit_lib"isis.ext'
; Now define the enumeration macros :
$     SET nds2_files      to (recv,send)
$     SET remote_files    to (rmrecv,rmsend)
; Now start the dependecies
```

230879-001

Figure 6. Second pass of MAKE file. Note how macros and iteration are added. (Continued)

```
; Set the dependency tree for three separate executable files.
$IF all > %"work_device"remote, send, recv THEN
$END

$IF %"work_device"serial.obj > %"work_device"serial.plm THEN
    %plm %"work_device"serial.plm
$END

$IF %"work_device"fileio.obj > %"work_device"fileio.plm, %comlit, %isis THEN
    %plm %"work_device"fileio.plm
$END

$IF %"work_device"remote.obj > %"work_device"remote.plm,
$%comlit, %"work_device"serial.ext, %work_device"fileio.ext THEN
    %plm %"work_device"remote.plm
$END

$FOR i IN %remote_files
; Build the send and receive modules for the remote system.
$     IF %work_device%"i".obj > %work_device%"i".plm, %comlit,
$     %"work_device"fileio.ext, %"work_device"serial.ext THEN
        %plm %work_device%"i".plm
$     END
$END

; Check the remote executable file
$IF %"work_device"remote > %"work_device"remote.obj,
$%"work_device"rmsend.obj, %"work_device"rmrecv.obj,
$%"work_device"fileio.obj, %"work_device"serial.obj,
$%plmlib, %syslib THEN
    %link %"work_device"remote.obj, &
      %"work_device"rmsend.obj, %"work_device"rmrecv.obj, &
      %"work_device"fileio.obj, %"work_device"serial.obj, &
      %plmlib, %syslib to %"work_device"remote.lnk
    %locate %"work_device"remote.lnk symbols lines &
      map print (%"work_device"remote.map)
$END

; Now that the remote program has been checked, check the two programs
; that run on the network.
$FOR i IN %nds2_files
; Check the NDS_II files RECV and SEND.
$     IF %i > %work _device%"i".plm, %comlit, %isis THEN
        %plm %work_device%"i".plm
        %link %work_device%"i".obj,%syslib, %plmlib to %work_device%"i".lnk
        %locate %work_device%"i".lnk
        copy %work_device%"i" to %i b
$   END
$END
```

230879-001

Figure 7. Final pass of MAKE file

```
Page 1          :F1:REMOTE.MKE

; MAKE file for the isis REMOTE program that runs on the iPDS system.
; Author : B. Valentine DSSO Applications Engineering 6/25/83

; First of all define the macros for the MAKE file.
;  Define the substitution macros:
;    Substitution macros are used as constant defines. This way, if
;    a major change is made, such as the source code device changes
;    from :f1: to :f2:, the only update to the MAKE file is to change
;    the macro define.

$    SET work_device       to ':f1:'
$    SET 8_bit_exe         to ':f2:'
$    SET 8_bit_lib         to ':f3:'
$    SET database          to ':f4:remote.db'
$    SET svcs_drive        to 'run :f5:'

$    SET plm               to '%"8_bit_exe"plm80'
; Note how macros may be nested and the macro is used with the %"<name>.
$    SET locate            to '%"8_bit_exe"locate'
$    SET link              to '%"8_bit_exe"link'
$    SET syslib            to '%"8_bit_lib"system.lib'
$    SET plmlib            to '%"8_bit_lib"plm80.lib'
$    SET comlit            to '%"8_bit_lib"common.lit'
$    SET get               to '%"svcs_drive"svcs_get %database'
$    SET put               to '%"svcs_drive"svcs_put %database'

; Now define the enumeration macros:
$    SET nds2_files        to (recv,send)
$    SET remote_files      to (remote,fileio,serial,rmrecv,rmsend)
$    SET files             to (%all(%nds2_files),%all(%remote_files))

; Tell make that we are going to be looking at the files in the database.
$ FOR i in %files
$    svcs %work_device%"i".plm   =%database (%i)
$    svcs %work_device%"i".obj   =%database (%i,,oj)
$END
$ svcs %"work_device"serial.ext  =%database (serial,,cp)
$ svcs %"work_device"fileio.ext  =%database (fileio,,cp)
$ svcs %"work_device"remote      =%database (exec,,oj)
$ svcs %"work_device"send        =%database (send,,cp)
$ svcs %"work_device"recv        =%database (recv,,cp)

; The include files are always required, so get them with the header.
$ HEADER
; Get all the externals and include files from the database
    %get (serial,,cp)     to %"work_device"serial.ext
    %get (fileio,,cp)     to %"work_device"fileio.ext
$END

; Now start the dependecies

;Set the dependency tree for three separate executable files.
;IF all > %"work_device"remote, %all(%work_device%nds2_files) THEN
$END
```

230879-001

Figure 7. Final pass of MAKE file (Continued)

```
Page 2              :F1:REMOTE.MKE

$FOR i IN %remote_files
; Build all the object files in the remote program.
$     IF %work_device%"i".obj > %work_device%"i".plm, %comlit,
$     %"work_device"fileio.ext, %"work_device"serial.ext THEN
         %get (%i) to %work_device%"i".plm
         %plm %work_device%"i".plm
         %put (%i,,oj) from %target
$     END
$END

; Check the remote executable file that runs on the iPDS system.
$ IF %"work_device"remote > %all(%work_device%"remote_files".obj),
$ %plmlib, %syslib THEN
$     FOR i in %remote_files
         %get (%i,,oj) to %work_device%"i".obj
$     END
     %link %depend to %"work_device"remote.lnk
     %locate %"work_device"remote.lnk symbols lines &
          map print(%"work_device"remote.map)
     %put (exec,,oj) from %target
$ END

; Now that the remote program has been checked, check the two programs
; that run on the network.


$FOR i IN %nds2_files.
; Check the NDS_II files RECV and SEND.
$     IF %work_device%i > %work_device%"i".plm THEN
         %get (%i) to %depend
         %plm %depend
         %put (%i,,oj) from %work_device%"i".obj
         %link %work_device%"i".obj, %syslib, %plmlib to %work_device%"i".lnk
         %locate %work_device%"i".ink
         %get (%i,,cp) to :bb:write
         %put (%i,,cp) from %target
$     END
$END
```

Figure 8 Breakdown of initial overhead of database.

Note: All numbers are in bytes

Byte Length		
266	Database header file	
62	Creating a unit in the database	
95	**Filling classes of a unit**	(Maximum)
	First Class	41
	Second Class	18
	Third Class	18
	Fourth Class	18
	TOTAL	95

Note: Class overhead is calculated for only the classes filled. Example, If a unit has only two classes filled, the overhead is 41 + 18 = 59.

In REMOTE, there are 10 units.
The overhead breaks down as follows:

Byte Length		
266	Database header file	
620	Unit header files 10 units × 62	
629	**Unit Classes**	
	2 units with 1 class filled 2×41	=82
	3 units with 2 classes filled 3×(41+18)	=162
	5 units with 3 classes filled 5×(41+18+18)	=385
	TOTAL	629
1,515	GRAND TOTAL	

230879-001

APPENDIX A

CONTENTS

PAGE

USERS MANUAL A-2
USER. MAN A-2

INCLUDE FILES A-3
FILEIO.EXT A-3
COMMON.LIT A-4
SERIAL.EXT A-4
ISIS.EXT A-5

REMOTE SOURCE FOR ISIS, IPDS ... A-7
REMOTE.PLM A-7
RMRECV.PLM A-10
RMSEND.PLM A-13
FILEIO.PLM A-15
SERIAL.PLM A-17

SOURCE CODE FOR RECV AND SEND NETWORK FILES A-17
RECV.PLM A-17
SEND.PLM A-19

CHANGES TO SERIAL.PLM TO RUN ON A SERIES III A-21
SERIAL.S2 ..... A-21

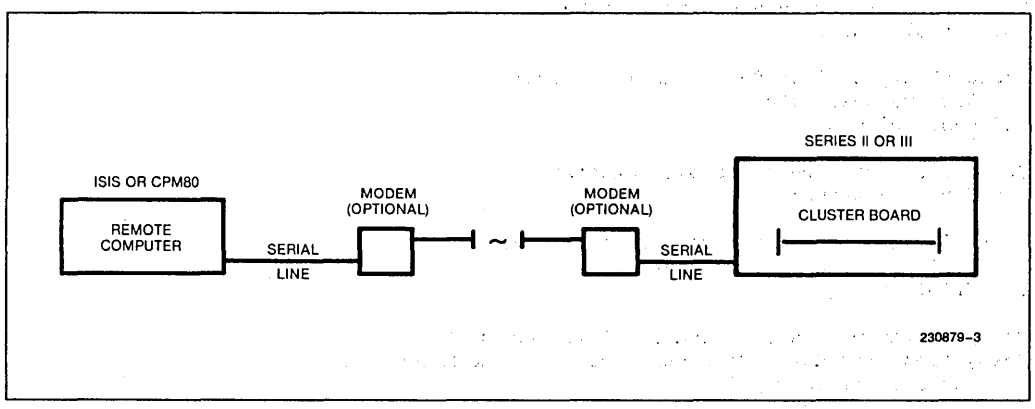
CHANGES TO FILEIO.PLM AND REMOTE.MKE TO RUN UNDER CPM A-22
FILEIO.CPM A-22
REMOTE.CPM A-26

USERS MANUAL

```
:F1:USER.MAN
```

This is the users manual for the PMT tutorial Remote program.

REMOTE is a program that runs on a remote computer connected to an NDS-II via an ISIS cluster board. The connection is an RS232 line and may include modems as shown below.



REMOTE, for the most part, turns the remote computer into a dumb terminal. ie. Characters entered on the remote computer keyboard are sent down the serial line and characters received up are echoed to the remote computers screen.

REMOTE internally collects the characters typed by the remote computer user and saves them in a buffer. When a <CR> is entered, REMOTE scans the buffer looking for three special commands - SEND, RECV and LOCAL. If these commands are not found - operation as a dumb terminal continues.

If one of these special commands are intercepted, REMOTE flips into file transfer mode (SEND or RECV commands) or back to standalone operation. Also, SEND or RECV cause a program to be activated on the cluster board to effect the file transfer. A simple protocol is used (STX, 128 data bytes, CHECKSUM, ETX) so some error checking is done.

REMOTE is written to be configurable. The I/O system is defined in FILEIO.PLM and the serial line configuration is SERIAL.PLM. All of the software is written in PLM. The systems currently supported are:

```
FILEIO.PLM - ISIS and CPM80

SERIAL.PLM - Series-II and iPDS
```

Thus four variants are currently available.

Note: Device 0 on the network must contain the SEND and RECV executable files.

230879-001

INCLUDE FILES

```
:F1:FILEIO.EXT

/*     In case of fatal errors */
Exit: Procedure external;
end Exit;

/*    Operating system dependant Console Routines */
Console$Input: Procedure byte external;
end Console$Input;

Console$Output: Procedure (char) external;
    Declare char byte;
end Console$Output;

Console$Status: Procedure byte external;
end Console$Status;

Print$String: Procedure (string$ptr) external;
    Declare string$ptr pointer;
end Print$String;

/*    Operating system dependant file routines */
Open$file: Procedure (file$ptr,mode) byte external;
    Declare file$ptr pointer,
            mode byte;
end Open$File;

Create$file: Procedure (file$ptr) byte external;
    Declare file$ptr pointer;
end Create$File;

Read$sector: Procedure (file$id, buffer$ptr) byte external;
    Declare file$id byte,
            buffer$ptr pointer;
end Read$sector;

Write$sector: Procedure (file$id, buffer$ptr) byte external;
    Declare file$id byte,
            buffer$ptr pointer;
end Write$sector;

Close$file: Procedure (file$id) byte external;
    Declare file$id byte;
end Close$File;

$list
```

230879-001

```
:F1:COMMON.LIT

/* Some useful defines for the remote program */

declare lit        literally 'literally';
declare word       lit       'address';
declare pointer    lit       'address';
declare connection lit       'address';

declare cr     lit '0dh',
        lf     lit '0ah',
        TAB    lit '09h',
        SOH    lit '01h',
        STX    lit '02h',
        ETX    lit '03h',
        EOT    lit '04h',
        ACK    lit '06h',
        NAK    lit '15h',
        XON    lit '11h',
        XOF    lit '13h',
        CAN    lit '18h',
        SUB    lit '1ah',
     RUBOUT    lit '7fh';

declare forever lit 'while 1';

declare false    lit '0',
        true     lit 'not false';

declare read$only    lit '1',
        write$only   lit '2',
        read$write   lit '3';

$list



:F1:SERIAL.EXT

/* Front end externals for the serial.plm link to the remote logon */
Serial$Status: Procedure byte external;
end Serial$Status;

Serial$Input: Procedure byte external;
end Serial$Input;

Serial$Output: Procedure (char) external;
    Declare char byte;
end Serial$Output;

Serial$Control: Procedure (value) external;
    Declare value byte;
end Serial$Control;

$list
```

230879-001

```
:F1:ISIS.EXT

isis: procedure (type, parameter$ptr) external;
    declare type byte,
            parameter$ptr address;
end isis;

open: procedure (conn$p, path$p, access, echo, status$p) external;
    declare (conn$p, path$p, access, echo, status$p) address;
end open:

close: procedure (conn, status$p) external;
    declare (conn, status$p) address;
end close;

read: procedure (conn, buff$p, count, actual$p, status$p) external;
    declare (conn, buff$p, count, actual$p, status$p) address;
end read;

write: procedure (conn, buff$p, count, status$p) external;
    declare (conn, buff$p, count, status$p) address;
end write;

seek: procedure (conn, mode, block$p, byte$p, status$p) external;
    declare (conn, mode, block$p, byte$p, status$p) address;
end seek;

rescan: procedure (con, status$p) external;
    declare (conn, status$p) address;
end rescan;

spath: procedure (path$p, info$p, status$p) external;
    declare (path$p, info$p, status$p) address;
end spath;

delete: procedure (path$p, status$p) external;
    declare (path$p, status$p) address;
end delete;

rename: procedure (old$p, new$p, status$p) external;
    declare (old$p, new$p, status$p) address;
end rename;

attrib: procedure (path$p, attrib, on$off, status$p) external;
    declare (path$p, attrib, on$off, status$p) address;
end attrib;

consol: procedure (ci$p, co$p, status$p) external;
    declare (ci$p, co$p, status$p) address;
end consol;

load: procedure (path$p, load$offset, switch, entry$p, status$p) external;
    declare (path$p, load$offset, switch, entry$p, status$p) address;
end load;

whocon: procedure (conn, buff$p) external;
    declare (conn, buff$p) address;
```

230879-001

```
:F1:ISIS.EXT

end whocon;

error: procedure (error$num) external;
    declare (error$num) address;
end error;

de$time: procedure (dt$p, status$p) external;
    declare (dt$p, status$p) address;
end de$time;

filinf: procedure (file$table$p, mode, file$info$p, status$p) external;
    declare (file$table$p, file$info$p, status$p) address,
            mode byte;
end filinf;

getd: procedure (did, conn$p, count, actual$p, table$p, status$p) external;
    declare (did, conn$p, count, actual$p, table$p, status$p) address;
end getd;

exit: procedure external;
end exit;

ci: procedure byte external;
end ci;

co: procedure (char) external;
    declare (char) byte;
end co;

ri: procedure byte external;
end ri;

po: procedure (char) external;
    declare (char) byte;
end po;

lo: procedure (char) external;
    declare (char) byte;
end lo;

csts: procedure byte external;
end csts;

iodef: procedure (type, entry) external;
    declare type byte,
            entry address;
end iodef;

iochk: procedure byte external;
end iochk;

ioset: procedure (value) external;
    declare value byte;
end ioset;

memck: procedure address external;
end memck;

$list
```

230879-001

REMOTE SOURCE FOR ISIS, iPDS

```
:F1:REMOTE.PLM

$DEBUG
Remote$Logan: do;

/*
    This program runs on a remote computer connected via a serial line to
    an ISIS Cluster board on an NDS-II system.
    The remote computer may be connected to the ISIS cluster board via a modem.
    It enables the remote computer to use all of the facilities of the NDS-II.
    For the most part the remote computer behaves as a dumb terminal; two
    commands (SEND and RECV) are intercepted to enable file tranfer
    between the remote computers file system and the NDS-II file system.
    Most remote computers will not be able to keep up with a high speed serial
    line so a XON/XOF protocol is used to slow the serial line down if required.
    Author: B. Valentine 6/22/83 - DSSO Applications Engineering
*/

$nolist include (:f3:common.lit)
$include(:f1:serial.ext)
$nolist include(:f1:fileio.ext)

Declare buffer$ptr              byte public;
Declare buffer(128)             byte public;
Declare save$buffer(128)        byte;
Declare save$buffer$ptr         byte;
Declare character               byte;
Declare time$out                byte;
Declare saved                   byte;
Declare i                       byte;

Send: Procedure external;
end Send;

Receive: Procedure external;
end Receive;

Uppercase: Procedure (char) byte;
/*If the character passed in is lowercase then convert it to uppercase.
*/
Declare char byte;
    if ((char>= 'a') AND (char <= 'z')) the return (char - 20h);
    return char;
end Uppercase;

Put$in$buffer: Procedure (character):
/* Put the character passed in into the input buffer, checking for EOLN
   and rubout.
*/
Declare character byte;
    character = uppercase (character);
    if character = RUBOUT then do;
        if buffer$ptr <> 0 then buffer$ptr = buffer$ptr - 1;
        return;
```

```
:F1:REMOTE.PLM

    end;
    if character = cr then buffer$ptr = 0;
    else if character = lf then do;
        /* Mark end of the buffer */
        buffer (buffer$ptr) = ' ';
        buffer$ptr = buffer$ptr + 1;
        buffer(buffer$ptr) = lf;
        return;
    end;
    else if character >= ' ' then do;
        buffer(buffer$ptr) = character;
        buffer$ptr = buffer$ptr + 1;
    end;
end Put$in$buffer;

Match$Keyword: Procedure byte;
/* Check command to see if it's one to process or just send down to the
   Network.
*/
Declare Keywords (4) structure (text(7) byte) data
    ('SEND   ',
     'RECV   ',
     'LOGOFF ',
     'LOCAL  ');
Declare (index,match,i) byte;
    do index = 0 to 3;
        match = true;
        do i = 0 to 6;
            if (Keywords(index).text (i) = ' ') and (match) then return index;
            if buffer(i) <> Keywords(index).text(i) then match = false;
        end;
    end;
return 4; /* No match */
end Match$Keyboard;

/************************* Program starts here ************************/
buffer$ptr = 0;
do i = 0 to 127;
    buffer(i) = ' ';
end;

/* Say Hello to the user */
call Print$String(.(cr,lf,
'REMOTE LOGON TO NDS 2. X1.8',cr,lf,
'--------------------',cr,lf,lf,
'Trying to establish connection......',cr,lf,'$'));

/* Kick start the serial line */
/* Send a BREAK */
call Serial$Control(00111111b);
call time(200);
call Serial$Control(00110111b);

call Serial$Output(cr);
```

230879-001

```
:F1:REMOTE.PLM

call Serial$Output(cr);

/* Set up as a transparent terminal */
do forever;
    if Console$Status then do;
        character = Console$Input and 7FH;
/* Need to scan for SEND and RECV commands */
        if character = cr then do;
            call put$in$buffer(lf);   /* Mark end of buffer */
            do case match$keyword;
                call send;
                call receive;
                do;             /* User typed Logoff, so do it */
                    call Serial$Output(cr);
                    call Exit;
                    end;
                do;
                    call Serial$Output(CAN);
                    call Exit;
                    end;
                ; /* Do nothing */
                end;
            do i = 0 to 127;    /* Clear buffer after comparison */
                buffer(i) = ' ';
                end;
            end;
        call put$in$buffer(character);
        call Serial$Output(character);
        end;

    if Serial$Status then do;
        character = Serial$Input and 7FH;

/* We need time to deal with a LF */
        if character = lf then do;
            call Serial$Output(XOF);
            call Serial$Output(XOF);
/* Stop characters being sent to me, note that a few will be on the way...
Collect them...
*/
            save$buffer(0) = lf;
            save$buffer$ptr = 1;
            do time$out = 0 to 100;
               call time(2); /* 100 microseconds */
               if Serial$Status then do;
                    save$buffer(save$buffer$ptr) = Seral$Input;
                    save$buffer$ptr = save$buffer$ptr + 1;
                    time$out = 0; /* Reset the timeout */
                    end;
               end;
/* Get here once no characters are waiting. Send saved characters to screen */
            do saved = 0 to save$buffer$ptr - 1;
               call Console$Output(save$buffer(saved));
               end;
/* We have now caught up so.... */
            call Serial$Output(XON);

            end;
        ELSE call console$output(character);
        end;
    end; /* Do forever */
end Remote$Logon;
```

230879-001

```
:F1:RMRECV.PLM

$DEBUG
Receive: do;

/*    This is part of the REMOTE_LOGON program. This module is called if the
      remote computer is to receive a file from the Network.
*/

/*    Declare the variables used from Remote$Logon */
Declare buffer$ptr    byte external;
Declare buffer(128)    byte external;
Declare file$id byte external;

$nolist include(:f3:common.lit)
$nolist include(:f1:fileio.ext)
$nolist include(:f1:serial.ext)

Declare delimit(4) byte data ('FROM');

Declare (character, i, j, match, status) byte;
Declare (count, checksum, received$checksum, loop$count, end$buffer) byte;

Wait$for$serial$input: Procedure (no$time$limit) byte;
Declare (no$time$limit, character, time$out) byte;
    do time$out = 0 to 100;
/* Has the user aborted the command? */
        if Console$Status then do;
            character = Console$Input and 7FH;
            if character = CAN then call Exit;
        end;
        if Serial$Status then return Serial$Input;
        call time(2); /* 100 microseconds */
        if no$time$limit then timeout = 0;
    end;
    call Print$String(.(cr,lf,lf,'Serial line lost, Program aborted',
                        cr,lf,'$'));
    call Exit;
end Wait$for$serial$input;

buf$all$blanks: Procedure (Buf$ptr) byte;
/* Check to see if the remainder of the buffer is blanks. */
Declare buf$ptr address,
        (buf based buf$ptr) (1) byte,
        i byte;
    i = 0;
    do while (buf(i) <> lf) and (buf(i) = ' ');
        i = i + 1;
    end;
    if buf(i) = lf then return true;
    return false;
end buf$all$blanks;

Receive: Procedure public;
/*    Copy a file from the NDS-II to the Remote Computer */
```

230879-001

```
:F1:RMRECV.PLM

/*    Say HELLO */
  call Print$String(.(cr,lf,'Receive V2.3',cr,lf,'$'));

/*    Skip through the command line looking for Remote Computer filename */
    loop$count = 0;
    end$buffer = 0;
    /* Find the end of good characters in the buffer */
    do while buffer(end$buffer) <> lf;
        end$buffer = end$buffer + 1;
    end;
    if end$buffer < 14 then do;
        /* Not enough characters in buffer to even get started. Must have
           at least "RECV A FROM B<lf>", which is 14 characters.
        */
        call Print$String(.('Command syntax error. Correct format is :',cr,lf,
            'RECV <remote_file> FROM <NDS_II_file>',cr,lf,'$'));
        call Serial$Output(CAN);
        return;
    end;
    i = 5;   /* Skip over the recv command word */
    if buf$all$blanks(.buffer(i)) then do;
        /* Buffer passed the length requirement but is all blanks after
           the RECV command word.
        */
        call print$String(.('Command syntax error. Correct syntax is:',cr,lf,
            'RECV <remote_file> FROM <NDS_II_file>',cr,lf,'$'));
        call Serial$Output(CAN);
        return;
    end;

    do while buffer(i) = ' ';        /* Skip blanks before local file name */
        i = i +1;
    end;

/*    We have found the filename */
/*    Check if it exists */

    file$id = Create$File(.buffer(i));
    if file$id = 0ffh then do;
        call Print$String(.('Local File already exists',cr,lf,'$'));
        call Serial$Output(CAN);    /* Abort ISIS command */
        return;
    end;

/* Now that the file is good - see if FROM is in the command string */

do while buffer(i) <> ' ';     /* Skip the local file name */
        i = i + 1;
    end;
        do while buffer(i) = ' '; /* Skip blanks before <FROM> */
        i = i + 1;
    end;
    match = true;
    do j = 0 to 3;             /* Check for <FROM> in command string */
        if buffer (i+j) <> delimit(j) then match = false;
```

230879-001

```
:F1:RMRECV.PLM

    end;
    if match then do;
/*    File OK, activate the ISIS RECV command */
      call Serial$Output(cr);

/*    Skip passed CR,LF sent from ISIS. If no problems at ISIS end of the link
      then we will be sent a STX.
*/
      character = Wait$for$serial$input(true); /* CR */
      character = Wait$for$serial$input(true); /* LF */
      do forever;
Try$Again:  character = Wait$for$serial$input(true); /* STX? */
            if character = ETX then do;
                status = Close$File(file$id);
                if status = 0ffh then call Print$String(.(
                    'Local disk write-protected',cr,lf,'$'));
                else call Print$String(.(
                    cr,lf,'File transfer complete',cr,lf,'$'));
                return;
            end;
            if character <> STX then do;
                 return;
            end;

/*    ISIS is about to send us a buffer */
            checksum = 0;
            do i = 0 to 127;
                character = Wait$for$serial$input(false);
                buffer(i) = character;
                checksum = checksum + character;
            end;
            received$checksum = Wait$for$serial$input(false);
            character = Wait$for$serial$input(false); /* ETX */
            if checksum <> received$checksum then do;
                /* Checksum error - request retransmission */
                call Console$Output('?');
                call Serial$Output (NAK);
                goto Try$Again;
            end;
/*    Buffer received OK */
/*    Write to disk */
            call Console$Output(loop$count + '0');
            loop$count = (loop$count + 1) MOD 10;
            status = Write$Sector(file$id, .buffer);
            if status <> 0 then do;
                status = Close$File(file$id);
                call Print$String(.('Local disk full',cr,'$'));
                return;
            end;
/*   Buffer written to disk, Look for some more..... */
            call Seral$Output(ACK);
       end; /* Do forever */

/*    String did not match, keep looking */
```

230879-001

```
:F1:RMRECV.PLM

    end;

/*    Have now scanned the complete command line */
    call Print$String(.('Missing <FROM>. Correct syntax is:',cr,lf,lf,
                'RECV <LOCAL_FILENAME> FROM <NDS_II_FILENAME>',cr,lf, '$'));
/*    Abort ISIS command too */
    call Serial$Output(CAN);
    end Receive;
end Receive;


Page 1            :F1:RMSEND.PLM

$DEBUG
Send: do;

/*    This is part of the REMOTE-LOGON program. This module called if
      user is going to send a file from the remote computer to the
      NDS-II.
*/

/*    Declare the variables used from Remote$Logon */
Declare buffer$ptr    byte external;
Declare buffer(128)    byte external;
Declare file$id byte public;

$nolist include(:f3:common.lit)
$nolist include(:f1:fileio.ext)
$nolist include(:f1:serial.ext)

Declare delimit(4) byte data (' TO ');

Declare (character, i, match, status, count) byte;
Declare (checksum, received$checksum, loop$count) byte;

Send: Procedure public;
/*    Copy a file from the Remote Computer to the NDS 2 */

/*    Say HELLO */
    call Print$String(.(cr,lf,'Send V2.1',cr,lf,'$'));

/*    Skip through the command line looking for Remote Computer filename */
    loop$count = 0;
    do i = 0 to buffer$ptr;
        if buffer(i) = ' ' then do;
            do while buffer(i) = ' ';
                i = i + 1;
                end;
            file$id = Open$File(.buffer(i),read$only);
            if file$id = 0ffh then do;
                call Print$String(.('Local file does not exist',cr,lf,'$'));
                call Serial$Output(CAN):    /* Abort ISIS command */
                return;
                end;
/*    File OK, activate the ISIS SEND command */
            call Serial$Output (cr);
```

230879-001

```
:F1:RMSEND.PLM

/*    Skip passed CR,LF sent from ISIS. If no problems at ISIS end of the link
      then we will be sent a ACK.
*/
            character = Serial$Input; /* CR */
            character = Serial$Input; /* LF */
            character = Serial$Input; /* ACK? */
            if character <> ACK then do;
                call console$output(character);
                return;
            end;

/*    Get a buffer ready to send */
            status = Read$sector(file$id, .buffer);
            do while status = 0;
                call Serial$Output(STX);
                checksum = 0;
                do i = 0 to 127;
                    character = buffer(i);
                    checksum = checksum + character;
                    call Serial$Output(character);
                    end;
                call Serial$Output(checksum);
                call Serial$Output(ETX);

/*    Buffer sent OK. Was it received OK */
                character = Serial$Input;
                if character = ACK then do;
                    call console$output(loop$count + '0');
                    loop$count = (loop$count + 1) MOD 10;
                    status = Read$sector(file$id, .buffer);
                    end;
                else do;
                    call Console$Output('?');
                    status = 0; /* Retransmit */
                    end;
                end;
/*    File sent */
            status = Close$file(file$id);
            if status <> 0 then call Print$String(.(cr,lf,
                'Could not close Local file', cr,lf,'$'));
            cal Serial$Output(ETX);
            return;
            end;
        end;
    end Send;
end Send;
```

```
:F1:FILEIO.PLM

$debug
File$io: do;

/*    This version contains all of the fileio definitions for ISIS */

$nolist include (:f3:common.lit)
$nolist include (:f3:isis.ext)
declare file$id byte external;

/*    Operating system dependant Console Routines */
Console$Input: Procedure byte public;
    return ci;
end Console$Input;

Console$Output: Procedure (char) public;
    Declare char byte;
    call co(char);
end Console$Output;

Console$Status: Procedure byte public;
    return csts;
end Console$Status;

Print$String: Procedure (string$ptr) public;
    Declare string$ptr pointer,
            text based string$ptr byte;
    do while text <> '$';
        call co(text);
        string$ptr = string$ptr + 1;
    end;
end Print$String;

/*    Operating system dependant file routines */
Open$file: Procedure (file$ptr,mode) byte public;
/*    Return OFFH if file does not exist, otherwise return file ID */
    Declare file$ptr pointer,
            mode byte,
            (aftn, status) word;
    call rename(file$ptr, file$ptr, .status);
    if status = 13 then return OFFH; /* File does not exist */
    call open(.aftn, file$ptr, mode, 0, .status);
    if status = 12 then return file$id;   /* 12 returned if file already open
                                             want it open, so it's ok.
                                          */
    if status <> 0 then return OFFH;
    return low(aftn);
end Open$File;

Create$file: Procedure (file$ptr) byte public;
/*    Return OFFH if file already exists, otherwise return file ID */
    Declare file$ptr pointer,
            (aftn, status) word;
    call rename(file$ptr, file$ptr, .status);
    if status <> 13 then return OFFH; /* File already exists */
```

230879-001

```
:F1:FILEIO.PLM

    call open(.aftn, file$ptr, read$write, 0, .status);
    if status <> 0 then return 0FFH;
    return low(aftn);
    end Create$File;

Read$sector: Procedure (file$id, buffer$ptr) byte public;
    Declare file$id byte,
            buffer$ptr pointer,
            (buffer based buffer$ptr) (1) byte,
            (actual, status, i) word;
    call read(double(file$id), buffer$ptr, 128, .actual, .status);
    if status <> 0 then return 0FFH;
    if actual = 0 then return 0FFH;
    if actual <> 128 then do i = actual to 128;
        buffer(i-1) = ' ';
    end;
    return 0;
end Read$sector;

Write$sector: Procedure (file$id, buffer$ptr) byte public;
    Declare file$id byte,
            buffer$ptr pointer,
            status word;
    call write(double(file$id), buffer$ptr, 128, .status);
    return not (status = 0);
end Write$sector;

Close$file: Procedure (file$id) byte public;
    Declare file$id byte,
            status word;
    call close(double(file$id), .status);
    return not (status = 0);
end Close$File;

end fileio;
```

230879-001

```
:F1:SERIAL.PLM

$DEBUG
Serial$IO$for$iPDS: do;

/* This module contains all of the iPDS specific serial IO routines */

Serial$Status: Procedure byte public;
    return ((input(091H) and 2) = 2);
end Serial$Status;

Serial$Input: Procedure byte public;
    do while not Serial$status;
        /* Wait */
    end;
    return (input(090H));
end Serial$Input;

Serial$Output: Procedure (char) public;
    Declare char byte;
    do while ((input(091H) and 1) = 0);
        /* Wait */
    end;
    output(090H) = char;
end Serial$Output;

Serial$Control: Procedure (value) public;
    Declare value byte;
    output(091H) = value;
end Serial$Control;

end Serial$IO$for$iPDS;
```

SOURCE CODE FOR RECV AND SEND NETWORK FILES

```
:F1:RECV.PLM

recv: do;

/*    This is an ISIS utility ptogram for use with a Remote Computer. */

/*    This utility will run on an ISIS cluster board which is connected
      to a remote computer rather than a dumb terminal.
*/

$nolist include(:f3:common.lit)
$nolist include(:f3:isis.ext)

Declare buffer(128)    byte;
Declare (actual, status, aftn) word;
Declare (i, j, checksum, character) byte;

/*    Read the remainder of the command line */
call read(1, .buffer, 128, .actual, .status);
```

230879-001

```
:F1:RECV.PLM

/*    Is the requested ISIS file available */
j = 0;
do i = 0 to 2;      /* Skip "RECV <FILENAME> FROM" in the command word.
                       Need to get the file on the NDS-II system.
                       Don't need to check for syntax - it is already done
                       by the program on the remote computer.
                    */
    do while buffer(j) <> ' ';
        j = j + 1;
    end;
    do while buffer(j) = ' ';
        j = j + 1;
    end;
end;
call open(.aftn, .buffer(j), 1, 0, .status);
if status <> 0 then do;
    call write(0,.(cr,lf,' NDS-II file does not exist',cr,lf), 32, .status);
    call exit;
end;

/*    File is OK */
/*    Get the first buffer of information */
call read(aftn, .buffer, 128, .actual, .status);

do while actual <> 0;
/*    and send it */
    if actual <> 128 then do i = actual to 128;
        buffer(i-1) = ' ';
    end;
    call co(STX);
    checksum = 0;
    do i = 0 to 127;
        call co(buffer(i));
        checksum = checksum + buffer(i);
    end;
    call co(checksum);
    call co(ETX);

/*    Did the Remote Computer receive this OK */
    character = ci and 7FH;

    if character = EOT then call exit;        /* Remote error */
    if character = ACK then call read(aftn, .buffer, 128, .actual, .status);

/*    otherwise assume a transmission error and resend */
end;

/*    Arrive here when the complete file has been sent */
call close(aftn, .status);
call co(ETX);
call exit;

end recv;
```

230879-001

```
:F1:SEND.PLM

send: do;

/*    This is an ISIS utility ptogram for use with a Remote Computer. */

      This utility will run on an ISIS cluster board which is connected
      to a remote computer rather than a dumb terminal.
*/

$nolist include(:f3:common.lit)
$nolist include(:f3:isis.ext)

Declare buffer(128)    byte;
Declare (actual, status, aftn) word;
Declare (i, j, match, count, checksum, received$checksum, character) byte;
Declare delimit(4) byte data (' TO ');

Uppercase: Procedure (char) byte;
Declare char byte;
    if ((char >= 'a') AND (char <= 'z')) then return (char - 20h);
    return char;
end uppercase,

/*    Read the remainder of the command line */
call read (1, .buffer, 128, .actual, .status);

do i = 0 to actual-1;
    if buffer(i) = ' ' then do;
        match = true;
        do j = 0 to 3;
            if uppercase(buffer(i+j)) <> delimit(j) then match = false;
        end;
        if match then do;

/*    We have found the filename */
            i = i + 3;
            do while buffer(i) = ' ';
                i = i + 1;
            end;
            call open(.aftn, .buffer(i), 3, 0, .status)

/*    Did the file already exist? */
                call read(aftn, .buffer, 1, .actual, .status);
                if actual = 1 then do;
                    call write(0.,(cr,lf,:'NDS-II file already exists',cr,lf),
                    30, .status);
                    call exit;
                end;

/*    File is OK, tell Remote Computer to proceed */
                call co(ACK);

/*    Receive the first buffer of information */
                do forever;
                    character = ci;    /* STX or ETX */
```

230879-001

```
:F1:SEND.PLM

                if character = ETX then do;
                    call close(aftn,.status);
                    call write (0,.(cr,lf,'File transfer complete',cr,lf), 26,
                    .status);
                    call exit;
                end;
                checksum = 0;
                do i = 0 to 127;
                    buffer(i) = ci;
                    checksum = checksum + buffer(i);
                end;
                received$checksum = ci;
                character = ci;             /* ETX */
                if received$checksum <> checksum then call co(NAK);
                else do;
                    call write(aftn, .buffer, 128, .status);
                    call co(ACK);
                end;
            end;
        end;    /* No match, keep looking */
    end;
end;        /* End of line */
call write(0, .(cr, lf,CR,LF, 'Missing <TO>. Correct syntax is:',cr,lf,
        'SEND <local_filename> TO <NDS-II_filename>',cr,lf), 84, .status);
call exit;

end send;
```

230879-001

CHANGES TO SERIAL.PLM TO RUN ON A SERIES III

```
:F1:SERIAL.S2

$DEBUG
Serial$IO$for$SII: do;

/* This module contains all of the SII specific serial IO routines */

Serial$Status: Procedure byte public;
    return ((input(0F7H) and 2) = 2);
end Serial$Status;

Serial$Input: Procedure byte public;
    do while not Serial$status;
        /* Wait */
    end;
    return (input(0F6H));
end Serial$Input;

Serial$Output: Procedure (char) public;
    Declare char byte;
    do while ((input(0F7H) and 1) = 0);
        /* Wait */
    end;
    output(0F6H) = char;
end Serial$Output;

Serial$Control: Procedure (value) public;
    Declare value byte;
    output(0F7H) = value;
end Serial$Control;

end Serial$IO$for$SII;
```

230879-001

CHANGES TO FILEIO.PLM AND REMOTE.MKE TO RUN UNDER CPM

```
:F1:FILEIO.CPM

$DEBUG
CPM$Interface$Library: do;

/* This module contains all of the definitions for FILEIO.EXT for CPM80 */

$nolist include(:f3:common.lit)
$list

Declare bdos$jump address data ( 5 ); /* Set up the address of where to
       go in memory to get to the CPM BDOS routines. This is done
       by using a call by address with parameters of which bdos routine
       and parameters to the routine. This is a clumsy way to do it because
       there is no way to read the return value of the routine. So
       as you will see in the procedure bdos, how the compiler is fooled into
       generating the asm code for a return value.
    */

/* PLM80 Declarations for CPM80 functions */
BDOS: Procedure (type, parameter) byte;
    Declare type byte,
            parameter word;
    if 1 = 2 then return 1; /* Let pass 2 of the compiler see a return for the
                               typed procedure. Then pass 3 will see 1 is
                               never = 2, so it will throw out the statement.
                               Now bdos has put the return value in the Acc.,
                               and the calling procedure that called this
                               procedure will get it out of the Acc.
                               Real clumsy but works.
                            */
    call bdos$jump (type,parameter);
end BDOS;

/* Some BDOS calls return a word; to conform to good PLM syntax we use:....*/
BDOSW:Procedure (type, parameter) word;
    Declare type byte,
            parameter word;
    if 1 = 2 then return 1;
    call bdos$jump (type,parameter);
end BDOSW;

/* Some BDOS calls return nothing; to conform to good PLM syntax we use:....*/
BDOSN:Procedure (type, parameter);
    Declare type byte,
            parameter word;
    call bdos$jump (type,parameter);
end BDOSN;

/*    In case of fatal errors */
Exit: Procedure public;
    call BDOSN(0,0);
    end Exit;
```

230879-001

```
:F1:FILEIO.CPM

/*    Operating system dependant Console Routines */

Declare waiting$char byte; /* This is the buffer for the CO,CI and
                              CSTS BDOS routines */
Console$Status: Procedure byte public;
    waiting$char = BDOS(6,0FFH);
    IF waiting$char = 0 THEN return false;
    return true;
end Console$Status;

Console$Input: Procedure byte public;
/* One problem with the ISIS to CPM BDOS conversion is in the console
   inputs, ISIS doesn't echo and CPM does. So, using the CPM
   BDOS direct console I/O to get around the echo problem.
*/
Declare dummy byte;
    do for$ever;
        IF waiting$char <> 0 THEN return waiting$char;
        dummy = console$status;
    end;
end Console$Input;

Console$Output: Procedure (char) public;
    Declare char byte;
    call BDOSN(6,double(char));
end Console$Output;

Print$String: Procedure (string$ptr) public;
    Declare string$ptr pointer;
    call BDOSN(9,string$ptr);
end Print$String;

/*    Operating system dependant file routines */
Declare FCB$free(6) byte initial (1,1,1,1,1,1);
Declare FCB(6) structure (item(36) byte);
Declare Blank$FCB(36 byte data (1,'           ',0,0,0,0,
                                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                0,0,0,0);

Get$FCB: Procedure byte;
    Declare i byte;
    do i = 0 to 5;
        if FCB$free(i) then do;
            FCB$free(i) = false;
            return i;
        end;
    end;
    call Print$String(.(cr,lf,'Too may files open, Program aborted',cr,lf,'$'));
    call exit;
end Get$FCB;

Format$FCB: Procedure(index, file$ptr);
    Declare (index, 1) byte,
            file$ptr pointer;
    Declare (text based file$ptr) (1) byte;
```

230879-001

```
:F1:FILEIO.CPM

/*Clear all the required fields */
    call move(36,.blank$FCB,.FCB(index));
/*    Skip over any leading spaces */
    do while text(0) = ' ';
        file$ptr = file$ptr + 1;
    end;
/*    Has a drive been specified ? */
    if text(1) = ':' then do;
        FCB(index).item(0) = text (0) - 'A' + 1;
        file$ptr = file$ptr + 2;
    end;

    file$ptr = file$ptr - 1;
    do i = 1 to 11;
        if text(i) = ' ' then return;
        if text(i) = cr then return;
        if text(i) = '.' then do;
            file$ptr = file$ptr + i - 8;
            i = 8;
        end;
        else FCB(index).item(i) = text(i);
    end;
end Format$FCB;

Set$DMA$Address: Procedure (value);
    Declare value word;
    call BDOSN(26, value);
end Set$DMA$Address;

Select$disk: Procedure (disk);
    Declare disk byte;
    call BDOSN(14,double(disk));
end Select$disk;

Open$file: Procedure (file$ptr,mode) byte public;
    Declare mode byte,
            (file$ptr, FCB$ptr) pointer,
            (index, status) byte;
    index = get$FCB
    call format$FCB(index, file$ptr);
    call Select$disk(FCB(index).item(0)-1);
    FCB(index).item(0) = 0;
    status = BDOS(15,.FCB(index));
    if status = 0FFH then return 0FFH;
    return index;
end Open$File;

Create$file: Procedure (file$ptr) byte public;
    Declare (file$ptr, FCB$ptr) pointer,
            (index, status) byte;
    index = get$FCB;
    call format$FCB(index, file$ptr);
    call Select$disk(FCB(index).item(0)-1);
    FCB(index).item(0) = 0;
    status = BDOS(22,.FCB(index));
```

230879-001

```
:F1:FILEIO.CPM

    if status = 0FFH then return 0FFH;
    return index;
end Create$File;

Read$sector: Procedure (file$id, buffer$ptr) byte public;
    Declare file$id byte,
            buffer$ptr pointer;
    call Set$DMA$Address(buffer$ptr);
    return BDOS(20,.FCB(file$id));
end Read$sector;

Write$sector: Procedure (file$id, buffer$ptr) byte public;
    Declare file$id byte,
            buffer$ptr pointer;
    call Set$DMA$Address(buffer$ptr);
    return BDOS(21,.FCB(file$id));
end Write$sector;

Close$file: Procedure (file$id) byte public;
    Declare file$id byte;
    FCB$free(file$id) = true;
    return BDOS(16,.FCB(file$id));
end Close$File;

end CPM$Interface$library;
```

```
:F1:REMOTE.CPM

; Make file for the ISIS REMOTE program that runs on the iPDS system.
; Author : B. Valentine DSSO Applications Engineering 6/25/83

; First of all define the macros for the MAKE file.
;  Define the subsitution macros :
;   Subsitution macros are used as constant defines. This way, if
;    a major change is made, such as the source code device changes
;    from :f1: to :f2:, the only update to the make file is to change
;    the macro define.

$    SET work_device        to ':f1:'
$    SET 8_bit_exe          to ':f2:'
$    SET 8_bit_lib          to ':f3:'
$    SET database           to ':f4:remote.db'
$    SET svcs_drive         to 'run :f5:'

$    SET plm                to '%"8_bit_exe"plm80'
; Note how macros may be nested and the macro is used with the %'<name>".
$    SET locate             to '%"8_bit_exe"Locate'
$    SET link               to '%"8_bit_exe"link'
$    SET syslib             to '%"8_bit_lib"system.lib'
$    SET plmlib             to '%"8_bit_lib"plm80.lib'
$    SET comlit             to '%"8_bit_lib"common.lit'
$    SET get                to '%"svcs_drive"svcs get %database'
$    SET put                to '%"svcs_drive"svcs put %database'

; Now define the enumeration macros:
$    SET nds2_files         to (recv,send)
$    SET remote_files       to (remote,fileio,serial,rmrecv,rmsend)
$    SET files              to (%all(%nds2_files),%all(%remote_fiels))

; Tell make that we are going to be looking at the files in the database.
$ FOR i in %files
$    svcs %work_device%"i".plm   = %database (%i)
$    svcs %work_device%"i".obj   = %database (%i,,oj)
$END
$ svcs %"work_device"serial.ext  = %database (serial,,cp)
$ svcs %"work_device"fileio.ext  = %database (fileio,,cp)
$ svcs %"work_device"remote      = %database (exec,,oj)
$ svcs %"work_device"send        = %database (send,,cp)
$ svcs %"work_device"recv        = %database (recv,,cp)

; The include files are always required, so get them with the header.
$ HEADER
; Get all the externals and include files from the database
   %get (serial,,cp)     to %"work_device"serial.ext
   %get (fileio,,cp)     to %"work_device"fileio.ext
$ END

; Now start the dependecies

; Set the dependency tree for three separate executable files.
$ IF all >"work_device" remote, %all(%work_device%nds2_files) THEN
$END
```

230879-001

```
:F1:REMOTE.CPM

$FOR i IN %remote_files
; Build all the object files in the remote program.
$    IF %work_device%"i".obj > %work_device%"i".plm, %comlit,
$    %"work_device"fileio.ext, %"work_device"serial.ext THEN
        %get (%i) to %work_device%"i".plm
        %plm %work_device%"i".plm
        %put (%i,,oj) from %target
$    END
$END

; Check the remote executable file that runs on the iPDS system.
$ IF %"work_device"remote > %all(%work_device%"remote_files".obj),
$ %plmlib, %syslib THEN
$    FOR i in %remote_files
        %get (%i,,oj) to %work_device%"i".obj
$    END
    %link %depend to %"work_device"remote.lnk
    %locate %"work_device"remote.lnk code(103H) symbols lines &
          map print(%work_device"remote.map)
        %put (exec,,oj) from %target
$ END

; Now that the remote program has been checked, check the two programs
; that run on the network.

$FOR i IN %nds2_files
; Check the NDS_II files RECV and SEND.
$    IF %work_device%i > %work_device%"i".plm THEN
        %get (%i) to %depend
        %plm %depend
        %put (%i,,oj) from %work_device%"i".obj
        %link %work_device%"i".obj, %syslib. %plmlib to %work_device%"i".lnk
        %locate %work_device%"i".lnk
        %get (%i,,cp) to :bb: write
        %put (%i,,cp) from %target
$    END
$END
```

230879-001

August 1982

Debugging Catches up with High-Level Programming

Stuart Vannerson
Electronic Design, June 24, 1982

Order Number: 210671-001

Software debugging at the statement and procedure level gives a high-level view of programs from creation to implementation.

Debugging catches up with high-level programming

Although high-level languages for microcomputers have made software design a state-of-the-art procedure, debugging technology has lagged behind. A high-level program debugger brings that technology up to date. By allowing users to monitor and scrutinize PL/M-86, Pascal-86, and Fortran-86 programs at the source level, it addresses some of the key problems faced by high-level language programmers.

The debugger, called Pscope, offers three major improvements over conventional tools:

- High-level debugging at the source statement and procedure level, in addition to the machine level.
- A powerful, reliable code-patching facility, which reduces editing-compiling-linking cycles.
- Symbolic access to all aspects of a user's program, including complex data structures, user-defined data types, dynamic variables, and numerics.

In the past, when microprocessor designs were primarily replacements of simple configurations that used logic gates, the software part of an application was usually written in assembly language. It made perfect sense to debug the application at the machine level, using in-circuit emulators, simulators, logic analyzers, and other discrete tools that worked at the CPU level. However, the increasing size and complexity of microprocessor software has generated a new set of requirements for program debugging.

Although most microprocessor applications today are programmed in high-level language, they employ the debugging tools used for assembly-level programs. In fact, most debuggers reduce high-level language programs to assembly-language equivalents, making debugging more difficult than programming.

An 8086-based software program, Pscope runs on a Series III microcomputer development system, along with the user's program being debugged. (It will be used as the software executive for future in-circuit emulators, to combine the benefits of high-level debugging with real time emulation.) Pscope's main advantage is that the user's view of the program during debugging is the same as during its implementation. Stepping, break-pointing, and tracing execution flow are performed on high-level constructs such as statements, procedures, and labels.

Stuart Vannerson, Software Product Manager
Intel Corp.
3065 Bowers Ave., Santa Clara, Calif. 95051

Tracking down bugs

The first thing a designer does once a program has been created, compiled, and linked for execution is to run it. What usually happens is that, due to some logic error, a program takes an incorrect branch and winds up executing in a place it is not supposed to. The designer's first inclination is to find out where that occurred and why.

This is where it is helpful to have some form of trace command. An emulator lets the designer examine the contents of a trace buffer, which gives the past 100 or so CPU instructions executed, plus other information. It even allows disassembly of the contents of the trace buffer. However, if the program went off into some infinite loop, the trace buffer will be filled with just those isolated addresses, and the place where the incorrect branch occurred will have been lost.

The trace facility within Pscope allows setting of trace points at high-level source statements, procedures, and labels. By putting a trace point on each procedure call, for example (as opposed to each CPU instruction), a programmer can look at the trace contents and see exactly the sequence of calls that led to the incorrect branch.

As an example of Pscope's trace capability, consider a program that takes a numeric expression, parses it into tokens, and evaluates it (Fig.1). By selecting different combinations of its 11 procedure calls to trace, the programmer can change the "granularity" of trace information. While parsing the numeric expression 23 + (19-5*3), and tracing 3, then 7, then all 11 pro-

cedures, Pscope generates first 12, then 23, then 74 trace messages, respectively. Tracing CPU instructions, although providing finer granularity, would be much less meaningful in this case.

A programmer debugging at the machine level might try single-stepping to find an incorrect branch. However, just like execution tracing, single stepping is done at the machine level. Working at this level is acceptable when the location of the bug has been determined, but offers little help in finding it. It would take several thousand steps to go through the parsing program in the previous example.

With Pscope, single stepping (like tracing) is done at the source level, using high-level statements and procedures. The LSTEP command executes a program one high-level statement at a time. The PSTEP command does the same, only it treats procedure calls as if the whole procedure were a single statement. In the example (Fig. 2), it takes five PSTEPs to step through the program. In this case, the procedures Sum, Difference, and Maxim were each executed with one PSTEP. If LSTEP were used, the procedures themselves would have been stepped through, and it would have taken 19 LSTEPs. A CPU-level debugger, however, would have stepped through each of the program's 177 machine instructions. It also would have stepped through the run-time routines that were linked in with the program and with the operating system, too. Pscope, in contrast, can differentiate between the user's program module and the run-time routines,

```
SERIES-III Pascal-86, V2.0

  1    1  0  0        program dc (input, output);

                        procedure error(e : error_class); (* print error & restart *)
 40   63  1  1        end (* error *);

                      procedure get_line; (* input line & set c to 1st char of line *)
 60   85  1  1        end (* get_line *);

                      procedure get_token; (* scan line & set t to its value *)

 62   90  1  0          function digit(c: char): boolean; (* true if c is a digit *)
                        end;

 64   95  1  0          function upper_case(c: char): boolean; (* true if c is upper case *)
                        end;

 66  100  1  0          function lower_case(c: char): boolean; (* true if c is lower case *)
                        end;

                        procedure get_char; (* set c to next char in line *)
 79  119  2  1        end (* get_char *);

                      begin (* get_token: scan line & set t to its value *)
135  170  1  1        end (* get_token *);

                      procedure factor(var factor_value : integer);
139  177  1  0        begin (* factor *)
168  202  1  1        end (* factor *);

                      procedure term(var term_value : integer);
173  210  1  0        begin (* term *)
188  223  1  1        end (* term *);

                      proceuure expression (var expression value : integer);
193  231  1  0        begin (* expression *)
221  254  1  1        end (* expression *);

                      procedure statement;
224  260  1  0        begin (* statement *)
228  264  1  1        end (* statement *);
```

(a)

```
229  267  0  0      begin (* main program *)
                      repeat (* forever *)
232  278  0  2          get_line;
233  279  0  2          get_token;
234  280  0  2          statement;
235  281  0  2        until false;
238  287  0  1      end.
```

AFN-02228A

eliminating a lot of tedious or wasted effort.

All of Pscope's commands allow users to examine and manipulate a program (both code and data) using the same symbolic constructs in which the program was developed. Breakpoints, like trace points, may be placed at procedure calls, procedure returns, statements, and labels. Thus, to debug a program with Pscope, all a programmer needs is a listing. Linkage maps, memory dumps, locate maps, and the like are unnecessary. In addition, the number of high-level breakpoints (and trace points) is unlimited with Pscope. On the other hand, since emulators use hardware breakpoints, they are limited to just a few.

Now suppose that the programmer has determined the specific procedure where the program took an incorrect branch. The next job is to find out why. Jumps and calls are usually based on the outcome of a particular condition—if the condition evaluates true, the program goes one way; if false, another. The conditions are often complex, however; they may involve several Boolean expressions and elements of some obscure data structures.

To track down the cause of the bug, a programmer begins examining the contents of these data elements, compares them with what he or she expected them to be, and moves a step further. This procedure usually involves stepping a statement at a time and looking at data values.

Unfortunately, the traditional low-level debuggers provide only symbolic access to variables of some basic

```
*load :f1:dc6.86
*
*
*dir procedure
DIR of :DC
ERROR
GET_LINE
GET_TOKEN
GET_TOKEN.DIGIT
GET_TOKEN.UPPER_CASE
GET_TOKEN.LOWER_CASE
GET_TOKEN.GET_CHAR
FACTOR
TERM
EXPRESSION
STATEMENT
*
*
```

(b)

```
*define trcreg t1 = error,get_line,get_token
*define trcreg t2 = factor,term,expression,statement
*namescope = get_token
*define trcreg t3 = digit,upper_case,lower_case,get_char
*
*
```

(c)

```
*go using t1 til get_line
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[Break at get_line]
*
*
```

(d)

```
*go using t1,t2 til get_line
[At get_token]
[At statement]
[At expression]
[At term]
[At factor]
[At get_token]
[At get_token]
[At get_token]
[At term]
[At factor]
[At get_token]
[At expression]
[At term]
[At factor]
[At get_token]
[At get_token]
[At term]
[At factor]
[At get_token]
[At get_token]
[Break at get_line]
*
*
```

(e)

```
*
*go using t1,t2,t3 til get_line
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At digit]
[At get_char]
[At digit]
[At get_char]
[At digit]
[At statement]
[At expression]
[At term]
[At factor]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At get_char]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At get_char]
[At term]
[At factor]
[At get_token]
[At lower_case]
```

```
[At upper_case]
[At digit]
[At digit]
[At get_char]
[At digit]
[At get_char]
[At digit]
[At expression]
[At term]
[At factor]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At get_char]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At get_char]
[At digit]
[At term]
[At factor]
[At get_token]
[At lower_case]
```

```
[At upper_case]
[At digit]
[At get_char]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At digit]
[At get_char]
[At digit]
[At factor]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[At get_char]
[At get_token]
[At lower_case]
[At upper_case]
[At digit]
[Break at get_line]
*
*
*
```

(f)

1. Pscope, a high-level program, allows different levels of "granularity" in high-level debugging. The program (a) contains 11 procedures, displayed (b) in their nested form with the DIR command. Three trace registers with different trace points in them have been defined (c). Executing the program with the trace points of t2 prints out 12 trace messages (d). Tracing more of the procedures during execution displays more trace messages—23 and 71 (e and f). A low-level debugger would have traced thousands of CPU instructions, providing a lot of unnecessary data and probably overflowing the capacity of the trace buffer.

AFN-02288A

program types. Complex structures, user-defined data types, stack-based dynamic variables, and numerics all are examples of data that requires more complex handling. For example, to access a field within a record using the ICE-86A emulator, users must give the byte offset from the beginning of the record (e.g., user_rec + 14). On the other hand, Pscope allows the designer to access fields by name, for example, user_rec. age. Representation of floating-point numbers requires binary-to-decimal conversion, a luxury many debuggers leave off. Pscope lets a designer examine and modify real numbers at three precision levels, providing conversion from binary into decimal back into binary. Examination and modification of all data is done symbolically in Pscope.

The advantage of all this is that data references are easier, and fewer mistakes are made (the designer does not have to calculate offsets). Thus the process of stepping, looking at data, evaluating expressions, and continuing are speeded up. In other words, bugs are tracked down faster.

Fixing bugs

Tracking down the location of bugs quickly is only half the battle. Correcting the problem is the other, time-consuming half.

For large applications the program-change cycle can be lengthy. Program changes are made with an editor; then the source is recompiled and the module linked with the run-time routines and other modules. Since programs can initially contain a lot of bugs, going through an editing-compiling-linking cycle for each bug can become extremely wearing after a while.

```
*pstep
[Step at :EXAMP#21]
*pstep
            INPUT TWO INTEGERS:
[Step at :EXAMP#22]
*pstep
                (input)  319 46
[Step at :EXAMP#23]
*pstep
                THE SUM IS  365
[Step at :EXAMP#24]
*pstep
          THE DIFFERENCE IS  273
[Step at :EXAMP#25]
*pstep
             THE MAXIMUN IS  319
             THE MINIMUM IS  46
[Step at :EXAMP#21]
*
```

2. This program illustrating the stepping features of Pscope, contains five statements in the main body, three of which are procedure calls. Five procedure steps are required to go through the program. It would have taken 19 statement-level steps, as each procedure would have been stepped though. In contrast, a CPU-level debugger would have stepped through all 177 instructions, as well as through the run-time system.

Programmers typically go to great measures to avoid such a necessity. Instead, they often try to patch the object module, in order to continue debugging.

Patching object code, however, may be difficult for a variety of reasons. First of all, the desired patch must be written in hex code or assembler mnemonics. Those debuggers that disassemble object code usually offer a line-by-line assembler as well. Patching a high-level program at the machine level can be a horrendous mess, though. The high-level language compiler may have adopted certain stack conventions, code optimizations, and register usage that make it difficult to understand what to patch, let alone how to patch it.

The patch is frequently a different size than the code to be patched, and that introduces more complications. An unfortunately common solution is to jump to an unused location, perform the patch, and jump back to the return address. Another problem is that even though the machine-level patch may work, incorporating the change into the source file later may generate entirely different code from that of the patch. Because of all these complications, patches are used only in simple cases, where programmers can easily determine the results. It is unfortunate, too, because a good patching mechanism could eliminate a lot of programmers' headaches.

In lieu of machine-level patching, the common methodology is to set a breakpoint at the location of the bug and correct the problem by hand. Correcting the problem usually means reassigning variables or reversing the outcome of some IF... THEN conditions. These methods are simpler than patching but introduce problems of their own: If the bug is located inside a loop, the "breakpoint and change by hand" approach must be done too frequently. Also, if the manual changes are more than a few assignments, the process becomes tedious. Lastly, only a few bugs can be changed in this fashion before it becomes difficult to keep track of them. As a result, programmers quickly resort back to extensive editing-compiling-linking cycles.

Pscope's approach to the problem is to provide an automated way of writing and managing high-level code patches. Rather than define changes to the program at the machine level, users may define code patches at statement numbers. With Pscope, the actual contents of the patch may be complex as well—DO... END blocks, IF... THEN... ELSE conditions, and REPEAT... WHILE... UNTIL loops make the command language as powerful as Pascal or PL/M.

Using high-level code patches is simple (Fig.3). After determining the location and cause of a particular bug, the programmer defines a patch. In this example, a multiplication took place at line 5, rather than an addition. The command define patch #5 til #6 = z=x+y causes the contents of the patch to be executed in place of the statement at line 5. The designer can

AFN-02228A

specify any starting point and any point to continue execution. Furthermore, patches are executed in all GO, LSTEP, and PSTEP commands without having to specify them. Perhaps the biggest help in managing patches is that it is easy to see where they are (DIR PATCH); in addition, they may be written out to disk (PUT filename PATCH). Thus, it becomes very easy to incorporate them into the source file later. Because the patch language is so similar to the source language, a patch that worked in Pscope is most likely to work in the modified program as well.

Many utility commands will be part of most simple debugging sessions. Pscope's "literally" feature allows users to abbreviate, redefine, and tailor the command language to suit their needs. For example, define literally d = 'define' lets the programmer use d for the define command. The HELP command displays (on the console) the usage and syntax of most commands and facilities in Pscope. The PUT and INCLUDE commands let users write and retrieve commands (usually definitions of break and trace registers, program patches, and "literally's") on disk, to use in later Pscope debugging sessions.

Pscope's command language is a powerful progamming language that is used for generating new commands (debugging procedures), the same way high-level code patches are defined. Debugging procedures allow you to define compound and conditional commands. Like procedures in high-level language, these procedures may have parameters, may supply return values, and may have their own local variables. Thus Pscope is in fact a programming language in its own right.

Debugging procedures may be called automatically upon reaching a breakpoint or a trace point. When a breakpoint is reached, Pscope can call a procedure that contains any sequence of Pscope commands. Conditional break and trace points may be set up this way. By evaluation on condition in the procedure, a return value of "true" or "false" determines whether the break (or trace message) should take place or not.

Debugging procedures (and code patches) are created with Pscope's built-in editor and may be stored on disk. The editor is a menu-driven, CRT-oriented program that is used to edit not only debugging procedures, but command lines as well. For example, when a syntax error occurs on a long command line, the user just hits < esc > on the keyboard and the editor comes up. The command will then be reexecuted when it is corrected.

The command language has conditional constructs (IF . . . THEN . . . ELSE), looping control (REPEAT . . . WHILE . . . UNTIL . . . COUNT), calls and returns for procedure nesting, and a full set of program data types. The data types correspond to the recognized types of the user's program (PL/M and Pascal). Debugging procedures can also access user-program variables (for example, debug_variable = prog_count + 1).

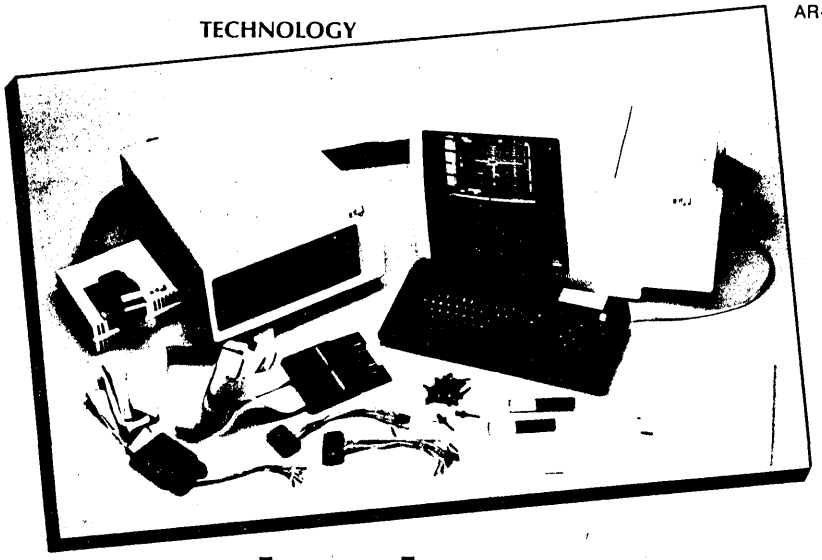
These procedures also allow program stubs to be expanded. Rather than resolve external program references with fully coded modules, subsystems can use empty stubs for resolving externals. During debugging, then, a procedure can be used to supply input values, take outputs and process them, supply return values and conditions, and so on. In essence, procedures can be set up so that all or part of a software system is modeled. Such flexibilty affords much greater independence in software implementation, as separate software modules can be developed and then debugged independently.

Lastly, debugging procedures can be used to automate the software testing process. Complete (or incomplete) systems may be executed over and over again, each time with new parameters and each time recording the results. The parameters can be selected by the designer or derived algorithmically using debugging procedures.□

```
*load :f1:maxmin.86
*
*
*go til #21
[Break at :EXAMP#21]
*go
             INPUT TWO INTEGERS: 19 4
                 THE SUM IS   76
          THE DIFFERENCE IS   15
             THE MAXIMUN IS   19
             THE MINIMUM IS    4
[Break at :EXAMP#21]
*
*
*  /*  looking at statement #5 in
**     the program, notice we multiplied
**     instead of added.  Let's patch it   */
*
*
*define patch #5 til #6 = z=x+y
*
*go
             INPUT TWO INTEGERS:  19 4
                 THE SUM IS   23
          THE DIFFERENCE IS   15
             THE MAXIMUN IS   19
             THE MINIMUM IS    4
[Break at :EXAMP#21]
*
```

3. High-level code patching can fix the bug in statement 5, which calls for multiplying, instead of adding, two integers. A patch is defined to replace this statement, and the program now executes correctly. Patches remain active during all LSTEP, and PSTEP, and GO commands until the patch is removed.

AFN-02228A



Integrated software-development instrumentation *can significantly reduce the development costs involved in bringing a product to market. The Intel I²ICE system comprises an in-circuit emulator for 16-bit microprocessors, a logic-timing and state analyzer and a high-level language debugger connected to a host development computer.*

Software development

PAUL MARITZ, Intel Corp.

New tools and approaches are boosting software-development productivity

The past year has seen a transformation in software development for microprocessor systems, involving more sophisticated processors, increased software content in the end product and a growing shortage of software talent. The integration of common human interfaces across heterogeneous systems, coupled with a tremendous focus on "friendly" and "productivity-based" features and the incorporation of classic hardware tools, such as in-circuit emulators, into the software-development environment have changed the very structure of the software lab. While in 1982 the concept of an integrated workstation combined an emulator with a logic analyzer, in 1984 an integrated workstation will combine software tools with hardware assistance to boost software productivity.

The cost to a company of a malfunctioning or poorly designed product can prove far more expensive than doubling its R&D expenditures or absorbing a significant increase in the product's cost. This is equally true for the software-development process. "Time to market is everything," and this consideration will become significantly more important over the next few years.

Increasing software productivity

During 1984, changes in computer systems will continue the evolution outlined above. Software tools will become available to guide the documentation and building of software systems, hardware will help software engineers evaluate software "completeness," and performance analysis and high-quality local-area networks (LANs) will be pervasive in every medium and large software environment. Just as logic analyzers, oscilloscopes and emulators have assisted hardware engineers, documentation aids, very high-performance distributed processing and the adaptation of emulators to software-intensive environments will lead the way to a greater measure of software productivity in the mid-1980s.

The key to software productivity lies in minimizing—or eliminating—a focus on learning to use the tools and maximizing the development and convenience of common human interfaces, high-level software tools and automated documentation and software-version control. No matter how well each individual tool works in and of itself, the effectiveness of the design aids available to the software engineer depends more on the interaction and interdependence of each tool than on any one tool's features.

The single most time-consuming task in the software-development cycle (Fig. 1) is verifying that the software works—that is, detecting and correcting bugs. One reason this process is so inefficient is that debugging is done at a low level. Most programs today are written in a high-level language. A software

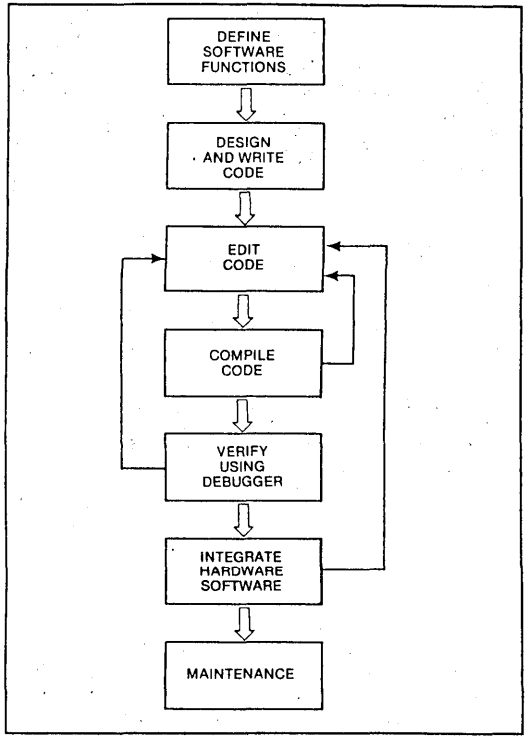


Fig. 1. In a typical software-development cycle, *problems in compiling the code, verifying it with a debugger and integrating hardware with software send a project back to the editing stage. Problems become more difficult to correct as development proceeds and particularly difficult to rectify after hardware is involved. A project that fails to use the appropriate tools throughout its life cycle risks slipping all the way back to the definition and design/writing phase.*

engineer uses a language translator that translates high-level terms and constructs that are closer to an application into low-level or processor-specific terms. For example, a programmer might write his program in Pascal, considering such entities as procedures, records and expressions. However, when he detects a bug in his program, he is forced by the available tools to operate at the processor level and to deal with such entities as hex numbers, registers and flags. Because the programmer has to translate manually back from a low level to a high level, productivity is lost.

Implementing high-level debugging

Why not, instead, have the debugging tool do a reverse translation? After all, the programmer submitted the high-level description (source code) of his program to a translator (compiler) to have it converted into low-level, processor-specific (object) code. The compiler could pass information about the program to the debugger, so that it could present the software engineer with information about the program in high-level terms (Fig. 2). This is an example of a human interface that is efficient, not just friendly.

The cost to a company of a malfunctioning or poorly designed product can prove far more expensive than doubling its R&D expenditures or absorbing a significant increase in the product's cost.

In microprocessor development, it is often necessary to complete the verification of the software by running the program in the target environment—the real-world environment of the application to be controlled. This is

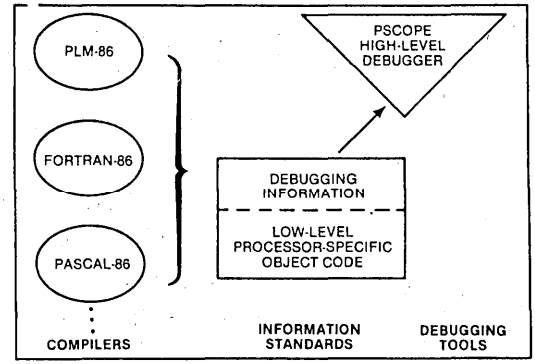


Fig. 2. Using a high-level debugger, *such as Intel Corp.'s PSCOPE, in the software-development process allows a developer to correct problems with code in a high-level language instead of in low-level, processor-specific terms. Such tools can greatly increase the efficiency of the debugging process.*

usually a necessary step because the interaction of the microprocessor and its external environment might be very complex and exceedingly difficult to simulate. For example, consider a microprocessor controlling a robot arm. The microprocessor must receive instructions on where to move the arm and, at the same time, monitor sensors reporting the state of the motors driving the arm. These inputs arrive in an unpredictable sequence and must be serviced within certain time limits if the robot arm is to perform as expected.

Simulating such an environment would be as much trouble as writing the target program. The ideal approach therefore involves simulating only those program modules that have a well-defined and simple input and output sequence and hence can be debugged easily in a simulated environment. The complete program, with its complex, time-dependent inter-

module relationships, can then be debugged in the actual target environment.

This two-stage approach requires two types of related debuggers: a software debugger that allows the software developer to simulate modules on his workstation and an in-circuit emulator that allows him to debug the software running in the target environment. To be most effective, these two types of debuggers should share the same human interface (Fig. 3), permitting an engineer to move easily from module-level simulation to in-target debugging without mentally shifting gears.

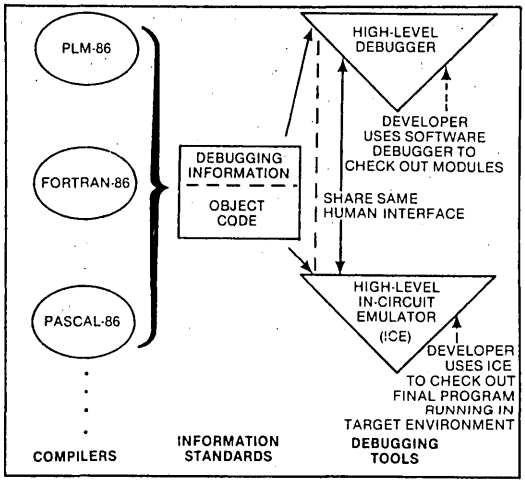


Fig. 3. High-level software and hardware debuggers *(yellow) sharing the same human interface speed software development. A developer can use a high-level debugger to exercise a software module on a workstation before all the modules of a program are available or before a prototype target system is constructed. When all modules and the prototype are ready, an in-circuit emulator, such as Intel's 12ICE, can be employed to debug the code running in real time in its real-word environment. Using a variety of compilers allows the developer to choose the optimum language for each sub-task and, in many cases, to use off-the-shelf software components written in a standard language.*

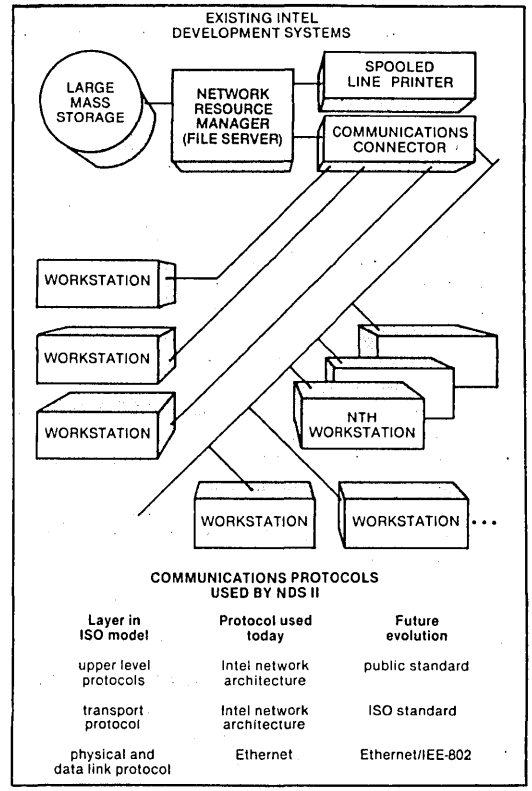


Fig. 4. An LAN can integrate shared and dedicated software-development resources. *A shared, central database allows the storage and management of project information. Dedicated, single-user workstations provide team members with processing power, large memory space and quick response. An LAN linking individual workstations furnishes communications between software developers and common access to database information through a network resource manager. An efficient LAN, such as Intel's NDS-II, will eventually be able to connect workstations from different manufacturers to serve changing software-development needs. This goal, however, requires further standardization of communications protocols, the aim of the International Standards Organization (ISO) and other groups.*

Managing software development

Typically, programmers generate at least three classes of information: documentation, source and object (processor-specific) code. More than one individual usually generates the information produced by a development project. In addition, the information usually undergoes changes over time, resulting in several different versions of the software. Furthermore, a typical project involves many variants of the information produced, such as one for floppy disks or one for Winchester disks.

On a multiengineer software-development project, the management of these different levels of information can become problematic. And, although the cause of the problems is usually simple, their effect is very costly. An engineer might waste days building a test system with an out-of-date document. Another problem that frequently arises is that of a "mysterious" change—an engineer changes a module and then fails to notify others of the modification.

Although these are simple management problems, a week lost on a 10-man engineering project because of an incorrect source file can mean $20,000 to $30,000 in direct staff costs and a serious slip in the product-development schedule.

Solving development problems

Automating software-management procedures can solve these types of problems by providing project members with a database in which to hold and control project information. It can also furnish the software-generation tools needed to build the correct software

systems from information held in the database (see "Automating software management," below).

A project database should be able to provide:

- automatic separation of information according to type, version and variant. For example, a user must be able to extract from the database "the source associated with module X, version 2—the floppy disk variant" or "the test data for module Y, version 3."
- control of access to information. A software developer must be able to lock, or "freeze," certain versions to prevent problems arising from mysterious changes to the base information.
- a guaranteed audit trail for all information—what changes were made, by whom, why and when—making it easier to track the changes made in going from "version 2.0 to version 3.0."

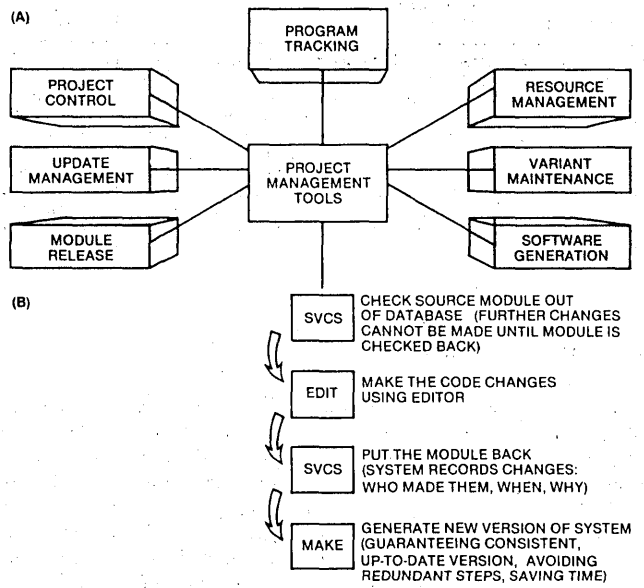
A software engineer should also be able to specify the desired software mix for the end product: the modules to be compiled and linked, the versions and variants to be used and the modules that rely on each other. From this description, a utility can extract the correct information modules out of the database and compile, assemble and link the source and object files to create up-to-date, consistent software. Ideally, the utility should be able to avoid redundant steps if the required information already exists. For example, there should be no need to recompile the source of module X as long as a good copy of the object for module X exists in the database. A single change in one module should not require the recompilation of all 60 modules in a project.

Shared resources + dedicated resources

In today's software-development environment, two conflicting requirements are placed on host systems. First, software developers must be able to communicate and *share* resources. Information-management tools typically require that members of a project share a central database in which project information is stored and managed. Software engineers must be able to communicate information, performing such functions ensure consistent, updated software and eliminate redundant steps. A programmer, for example, can use these project-management tools to alter a program module, put the module back in the system and generate a new, consistent version of the system (B).

AUTOMATING SOFTWARE MANAGEMENT

Creating a new software product is a complex, multistage process usually involving many software-development team members, three kinds of information and code (documentation, source and object code) and several versions and variants of a package. For example, developing an Intel Corp. compiler for the 8086 microprocessor involved 175 modules totaling 200K bytes of code, four engineers and 10 hours of program-generation time. Correctly managing such a project and avoiding costly mistakes increasingly requires automated project-management tools (A), such as Intel's Software Version Control System (SVCS) and MAKE. The SVCS system furnishes a database that permits project members to track and control versions and variants of the software. Database information can be protected against accidental or simultaneous changes by two or more developers. The system can also record when a change in a software module was made and the reason for and initiator of the change. The MAKE facility can determine what compiles, assembles and links must be performed on various modules to construct a software product from its constituents. The utility uses module-dependency information (what modules affect certain other modules) to



as sending and receiving electronic mail. This trend favors the use of minicomputers that let users share a database, communicate and cooperate.

Second, software developers' tools are becoming increasingly sophisticated. The price of this sophistication, however, is more powerful computing resources. These tools require *dedicated* processing power, large memory space and quick response to perform efficiently. This means newer tools will have to be hosted single-user workstations, in which software developers can be guaranteed a certain level of computing resources.

Single-user workstations connected to a local-area network (LAN) can resolve these conflicting trends (Fig. 4). Such a distributed-processing approach offers the best of both worlds. Each user has a dedicated set of computing resources in his workstation, uses a central, shared database located at the file server and can easily communicate with other users.

If the LAN architecture is correctly designed, distributed processing can offer other benefits as well. Different types of workstations can be attached to the network, according to user needs. Thus, in a software-engineering environment, most of the workstations could be optimized for software developers, with only a few reserved for hardware debugging. To meet these needs, the LAN should become the standard information bus of the software-development team.

The distributed processing afforded by an LAN will also provide a measure of protection against obsolescence. Newer stations can be added to replace older ones, as required. Now, the unit of growth for software-development systems is the workstation, not the mainframe.

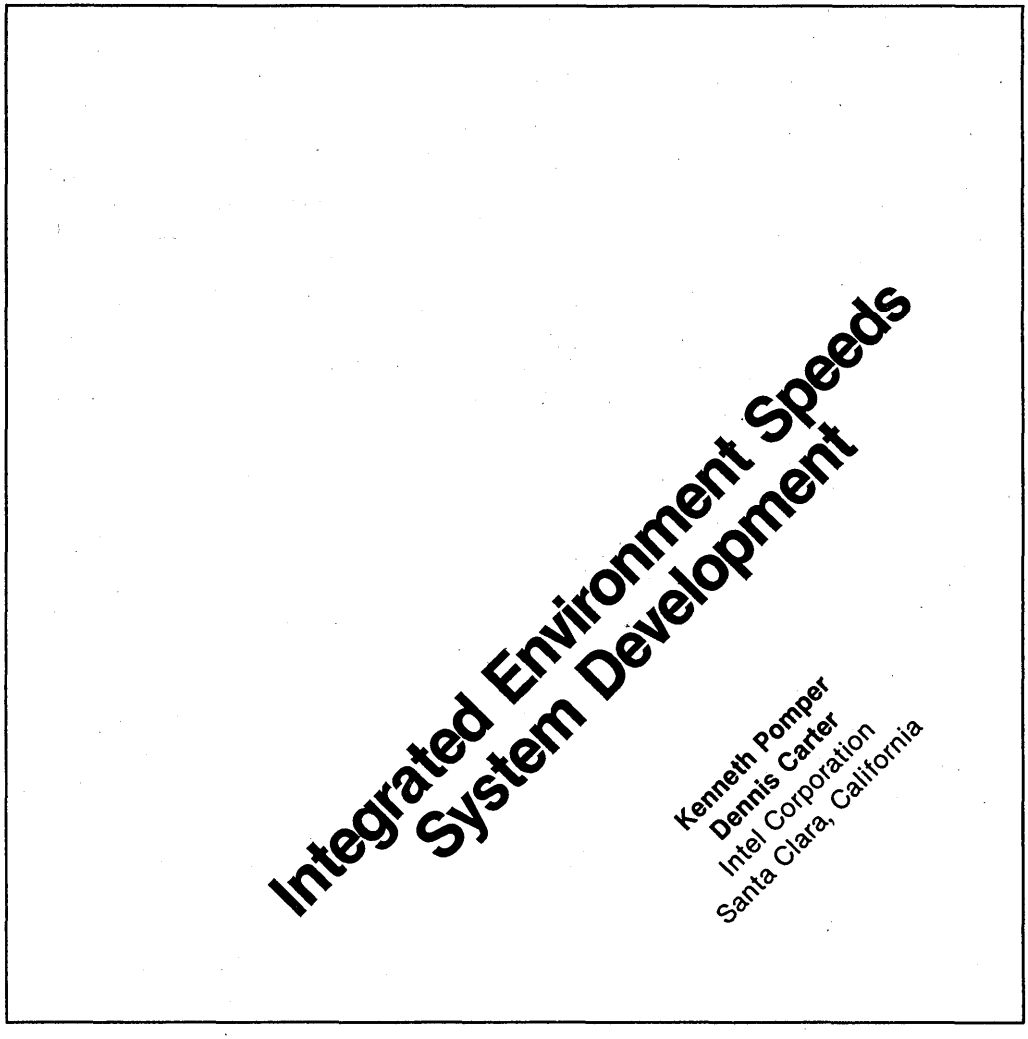
The trend toward workstations connected by an LAN weighs heavily against the cost advantage of timesharing over distributed processing. The push for software productivity will therefore be the most pressing reason for the adoption of distributed processing in modern software-development environments. □

Paul Maritz is software tools planning chairman at Intel Corp.'s Development Systems Operations, Santa Clara, Calif.

Reprinted from **MINI-MICRO SYSTEMS** December 1983

ARTICLE
REPRINT

AR-352



Reprinted from *Computer Design*, November 1984

ORDER NUMBER:231257-001

INTEGRATED ENVIRONMENT SPEEDS SYSTEM DEVELOPMENT

By integrating source and version control, electronic mail, standard interfaces for programming languages, and common interfaces to operating systems, a total development environment can accelerate the software task faster than adding staff.

by Kenneth Pomper and Dennis Carter

If a project is running behind schedule, adding staff members is not always the best tactic for getting it back on schedule: as the saying goes, adding manpower to a late software project makes it later. Often the best solution is to coordinate programming efforts and project management through an integrated development environment. This type of system stimulates greater efficiency by combining management, programming, and debugging tools in one environment. Productivity increases especially with microprocessor systems with separate target and host development systems. As a result, industries can meet critical delivery schedules without needing additional programmers.

System development is a complex process involving several different stages that continually pass information between each other. The development environment should be more than a collection of assorted tools that are poorly linked. It must efficiently coordinate the diverse stages of development in a single coherent environment, allowing information to flow easily between different tiers of the project (Fig 1).

An efficient development cycle has two parts. Managers must have a clear view of the project from inception through test and implementation. Thus, planning

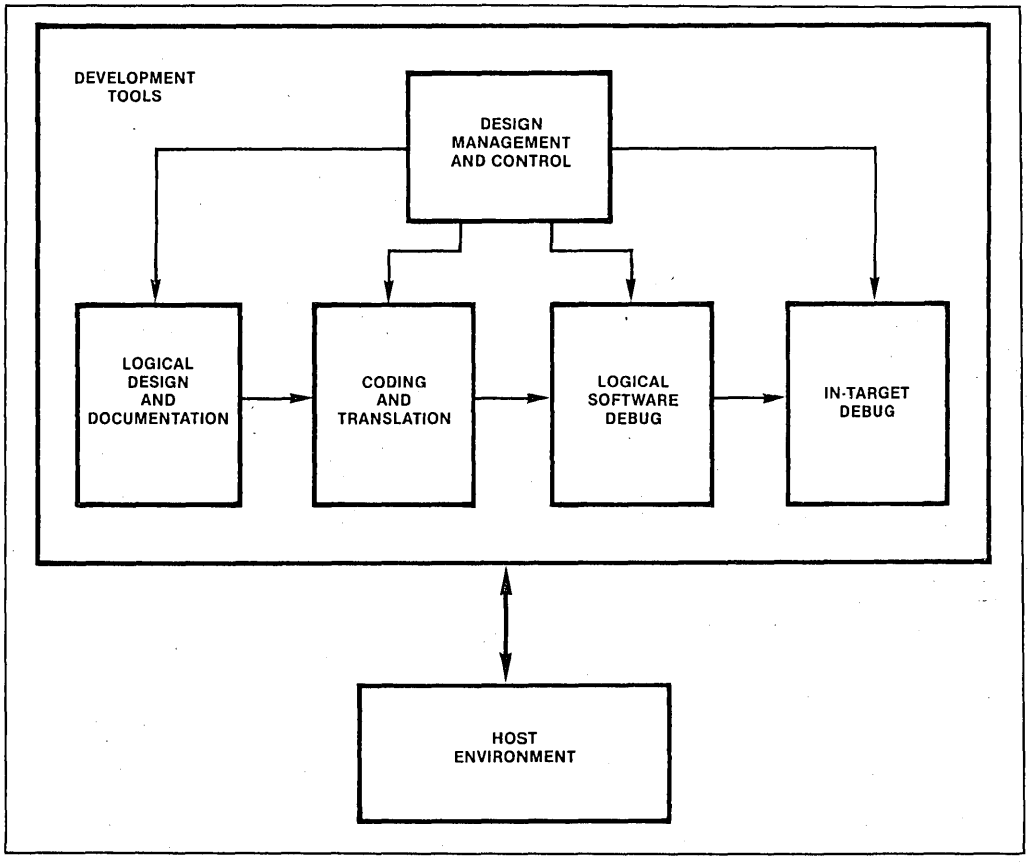


Fig 1. An integrated development environment must do more than act as a library for development tools. It must ensure that information flows smoothly between components. As organizations shift to new development policies and expand development hardware, the system must be able to migrate smoothly to the new host environment.

work schedules and anticipating design bottlenecks is easier. Software engineers must share their ideas, designs, and programs, passing information throughout the different development stages.

Yet, in developing products for other target machines, an integrated environment for the host development system alone is not enough. Unless a smooth transition to the final target environment is provided, the project will bog down during the critical target system integration and test. The transition from host to target development environments is one of the two major factors affecting the project cost. According to R.W. Jensen, changing environments can increase costs as much as 122 percent.

Not only must engineers deal with different target hardware in different projects, but also they must work on a shifting host hardware base as companies expand their development resources. Rather than losing previous investments in tools or training, the company must be able to shift the entire environment smoothly as the company shifts to different development strategies. For example, engineers using Intel's Intellec® Series IV workstation maintain the same fundamental development environment when they move to the NDS-II distributed development environment.

With its multiple stages, development can turn into a logistical headache for managers and engineers alike. Managers supervising several programming teams, each developing different versions of programs, can easily lose the thread of revisions to source code. Similarly, programmers can find themselves working at cross-purposes in their attempts to generate and test the most recent versions of code, rather than a hybrid of current and obsolete code versions.

An integrated system can help prevent these problems by combining different tools and making them work well together. For example Intel's configuration management tools, Source Version Control System (SVCS) and MAKE, manage multiple versions of a program. The tools can automatically combine the most current versions of several modules in larger programs. Similarly, Intel's debugging aids, PSCOPE and Integrated Instrumentation and In-Circuit Emulation (I^2ICE^{TM}) package, use information implanted by compilers to permit programmers to debug during the integration process at the source-level. Such an integrated environment increases efficiency through good allocation of available resources.

MANAGEMENT AND CONTROL

Modular design helps software engineers break a large complex problem into a set of small simple programs. Unfortunately, a modular design system requires more overhead for managing a large number of modules and different versions of the same module. If the logistics become too troublesome, programmers might even collapse several modules into a single file to save themselves the trouble of manipulating the separate modules. Project management tools can free engineers from the housekeeping chores associated with program development (Fig 2).

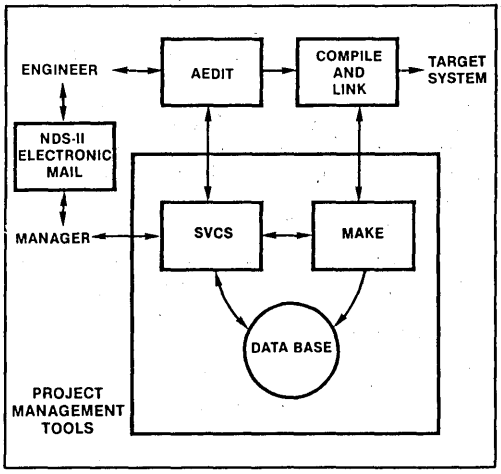


Fig 2. Besides controlling changes to the source files in its data base, SVCS helps managers audit source updates. Automatically generating the software for the target system, MAKE reduces generation time by about 50 percent, leaving engineers more time to concentrate on development.

Programmers keep track of major changes in their programs by either creating copies of the new version or changing an older version. The result is a series of similar programs that lack proper documentation to indicate the change and reason for the change. SVCS provides an automated approach to this record keeping. It tracks changes to the baseline version of a program, and demands that programmers record their reasons.

When software engineers need a particular version of a file, whether the current or some older copy, SVCS automatically retrieves the correct version from its data base of updates and baseline versions. Similarly, after the programmers have added changes, SVCS records the updates and the reasons for the changes, adding as little as a 3 percent overhead. In addition, SVCS helps project managers exercise precise control in large team projects by preventing certain engineers from making changes independently.

While programmers work directly with SVCS to manage different versions of programs, MAKE works closely with SVCS facilities to generate current versions of systems. While generating large systems from several different modules, programmers often find that one or two modules have been updated since the last compilation. This problem is compounded when modules depend on a series of other submodules. MAKE automates the manual procedures often resorted to by software engineers to track current object modules.

Using templates that detail the modules' interdependence, MAKE ensures that only current versions of modules are included in the system generation. If it finds that a required object module is obsolete, MAKE automatically compiles the appropriate source module to produce the current version of the object module. Furthermore, if source modules depend on submodules, MAKE continues searching through its templates to ensure it recompiles modules using the current submodules for these source modules.

MAKE selectively compiles the needed modules. Only if a module or one of its submodules is obsolete does MAKE execute a recompilation. This cuts the inefficient massive compilation procedures commonly used to ensure that object modules are current.

In addition to the project management tools handling version control and system generation, a complete integrated development environment should also facilitate communication among users. Acting as an electronic central distribution center, the NDS-II electronic mail facility maintains mailboxes for individual users and groups of users on the network, and an electronic bulletin board for all users. In addition to supporting document distribution, electronic mail manages a file transfer facility. Team members can transmit both source and object modules to any other user on the network.

Another feature, NDS-II's network resources manager (NRM), provides extensive support for file management and resource sharing. The NRM manages files with a hierarchical structure that arranges files into volumes and multiple subdirectories. The NRM also improves allocation of resources through its distributed job control (DJC) facility. DJC permits users on private workstations to export a batch job to the NRM for remote execution. The NRM then moves the job to a free workstation for execution, returning the completed job status to the user's directory.

LOGICAL DESIGN

An integral part of the software development environment and its primary interface with the user is the text editor. Because software engineers typically spend 40-50 percent of their time using a system editor, it is a critical element in software development and can greatly enhance productivity if used well. For example, programmers often need to work simultaneously on two separate files, such as two different source programs or a program and a specification document. Editors such as Intel's AEDIT permit them to edit two files of any size simultaneously and transfer text between them.

AEDIT's ability to store a sequence of edit commands also simplifies the use of edit macros. With AEDIT, programmers build macros simply by typing in their commands. They can re-execute the command series and even save it on disk for later use. AEDIT also helps software engineers with structured programming techniques through its automatic text indentation. Furthermore, AEDIT protects programmers' efforts by optionally creating back-up copies of files being edited.

Although a text editor serves as the primary interface between the development system and programmer, programming languages serve as the principal interface between design concepts and the target hardware. With the right set of programming languages and support tools, software professionals can develop the optimal solution for a particular situation, without the design bias often seen when designers plan projects with an eye on their eventual implementation.

For example, different programming languages like assembler, PL/M, C, Pascal, and Fortran enjoy certain advantages over each other. Software developers should be able to draw on the most appropriate language to implement the different facets of a design. In order to support this kind of free choice, however, the development environment must be able to coordinate the use of a mix of programming languages, so that programmers can use different languages without concern about how the different modules will eventually be combined.

Like natural languages, the virtue of programming languages lies in their ability to represent abstract ideas in concrete terms. Just as it may be easier to express a certain idea with a particular natural language than another, programming languages vary in their ability to represent certain design concepts. For example, software engineers find that Pascal represents structured designs more faithfully than a language like Fortran. Also, languages like PL/M or C, which closely reflect the hardware base of a design, or assembly language, which provides the ultimate visibility into the hardware, are powerful tools for developing real-time embedded systems.

Still, programming languages share another feature with natural languages—varying degrees of popularity. For example, Fortran remains one of the most popular programming languages. Its continued strong momentum translates into a large installed base of

software. For managers, this large installed base provides a ready source of existing code. On the other hand, managers must remain ready to incorporate newer languages like ADA into designs without starting from scratch.

In many software development projects, managers often look for a way to juggle several programming languages simultaneously. Software engineers can usually adapt quickly to new programming languages —particularly when they are supported by project management tools. On the other hand, the development environment often acts as a bottleneck in mixing several different languages in the same target system because of its inability to match the varying program and system interfaces of different languages.

The Intel development environment integrates different languages through a common object module format (OMF). A standard OMF works at several levels. During link time, OMF presents a standard method for indicating data type information, which the linker uses to build its memory allocation tables. Furthermore, debuggers exploit OMF's standard arrangement of symbolic information for handling symbolic debugging.

Two other aspects of the standard development environment include the definition of standard conventions for passing parameters between different programs—regardless of their implementation language —and standard interfaces to the operating environment. Besides accounting for critical implementation details another key measure of the effectiveness of a development environment is its support of application level standards like IEEE 754 for floating point operations or IEEE 802 for Ethernet.

For those areas currently without standards, the development environment takes the initiative with a baseline for the operating environment. Here, Intel's universal development interface (UDI) defines a system-independent interface between application programs and the operating environment. Rather than write their programs with system-dependent calls to operating system utilities, software developers use the same UDI call to allocate memory, for example, regardless of the target operating system. During link-time, the linker uses this UDI call to link in the appropriate system utility in iRMX™, for example (Fig 3). Consequently, programs that use the UDI can be ported between ISIS, iRMX, and Microsoft's Xenix simply by loading the modules into the new environment. Thus, if the design calls for a realtime operating environment like iRMX, engineers can develop the application under ISIS without fear that their work will be lost when the system is transported to the iRMX environment.

For the manager trying to improve productivity, no faster method exists than simply porting existing code to a new environment. Besides IEEE standards, which provide a common application environment, the use of a common object format and universal development interface provide a clear migration path between operating environments.

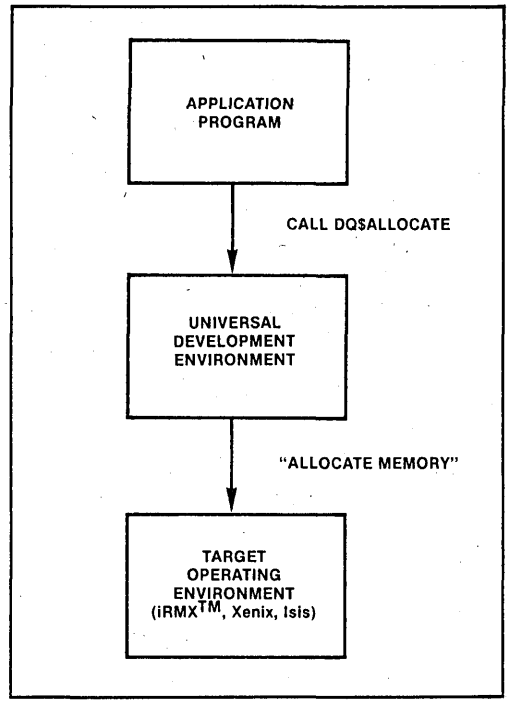


Fig 3. Where applications standards do not already exist, a development system should follow some baseline. The universal development interface (UDI) sets a baseline for interactions between application programs and operating software. For example, an application that requires memory uses a UDI call (DQ$ ALLOCATE) which is later translated into the appropriate call for target operating environment.

SAME INTERFACE

In the kind of cross-development environments commonly used for creating microprocessor-based products, engineers work most effectively if they are able to split debugging into two phases. In the first phase, debugging occurs in parallel for the target hardware system and for the software. Here, engineers use the host environment to debug the basic logic of the software system. Once they are satisfied both with the logic of the software and with the operation of the hardware, the engineers then load the software into

the target system for the second phase—integration and test.

This in-target phase is the critical step where hardware and software are finally integrated as a total system. As noted earlier, differences between the host and target environments can more than double costs. Consequently, a key feature of an integrated environment is a common debug interface between host and target.

Intel's PSCOPE debugger permits programmers to check out programs at the source-level both during logic debug and during in-target test. Because PSCOPE shows up again as one of the three major components of the I^2ICE system, software engineers are assured of a smooth transition between host and target. Along with PSCOPE, I^2ICE's in-circuit emulation and logic timing analyzer (LTA) give developers a full view simultaneously into the hardware and software components of their systems. Without this kind of coordinated approach to system integration and test, developers can never deal with the hardware and software as an integrated system, but are forced to switch continually between hardware testing and software debugging.

Supporting system integration at the most fundamental level, in-circuit emulation provides a transparent, full speed emulation of the iAPX 86 and iAPX 286 families of processors. Besides handling multiple level breakpoints and traces in single microprocessors, I^2ICE extends its support to multiprocessor environments. Developers can emulate a system of up to four microprocessors and examine complex processor interactions like synchronization. For example, I^2ICE lets engineers define events like breaks and traces conditionally, so that a microprocessor will break when another defined event occurs in a different microprocesor.

While I^2ICE and PSCOPE provide the fundamental support for a system's underlying hardware and software, the LTA also serves as a key element of the system's integrated package. Displaying 16 channels of

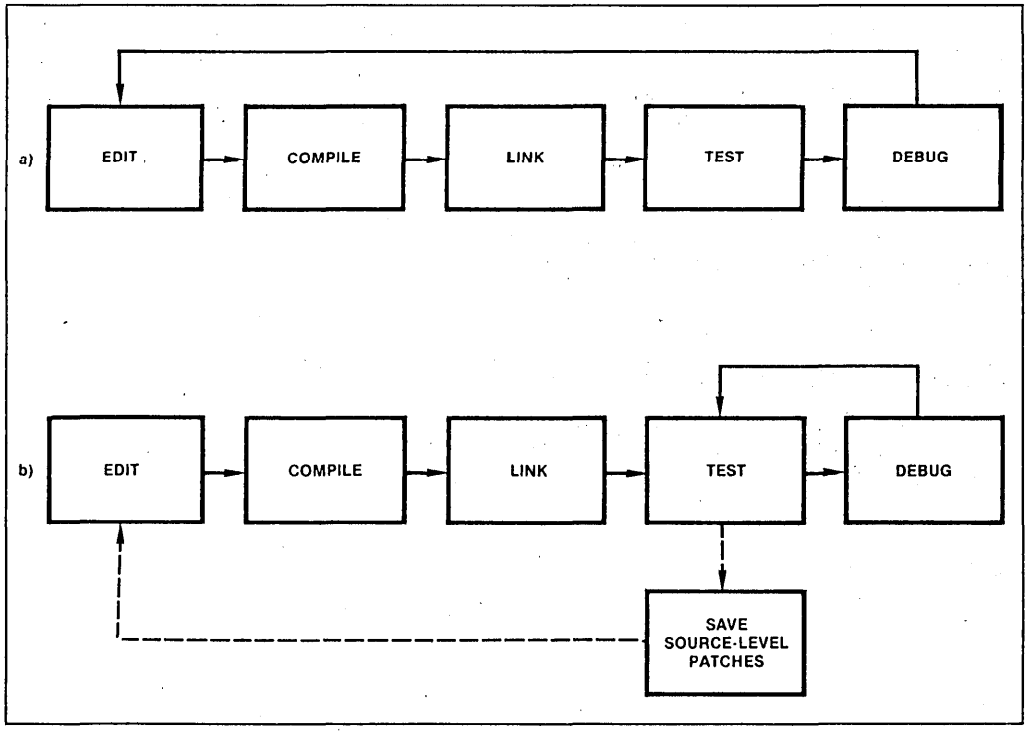


Fig 4 a/b. *In the past, engineers have needed to iterate through a lengthy development cycle in order to* **debug source code in the target system (a). On the other hand, PSCOPE lets engineers use source level code to debug and patch target systems and continue debugging, then finally, after many bugs are found, save the source-level patches on disk for later addition to the original source files (b).**

logic and timing information, the LTA helps isolate critical state and timing problems. In order to speed the analysis process, this menu-oriented system also permits engineers to save debugging setups and waveforms on disk.

A key advantage of an integrated environment is its ability to present information, through a consistent command language, in a familiar form. With I^2ICE, this feature extends to logic and timing analysis. Rather than present a morass of digits, the LTA displays most information in easy to understand waveform diagrams.

Just as the LTA has moved system integration and test above the bit level, PSCOPE shortens software debugging by permitting engineers to test programs using their own symbols, rather than machine code. With the traditional machine code debugger, if they wanted to patch a section of machine code, programmers would spend hours converting machine code between different formats, like binary and hex, and calculating the machine code equivalents of assembler instructions. Even somewhat more sophisticated debuggers that disassemble machine code are little help in retaining the sense of a program as expressed through its use of symbols.

Instead, even though it helps software engineers deal with machine code when necessary, PSCOPE can handle debugging at the level of the original source code. Consequently, programmers can set an unlimited number of breakpoints by statement number, step through a single source statement at a time, and trace execution by statement number, procedure name, or label (regardless of whether they are working with the host or target system).

From the user's point of view, the utility of PSCOPE lies in its built-in, CRT-oriented editor and in its command languge that resembles a high level structured programming language (see the Table). Using PSCOPE's editor, engineers write extensive procedures in the command language for testing code and even patch existing code with new or revised source statements.

PSCOPE's ability to handle source-level patches avoids the conventional development scenario where software developers go through a continual cycle of edit-compile-link-test-debug [Fig 4(a)]. Source-level patching short-circuits this loop; programmers can remain in the debug phase—patching at the source-level and even saving the source-level patch on disk for later incorporation into the original source-code files maintained under SVCS [Fig 4(b)].

The advantages of an integrated environment show up here very dramatically. During compilation, the compiler places symbolic information associated with a program into the object modules it generates. In turn, the linker carries this information along into the run time image. Both PSCOPE and I^2ICE draw on this symbolic information for their source-level debugging. Consequently, during system debugging, developers see familiar procedure and data names, rather than a confusing series of machine codes or disassembled mnemonics. Furthermore, because it maintains this symbolic information in a virtual table, PSCOPE is able to handle arbitrarily long symbol tables—it just brings a new page of symbols from disk, if necessary.

As a result of its ability to coordinate its tools for the various stages of development, the Intel development environment lets system engineers concentrate on product development, rather than administrative chores. For the development manager, this translates into on-time product delivery, without the costs of additional resources.

Microcomputer Development Languages

4

iAPX 286 SOFTWARE DEVELOPMENT PACKAGE

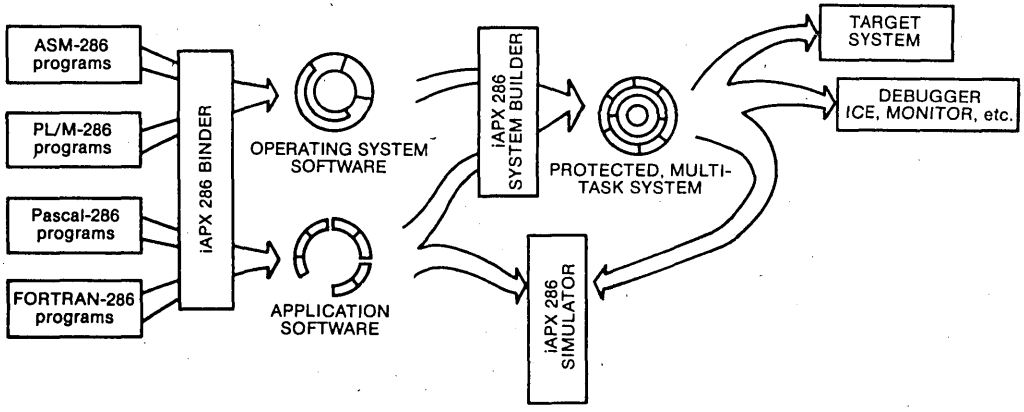
- **Complete System Development Capability for High-Performance iAPX 286 Applications.**
- **Allows creation of Multi-User, Virtual Memory, and Memory-Protected Systems.**
- **Macro Assembler for Machine-Level Programming.**
- **System Utilities for Program Linkage and System Building.**
- **Software Simulator for Execution and Symbolic Debugging on Intel Development System.**
- **Package Supports Program Development with PL/M-286, Pascal-286, and FORTRAN 286.**
- **Extends Existing Intellec® Development Systems to Provide Broad Support for the iAPX 286 Microprocessor.**

The iAPX 286 is a 16-bit microprocessor system with 32-bit virtual addressing, integrated memory protection, and instruction pipelining for high performance. The iAPX 286 Software Development Package is a cohesive set of software design aids for programming the iAPX 286 microprocessor system. The package enables system programmers to design protected, multi-user and multi-tasking operating system software, and enables application programmers to develop tasks to run on a protected operating system.

The iAPX 286 Software Development package contains a macro assembler, a program binder (for linking separately compiled modules together), a system builder (for configuring protected multiple-task systems), and a software simulator (for execution and symbolic debugging).

The memory protection features of the iAPX 286 architecture are invisible to application programmers, who use language translators and the program binder. System programmers may use special memory protection features in ASM-286 or PL/M 286, and use the system builder for initializing and managing protection features. The Simulator duplicates the operation of the 80286 CPU, as well as the floating point operations of the 80287.

All the utilities in the Software Development Package run on the Intel Microcomputer Development Systems (Series III/Series IV).



The iAPX 286 Software Development Package keeps the protection mechanism invisible to the application programmer, yet easy to configure for the system programmer.

JUNE 1984
ORDER NUMBER: 210565-001

iAPX 286 MACRO ASSEMBLER

- **Instruction Set and Assembler Mnemonics Are Upward Compatible with ASM-86/88.**
- **Powerful and Flexible Text Macro Facility.**
- **Type-Checking at Assembly Time Helps Reduce Errors at Run-Time.**
- **Structures and RECORDS Provide Powerful Data Representation.**
- **"High-Level" Assembler Mnemonics Simplify the Language.**
- **Supports Full Instruction Set of the iAPX 286/20, Including Memory Protection and Numerics.**

ASM-286 is the "high-level" macro assembler for the iAPX 286 assembly language. ASM-286 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM-86/88 mnemonics; new ones have also been added to support the new iAPX 286 instructions. The segmentation directives have been greatly simplified.

The iAPX 286 assembly language includes approximately 150 instruction mnemonics. From these few mnemonics the assembler can generate over 4,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 150 mnemonics to generate all possible machine instructions. ASM-286 will generate the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM-286 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM-286 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This means that many programming errors will be detected when the program is assembled, long before it is being debugged.

ASM-286 object modules conform to a thorough, well-defined format used by all 286 high-level languages and utilities. This makes it easy to call (and be called from) HLL object modules.

Key Benefit:

For programmers who wish to use assembly language, ASM-286 provides many powerful "high-level" capabilities that simplify program development and maintenance.

AFN-00378B

iAPX 286 BINDER

- **Links Separately Compiled Program Modules Into an Executable Task.**
- **Makes the iAPX 286 Protection Mechanism Invisible to Application Programmers.**
- **Works with PL/M-286, Pascal-286, FORTRAN-286 and ASM-286 Object Modules.**
- **Performs Incremental Linking with Output of Binder and Builder.**
- **Resolves PUBLIC/EXTERNAL Code and Data References, and Performs Intermodule Type-Checking.**
- **Provides Print File Showing Segment Map, Errors and Warnings.**
- **Assigns Virtual Addresses to Tasks in the 2^{32} Address Space.**
- **Generates Linkable or Loadable Module for Debugging.**

BND-286 is a utility that combines iAPX 286 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules written in ASM-286, PL/M-286, Pascal-286 or FORTRAN-286, and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND-286 generates a print file displaying a segment map, and error messages.

The Binder will be used by system programmers and application programmers. Since application programmers need to develop software independent of any system architecture, the 286 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged, as well.) System protection features are specified later in the development cycle, using the 286 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of use of the 286 utilities.

Key Benefit:

The Binder is the only utility an application programmer needs to develop and debug an individual task. Users of the Binder need not be concerned with the architecture of the target machine, making application program development for the 286 very simple.

---

iAPX 286 MAPPER

- **Flexible Utility to Display Object File Information.**
- **MAP-286 Selectively Purges Symbols from a Load Module.**
- **Provides Inter-Module Cross-Referencing for Modules Written in All Languages.**
- **Mapper Allows Users to Display:**

Protection Information:	Debug Information:
SEGMENT TABLES	MODULE NAMES
GATE TABLES	PROGRAM SYMBOLS
PUBLIC ADDRESSES	LINE NUMBERS

Key Benefit:

A cross-reference map showing references *between* modules simplifies debugging; the map also lists and controls all symbolic information in one easy-to-read place.

iAPX 286 LIBRARIAN

- **Fast, Easy Management of iAPX 286 Object Module Libraries.**
- **Only Required Modules Are Linked, When Using the Binder or Builder.**
- **Librarian Allows Users to:**

  Create Libraries
  Add Modules
  Replace Modules
  Delete Modules
  Copy Modules from Another Library
  Save Library Module to Object File
  Create Backup
  Display Module Information (creation date, publics, segments)

Key Benefit:

Program libraries improve management of program modules, and reduce software administrative overhead.

iAPX 286 SYSTEM BUILDER

- **Supports Complete Creation of Protected, Multi-task Systems.**
- **Resolves PUBLIC/EXTERNAL Definitions (between protection levels).**
- **Supports Memory Protection by Building System Tables, Initializing Tasks, and Assigning Protection Rights to Segments.**
- **Creates a Memory Image of a 286 System for Cold-start Execution.**
- **Target System may be Boot-loadable, Programmed into ROM, or Loaded From Mass-store.**
- **Generates Print File with Command Listing and System Map.**

BLD-286 is the utility that lets system programmers configure multi-tasking, protected systems from an operating system and discrete tasks. The Builder generates a cold-start execution module, suitable for ROM-based or disk-based systems.

The Builder accepts input modules from iAPX 286 translators or the iAPX 286 Binder. It also accepts a "Build File" containing definitions and initial values for the 286 protection mechanism—descriptor tables, gates, segments, and tasks. BLD-286 generates a Loadable or bootloadable output module, as well as a print file with a detailed map of the memory-protected system.

Using the Builder command Language, system programmers may perform the following functions:

- — Assign physical addresses to segments; also set segment access rights and limits.
- — Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.
- — Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.
- — Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.
- — Create Task State Segments and Task Gates for multi-task applications.
- — Resolve inter-module and inter-level references, and perform type-checking.
- — Automatically select required modules from libraries.
- — Configure the memory image into partitions in the address space.
- — Selectively generate an object file and various sections of the print file.

Key Benefit:

Allows a system programmer to define the configuration of a protected system in *one* place, with one easy-to-use Utility. This specification may then be adopted by all project members, using either the Builder *or just the Binder.* The flexibility simplifies program development for all users.

AFN-00378B

iAPX 286 SIMULATOR

- **Supports Symbolic Debugging of Complete, Protected 286 Systems.**
- **Allows 286 Program Execution and Debugging in Absence of iAPX 286 Hardware Execution Vehicle.**
- **Functionally Duplicates the Operation of the iAPX 286 Microprocessor, Including Memory Protection.**
- **Executes Full Instruction Set, Including 80287 Numerics.**
- **Symbolic Access to Program Variables as well as Descriptor Tables.**
- **Two Execution Timers for Program Benchmarking and Interrupt Simulation.**
- **UDI File System Support for User Program.**

SIM-286 is an 8086-resident program designed to support development of iAPX 286 O.S. kernels, systems, and applications. All of these may be developed and debugged without the use of a 286 hardware execution vehicle.

The Simulator consists of a human interface layer, and software executors for the 80286 CPU and 80287 Numeric Data Processor. The human interface receives commands with symbolic names, and passes control to the executor as though it were a 286-resident monitor.

SIM-286 lets designers manipulate a 286 program using the symbolic names given for code and data. It also lets users symbolically examine and modify the protection features (such as system tables, access rights, etc.), if it is desired.

SIM-286 contains two instruction timers. One may be set and incremented during execution; this allows program sequences to be benchmarked in clock cycles and microseconds. The second, an interval timer, may be set to generate interrupts every η clock cycles, to simulate event-driven processing. These timers are extremely useful for developing system kernels.

For programs that make operating system calls for file I/O, SIM-286 provides access to these services through the Universal Development Interface.

Key Benefit:

Symbolic system debugging (for protected 286 software) may be performed in the absence of a 286-based target.

SPECIFICATIONS

OPERATING ENVIRONMENT

Intel Microcomputer Development Systems
(Series III/Series IV)

DOCUMENTATION

ASM 286 Language Reference Manual
ASM 286 Macro Assembler Operating Instructions
iAPX 286 Utilities User's Guide
iAPX 286 System Builder User's Guide
iAPX 286 Simulator User's Guide
Pocket Reference for all the above:
- ASM 286
- Utilities
- SIM 286

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

ORDERING INFORMATION

Product Code	Description
iMDX-321	iAPX 286 Software Development Package

AFN-00378B

PASCAL-286 SOFTWARE PACKAGE

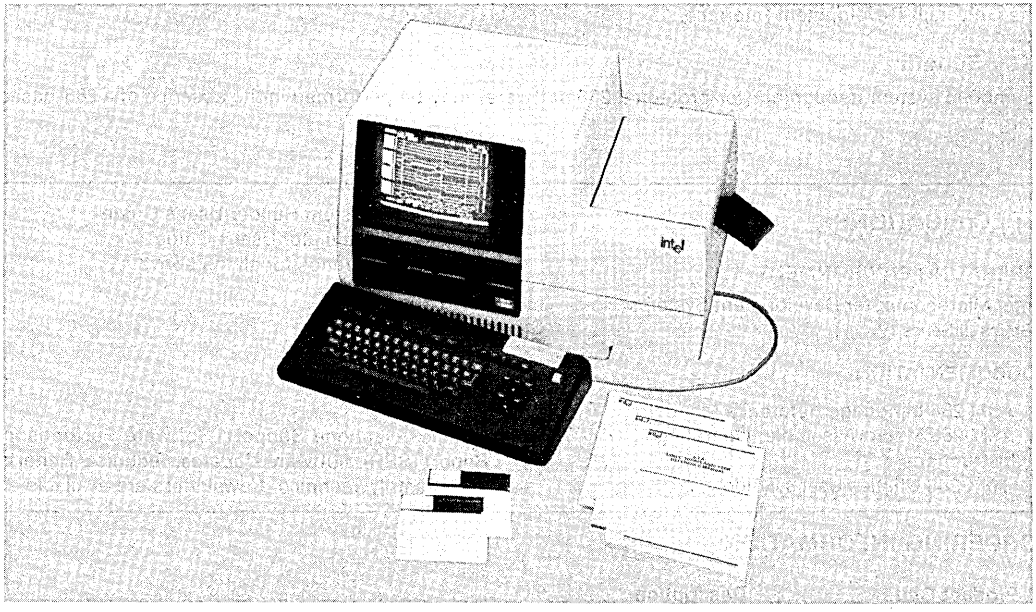
- **High-level programming language for the protected virtual mode iAPX 286**
- **Implements ISO standard Pascal. Many useful extensions may be enabled via a compiler switch**
- **Upward compatible with Pascal-86 for software portability**
- **Produces relocatable object code which is linkable to object modules generated by other iAPX 286 translators**
- **Supports full symbolic debugging with iAPX 286 software and ICE™ debuggers**
- **Fully supports the 80287 numeric processor using the IEEE floating point standard**

Pascal-286 is a powerful, structured, applications programming language for the protected virtual address mode of the iAPX 286. Pascal-286 is upward compatible with Pascal-86 so that 8086 Pascal source code can be ported to the iAPX 286 in protected mode.

Pascal-286 implements strict ISO standard Pascal, but with many useful extensions. These include separate compilation of modules, interrupt handling, port I/O, and 80287 numerics support. A control is provided in the compiler to flag all non-ISO features used.

Pascal-286 produces relocatable object code which can be linked with object code produced by other iAPX 286 translators such as ASM-286 and PL/M-286. Thus, a combination of translators can be used to provide great programming flexibility.

Type and symbol information needed by software and in-circuit debuggers is added to the object code by the Pascal-286 compiler. This information can be stripped off by the compiler or linker for the final production version.



NOVEMBER 1983
ORDER NUMBER: 230863-001

FEATURES

Conforms to ISO Standard Pascal

Pascal has gained wide acceptance as a portable language for microcomputer applications. However, portability can result only if standards are adhered to. Pascal-286 is a strict implementation of ISO standard Pascal. Extensions are provided to make the language more powerful for microprocessor applications. All extensions are clearly highlighted in the documentation. In addition, the compiler provides a control to flag any non ISO feature used. Pascal-286 will evolve to track future enhancements to standard Pascal.

Upward Compatible with Pascal-86

The Pascal-286 compiler produces object code for the protected virtual address mode of the iAPX 286 language. However, no 286 architecture specific features have been added to the Pascal-286 language. This makes Pascal-286 source code upward compatible with Pascal-86, which allows for porting of 8086 software to the protected 286 with relative ease.

Compatible With Other iAPX 286 Translators

All Intel iAPX 286 translators output object code in a standardized format. This allows 286 programs to be written in a mixture of languages. Systems routines which need access to architectural features can be coded in PL/M-286 or ASM-286. Pascal-286 may be better suited for the applications routines. The systems and application routines can then be combined using the 286 linker (BIND-286).

Standardized Run Time Support

Programs compiled with Pascal-286 can be moved from the development host environment to the target environment with ease. This is the result of standardizing run-time operating system interfaces required by the compiled program into a well defined and well documented set of routines. After programs are developed on a development host, they can then be executed in the target using the same set of system interfaces.

Extensions for Microprocessor Programming

Pascal-286 provides extensions that make it powerful for microprocessor applications. Built-in procedures allow I/O directly from the ports of the iAPX 286. This speeds up I/O as it is done by direct communication with the microprocessor. Interrupt processing is also supported by built in procedures. Examples are: ENABLEINTERRUPTS, DISABLEINTERRUPTS, CAUSEINTERRUPT. Many built in procedures and variables are provided for communicating with the 80287 for numeric computations.

Compiler Controls

The Pascal-286 compiler provides many controls which can be used at invocation time to enhance programming flexibility. Examples are: CODE/NOCODE, DEBUG/NODEBUG, INCLUDE (file), LIST/NOLIST, OPTIMIZE (n), EXTENSIONS/NOEXTENSIONS. All controls have default values that are active unless the opposite is specified during invocation. Thus, for most compiles, no controls need be specified.

Support for IEEE Standard Numerics

Pascal-286 provides full support for the 80287 numerics co-processor. All floating point operations are done according to the IEEE floating point standard. The benefits are predictable, accurate and consistent results. Built-in procedures to support the 80287 include GET8087ERRORS and MASK 8087ERRORS. A full set of 80287 library routines are supplied with the compiler.

Optimizations

The Pascal-286 compiler produces highly optimized code, both in size and execution time. This is achieved by:

—Use of powerful iAPX 286 instructions, in particular, for string handling, 80287 numerics and subroutine linkage
—Short circuit evaluation of boolean expressions, constant folding and strength reduction of multiplications and additions
—Elimination of superfluous branches, optimization of span dependent jumps

SPECIFICATIONS

Operating Environment

Intel 8086 based microcomputer
Development systems (Series III, Series IV)

Documentation Package

Pascal-286 User's Guide
Pascal-286 Pocket Reference

ORDERING INFORMATION

Part Number	Description
iMDX-324	Pascal-286 Software Package

Requires Software License

Support

Hotline service, SPR (Software Performance Reports), Updates and technical newsletters are available.

PL/M 286 SOFTWARE PACKAGE

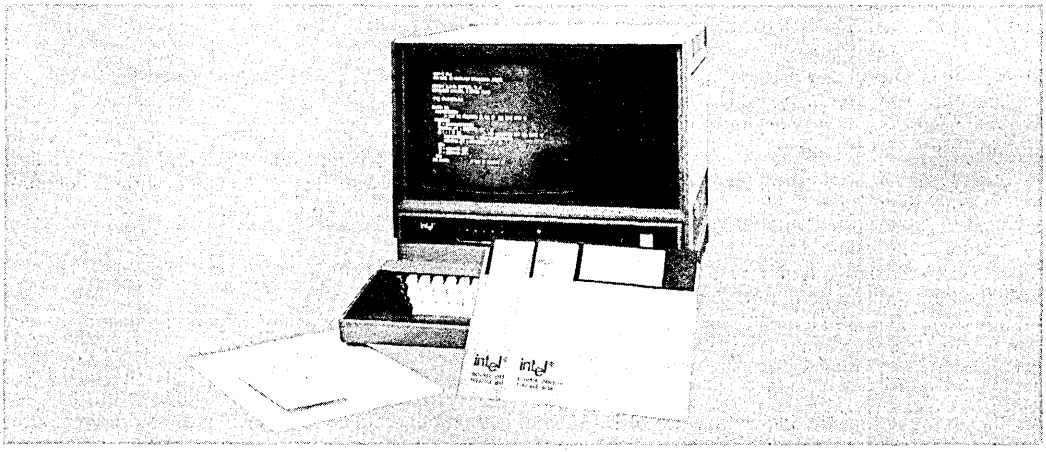
- **Systems programming language for the protected virtual address mode iAPX 286**
- **Upward compatible with PL/M 86 and PL/M 80 assuring software portability**
- **Enhanced to support design of protected, multi-user, multi-tasking, virtual memory operating system software**
- **Advanced, structured system implementation language for algorithm development**
- **Produces relocatable object code which is linkable to object modules generated by all other iAPX 286 language translators**
- **Multiple levels of optimization**
- **Resident on Intel microcomputer development systems (Series III, IV)**

PL/M 286 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode iAPX 286. PL/M 286 has been enhanced to utilize iAPX 286 features—memory management and protection—for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M 286 is upward compatible with PL/M 86 and PL/M 80. Existing systems software can be re-compiled with PL/M 286 to execute in protected virtual address mode on the iAPX 286.

PL/M 286 is the high-level alternative to assembly language programming on the iAPX 286. For the majority of iAPX 286 system programs, PL/M 286 provides the features needed to access and to control efficiently the underlying iAPX 286 hardware and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M 286 compiler has been designed to efficiently support all phases of software development. Features such as a built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed.



MAY 1983
ORDER NUMBER:210536-002

FEATURES

Major features of the Intel PL/M 286 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, for example).

The use of modules and procedures to break down a large problem leads to productive software development. The PL/M 286 implementation of block structure allows the use of REENTRANT procedures, which are especially useful in system design.

Language Compatibility

PL/M 286 object modules are compatible with object modules generated by all other 286 translators. This means that PL/M programs may be linked to programs written in any other 286 language.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with full symbolic debugging capabilities.

PL/M 286 language is upward compatible with PL/M 86 and PL/M 80 so that application programs may be easily ported to run on the protected mode iAPX 286.

Supports Seven Data Types

PL/M makes use of seven data types for various applications. These data types range from one to four bytes and facilitate various arithmetic, logic, and addressing functions:

—Byte: 8-bit unsigned number
—Word: 16-bit unsigned number
—Dword: 32-bit unsigned number
—Integer: 16-bit signed number
—Real: 32-bit floating-point number
—Pointer: 16-bit or 32-bit memory address indicator
—Selector: 16-bit pointer base.

Another powerful facility allows the use of BASED variables which permit run-time mapping of variables to memory locations. This is especially useful for passing parameters, relative and absolute addressing, and dynamic memory allocation.

Two Data Structuring Facilities

In addition to the seven data types and based variables, PL/M supports two powerful data structuring facilities. These help the user to organize data into logical groups.

—Array: Indexed list of same type data elements
—Structure: Named collection of same or different type data elements
—Combinations of both: Arrays of structures or structures of arrays.

Numerics Support

PL/M programs that use 32-bit REAL data are executed using the 80287 Numeric Data Processor for high performance. All floating-point operations supported by PL/M are executed on the 80287 according to the IEEE floating-point standard. PL/M 286 programs can use built-in functions and predefined procedures—INIT$REAL$MATH$UNIT, SET$REAL$MODE, GET$REAL$ERROR, SAVE$REAL$STATUS, RESTORE$REAL$STATUS—to control the operation of the 80287 within the scope of the language.

Built-In String Handling Facilities

The PL/M 286 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Built-In Port I/O

PL/M 286 directly supports input and output from the iAPX 286 ports for single BYTE and WORD transfers. For BLOCK transfers, PL/M 286 programs can make calls to predefined procedures.

Interrupt Handling

PL/M 286 has the facility for generating and handling interrupts on the iAPX 286. A procedure may be defined as an interrupt handler through use of the INTERRUPT attribute. The compiler will then generate code to save and restore the processor status on each execution of the user-defined

AFN-00643B

interrupt handler routine. The PL/M statement CAUSE$INTERRUPT allows the user to trigger a software interrupt from within the program.

Protection Model

PL/M 286 supports the implementation of protected operating system software by providing built-in procedures and variables to access the protection mechanism of the iAPX 286. Predefined variables—TASK$REGISTER, LOCAL$TABLE, MACHINE$STATUS, etc.—allow direct access and modification of the protection system. Untyped procedures and functions—SAVE$GLOBAL$TABLE, RESTORE$GLOBAL$TABLE, SAVE$INTERRUPT$TABLE, RESTORE$INTERRUPT$TABLE, CLEAR$TASK$SWITCHED$FLAG, GET$ACCESS$RIGHTS, GET$SEGMENT$LIMIT, SEGMENT$READABLE, SEGMENT$WRITABLE, ADJUST$RPL—provide all the facilities needed to implement efficient operating system software.

Compiler Controls

The PL/M 286 compiler offers controls that facilitate such features as:

—Optimization
—Conditional compilation
—The inclusion of additional PL/M source files from disk
—Cross-reference of symbols
—Optional assembly language code in the listing file
—The setting of overflow conditions for run-time handling.

Addressing Control

The PL/M 286 compiler uses the SMALL, COMPACT, MEDIUM, and LARGE controls to generate optimum addressing instructions for programs. Programs of any size can be easily modularized into "subsystems" to exploit the most efficient memory addressing schemes. This lowers total memory requirements and improves run-time execution of programs.

Code Optimization

The PL/M 286 compiler offers four levels of optimization for significantly reducing overall program size.

—Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions
—"Strength reductions": a shift left rather than multiply by 2; and elimination of common subexpressions within the same block
—Machine code optimizations; elimination of superfluous branches; reuse of duplicate code; removal of unreachable code
—Optimization of based-variable operations and cross-statement load/store.

Error Checking

The PL/M 286 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed and helpful set of programming and compilation error messages is provided by the compiler and user's guide.

BENEFITS

PL/M 286 is designed to be an efficient, cost-effective solution to the special requirements of protected mode iAPX 286 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 286 is easy to learn and use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 286, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 286 is designed to aid in the development of reliable software (PL/M 286 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in

systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

Cost-Effective Alternative to Assembly Language

PL/M 286 programs are code efficient. PL/M 286 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the iAPX 286 architecture. This includes language features for control of the iAPX 286 protection mechanism. Consequently, for the development of systems software, PL/M 286 is the cost-effective alternative to assembly language programming.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development System (Series III/Series IV)

Documentation Package

PL/M 286 User's Guide

ORDERING INFORMATION

Part Number	Description
iMDX 323	PL/M 286 Software Package

Requires Software License

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

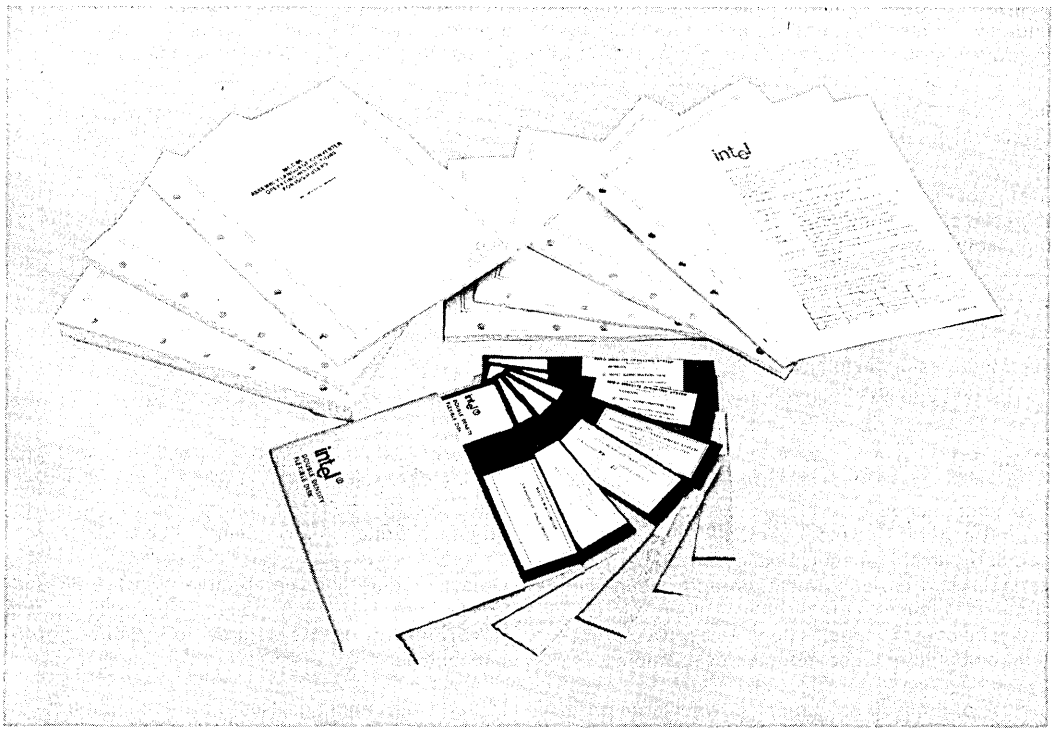
iAPX 86,88
SOFTWARE DEVELOPMENT PACKAGES FOR SERIES II/PDS

- **PL/M 86/88 High Level Programming Language**
- **ASM 86/88 Macro Assembler for iAPX 86,88 Assembly Language Programming**
- **LINK 86/88 and LOC 86/88 Linkage and Relocation Utilities**
- **CONV 86/88 Converter for Conversion of 8080/8085 Assembly Language Source Code to iAPX 86, 88 Assembly Language Source Code**
- **OH 86/88 Object-to-Hexadecimal Converter**
- **LIB 86/88 Library Manager**

The iAPX 86,88 Software Development Packages for Series II provide a set of software development tools for the iAPX 86/88 CPUs and the iSBC 86/12A single board computer. The packages operate under the ISIS-II operating system on Intel Microcomputer Development Systems—Model 800, Series II or the Personal Development System (PDS)—thus minimizing requirements for additional hardware or training for Intel Microcomputer Development System users.

These packages permit 8080/8085 users to efficiently upgrade existing programs into iAPX 86/88 code from either 8080/8085 assembly language source code or PL/M 80 source code.

For the new Intel Microcomputer Development System user, the packages operating on a PDS or an Intellec Series II, such as a Model 235, provide total iAPX 86,88 software development capability.



MAY 1983

AFN-01239E

PL/M 86/88 COMPILER FOR SERIES II/PDS

- **Language is Upward Compatible from PL/M 80, Assuring MCS-80/85™ Design Portability**
- **Supports 16-bit Signed Integer and 32-bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-to-Learn, Block-Structured Language Encourages Program Modularity**
- **Produces Relocatable Object Code Which is Linkable to All Other 8086 Object Modules**
- **Supports Full Extended Addressing Features of the iAPX 86/10 and 88/10 Microprocessors (Up to 1 Mbyte)**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**

Like its counterpart for MCS-80/85 program development, PL/M 86/88 is an advanced, structured high-level programming language. The PL/M 86/88 compiler was created specifically for performing software development for the Intel iAPX 86,88 Microprocessors.

PL/M 86/88 has significant new capabilities over PL/M 80 that take advantage of the new facilities provided by the iAPX 86,88 microsystem, yet the PL/M 86/88 language remains compatible with PL/M 80.

With the exception of hardware-dependent modules, such as interrupt handlers, PL/M 80 applications may be recompiled with PL/M 86/88 with little need for modification. PL/M 86/88, like PL/M 80, is easy to learn, facilitates rapid program development, and reduces program maintenance costs.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 86/88 compiler efficiently converts free-form PL/M language statements into equivalent 86/88 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

FEATURES

Major features of the Intel PL/M 86/88 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, global to a public module, for example).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 86/88 implementation of a block structure allows the use of REENTRANT which is especially useful in system design.

Language Compatibility

PL/M 86/88 object modules are compatible with object modules generated by all other 86/88 translators. This means that PL/M programs may be linked to programs written in any other 86/88 language.

Object modules are compatible with ICE-88 and ICE-86 units; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 86/88 Language is upward-compatible with PL/M 80, so that application programs may be easily ported to run on the iAPX 86 or 88.

Supports Five Data Types

PL/M makes use of five data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

—Byte: 8-bit unsigned number
—Word: 16-bit unsigned number
—Integer: 16-bit signed number
—Real: 32-bit floating point number
—Pointer: 16-bit or 32-bit memory address indicator

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

—Array: Indexed list of same type data elements
—Structure: Named collection of same or different type data elements
—Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the 8087 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or emulator takes place at link-time, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 86/88 language contains built-in functions for string manipulaiton. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for generating interrupts to the iAPX 86 or 88 via software. A procedure may be defined with the INTERRUPT attribute, and the compiler will automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to same and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET$INTERRUPT, the function retuning an INTERRUPT$PTR, and the PL/M statement CAUSE$INTERRUPT all add flexibility to user programs involving interrupt handling.

Segmentation Control

The PL/M 86/88 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 86/88 compiler offers four levels of optimization for significantly reducing overall program size.

—Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions.
—"Strength reductions" (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block.
—Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreadable code.
—Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations.

Compiler Controls

The PL/M 86/88 compiler offers more than 25 controls that facilitate such features as:

—Conditional compilation
—Intra- and Inter-module cross reference
—Corresponding assembly language code in the listing file
—Setting overflow conditions for run-time handling

BENEFITS

PL/M 86/88 is designed to be an efficient, cost-effective solution to the special requirements of iAPX 86 or 88 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 86/88 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 86/88, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 86/88 is designed to aid in the development of reliable software (PL/M 86/88 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

iAPX 86,88 MACRO ASSEMBLER FOR SERIES II/PDS

- **Powerful and Flexible Text Macro Facility with Three Macro Listing Options to Aid Debugging**
- **Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions**
- **"Strongly Typed" Assembler Helps Detect Errors at Assembly Time**
- **High-Level Data Structuring Facilities Such as "STRUCTUREs" and "RECORDs"**
- **Over 120 Detailed and Fully Documented Error Messages**
- **Produces Relocatable and Linkable Object Code**

ASM 86/88 is the "high-level" macro assembler for the iAPX 86,88 assembly language. ASM 86/88 translates symbolic 86/10, 88/10 assembly language mnemonics into 86/10, 88/10 relocatable object code.

ASM 86/88 should be used where maximum code efficiency and hardware control is needed. The iAPX 86,88 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 86/10, 88/10 machine instructions. ASM 86/88 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

ASM 86/88 offers many features normally found only in high-level languages. The iAPX 86,88 assembly language is strongly typed. The assembler performs extensive checks on the usage of variables and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

FEATURES

Major features of the Intel iAPX 86,88 assembler and assembly language include:

Powerful and Flexible Text Macro Facility

- Macro calls may appear anywhere
- Allows user to define the syntax of each macro
- Built-in functions
  conditional assembly (IF-THEN-ELSE, WHILE)
  repetition (REPEAT)
  string processing functions (MATCH)
  support of assembly time I/O to console (IN, OUT)
- Three Macro Listing Options include a GEN mode which provides a complete trace of all macro calls and expansions

High-Level Data Structuring Capability

- STRUCTURES: Defined to be a template and then used to allocate storage. The familiar dot notation may be used to form instruction addresses with structure fields.
- ARRAYS: Indexed list of same type data elements.
- RECORDS: Allows bit-templates to be defined and used as instruction operands and/or to allocate storage.

Fully Supports iAPX 86,88 Addressing Modes

- Provides for complex address expressions involving base and indexing registers and (structure) field offsets.
- Powerful EQU facility allows complicated expressions to be named and the name can be used as a synonym for the expression throughout the module.

Powerful STRING MANIPULATION INSTRUCTIONS

- Permit direct transfers to or from memory or the accumulator.
- Can be prefixed with a repeat operator for repetitive execution with a count-down and a condition test.

Over 120 Detailed Error Messages

- Appear both in regular list file and error print file.
- User documentation fully explains the occurrence of each error and suggests a method to correct it.

Support for ICE-86™ Emulation and Symbolic Debugging

- Debug options for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging.

Generates Relocatable and Linkable Object Code—Fully Compatible with LINK 86/88, LOC 86/88 and LIB 86/88

- Permits ASM 86/88 programs to be developed and debugged in small modules. These modules can be easily linked with other ASM 86/88 or PL/M 86/88 object modules and/or library routines to form a complete application system.

BENEFITS

The iAPX 86,88 macro assembler allows the extensive capabilities of the 86/88 CPU's to be fully exploited. In any application, time and space critical routines can be effectively written in ASM 86/88. The 86,88 assembler outputs relocatable and linkable object modules. These object modules may be easily combined with object modules written in PL/M 86/88—Intel's structured, high-level programming language. ASM 86/88 compliments PL/M 86/88 as the programmer may choose to write each module in the language most appropriate to the task and then combine the modules into the complete applications program using the iAPX 86,88 relocation and linkage utilities.

CONV 86/88
MCS®-80/85 to iAPX 86,88 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

- **Translates 8080/8085 Assembly Language Source Code to iAPX 86,88 Assembly Language Source Code**
- **Provides a Fast and Accurate Means to Convert 8080/8085 Programs to the iAPX 86/88 Facilitating Program Portability**
- **Automatically Generates Proper ASM 86/88 Directives to Set Up a "Virtual 8080" Environment that is Compatible with PL/M 86/88**

In support of Intel's commitment to software portability, CONV 86/88 is offered as a tool to move 8080/8085 programs to the iAPX 86/88. A comprehensive manual, "MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users," covers the entire conversion process. Detailed methodology of the conversion process is fully described therein.

— CONV 86/88 will accept as input an error-free 8080/8085 assembly-language source file and optional controls, and produce as output, optional PRINT and OUTPUT files.

— The PRINT file is a formatted copy of the 8080/8085 source and the 86/88 source file with embedded caution messages.

— The OUTPUT file is an 86/88 source file.

— CONV 86/88 issues a caution message when it detects a potential problem in the converted 86/88 code.

— A transliteration of the 8080/8085 programs occurs, with each 8080/8085 construct mapped to its exact 86/88 counterpart:

Registers
Condition flags
Instruction
Operands
Assembler directives
Assembler control lines
Macros

Because CONV 86/88 is a transliteration process, there is the possibility of as much as a 15%–20% code expansion over the 8080/8085 code. For compactness and efficiency it is recommended that critical portions of programs be re-coded in iAPX 86,88 assembly language.

Also, as a consequence of the transliteration, some manual editing may be required for converting instruction sequences dependent on:

—instruction length, timing, or encoding
—interrupt processing*
—PL/M parameter passing conventions*

*Mechanical editing procedures for these are suggested in the converter manual.

The accompanying figure illustrates the flow of the conversion process. Initially, the abstract program may be represented in 8080/8085 or iAPX 86,88 assembly language to execute on that respective target machine. The conversion process is porting a source destined for the 8080/8085 to the 86/88 via CONV 86/88.

AFN-01239E

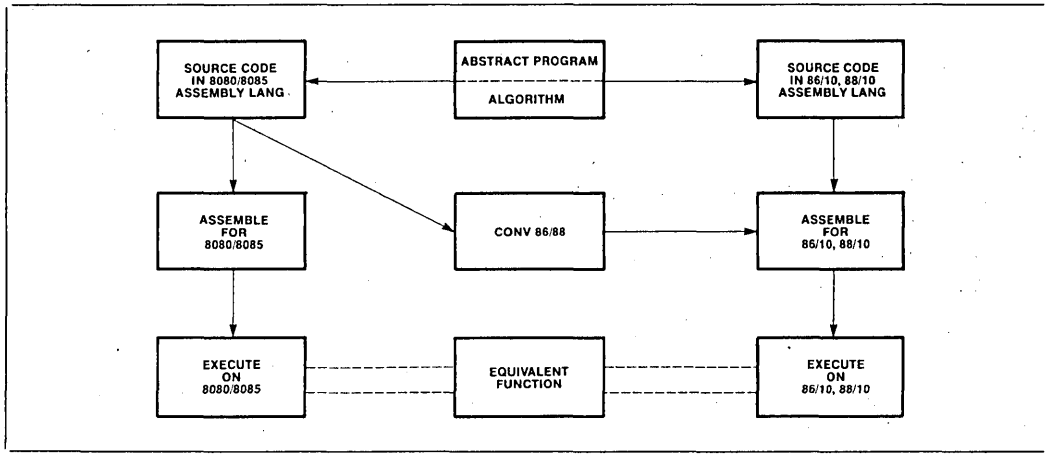


Figure 1. Porting 8080/8085 Source Code to the iAPX 86/10 and 88/10

LINK 86/88

- **Automatic Combination of Separately Compiled or Assembled iAPX 86, 88 Programs Into a Relocatable Module**
- **Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**
- **Automatic Generation of a Summary Map Giving Results of the LINK 86/88 Process**
- **Abbreviated Control Syntax**
- **Relocatable Modules may be Merged into a Single Module Suitable for Inclusion in a Library**
- **Supports "Incremental" Linking**
- **Supports Type Checking of Public and External Symbols**

LINK 86/88 combines object modules specified in the LINK 86/88 input list into a single output module. LINK 86/88 combines segments from the input modules according to the order in which the modules are listed.

LINK 86/88 will accept libraries and object modules built from PL/M 86/88, ASM 86/88, or any other translator generating Intel's iAPX 86/88 Relocatable Object Modules.

Support for incremental linking is provided since an output module produced by LINK 86/88 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

LINK 86/88 supports type checking of PUBLIC and EXTERNAL symbols reporting an error if their types are not consistent.

LINK 86/88 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the LINK 86/88 process and to control the content of the output module.

LINK 86/88 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using LOC 86/88 and enter final testing with much of the work accomplished.

LIB 86/88

- **LIB 86/88 is a Library Manager Program which Allows You to:**
  - **Create Specially Formatted Files to Contain Libraries of Object Modules**
  - **Maintain These Libraries by Adding or Deleting Modules**
  - **Print a Listing of the Modules and Public Symbols in a Library File**
- **Libraries Can be Used as Input to LINK 86/88 Which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program LIB 86/88 creates and maintains files containing object modules. The operation of LIB 86/88 is controlled by commands to indicate which operation LIB 86/88 is to perform. The commands are:

CREATE: creates an empty library file
ADD: adds object modules to a library file
DELETE: deletes modules from a library file
LIST: lists the module directory of library files
EXIT: terminates the LIB 86 program and returns control to ISIS-II

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

LOC 86/88

- **Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Lengths, and Debug Symbols and their Addresses**
- **Extensive Capability to Manipulate the Order and Placement of Segments in iAPX 86/88 Memory**
- **Abbreviated Control Syntax**
- **Automatic and Independent Relocation of Segments. Segments May Be Relocated to Best Match Users Memory Configuration**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

LOC 86/88 converts relative addresses in an input module to absolute addresses. LOC 86/88 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

LOC 86/88 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the LOC 86/88 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program on the Intellec development system and then simply relocate the object code to suit your application.

OH 86/88

- **Converts an iAPX 86/88 Absolute Object Module to Symbolic Hexadecimal Format**
- **Facilitates Preparing a File for Later Loading by a Symbolic Hexadecimal Loader, such as the iSBC™ Monitor SDK-86 Loader, or Universal PROM Mapper**
- **Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging**

The OH 86/88 utility converts an 86/88 absolute object module to the hexadecimal format. This conversion may be necessary to format a module for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or Universal PROM Mapper. The conversion may also be made to put the module in a more readable format than can be displayed or printed.

The module to be converted must be in absolute format; the output from LOC 86/88 is in absolute format.

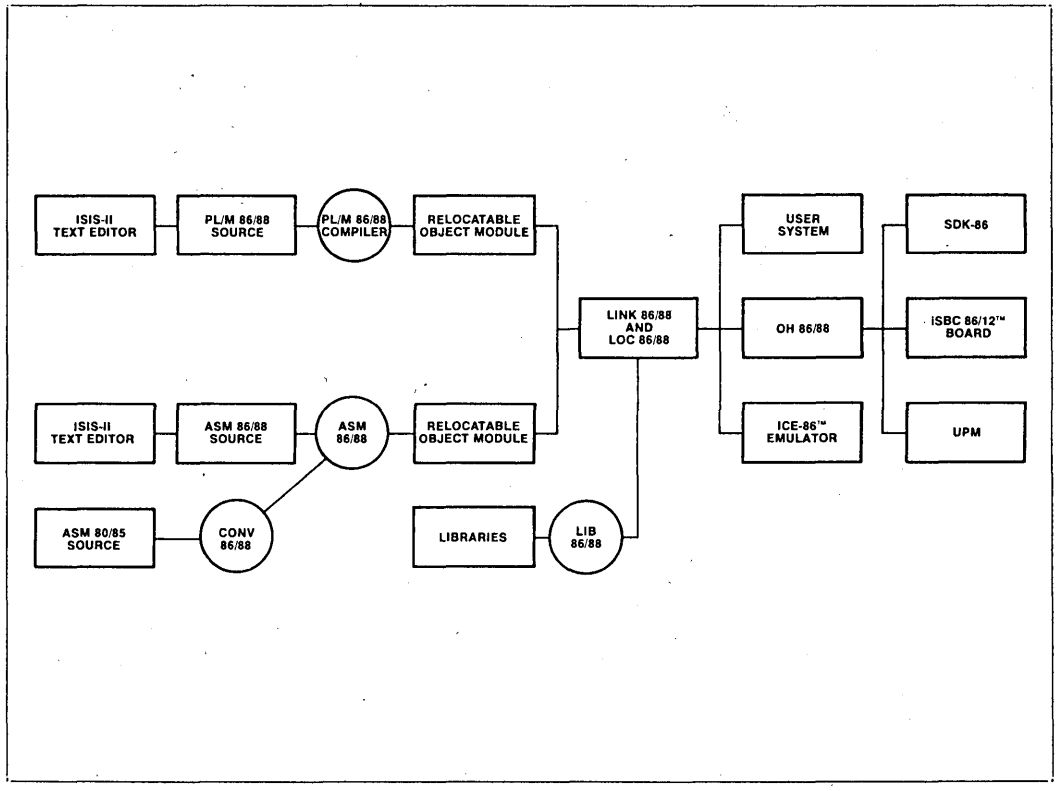


Figure 2. iAPX 86,88 Software Development Cycle

AFN-01239E

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems
Intel Personal Development System

Documentation

PL/M-86 Programming Manual

ISIS-II PL/M-86 Compiler Operator's Manual

MCS-86 User's Manual

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users

MCS-86 Macro Assembly Language Reference Manual

MCS-86 Macro Assembler Operating Instructions for ISIS-II Users

MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users

Universal PROM Programmer User's Manual

ORDERING INFORMATION

iAPX 86,88 Software Development Packages for Series II:

Part No.	Description
MDS-308*	Assembler and Utilities Package
MDS-309*	PL/M compiler and Utilities Package
MDS-311*	PL/M compiler, Assembler, and Utilities Package

All Packages Require Software Licenses

SUPPORT:

Hotline Telephone Support, Software Performance Reports (SPR), Software Updates, Technical Reports, Monthly Newsletters are available.

*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.

86/88/186/188 SOFTWARE PACKAGES

FORTRAN 86/88 Software Package

- Features High-Level Language Support for Floating-Point Calculation, Transcendentals, Interrupt Procedures, and run-time exception handling
- Meets ANS FORTRAN 77 Subset Language Specifications
- Supports Complex Data Types

PASCAL 86/88 Software Package

- Resident on iAPX 86 Based Intel Microcomputer Development Systems
- Object Compatible and Linkable with PL/M 86/88, ASM 86/88 and FORTRAN 86/88
- Supports Large Array Operation

PL/M 86/88/186/188 Software Package

- Advanced Structured System Implementation Language for Algorithm Development
- Supports 16-bit Signed Integer and 32-bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard
- Easy-to-Learn Block-Structured Language Encourages Program Modularity

iC-86 C Compiler for the 8086

- Implements Full C Language
- Produces High Density Code Rivaling Assembler
- Supports Intel Object Module Format (OMF)

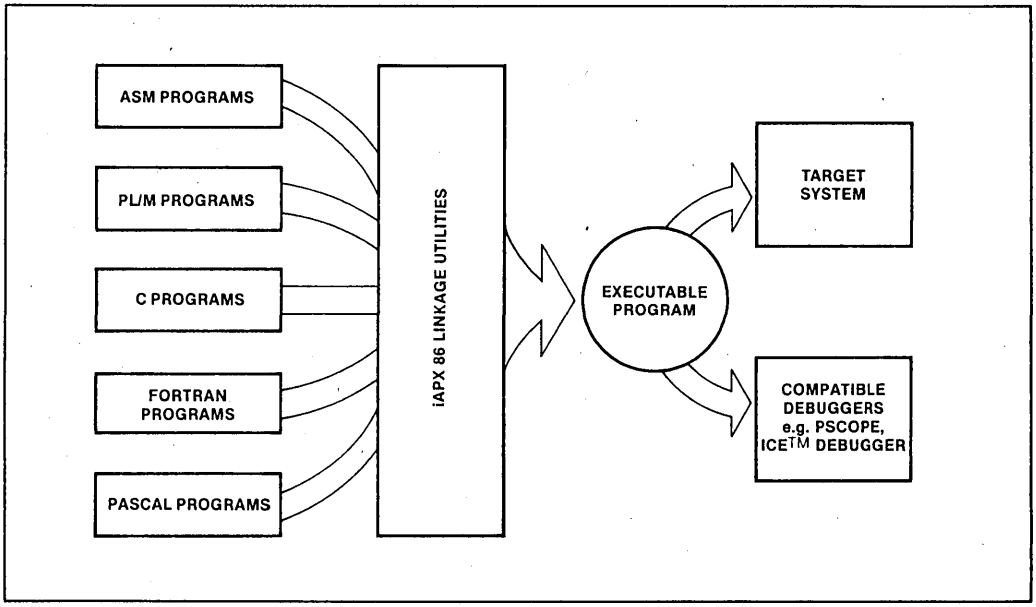


Figure 1. Program modules compiled with any of the iAPX 86 languages may be linked together. Each language is compatible with Intel's debug tools.

SEPTEMBER 1984
ORDER NUMBER: 210689-003

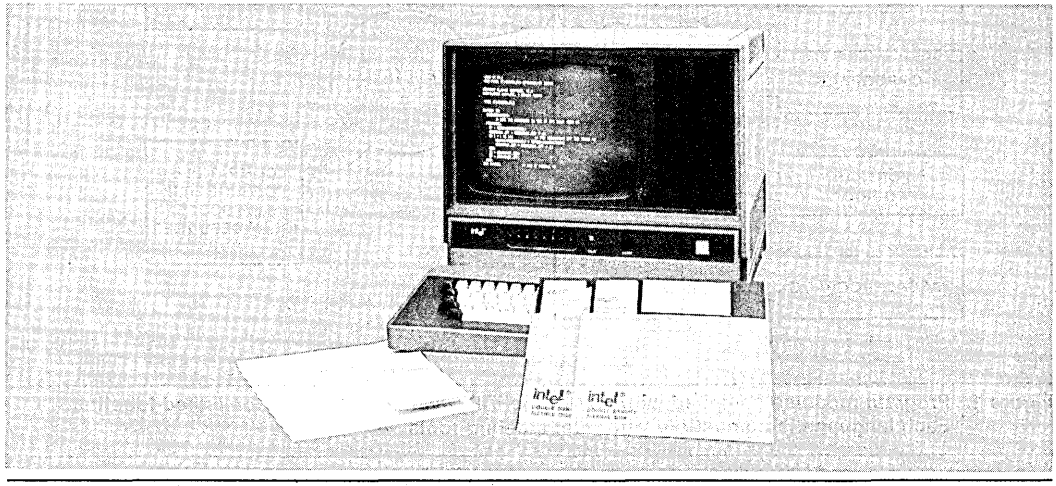
FORTRAN 86/88
SOFTWARE PACKAGE

- **Features high-level language support for floating-point calculations, transcendentals, interrupt procedures, and run-time exception handling**
- **Meets ANS FORTRAN 77 Subset Language Specifications**
- **Supports iAPX 86/20, 88/20 Numeric Data Processor for fast and efficient execution of numeric instructions**
- **Uses REALMATH Floating-Point Standard for consistent and reliable results**
- **Supports Arrays Larger Than 64K**
- **Unlimited User Program Symbols**
- **Offers powerful extensions tailored to microprocessor applications**
- **Offers upward compatibility with FORTRAN 80**
- **Provides FORTRAN run-time support for iAPX 86,88,186,188-based design**
- **Provides users ability to do formatted and unformatted I/O with sequential or direct access methods**
- **ICE™ Symbolic Debugging Fully Supported**
- **PSCOPE Source Level Debugging Fully Supported**
- **Supports complex data types**

FORTRAN 86/88 meets the ANS FORTRAN 77 Language Subset Specification and includes many features of the full standard. Therefore, the user is assured of portability of most existing ANS FORTRAN programs and of full portability from other computer systems with an ANS FORTRAN 77 Compiler.

FORTRAN 86/88 programs developed and debugged on the Intel Microcomputer Development Systems may be tested with the prototype using ICE symbolic debugging, and executed on an RMX-86 operating system, or on a user's iAPX 86,88,186,188-based operating system.

FORTRAN 86/88 is one of a complete family of compatible programming languages for iAPX 86,88,186,188 development: PL/M, Pascal, FORTRAN, and Assembler. Therefore, users may choose the language best suited for a specific problem solution.



MAY 1983

FEATURES

Extensive High-Level Language Numeric Processing Support

Single (32-bit), double (64-bit), and double extended precision (80-bit) floating-point data types

REALMATH Proposed IEEE Floating-Point Standard) for consistent and reliable results

Full support for all other data types: integer, logical, character

Ability to use hardware (iAPX 86/20, 88/20 Numeric Data Processor) or software (simulator) floating-point support chosen at link time

ANS FORTRAN 77 Standard

Intel® Microprocessor Support

FORTRAN 86/88 language features support of iAPX 86/20, 88/20 Numeric Data Processor

Compiler generates in-line iAPX 86/20, 88/20 Numeric Data Processor object code for floating-point arithmetic (See Figure 1)

Intrinsics allow user to control iAPX 86/20, 88/20 Numeric Data Processor

iAPX 86,88,186,188 architectural advantages used for indexing and character-string handling

Symbolic debugging of application using ICE emulators

Source level debugging using PSCOPE.

FLOATING-POINT-STATMENT

```
 TEMPER = (PRESS - VOLUM / QUEK) - 3.45 / (PRESS - VOLUM / QUEK)
&    - (PRESS - VOLUM / QUEK) * (PRESS - VOLUM / QUEK)
```

OBJECT CODE GENERATED

```
        Intel FORTRAN-86 Compiler

        IAPX 86/20, 88/20       ASSEMBLER MNEMONICS
        MACHINE CODE                                  ; STATEMENT # 2
        0013  9BD9060C00        FLD     VOLUM
        0018  9BD8360000        FDIV    QUEK
        001D  9BD82E0800        FSUBR   PRESS
        0022  9BDDD1            FST     TOS+1H
        0025  9B2ED83E0000      FDIVR   CS:@CONST
        002B  9BD9C9            FXCHG   TOS+1H
        002E  9BDDD2            FST     TOS+2H
        0031  9BDEE9            FSUBRP
        0034  9BD9C1            FLD     TOS+1H
        0037  9BD8C8            FMUL    TOS
        003A  9BDDC2            FFREE   TOS+2H
        003D  9BDEE1            FSUBP
        0040  9BD91E0400        FSTP    TEMPER
        0045  9B                WAIT
```

Figure 2. Object Code Generated by FORTRAN 86/88 for a Floating-Point Calculation Using iAPX 86/20, 88/20 Numeric Processor.

AFN-01653B

Microprocessor Application Support

—Direct byte- or word-oriented port I/O

—Reentrant procedures

—Interrupt procedures

Flexible Run-Time Support

Application object code may be executed in iAPX 86, 88,186,188-based environment of user's choice:

—a Series III or Series IV Intellec Development System

—an iAPX 86,88,186,188-based system with iRMX-86 Operating System

—an iAPX 86,88,186,188-based system with user-designed Operating System

Run-time exception handling for fixed-point numerics, floating-point numerics, and I/O errors

Relocatable object libraries for complete run-time support of I/O and arithmetic functions. In-line code execution is generated for iAPX 86/20, 88/20 Numeric Data Processor

BENEFITS

FORTRAN 86/88 provides a means of developing application software for the Intel iAPX 86,88,186,188 products lines in a familiar, widely accepted, and industry-standard programming language. FORTRAN 86,88 will greatly enhance the user's ability to provide cost-effective software development for Intel microprocessors as illustrated by the following:

Early Project Completion

FORTRAN is an industry-standard, high-level numerics processing language. FORTRAN programmers can use FORTRAN 86/88 on microprocessor projects with little retraining. Existing FORTRAN software can be compiled with FORTRAN 86/88 and programs developed in FORTRAN 86/88 can run on other computers with ANS FORTRAN 77 with little or no change. Libraries of mathematical programs using ANS 77 standards may be compiled with FORTRAN 86/88.

Application Object Code Portability for a Processor Family

FORTRAN 86/88 modules "talk" to the resident Intellec development operating system using Intel's standard interface for all development-system software. This allows an application developed under the ISIS-II operating system to execute on iRMX/86, or a user-supplied operating system by linking in the iRMX/86 or other appropriate interface library. A standard logical-record interface enables communication with non-standard I/O devices.

Comprehensive, Reliable and Efficient Numeric Processing

The unique combination of FORTRAN 86/88, iAPX 86/20, 88/20 Numeric Data Processor, and REALMATH (Proposed IEEE Floating-Point Standard) provide universal consistency in results of numeric computations and efficient object code generation.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Series III/Series IV)

Documentation Package

FORTRAN 86/88 User's Guide

AFN-01653B

ORDERING INFORMATION

Part Number	Description
MDS*-315	FORTRAN 86/88 Software Package

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*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.

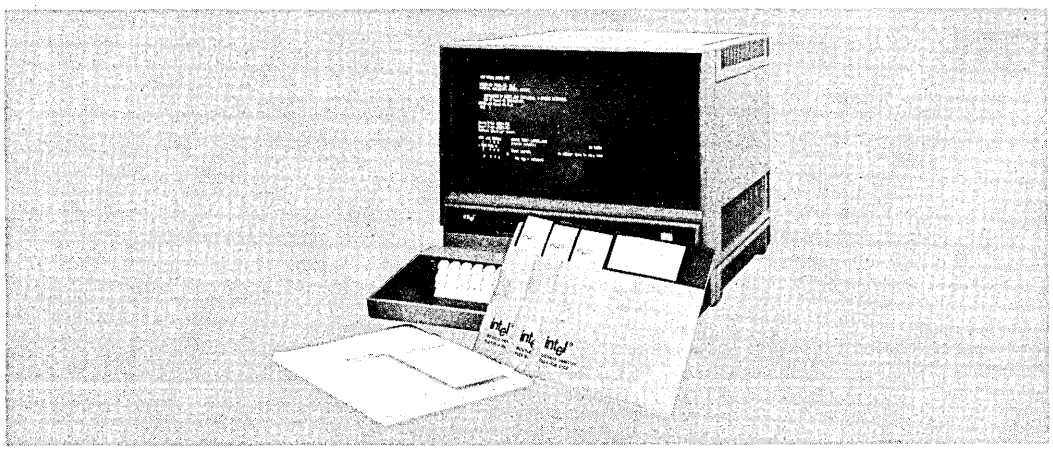
PASCAL 86/88
SOFTWARE PACKAGE

- **Resident on iAPX 86 Based Intel Microcomputer Development Systems**
- **Object Compatible and Linkable with PL/M 86/88, ASM 86/88 and FORTRAN 86/88**
- **ICE™ Symbolic Debugging Fully Supported**
- **PSCOPE Source Level Debugging Fully Supported**
- **Implements REALMATH for Consistent and Reliable Results**
- **Supports large array operation**
- **Unlimited User Program Symbols**
- **Supports iAPX86/20, 88/20 Numeric Data Processors**
- **Strict Implementation of ISO Standard Pascal**
- **Useful Extensions Essential for Microcomputer Applications**
- **Separate Compilation with Type-Checking Enforced Between Pascal Modules**
- **Compiler Option to Support Full Run-Time Range-Checking**

PASCAL 86/88 conforms to and implements the ISO Draft Proposed Pascal standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. To assist the development of portable software, the compiler can be directed to flag all non-standard features.

The PASCAL 86/88 compiler runs on Series III and Series IV Microcomputer Development Systems. A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs as an alternate to the development system environment. Program modules compiled under PASCAL 86/88 are compatible and linkable with modules written in PL/M 86/88, ASM 86/88 or FORTRAN 86/88. With a complete family of compatible programming languages for the iAPX 86, 88, 186, 188 one can implement each module in the language most appropriate to the task at hand.

PASCAL 86/88 object modules contain symbol and type information for program debugging using ICE™ emulators and PSCOPE source language debugger. For final production version, the compiler can remove this extra information and code.



FEATURES

Includes all the language features of Jensen & Wirth Pascal as defined in the ISO Draft Proposed Pascal Standard.

Supports required extensions for microcomputer applications.

—Interrupt handling
—Direct port I/O

Separate compilation extensions allow:

—Modular decomposition of large programs
—Linkage with other Pascal modules as well as PL/M 86/88/186/188, ASM 86/88/186/188 and FORTRAN 86/88.
—Enforcement of type-checking at LINK-time

Supports numerous compiler options to control the compilation process, to INCLUDE files, flag non-standard Pascal statements and others to control program listings and object modules.

Utilizes the IEEE standard for Floating-Point Arithmetic (the Intel REALMATH standard) for arithmetic operations.

Well-defined and documented run-time operating system interfaces allow the user to execute the applications under user-designed operating systems.

Predefined type extensions allow:

—Create precision in read, integer, and unsigned calculations.
—Means to check 8087 errors
—Circumvention of rigid type checking on calls to non-Pascal routines

BENEFITS

Provides a standard Pascal for iAPX 86, 88, 186, 188 based applications.

—Pascal has gained wide acceptance as the portable application language for microcomputer applications
—It is being taught in many colleges and universities around the world
—It is easy to learn, originally intended as a vehicle for teaching computer programming
—Improves maintainability: Type mechanism is both strictly enforced and user extendable
—Few machine specific language constructs

Strict implementation of the proposed ISO standard for Pascal aids portability of application programs. A compile time option checks conformance to the standard making it easy to write conforming programs.

PASCAL 86/88 extensions via predefined procedures for interrupt handling and direct port I/O make it possible to code an entire application in Pascal without compromising portability.

Standard Intel REALMATH is easy to use and provides reliable results, consistent with other Intel languages and other implementations of the IEEE proposed Floating-Point standard.

Provides run-time support for co-processors. All real-type arithmetic is performed on the 86/20 numeric data processor unit or software emulator. Run-time library routines, common between Pascal and other Intel languages (such as FORTRAN), permit efficient and consistently accurate results.

Extended relocation and linkage support allows the user to link Pascal program modules with routines written in other languages for certain parts of the program. For example, real-time or hardware dependent routines written in ASM 86/88/186/188 or PL/M 86/88/186/188 can be linked to Pascal routines, further extending the user's ability to write structured and modular programs.

PASCAL 86/88 programs "talk" to the resident operating system using Intel's standard interface for translated programs. This allows users to replace the development operating system by their own operating systems in the final application.

PASCAL 86/88 takes full advantage of iAPX 86, 88, 186, 188 high level language architecture to generate efficient machine code.

Compiler options can be used to control the program listings and object modules. While debugging, the user may generate additional information such as the symbol record information required and useful for debugging using PSCOPE or ICE emulation. After debugging, the production version may be streamlined by removing this additional information.

AFN-01652B

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

Intel Microcomputer Development Systems (Series III, Series IV)

Documentation Package

PASCAL 86 User's Guide

ORDERING INFORMATION

Part Number	Description
MDS*-314	PASCAL 86/88 Software Package

Requires software license.

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

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

AFN-01652B

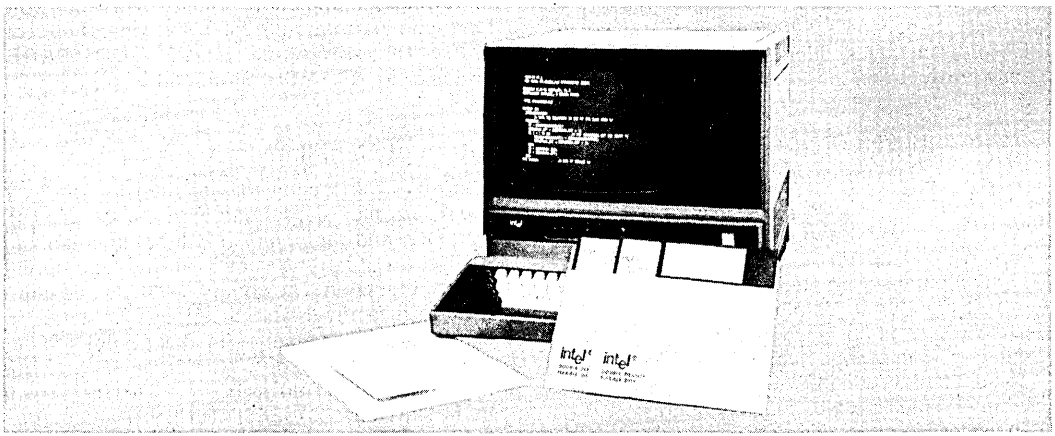
PL/M 86/88/186/188 Software Package

- **Systems Programming Language for the iAPX 86/88/186/188 Processors**
- **Language Is Upward Compatible from PL/M 80, Assuring MCS®-80/85 Design Portability**
- **Advanced Structured System Implementation Language for Algorithm Development**
- **Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-to-Learn Block-Structured Language Encourages Program Modularity**
- **Improved Compiler Performance Now Supports More User Symbols and Faster Compilation Speeds**
- **Produces Relocatable Object Code Which Is Linkable to All Other 8086 Object Modules**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**
- **Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors**
- **Resident on iAPX 86 Intel Microcomputer Development Systems**

PL/M 86 is an advanced, structured, high-level systems programming language. The PL/M 86 compiler was created specifically for performing software development for the Intel 8086, 8088, 80186 and 80188 Microprocessors. PL/M was designed so that program statements naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 86 compiler efficiently converts free-form PL/M language statements into machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-up maintenance costs for the user.



NOTE: The Intellec® Development System pictured here is not included with the PL/M 86/88 Software package but merely depicts a language in its operating environment.

MAY 1983

FEATURES

Major features of the Intel PL/M 86 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 86 implementation of a block structure allows the use of REENTRANT (recursive) procedures, which are especially useful in system design.

Language Compatibility

PL/M 86 object modules are compatible with object modules generated by all other iAPX 86 translators. This means that PL/M programs may be linked to programs written in any other iAPX 86 language.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 86 Language is upward compatible with PL/M 80, so that application programs may be easily ported to run on the iAPX 86.

Supports Seven Data Types

PL/M makes use of seven data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

—Byte: 8-bit unsigned number
—Word: 16-bit unsigned number
—DWORD: 32-bit unsigned number
—Integer: 16-bit signed number
—Read: 32-bit floating point number
—Pointer: 16-bit or 32-bit memory address indicator
—Selector: 16-bit base portion of a pointer

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These help the user to organize data into logical groups.

— Array: Indexed list of same type data elements
— Structure: Named collection of same or different type data elements
— Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the iAPX 86/20 or 88/20 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or Emulator takes place at linktime, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 86 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for handling interrupts. A procedure may be defined with the INTERRUPT attribute, and the compiler will automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to save and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET$INTERRUPT, the function retuning an INTERRUPT$PTR, and the PL/M statement CAUSE$INTERRUPT all add flexibility to user programs involving interrupt and handling.

AFN-01661C

Compiler Controls

Including several that have been mentioned, the PL/M 86 compiler offers more than 25 controls that facilitate such features as:

— Conditional compilation
— Including additional PL/M source files from disk
— Corresponding assembly language code in the listing file
— Setting overflow conditions for run-time handling

Segmentation Control

The PL/M 86 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 86 compiler offers four levels of optimization for significantly reducing overall program size.

— Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions
— "Strength reductions" (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block
— Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreachable code
— Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations

Error Checking

The PL/M 86 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation errors is provided by the compiler.

```
M:DO; /* Beginning of module */
      SORTPROC: PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX) PUBLIC;
                DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER,
      /* Parameters:
           PTR is pointer to first record.
           COUNT is number of records to be sorted.
           RECSIZE is number of bytes in each record—max is 128.
           KEYINDEX is byte position within each record of a BYTE scalar
                    to be used as sort key. */
                DECLARE RECORD BASED PTR (1) BYTE,
                        CURRENT (128) BYTE,
                        (I, J) INTEGER;
      SORT:     DO J=1 TO COUNT-1;
                     CALL MOVB(@RECORD(J*RECSIZE), @CURRENT, RECSIZE);
                     I=J;
      FIND:          DO WHILE I >0
                               AND RECORD((I-1)*RECSIZE - KEYINDEX)
                                  >CURRENT(KEYINDEX);
                          CALL MOVB(@RECORD((I-1)*RECSIZE),
                                    @RECORD(I*RECSIZE),
                                    RECSIZE);
                          I=I-1;
                     END FIND;
                     CALL MOVB(@CURRENT, @RECORD(I*RECSIZE), RECSIZE);
                END SORT;
           END SORTPROC;
END M;     /*End of module*/
```

PUBLIC and EXTERNAL attributes promote program modularity.

"Based" Variables allow manipulation of external data by passing the base of the data structure (a pointer). This minimizes the STACK space used for parameter passing, and the execution time to perform many STACK operations.

The "AT" operator returns the *address* of a variable, instead of its contents. This is very useful in passing pointers for based variables.

One of several PL/M built-in procedures for string manipulation.

Figure 3. Sample PL/M 86 Program.

AFN-01661C

BENEFITS

PL/M 86 is designed to be an efficient, cost-effective solution to the special requirements of iAPX 86 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Cost-Effective Alternative to Assembly Language

PL/M 86 programs are code efficient. PL/M 86 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the iAPX 86 architecture. Consequently, for the development of systems software, PL/M 86 is the cost-effective alternative to assembly language programming.

Low Learning Effort

PL/M is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 86, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because fewer programming resources are required for a given programmed function.

Increased Reliability

PL/M 86 is designed to aid in the development of reliable software (PL/M 86 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE:

Intel Microcomputer Development Systems (Series III/Series IV)

Documentation Package

PL/M-86 User's Guide for 8086-based Development Systems (121636)

SUPPORT:

Hotline Telephone Support, Software Performance Reporting (SPR), Software Updates, Technical Reports, Monthly Newsletter available.

ORDERING INFORMATION

Part Number	Description
MDS-313*	PL/M 86 Software Package

Requires Software License

*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.

iC-86
C COMPILER FOR THE 8086

- **Implements full C Language**
- **Produces high density code rivaling assembler**
- **Supports Intel Object Module Format (OMF)**
- **Runs under the Intel UDI on Intel Development Systems and iRMX™ 86**
- **Available for the VAX/VMS* Operating System**
- **Supports both small and large models of computation**
- **Supports PSCOPE-86 and I²ICE™**
- **Supports IEEE Floating Point Math with 8087 coprocessor**
- **Supports Bit Fields**
- **Supports full standard I/O Library (STDIO)**
- **Written in C**

The C Programming Language was originally designed in 1972 and has become increasingly popular as a systems development language. C is not a "very high level" language and is not tied to any specific application area. Although it is used for writing operating systems, it has been used equally well to write numerical, text-processing and data base programs. C combines the flexibility and programming speed of a higher level language with the efficiency and control of assembly language.

Intel iC-86 brings the full power of the C programming language to 8086 and 8088 based microprocessor systems.

Intel iC-86 supports the full C language as described in the Kernighan and Ritchie book, "The C Programming Lanugage," (Prentice-Hall, 1978). Also included are the latest enhancements to the C language: structure assignments, functions taking structure arguments and returning structures, and the "void" and "enum" data types.

C is rapidly becoming the standard microprocessor system implementation language because it provides:

1. the ability to manipulate the fundamental objects of the machine (including machine addresses) as easily as assembly language.
2. the power and speed of a structured language supporting a large number of data types, storage classes, expressions and statements,
3. processor independence (most programs developed for other processors can be easily transported to the 8086), and
4. code that rivals assembly language in efficiency

INTEL iC-86 COMPILER DESCRIPTION

The iC-86 compiler operates in four phases: preprocessor, parser, code generator, and optimizer. The preprocessor phase interprets directives in C source code, including conditional compilations (# define). The parser phase converts the C program into an intermediate free form and does all syntactic and semantic error checking. The code generator phase converts the parser's output into an efficient intermediate binary code, performs constant folding, and features an extremely efficient register allocator, ensuring high quality code. The optimizer phase converts the output of the code generator into

relocatable Intel Object Module Format (OMF) code, without creating an intermediate assembly file. Optionally, the iC-86 compiler can produce a symbolic assembly like file. The iC-86 optimizer eliminates common code, eliminates redundant loads and stores, and resolves span dependencies (shortens branches) within a program.

The iC-86 runtime library consists of a number of functions which the C programmer can call. The runtime system includes the standard I/O library (STDIO), conversion routines, routines for manipulating strings, special routines to perform functions not available on the 8086 (32-bit arithmetic and emulated floating point), and (where appropriate) routines for interfacing with the operating system.

iC-86 uses Intel's linker and locator and generates debug records for symbols and lines on request, permitting access to Intel's PSCOPE AND I²ICE™ to aid in program testing.

FEATURES

Support for Small and Large Models

Intel iC-86 supports both the SMALL and LARGE modes of segmentation. A SMALL model program can have up to 64K bytes of code and 64K bytes of data, with all pointers occupying two bytes. Because two byte pointers permit the generation of highly compact and efficient code, this model is recommended for programs that can meet the size restrictions. The LARGE segmentation model is used by programs that require access to the full addressing space of the 8086/8088 processors. In this model, each source file generates a distinct pair of code and data segments of up to 64K bytes in length. All pointers are four bytes long.

Preprocessor Directives

#define—defines a macro

#include—includes code outside of the program source file

#if—conditionally includes or excludes code

Other preprocessor directives include #undef, #ifdef, #ifndef, #else, #endif, and #line.

Statements

The C language supports a variety of statements:

Conditionals: IF, IF-ELSE

Loops: WHILE, DO-WHILE, FOR

Selection of cases: SWITCH, CASE, DEFAULT

Exit from a function: RETURN

Loop control: CONTINUE, BREAK

Branching: GOTO

Expressions and Operators

The C language includes a rich set of expressions and operators.

Primary expression: invoke functions, select elements from arrays, and extract fields from structures or unions

Arithmetic operators: add, subtract, multiply, divide, modulus

Relational operators: greater than, greater than or equal, less than, less than or equal, not equal

Unary operators: indirect through a pointer, compute an address, logical negation, ones complement, provide the size in bytes of an operand.

Logical operators: AND, OR

Bitwise operators: AND, exclusive OR, inclusive OR, bitwise complement

Data Types and Storage Classes

Data in C is described by its type and storage class. The type determines its representation and use, and the storage class determines its lifetime, scope, and storage allocation. The following data types are fully supported by iC-86.

char
an 8 bit signed integer

int
a 16 bit signed integer

short
same as int (on the 8086)

long
a 32 bit signed integer

unsigned
a modifier for integer data types (char, int, short, and long) which doubles the positive range of values

float
a 32 bit floating point number which utilizes the 8087 or a software floating point library

double
a 64 bit floating point number

void
a special type that cannot be used as an operand in expressions; normally used for functions called only for effect (to prevent their use in contexts where a value is required).

enum
an enumerated data type

These fundamental data types may be used to create other data types including: arrays, functions, structures, pointers, and unions.

The storage classes availabe in iC-86 include:

register
suggests that a variable be kept in a machine register, often enhancing code density and speed

extern
a variable defined outside of the function where it is declared; retaining its value throughout the entire program and accessible to other modules

auto
a local variable, created when a block of code is entered and discarded when the block is existed

static
a local variable that retains its value until the termination of the entire program

typedef
defines a new data type name from existing data types

BENEFITS

Faster Compilation

Intel iC-86 compiles C programs substantially faster than standard C compilers because it produces Intel OMF code directly, eliminating the traditional intermediate process of generating an assembly file.

Portability of Code

Because Intel iC-86 supports the STDIO and produces Intel OMF code, programs developed on a variety of machines can easily be transported to the 8086.

Rapid Program Development

Intel iC-86 provides the programmer with detailed error messages and access to PSCOPE-86 and I²ICE™ to speed program development.

Full Manipulation of the 8086

Intel iC-86 enables the programmer to utilize features of the C language to control bit fields, pointers, addresses and register allocation, taking full advantage of the fundamental concepts of the 8086.

SPECIFICATIONS

Operating Environment

The iC-86 compiler runs host resident on both the Intel Series III Microcomputer Development System under ISIS-II and on the System 86/330 under the iRMX™ 86 operating system. iC-86 can also run as a cross compliler on a VAX 11/780 computer under the VMS operating system 128 KBytes of User Memory is required on all versions. Specify desired version when ordering.

Required Hardware

Development System Version

—Intellec® Microcomputer Development System; Series III or Series IV

—Dual Diskette Drives, Single or Double Density

—System Console; CRT or Hardcopy Interactive Device

iRMX 86 version:

—Any iAPX 86/88, iSBC® 86/88, iTPS 86/XXX, or SYS 86/3XX based system capable of running the iRMX 86 Operating System

VAX version:

—Digital Equipment Corporation VAX 11/780 or compatible computer

AFN-00144C

Optional Hardware

ISIS-II version:

—ICE-86, I²ICE-86

iRMX 86 version:

—Numeric Data Processors for support of the REALMATH standard

VAX version:

—None

Required Software

ISIS-II version:

—ISIS-II Diskette Operating System

—Series III or Series IV Operating System

iRMX 86 version:

—iRMX 86 Realtime Multiprogramming Operating System

—iRMX 860 Utilities Package

VAX version:

—VMS Operating System

Optional Software

Development System Version:

—None

iRMX 86 version:

—None

VAX version:

—MDS*-384 Kit-Mainframe Link for distributed development, or iMDX-394 Asynchronous Communications Link.

—VAX iAPX 86/88/186 MACRO Assembler and utilities package (iMDX-341VX)

Documentation Package

The C Programming Language by Kernighan and Ritchie (1978 Prentice-Hall)

iC-86 User Manual

Shipping Media

Development System Version:

—Two single and one double density ISIS-II format 8" diskettes, one 5 1/4" Series IV Format

iRMX 86 version:

—Double Density iRMX 86 format 8" diskette

—Double Density iRMX 86 format 5¼" diskette

VAX version:

—1600 bpi, 9 track Magnetic tape

ORDERING INFORMATION

Order Code	Description
iMDX-317	iC-86 Compiler for ISIS-II
iRMX-866	iC-86 Compiler for iRMX 86
iMDX-347	iC-86 Cross Compiler for VAX/VMS

Intel Software License required.

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*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.

FORTRAN 80
8080/8085 ANS FORTRAN 77
INTELLEC® RESIDENT COMPILER

- **Meets ANS FORTRAN 77 Subset Language Specification plus adds Intel® microprocessor extensions**
- **Supports Intel Floating Point Standard with the FORTRAN 80 software routines, the iSBC-310™ High Speed Mathematics Board, or the iSBC-332™ math multimodule**
- **Executes on Intellec Microcomputer Development System, Intellec Series II Microcomputer Development System, and Personal Development System**
- **Supports full symbolic debugging with ICE-80™ and ICE-85™**
- **Produces relocatable and linkable object code compatible with resident PL/M 80 and 8080/8085 Macro Assembler**
- **Provides optional run-time library to execute in RMX-80™ environment**
- **Has well defined I/O interface for configuration with user-supplied drivers**

FORTRAN 80 is a computer industry-standard, high-level programming language and compiler that translates FORTRAN statements into relocatable object modules. When the object modules are linked together and located into absolute program modules, they are suitable for execution on Intel 8080/8085 Microprocessors, iSBC-80 OEM Computer Systems, Intellec Microcomputer Development Systems and Personal Development Systems. FORTRAN 80 meets the ANS FORTRAN 77 Language Subset Specification[1]. In addition, extensions designed specifically for microprocessor applications are included. The compiler operates on the Intellec Microcomputer Development System and Personal Development System under the ISIS-II Disk Operating Systems and produces efficient relocatable object modules that are compatible for linkage with PL/M 80 and 8080/8085 Macro Assembler modules.

The ANS FORTRAN 77 language specification offers many powerful extensions to the FORTRAN language that are especially well suited to Intel 8080/8085 Microprocessor software development. Because FORTRAN 80 conforms to the ANS FORTRAN 77 standard, the user is assured of compatibility with existing FORTRAN software that meets the standard as well as a guarantee of upward compatibility to other computer systems supporting an ANS FORTRAN 77 Compiler.

[1] ANSI X3J3/90

MAY 1983
ORDER NUMBER:400610-001

FORTRAN 80 LANGUAGE FEATURES

Major ANS FORTRAN 77 features supported by the Intel FORTRAN 80 Programming Language include:

- Structured Programming is supported with the IF ... THEN ... ELSE IF ... ELSE ... END IF constructs.
- CHARACTER data type permits alphanumeric data to be handled as strings rather than characters stored in array elements.
- Full I/O capabilities include:
  - — Sequential and Direct Access files
  - — Error handling facilities
  - — Formatted, Free-formatted, and Unformatted data representation
  - — Internal (in-memory) file units provide capability to format and reformat data in internal memory buffers
  - — List Directed Formatting
- Supports arrays of up to seven dimensions.
- Supports logical operators
  - .EQV. — Logical equivalence
  - .NEQV. — Logical nonequivalence

Major extensions to FORTRAN 77 in Intel FORTRAN-80 include:

- Direct 8080/8085 port I/O supported by intrinsic subroutines.
- Binary and Hexadecimal integer constants.
- Well defined interface to FORTRAN-80 I/O statements (READ, OPEN, etc.), allowing easy use of user-supplied I/O drivers.
- User-defined INTEGER storage lengths of 1, 2 or 4 bytes.
- User-defined LOGICAL storage lengths of 1, 2 or 4 bytes.
- REAL STORAGE lengths of 4 bytes.
- Bitwise Boolean operations using logical operators on integer values.
- Hollerith data constants.
- Implicit extension of the length of an integer or logical expression to the length of the left-hand side in an assignment statement.
- A format descriptor to suppress carriage return on a terminal output device at the end of the record.

FORTRAN 80 COMPILER FEATURES

- Supports multiple compilation units in single source file.
- Optional Assembly Language code listing.
- Comprehensive cross-reference, symbol attribute and error listing.
- Compiler controls and directives are compatible with other Intel language translators.
- Optional Reentrancy.
- User-defined default storage lengths.
- Optional FORTRAN 66 Do Loop semantics.
- Source files may be prepared in free format.
- The INCLUDE control permits specified source files to be combined into a compilation unit at compile time.
- Transparent interface for software and hardware floating point support, allowing either to be chosen at time of linking.

FORTRAN 80 BENEFITS

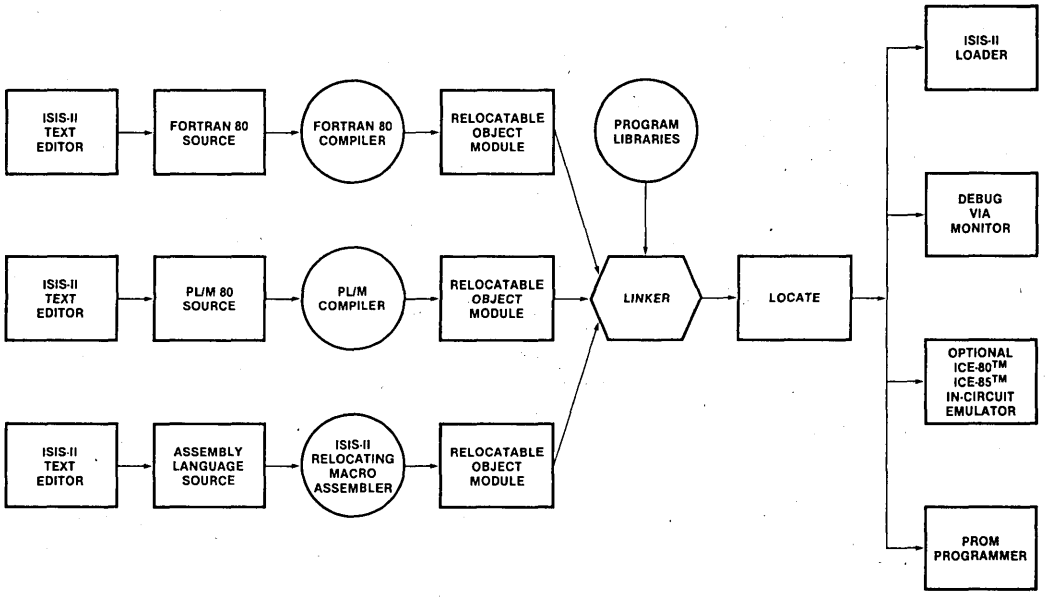
FORTRAN 80 provides a means of developing application software for Intel MCS-80/85 products in a familiar, widely accepted, and computer industry-standardized programming language. FORTRAN 80 will greatly enhance the user's ability to provide cost-effective solutions to software development for Intel microprocessors as illustrated by the following:

- *Completely Complementary to Existing Intel Software Design Tools* — Object modules are linkable with new or existing Assembly Language and PL/M Modules.
- *Incremental Runtime Library Support* — Runtime overhead is limited only to facilities required by the program.
- *Low Learning Effort* — FORTRAN 80, like PL/M, is easy to learn and use. Existing FORTRAN software can be ported to FORTRAN 80, and programs developed in FORTRAN 80 can be run on any other computer with ANS FORTRAN 77.
- *Earlier Project Completion* — Critical projects are completed earlier than otherwise possible because FORTRAN 80 will substantially increase programmer productivity, and is complementary to PL/M Modules by providing comprehensive arithmetic, I/O formatting, and data management support in the language.
- *Lower Development Cost* — Increases in programmer productivity translates into lower software development costs because less programming resources are required for a given function.
- *Increased Reliability* — The nature of high-level languages, including FORTAN 80, is that they lend themselves to simple statements of the program algorithm. This substantially reduces the risk of costly errors in systems that have already reached production status.
- *Easier Enhancements and Maintenance* — Like PL/M, program modules written in FORTRAN 80 are easier to read and understand than assembly language. This means it is easier to enhance and maintain FORTRAN 80 programs as system capabilities expand and future products are developed.
- *Comprehensive, Yet Simple Project Development* — The Intellec Microcomputer Development System and Personal Development System, with the 8080/8085 Macro Assembler, PL/M 80 and FORTRAN 80 are the most comprehensive software design facilities available for the Intel MCS-80/85 Microprocessor family. This reduces development time and cost because expensive (and remote) timesharing or large computers are not required.

SAMPLE FORTRAN-80 SOURCE PROGRAM LISTING

```
*    **  THIS PROGRAM IS AN EXAMPLE OF ISIS-II FORTRAN-80 THAT
*    **  CONVERTS TEMPERATURE BETWEEN CELSIUS AND FARENHEIT

      PROGRAM CONVRT

      CHARACTER*1 CHOICE, SCALE

      PRINT 100
*     **  ENTER CONVERSION SCALE (C OR F)
10    PRINT 200
      READ (5,300) SCALE

        IF (SCALE .EQ. 'C')
     +     THEN
              PRINT 400
*             **  ENTER THE NUMBER OF DEGREES FARENHEIT
              READ (5,*) DEGF
              DEGC = 5./9.*(DEGF-32)
*             **  PRINT THE ANSWER
              WRITE (6,500) DEGF,DEGC
*             **  RUN AGAIN?
20            PRINT 600
              READ (5,300) CHOICE
                 IF (CHOICE .EQ. 'Y')
     +              THEN
                       GOTO 10
                 ELSE IF (CHOICE .EQ. 'N')
     +              THEN
                       CALL EXIT
                 ELSE
                    GOTO 20
                 END IF
        ELSE IF (SCALE .EQ. 'F')
     +     THEN
*             **  CONVERT FROM FARENHEIT TO CELSIUS
              PRINT 700
              READ (5,*) DEGC
              DEGF = 9./5.*DEGC+32.
*             **  PRINT THE ANSWER
              WRITE (6,800) DEGC,DEGF
              GOTO 20
        ELSE
*          **  NOT A VALID ENTRY FOR THE SCALE
           WRITE (6,900) SCALE
           GOTO 10
        END IF
100   FORMAT(' TEMPERATURE CONVERSION PROGRAM',//,
     +' TYPE C FOR FARENHEIT TO CELSIUS OR',/,
     +' TYPE F FOR CELSIUS TO FARENHEIT',//)
200   FORMAT(/,' CONVERSION? ',$)
300   FORMAT(A1)
400   FORMAT(/,'ENTER DEGREES FARENHEIT: ',$)
500   FORMAT(/,F7.2,' DEGREES FARENHEIT = ',F7.2,' DEGREES CELSIUS')
600   FORMAT(/,' AGAIN (Y OR N)? ',$)
700   FORMAT(/,' ENTER DEGREES CELSIUS: ',$)
800   FORMAT(/,F7.2,' DEGREES CELSIUS = ',F7.2,' DEGREES FARENHEIT',/)
900   FORMAT(/,1H ,A1,' NOT A VALID CHOICE - TRY AGAIN!',/)
      END
```

The FORTRAN 80 Compiler is an efficient, multiphase compiler that accepts source programs, translates them into relocatable object code, and produces requested listings. After compilation, the object program may be linked to other modules, located to a specific area of memory, then executed. The diagram shown below illustrates a program development cycle where the program consists of modules created by FORTRAN 80, PL/M 80 and the 8080/8085 Macro Assembler.



SPECIFICATIONS

OPERATING ENVIRONMENT

Required Hardware:

1. Intel Microcomputer Development Systems
   —MDS-800 and Series II

   or

2. Personal Development System

DOCUMENTATION PACKAGE

FORTRAN-80 Programming Manual

ISIS-II FORTRAN-80 Compiler Operator's Manual

FORTRAN-80 Programming Reference Card

ORDERING INFORMATION

PART NO.	DESCRIPTION
Model MDS-301	FORTRAN 80 Compiler for Intellec Microcomputer Development Systems

Requires Software License.

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*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.

PASCAL 80
SOFTWARE PACKAGE

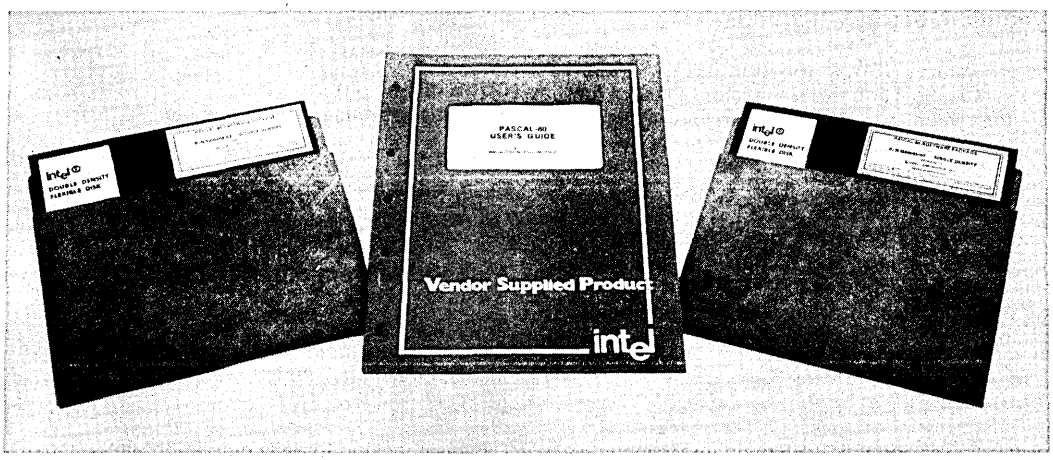
- **Offers a Superset of Standard Pascal**
- **Provides Highly Structured Language with Powerful Data Type Definitions to Suit Applications**
- **Compiles Pascal Source Code into Intermediate Code to Optimize Execution Speed and Storage**
- **Executes Compiler and Interprets the Intermediate Code on Intellec® Microcomputer Development Systems**
- **Provides a Utility to Produce Relocatable Object Modules Compatible with Other Intel® Languages**
- **Can Call Routines Written in PL/M 80, FORTRAN 80, or 8080/8085 Macro Assembler**
- **Allows Modular Breakdown of Large Programs and Separate Compilation of Individual Modules**
- **Gives Application Control Over Run-Time Errors by Providing User-Declared Error Procedures**

PASCAL 80 Software Package consists of a compiler and an interactive Run-Time System designed to provide the Pascal programming language as a software development tool for Intellec Development System Users.

Pascal is a highly-structured, block-oriented programming language that is now gaining wide acceptance as a powerful software development tool. Its rigid structure encourages and enforces good programming techniques, which, combined with a high level of readability, helps produce more reliable software.

Standard Intel development tools, such as CREDIT editor can be used to create and modify Pascal source programs. The compiler compiles this source and creates a P-Code file. The Run-Time System executes this P-Code in an interpretive manner under ISIS-II.

*Pascal language as defined in *PASCAL User Manual and Report*, Second Edition, Kathleen Jenson and Niklaus Wirth.



LANGUAGE FEATURES

Data Structures

Pascal allows the user to define labels, constants, data types, variables, procedures, and functions.

Variable Types

Variables can be defined according to the following system-defined data types: boolean, integer, real, character, array, record, string, set, file, and pointer.

User-Defined Types

New types can be defined by the user for added flexibility.

File Handling Procedures

Pascal provides procedures to allow a user's program to interface with the ISIS-II file manager. Routines provided are: RESET, REWRITE, CLOSE, PUT, GET, SEEK, and PAGE.

Input/Output Procedures

Routines are provided to interact with the console or an ISIS file. These procedures are: READ, WRITE, READLN, WRITELN, plus BUFFER and BLOCK Read and Write.

Dynamic Memory Allocation

The procedures NEW, MARK, and RELEASE allow the user to obtain and release memory space at run-time for dynamically allocating variable storage.

String Handling

Pascal provides powerful tools for defining and manipulating strings and character arrays. These facilities enable concatenation of strings, character and pattern scans, insertion, deletion, and pointer manipulation.

Recursion

Pascal allows a PROCEDURE definition to include a call to itself, a powerful construct in many mathematical algorithms.

PROGRAM TRACING FACILITY

The PASCAL 80 System incorporates a program tracing facility which allows for selectively monitoring the execution of a Pascal program. When the TRACE flag is set, the line number of each program statement being executed is output to the console.

The TRACE flag may be manipulated in two ways:

—The TRACEON command (of the Run-Time System) will set the flag, and the TRACEOFF command will reset the flag.

—Pressing the Interrupt 4 switch on the Intellec System front panel will toggle the TRACE flag; i.e., the flag will be set if it was reset, and vice-versa.

COMPILER DIRECTIVES (PARTIAL LIST)

Compiler Command Line Directives

NOLIST
No list file is produced; used for fast compilation of "clean" programs.

NOCODE
No code file is produced; used for syntax error checking.

ERRLIST
List file is limited to only those Pascal lines that contain errors, along with the error messages produced.

LIST (file-name)
Specifies the name of the list file.

CODE (file-name)
Specifies the name of the code file.

NOECHO
Error lines are echoed on the console unless this directive is specified.

Embedded Compiler Directives

$C text
Causes text to appear in code file (allows for comments, copyrights, etc.).

$I+
Causes checking for I/O completion after each I/O transfer. Failure results in a run-time error. ($I− causes no checking, and no errors on I/O failure.)

AFN-01233B

$R+
Causes Range Checking to occur, so that an out-of-range value causes a Run-Time error. ($R− suppresses generation of code for Range Checking.)

$O+
Causes the compiler to operate in overlay mode. Overlays allow less source code to reside in memory. ($O− causes no overlays, which decreases compile time, since there are fewer disk accesses.)

$T+
Causes the compiler to generate tracing instructions to be used by the TRACE facility. ($T− suppresses tracing instructions.)

BENEFITS

Brings Pascal to Intellec Microcomputer Development Systems:

—Pascal is a block-structured, highly-readable programming language, suitable for a wide-range of applications.

—Pascal is being acclaimed as the programming language of the future; it is being taught in many colleges and universities around the country.

—PASCAL 80 Run-Time System provides great ease in programming formatted I/O operations.

PASCAL 80 provides a portable language for application programs running under ISIS-II.

PASCAL 80 can be used to evaluate complicated algorithms using a natural language.

PASCAL 80 compiler generates intermediate Pseudo-code.

—P-code is optimized for speed and storage space.

—P-code is approximately 50% to 70% smaller than corresponding machine code.

—P-code is machine independent, providing code portability to any CPU.

Makes the Intellec Development System a more valuable tool. Extension of software support to include Pascal makes software development and resource management more flexible.

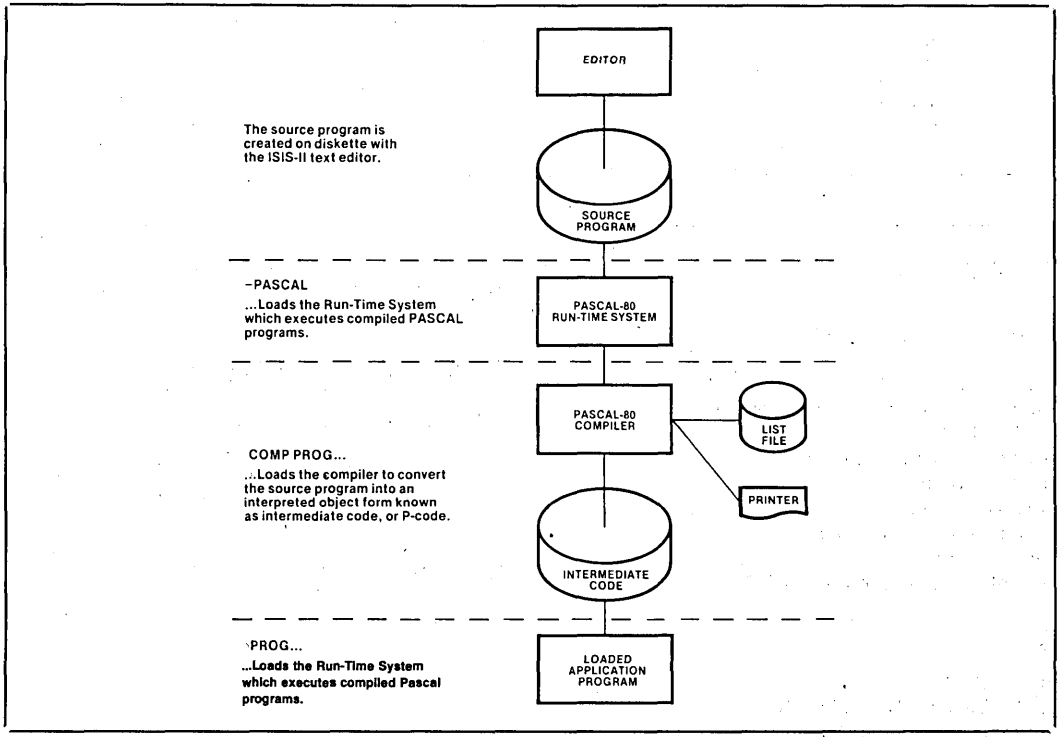


Figure 1. Program Development Cycle

AFN-01233B

Table 1. Sample Program Listing Showing Nesting Levels

BUFFER.PAS Program Listing

```
Line  Seg  Proc  Lev  Disp
   1    1     1          1  program example;
   2    1     1          3
   3    1     1          3  { Example using bufferread and bufferwrite with break characters }
   4    1     1          3
   5    1     1          3  var buffer: string;
   6    1     1         44      disk_storage: file;
   7    1     1         64      break: char;
   8    1     1         65      new_len, len: integer;
   9    1     1         67      buff_array: packed array[0..80] of char;
  10    1     1        108
  11    1     1    0     0  begin
  12    1     1    1     0    rewrite (disk_storage, 'data');
  13    1     1    1    27    writeln('Input a line of text: ');
  14    1     1    1    68    readln (buffer);
  15    1     1    1    87    len := bufferwrite(disk_storage, buffer[1], length(buffer));
  16    1     1    1   109    repeat
  17    1     1    2   109      reset(disk_storage);
  18    1     1    2   116      writeln; writeln;
  19    1     1    2   132      write('Input break char [cntrl Z to stop]: ');
  20    1     1    2   179      readln(break);
  21    1     1    2   197      if not eof(input) then
  22    1     1    3   208        begin
  23    1     1    4   208          new_len := bufferread(disk_storage, buff_array, len, ord(break));
  24    1     1    4   226          writeln('The buffer read: ');
  25    1     1    4   262          writeln(copy(buffer, 1, abs(new_len)));
  26    1     1    4   292          writeln('Length: ', abs(new_len):0);
  27    1     1    4   331          if new_len < 0 then writeln('(Break char not found)');
  28    1     1    3   378        end;
  29    1     1    1   378    until eof(input);
  30    1     1    0   388  end.
```

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE
Intellec® Microcomputer Development System
—Model 800
—Series II Model 220, Model 230, Model 240
64KB of Memory
Dual-Diskette Drives
—Single- or Double-Density*
System Console
—Intel® CRT or non-Intel® CRT

*Recommended.

REQUIRED SOFTWARE
ISIS-II Diskette Operating System
—Single- or Double-Density

OPTIONAL SOFTWARE
ISIS-II CREDIT™ (CRT-Based Text Editor)

Documentation Package

PASCAL 80 User's Guide (9801015-01)

PASCAL User Manual and Report, Second Edition, Kathleen Jensen and Niklaus Wirth

Shipping Media

Flexible Diskettes
—Single- and Double-Density

AFN-01233B

ORDERING INFORMATION

Part Number	Description
MDS-381*	PASCAL 80 Software Package

Requires Software License

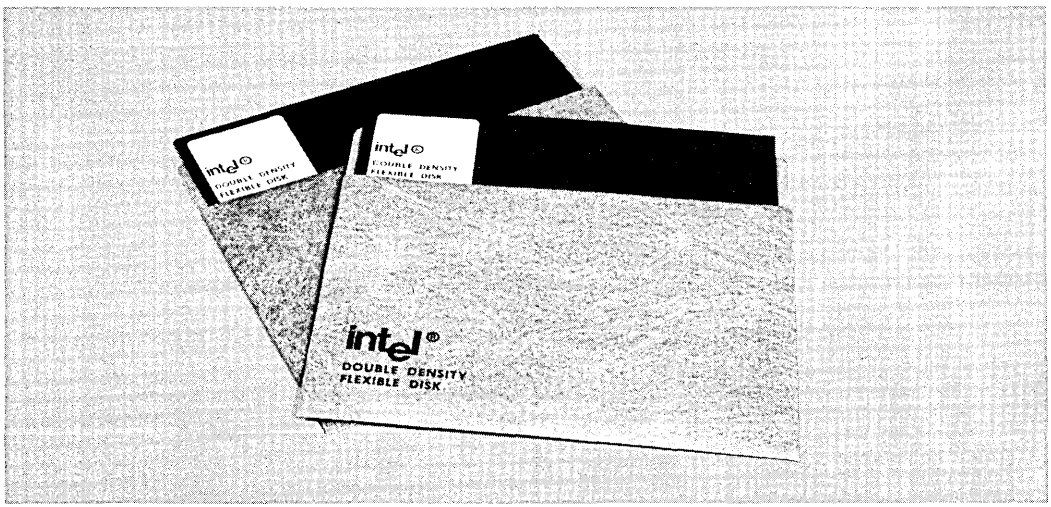
*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

SUPPORT CATEGORY: Level D

PL/M 80
HIGH LEVEL PROGRAMMING LANGUAGE

- **Provides Resident Operation on Intellec® Microcomputer Development System and Intellec® Series II Microcomputer Development Systems**
- **Produces Relocatable and Linkable Object Code**
- **Sophisticated Code Optimization Reduces Application Memory Requirements**
- **Speeds Project Completion with Increased Programmer Productivity**
- **Cuts Software Development and Maintenance Costs**
- **Improves Product Reliability with Simplified Language and Consequent Error Reduction**
- **Eases Enhancement as System Capabilities Expand**

The PL/M 80 High Level Programming Language Intellec Resident Compiler is an advanced, high level programming language for Intel 8080 and 8085 microprocessors, iSBC-80 OEM computer systems, and Intellec microcomputer development systems. PL/M has been substantially enhanced since its introduction in 1973 and has become one of the most effective and powerful microprocessor systems implementation tools available. It is easy to learn, facilitates rapid program development and debugging, and significantly reduces maintenance costs. PL/M is an algorithmic language in which program statements naturally express the algorithm to be programmed, thus freeing programmers to concentrate on system development rather than assembly language details (such as register allocation, meanings of assembler mnemonics, etc.). The PL/M compiler efficiently converts free-form PL/M programs into equivalent 8080/8085 instructions. Substantially fewer PL/M statements are necessary for a given application than would be using assembly language or machine code. Since PL/M programs are problem oriented and thus more compact, programming in PL/M results in a high degree of productivity during development efforts, resulting in significant cost reduction in software development and maintenance for the user.



MAY 1983
ORDER NUMBER:210327-002

FUNCTIONAL DESCRIPTION

The PL/M compiler is an efficient multiphase compiler that accepts source programs, translates them into object code, and produces requested listings. After compilation, the object program may be first linked to other modules, then located to a specific area of memory, and finally executed. The diagram shown in Figure 1 illustrates a program development cycle where the program consists of three modules: PL/M, FORTRAN, and assembly language. A typical PL/M compiler procedure is shown in Table 1.

Features

Major features of the Intel PL/M 80 compiler and programming language include:

Resident Operation — on Intellec microcomputer development systems eliminates the need for a large in-house computer or costly timesharing system.

Object Code Generation — of relocatable and linkable object codes permits PL/M program development and debugging in small modules, which may be easily linked with other modules and/or library routines to form a complete application.

Extensive Code Optimization — including compile time arithmetic, constant subscript resolution, and common subexpression elimination, results in generation of short, efficient CPU instruction sequences.

Symbolic Debugging — fully supported in the PL/M compiler and ICE-85 in-circuit emulators.

Compile Time Options — includes general listing format commands, symbol table listing, cross reference listing, and "innerlist" of generated assembly language instructions.

Block Structure — aids in utilization of structured programming techniques.

Access — provided by high level PL/M statements to hardware resources (interrupt systems, absolute addresses, CPU input/output ports).

Data Definition — enables complex data structures to be defined at a high level.

Re-entrant Procedures — may be specified as a user option.

Benefits

PL/M is designed to be an efficient, cost-effective solution to the special requirements of microcomputer software development as illustrated by the following benefits of PL/M use:

Low Learning Effort — even for the novice programmer, because PL/M is easy to learn.

Earlier Project Completion — on critical projects, because PL/M substantially increases programmmer productivity while reducing program development time.

Lower Development Cost — because increased programmer productivity requiring less programming resources for a given function translates into lower software development costs.

Increased Reliability — because of PL/M's use of simple statements in the program algorithm, which are easier to correct and thus substantially reduce the risk of costly errors in systems that have already reached full production status.

Easier Enhancement and Maintenance — because programs written in PL/M are easier to read and easier to understand than assembly language, and thus are easier to enhance and maintain as system capabilities expand and future products are developed.

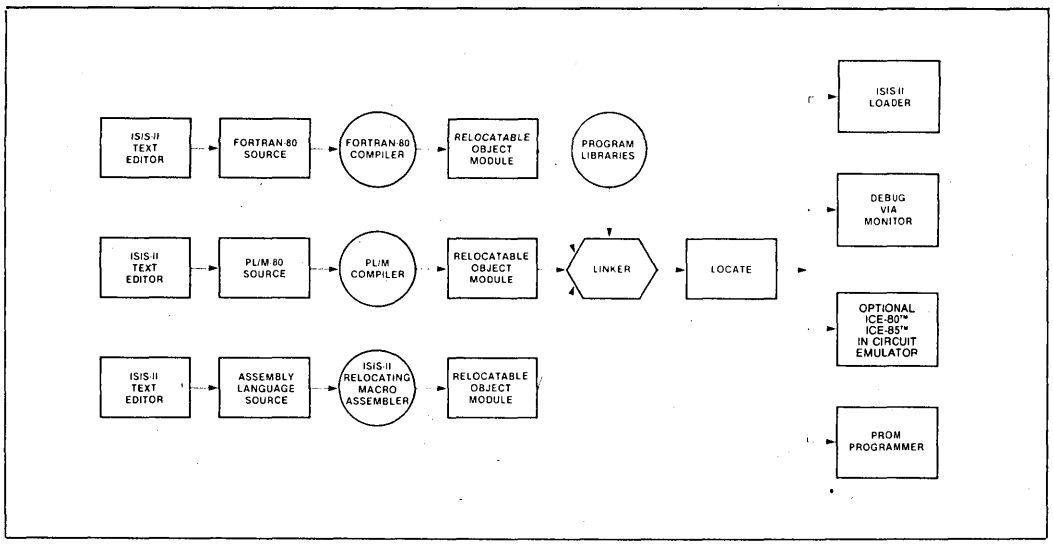


Figure 1. Program Development Cycle Block Diagram

AFN-00818B

Simpler Project Development — because the Intellec microcomputer development system with resident PL/M 80 is all that is needed for developing and debugging software for 8080 and 8085 microcomputers, and the use of expensive (and remote) timesharing or large computers is consequently not required.

Table 1. PL/M-80 Compiler Sample Factorial Generator Procedure

```
                                $OBJECT(:F1:FACT.OB2)
                                $DEBUG
                                $XREF
                                $TITLE('FACTORIAL GENERATOR — PROCEDURE')
                                $PAGEWIDTH(80)

 1                              FACT:
                                DO;

 2    1                         DECLARE NUMCH BYTE PUBLIC;

 3    1                         FACTORIAL: PROCEDURE (NUM,PTR) PUBLIC;
 4    2                             DECLARE NUM BYTE, PTR ADDRESS;
 5    2                             DECLARE DIGITS BASED PTR (161) BYTE;
 6    2                             DECLARE (I,C,M) BYTE;

 7    2                             NUMCH = 1; DIGITS(1) = 1;
 9    2                             DO M = 1 TO NUM;
10    3                                 C = 0;
11    3                                 DO I = 1 TO NUMCH;
12    4                                     DIGITS(I) = DIGITS(I)*M + C;
13    4                                     C = DIGITS(I)/10;
14    4                                     DIGITS(I) = DIGITS(I) — 10*C;
15    4                                 END;

16    3                                 IF C<>0 THEN
17    3                                 DO;
18    4                                     NUMCH = NUMCH + 1; DIGITS(NUMCH) = C;
20    4                                     C = DIGITS(NUMCH)/10;
21    4                                     DIGITS(NUMCH) = DIGITS(NUMCH) — 10*C;
22    4                                 END
                                    END;

24    2                         END FACTORIAL;

25    1                         END;
```

SPECIFICATIONS

OPERATING ENVIRONMENT

Intel Microcomputer Development Systems
(Series II, Series III, Series IV)
Intel Personal Development System

DOCUMENTATION

PL/M 80 Programming Manual
ISIS-II PL/M 80 Compiler Operator's Manual

ORDERING INFORMATION

Product Code	Description
MDS*-PLM	PL/M 80 High Level Language Compiler. Needs Software License.

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

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

8087
SOFTWARE SUPPORT PACKAGE

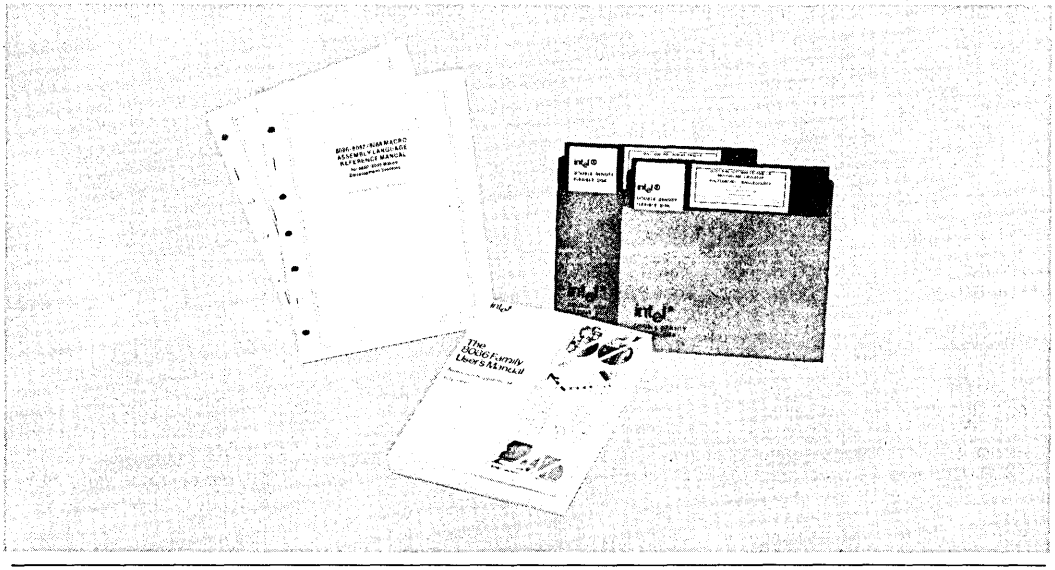
- **Program Generation for the 8087 Numeric Data Processor on 8080/8085 Based Intel Microcomputer Development Systems**
- **Consists of: 8086/8087/8088 Macro Assembler, 8087 Software Emulator**
- **Macro Assembler Generates Code for 8087 Processor or Emulator, While Also Supporting the 8086/8088 Instruction Set**
- **8087 Emulator Duplicates Each 8087 Floating-Point Instruction in Software, for Evaluation of Prototyping, or for Use in an End Product**
- **Macro Assembler and 8087 Emulator are Fully Compatible with Other 8086/8088 Development Software**
- **Implementation of the IEEE Proposed Floating-Point Standard (the Intel® Realmath Standard)**

The 8087 Software Support Package is an optional extention of Intel's 8086/8088 Software Development Package.

The 8087 Software Support Package consists of the 8086/8087/8088 Macro Assembler, and the Full 8087 Emulator. The assembler is a functional superset of the 8086/8088 Macro Assembler, and includes instructions for over sixty new floating-point operations, plus new data types supported by the 8087.

The 8087 Emulator is an 8086/8088 object module that simulates the environment of the 8087, and executes each floating-point operation using software algorithms. This emulator functionally duplicates the operation of the 8087 Numeric Data Processor.

Also included in this package are interface libraries to link with 8086/8087/8088 object modules, which are used for specifying whether the 8087 Processor or the 8087 Emulator is to be used. This enables the run-time environment to be invisible to the programmer at assembly time.



SEPTEMBER 1984
ORDER NUMBER:402150-002

FUNCTIONAL DESCRIPTION

8086/8087/8088 Macro Assembler

The 8086/8087/8088 Macro Assembler translates symbolic macro assembly language instructions into appropriate machine instructions. It is an extended version of the 8086/8088 Macro Assembler, and therefore supports all of the same features and functions, such as limited type checking, conditional assembly, data structures, macros, etc. The extensions are the new instructions and data types to support floating-point operations. Realmath floating-point instructions (see Table 1) generate code capable of being converted to either 8087 instructions or interrupts for the 8087 Emulator. The Processor/Emulator selection is made via interface libraries at LINK-time. In addition to the new floating-point instructions, the macro assembler also introduces two new 8087 data types: QWORD (8 bytes) and TBYTE (ten bytes). These support the highest precision of data processed by the 8087.

Full 8087 Emulator

The Full 8087 Emulator is a 16-kilobyte object module that is linked to the application program for floating-point operations. Its functionality is identical to the 8087 chip, and is ideal for prototyping and debugging floating-point applications. The Emulator is an alternative to the use of the 8087 chip, although the latter executes floating-point applications up to 100 times faster than an 8086 with the 8087 Emulator. Furthermore, since the 8087 is a "co-processor," use of the chip will allow many operations to be performed in parallel with the 8086.

Table 1. 8087 Instructions

Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Table 1. 8087 Instructions (cont'd)

Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^x-1
FYL2X	$Y \bullet \log_2 X$
FYL2XP1	$Y \bullet \log_2(X+1)$

Constant Instructions

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

Intel Microcomputer Development Systems
—Series II
—Personal Development System
—Series IV

REQUIRED SOFTWARE

8086/8088 Software Development Package

Documentation Package

8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems

8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems

The 8086 Family Users Manual Supplement for the 8087 Numeric Data Processor

ORDERING INFORMATION

Part Number	Description
MDS*-387	8087 Software Support Package

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

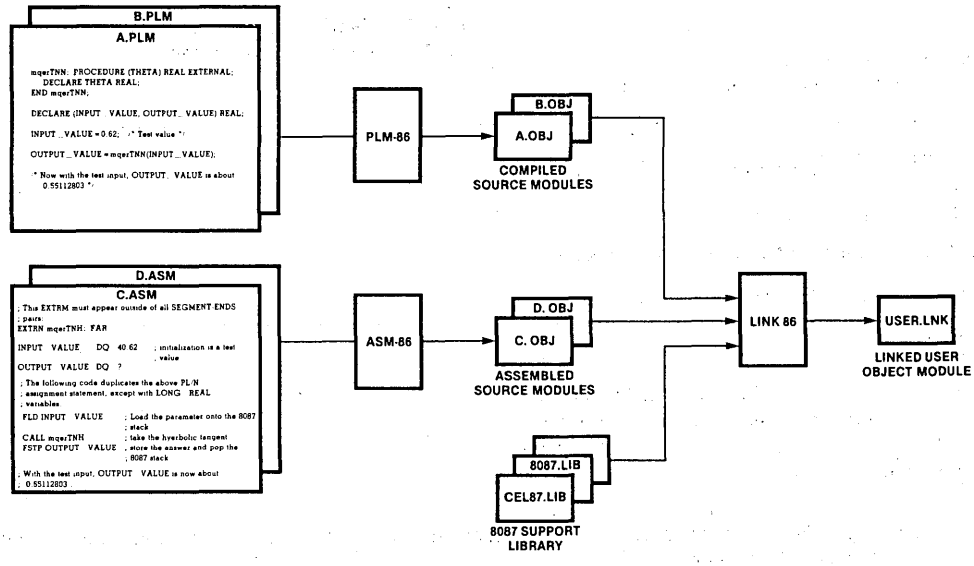
*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

8087 SUPPORT LIBRARY

- **Library to support floating point arithmetic in PL/M-86 and ASM-86**
- **Common elementary function library provides trigonometric, logarithmic and other useful functions**
- **Decimal conversion module supports binary-decimal conversions**
- **Error-handler module simplifies floating point error recovery**
- **Full 8087 Software Emulator for software debugging without the 8087 component**
- **Accurate, verified and efficient implementation of algorithms for functions**
- **Supports proposed IEEE Floating Point Standard for high accuracy and software portability**

The 8087 Support Library provides PL/M-86 and ASM-86 users with the equivalent numeric data processing capability of Fortran-86. With the Library, it is easy for PL/M-86 and ASM-86 programs to do floating point arithmetic. Programs can link in modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. The 8087 Support Library implements Intel's REALMATH standard and also supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the PL/M-86 user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable—his software investment is maintained.

The 8087 Support Library consists of the common elementary function library, the decimal conversion module, the error handler module, the full 8087 Software emulator and interface libraries to the 8087 and to the 8087 emulator.



MAY 1983
ORDER NUMBER:121653-001

CEL87.LIB
THE COMMON ELEMENTARY FUNCTION LIBRARY

CEL87.LIB contains commonly used floating point functions. It is used along with the 8087 numeric coprocessor or the 8087 emulator and it provides a complete package of elementary functions, giving valid results for all appropriate inputs. This library provides PL/M-86 and ASM-86 users all the math functions supported intrinsically by the Fortran-86. Following is a summary of CEL87 functions, grouped by functionality.

Rounding and Truncation Functions:

mqerIEX, mqerIE2, and mqerIE4 round a real number to the nearest integer; to the even integer if there is a tie. The answer returned is real, a 16-bit integer or a 32-bit integer respectively.
mqerIAX, mqerIA2, mqerIA4 round a real number to the nearest integer, to the integer away from zero if there is a tie; the answer returned is real, a 16-bit integer or a 32-bit integer, respectively.
mqerICX, mqerIC2, mqerIC4 truncate the fractional part of a real input; the answer is real, a 16-bit integer or a 32-bit integer, respectively.

Logarithmic and Exponential Functions:

mqerLGD computes decimal (base 10) logarithms.
mqerLGE computes natural (base e) logarithms.
mqerEXP computes exponentials to the base e.
mqerY2X computes exponentials to any base.
mqerYI2 raises an input real to a 16-bit integer power.
mqerYI4 is as mqerYI2, except to a 32-bit integer power.
mqerYIS is as mqerYI2, but it accommodates PL/M-86 users.

Trigonometric and Hyperbolic Functions:

mqerSIN, mqerCOS, mqerTAN compute sine, cosine, and tangent.
mqerASN, mqerACS, mqerATN compute the corresponding inverse functions.
mqerSNH, mqerCSH, mqerTNH compute the corresponding hyperbolic functions.
mqerAT2 is a special version of the arc tangent function that accepts rectangular coordinate inputs.

Other Functions:

mqerDIM is FORTRAN's positive difference function.
mqerMAX returns the maximum of two real inputs.
mqerMIN returns the minimum of two real inputs.
mqerSGH combines the sign of one input with the magnitude of the other input.
mqerMOD computes a modulus, retaining the sign of the dividend.
mqerRMD computes a modulus, giving the value closest to zero.

DCON87.LIB
THE DECIMAL CONVERSION LIBRARY

DCON87.LIB is a library of procedures which convert binary representations of floating point numbers and ASCII-encoded string of digits.

The binary-to-decimal procedure mqcBIN DECLOW accepts a binary number in any of the formats used for the representation of floating point numbers in the 8087. Because there are so many output formats for floating point numbers, mqcBIN_DECLOW does not attempt to provide a finished, formatted text string. Instead, it provides the "building blocks" for you to use to construct the output string which meets your exact format specification.

The decimal-to-binary procedure mqcDEC__BIN accepts a text string which consists of a decimal number with optional sign, decimal point, and/or power-of-ten exponent. It translates the string into the caller's choice of binary formats.

Decimal-to-binary procedure mqcDECLOW__BIN is provided for callers who have already broken the decimal number into its constituent parts.

The procedures mqcLONG__TEMP, mqcSHORT__TEMP, mqcTEMP__LONG, and mqcTEMP__SHORT convert floating point numbers between the longest binary format, TEMP__REAL, and the shorter formats.

EH87.LIB
THE ERROR HANDLER MODULE

EH87.LIB is a library of five utility procedures which a user can utilize for writing trap handlers. Trap handlers are called when an unmasked 8087 error occurs.

The 8087 error reporting mechanism can be used not only to report error conditions, but also to let software implement IEEE standard options not directly supported by the chip. The three such extensions to the 8087 are: normalizing mode, non-trapping not-a-number (NaN), and non-ordered comparison. The utility procedures support these extra features.

DECODE is called near the beginning of the trap handler. It preserves the complete state of the 8087, and also identifies what function called the trap handler, and returns available arguments and/or results. DECODE eliminates much of the effort needed to determine what error caused the trap handler to be called.

NORMAL provides the "normalizing mode" capability for handling the "D" exception. By calling NORMAL in your trap handler, you eliminate the need to write code in your application program which tests for non-normal inputs.

SIEVE provides two capabilities for handling the "I" exception. It implements non-trapping NaN's and non-ordered comparisons. These two IEEE standard features are useful for diagnostic work.

ENCODE is called near the end of the trap handler. It restores the state of the 8087 saved by DECODE, and performs a choice of concluding actions, by either retrying the offending function or returning a specified result.

FILTER calls each of the above four procedures. If your error handler does nothing more than detect fatal errors and implement the features supported by SIEVE and NORMAL, then your interface to EH87.LIB can be accomplished with a single call to FILTER.

E8087
THE FULL 8087 EMULATOR

E8087 is an object module that functionally emulates the 8087 coprocessor chip. It is ideal for use during prototyping and debugging floating point programs. However, the target system should use the 8087 component because it executes 1000 times faster and uses significantly less memory.

E8087.LIB, 8087.LIB, 87NULL.LIB INTERFACE LIBRARIES

E8087. LIB, 8087.LIB and 87NULL. LIB libraries configure a user's application program for his run-time environment: running with the emulator, with the 8087 component or without floating point arithmetic, respectively.

SPECIFICATIONS

TARGET ENVIRONMENT

8086/8088 Based Microcomputer System

DEVELOPMENT ENVIRONMENT

Required Hardware

All Intel Microcomputer Development Systems (Series II, Series III/Series IV)

Required Software

For Series II:
8086/8088 Software Development Package

Documentation Package

Numeric Support Library Manual

*Recommended

ORDERING INFORMATION

Part Number	Description
MDS*-319	8087 Support Library

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*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.

AFN-02063B

80287 SUPPORT LIBRARY

- **Library to support floating point arithmetic in Pascal-286, PL/M-286 and ASM-286**
- **Common elementary function library provides trigonometric, logarithmic and other useful functions**
- **Decimal conversion module supports binary-decimal conversions**
- **Error-handler module simplifies floating point error recovery**
- **Supports proposed IEEE Floating Point Standard for high accuracy and software portability**

The 80287 Support Library provides Pascal-286, PL/M-286 and ASM-286 users with numeric data processing capability. With the Library, it is easy for programs to do floating point arithmetic. Programs can bind in library modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. Figure 1 below illustrates how the 80287 Support Library can be bound with PL/M-286 and ASM-286 user code to do this. The 80287 Support Library supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable--the software investment is maintained.

The 80287 Support Library consists of the common elementary function library (CEL287.LIB), the decimal conversion library (DC287.LIB), the error handler module (EH287.LIB) and interface libraries (80287.LIB, NUL287.LIB).

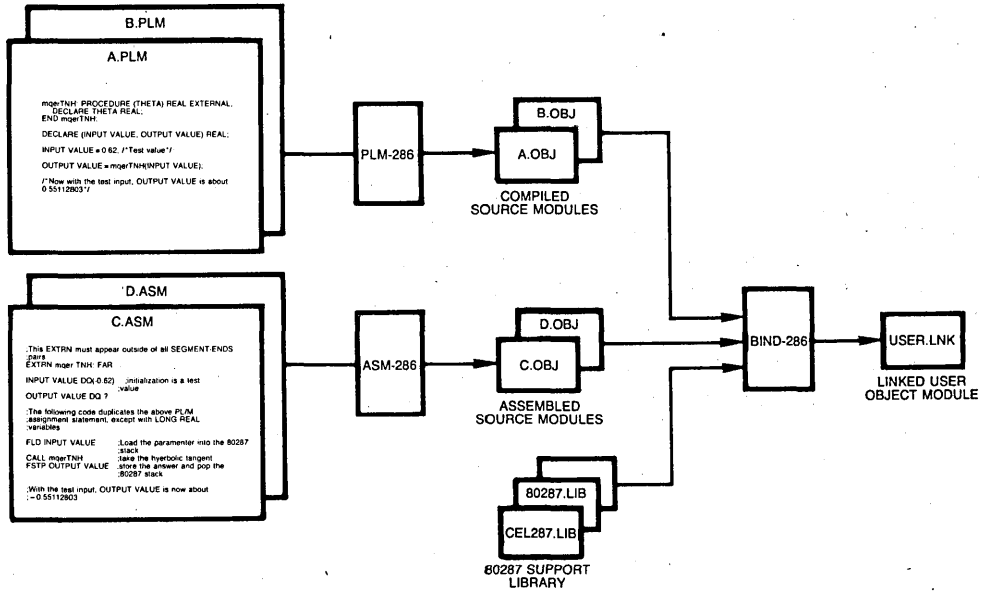


Figure 1. Use of 80287 Support Library with PL/M-286 and ASM-286.

MARCH 1984
ORDER NUMBER: 231041-001

CEL287.LIB
THE COMMON ELEMENTARY FUNCTION LIBRARY

FUNCTIONS

CEL287.LIB contains commonly used floating point functions. It is used along with the 80287 numeric coprocessor. It provides a complete package of elementary functions, giving valid results for all appropriate inputs. Following is a summary of CEL287 functions, grouped by functionality.

Rounding and Truncation Functions:

mqerIEX, mqerIE2, and mqerIE4. Round a real number to the nearest integer; to the even integer if there is a tie. The answer returned is real, a 16-bit integer or a 32-bit integer respectively.

mqerIAX, mqerIA2, mqerIA4. Round a real number to the nearest integer, to the integer away from zero if there is a tie; the answer returned is real, a 16-bit integer or a 32-bit integer, respectively.

mqerICX, mqerIC2, mqerIC4. Truncate the fractional part of a real input; the answer is real, a 16-bit integer or 32-bit integer, respectively.

Logarithmic and Exponential Functions:

mqerLGD computes decimal (base 10) logarithms.
mqerLGE computes natural (base e) logarithms.
mqerEXP computes exponentials to the base e.
mqerY2X computes exponentials to any base.
mqerY12 raises an input real to a 16-bit integer power.
mqerY14 is as mqerY12, except to a 32-bit integer power.
mqerYIS is as mqerY12, but it accommodates PL/M-286 users.

Trigonometric and Hyperbolic Functions:

mqerSIN, mqerCOS, mqerTAN compute sine, cosine, and tangent.

mqerASN, mqerACS, mqerATN compute the corresponding inverse functions.

mqerSNH, mqerCSH, mqerTNH compute the corresponding hyperbolic functions.

mqerAT2 is a special version of the arc tangent function that accepts rectangular coordinate inputs.

Other Functions:

mqerDIM is FORTRAN's positive difference function.
mqerMAX returns the maximum of two real inputs.
mqerMIN returns the minimum of two real inputs.
mqerSGH combines the sign of one input with the magnitude of the other input.
mqerMOD computes a modulus, retaining the sign of the dividend.
mqerRMD computes a modulus, giving the value closest to zero.

DC287.LIB
THE DECIMAL CONVERSION LIBRARY

DC287.LIB is a library of procedures which convert binary representations of floating point numbers and ASCII-encoded string of digits.

The binary-to-decimal procedure mqcBIN_DECLOW accepts a binary number in any of the formats used for the representation of floating point numbers in the 80287. Because there are so many output formats for floating point numbers, mqcBIN_DECLOW does not attempt to provide a finished, formatted text string. Instead, it provides the "building blocks" for you to use to construct the output string which meets your exact format specification.

The decimal-to-binary procedure mqcDEC_BIN accepts a text string which consists of a decimal number with optional sign, decimal point, and/or power-of-ten exponent. It translates the string into the caller's choice of binary formats.

Decimal-to-binary procedure mqcDECLOW_BIN is provided for callers who have already broken the decimal number into its constituent parts.

The procedures mqcLONG_TEMP, mqcSHORT_TEMP, mqcTEMP_LONG, and mqcTEMP_SHORT convert floating point numbers between the longest binary format, TEMP_REAL, and the shorter formats.

231041-001

EH287.LIB
THE ERROR HANDLER MODULE

EH287.LIB is a library of five utility procedures for writing trap handlers. Trap handlers are called when an unmasked 80287 error occurs.

The 80287 error reporting mechanism can be used not only to report error conditions, but also to let software implement IEEE standard options not directly supported by the chip. The three such extensions to the 80287 are: normalizing mode, non-trapping not-a-number (NaN), and non-ordered comparison. The utility procedures support these extra features.

DECODE is called near the beginning of the trap handler. It preserves the complete state of the 80287, and also identifies what function called the trap handler, and returns available arguments and/or results. DECODE eliminates much of the effort needed to determine what error caused the trap handler to be called.

NORMAL provides the "normalizing mode" capability for handling the "D" exception. By calling NORMAL in your trap handler you eliminate the need to write code in your application program which tests for non-normal inputs.

SIEVE provides two capabilities for handling the "I" exception. It implements non-trapping NaN's and non-ordered comparisons. These two IEEE standard features are useful for diagnostic work.

ENCODE is called near the end of the trap handler. It restores the state of the 80287 saved by DECODE, and performs a choice of concluding actions, by either retrying the offending function or returning a specified result.

FILTER calls each of the above four procedures. If your error handler does nothing more than detect fatal errors and implement the features supported by SIEVE and NORMAL, then your interface to EH287.LIB can be accomplished with a single call to FILTER.

80287.LIB, NUL287.LIB
INTERFACE LIBRARIES

80287.LIB and NUL287.LIB libraries configure a user's application program for his run-time environment; running with the 80287 component or without floating point arithmetic, respectively.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Series III, Series IV)

Documentation Package

80287 Support Library Reference Manual

Related Software

A 80287 software emulator is available as part of the 8086 software toolbox (iMDX364)

ORDERING INFORMATION

Part Number	Description
iMDX329	80287 Support Library

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

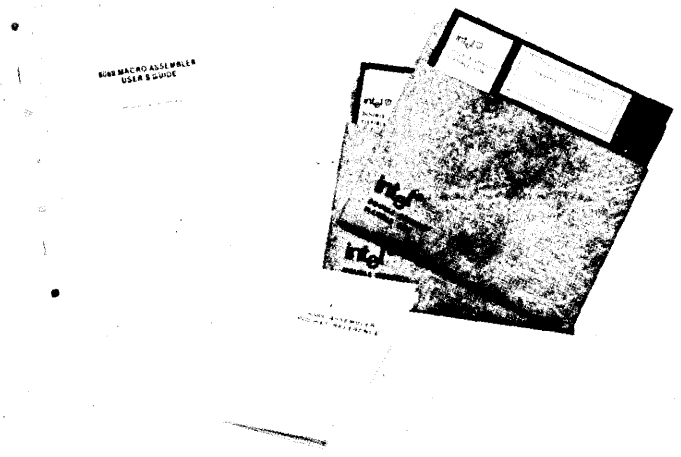
231041-001

8089 IOP
SOFTWARE SUPPORT PACKAGE
#407200

- **Program Generation for the 8089 I/O Processor on the Intellec® Microcomputer Development System**
- **Contains 8089 Macro Assembler, plus Relocation and Linkage Utilities**
- **Relocatable Object Module Compatible with All iAPX 86 and iAPX 88 Object Modules**
- **Fully Supports Symbolic Debugging with the RBF-89 Software Debugger**
- **Supports 8089-Based Addressing Modes with a Structure Facility that Enables Easy Access to Based Data**
- **Powerful Macro Capabilities**
- **Provides Timing Information in Assembly Listing**
- **Fully Detailed Set of Error Messages**

The IOP Software Support Package extends Intellec Microcomputer Development System support to the 8089 I/O Processor. The macro assembler translates symbolic 8089 macro assembly language instructions into relocatable machine code. The relocation and linkage utilities provide compatibility with iAPX 86, iAPX 88, and 8089 modules, and make structured, modular programming easier.

The macro assembler also provides symbolic debugging capability when used with the RBF-89 software debugger. 8089 program modularity is supported with inter-segment jumps and calls. The macro assembler also provides instruction cycle counts in the listing file, for giving the programmer execution timing information. The programs in the 8089 Software Support Package run on any Intellec Series II or Model 800 with 64K bytes of memory.



MAY 1983

ORDER NUMBER:210853-002

Table 1. Sample Program Listing

```
8089 MACRO ASSEMBLER


ISIS-II 8089 MACRO ASSEMBLER X105 ASSEMBLY OF MODULE TASK
OBJECT MODULE PLACED IN :F1:TASK.OBJ
ASSEMBLER INVOKED BY:  asm89 :f1:task a89 gen macro debug pagewidth(132) print(:f1:taskx.lst)


LOC   OBJECT CODE              TIMING    INC MNC  LINE SOURCE

                                                    1  ;*************************************************
                                                    2  ;*                                               *
                                                    3  ;*                8089 TASK PROGRAM              *
                                                    4  ;*                                               *
                                                    5  ;*************************************************
                                                    6
                                                    7  name TASK
0000                                                8  TASK segment
                                                    9  ;
                                                   10  ;       In the first part of this sample program data is moved from
                                                   11  ;       8086 system RAM to memory local to the 8089 IOP.  In the second
                                                   12  ;       part, the data is moved from the local memory to a data port
                                                   13  ;       also in the 8089 I/O space.
                                                   14  ;
  C000                                             15  data@port@8251  equ     0c000h          ;8251 DP on 8089 local bus
  C001                                             16  command@port@8251 equ   0c001h          ;8251 CP on 8089 local bus
  0200                                             17  buffer@8089     equ     0200h           ;RAM buffer in 8089 I/O space
                                                   18  ;
                                                   19  extrn   buffer@8086                     ;RAM buffer in 8086 system memory
                                                   20  extrn   y                               ;location of the buffer count
                                                   21  ;
                                                   22  %*define (macro_1)
                                                   23  (       movi    gb,buffer@8089          ; Move buffer address into GB
                                                   24          lpdi    gc,y                    ; Load pointer to count into GC
                                                   25          movb    bc,[gc]                 ; Move byte count into BC
                                                   26  )
                                                   27  %*define (macro_2(param_1, param_2)) local loop
                                                   28  (       inc     %param_1                ; Increment pointer into source
                                                   29          dec     %param_2                ; Decrement byte count
                                                   30          jnz     %param_2,%loop          ; Loop back if byte count > 0
                                                   31  )
0000   1100 00000000            38    46          32  ONE:    lpdi    ga,buffer@8086          ; Load register GA with address
                                                   33                                          ; of 8086 buffer
                                                   34  %macro_1
0006   3130 0002                47    71    +1    35          movi    gb,buffer@8089          ; Move buffer address into GB
0008   5180 00000000            77   117    +1    36          lpdi    gc,y                    ; Load pointer to count into GC
0010   6002                     92   139    +1    37          movb    bc,[gc]                 ; Move byte count into BC
                                          +1    38
0012   0050 00CD               124   185          39  loop@8: movb    [gb],[ga]               ; Move byte from 8086 to 8089 buffer
0016   0030                     134   202          40          inc     ga                      ; Increment pointer into 8086 buffer
                                                   41  %macro_2(gb,gc)
0018   2030                     144   219    +1    42          inc     %PARAM_1
                                          +2    43      gb              ; Increment pointer into source
001A   403C                     154   236    +1    44          dec     %PARAM_2
                                          +2    45      gc              ; Decrement byte count
001C   4040 F3                  173   259    +1    46          jnz     %PARAM_2
                                          +2    47      gc,%LOOP
                                          +2    48          LOOP@8          ; Loop back if byte count > 0
                                          +1    49
001F   1130 00C8                191*  294          50  TWO:    movi    ga,data@port@8251       ; load GA with address of 8251 DP
0023   1130 01C8                209   309          51          movi    gc,command@port@8251    ; load GC with address of 8251 CP
                                                   52  %macro_1
0027   3130 0002                227   334    +1    53          movi    gb,buffer@8089          ; Move buffer address into GB
0028   5180 00000000            242   388    +1    54          lpdi    gc,y                    ; Load pointer to count into GC
0031   6002                     281   402    +1    55          movb    bc,[gc]                 ; Move byte count into BC
                                          +1    56
0033   000A FD                  306*  434         57  loop@1: jnbt    [gc],0,loop@1           ; loop until 8251 transmit ready
0036   0091 00CC                338   488         58          movb    [ga],[gb]               ; move message into buffer
                                                   59  %macro_2(gb,gc)
003A   2030                     348   497    +1    60          inc     %PARAM_1
                                          +2    61      gb              ; Increment pointer into source
003C   403C                     358   514    +1    62          dec     %PARAM_2
                                          +2    63      gc              ; Decrement byte count
003E   4040 F2                  377   537    +1    64          jnz     %PARAM_2
                                          +2    65      gc,%LOOP
                                          +2    66          LOOP@1          ; Loop back if byte count > 0
                                          +1    67
0041   2040                     399   562         68          hlt
0043                                              69  TASK    ends
                                                  70  END


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

FUNCTIONAL DESCRIPTION

The IOP Software Support Package contains:

ASM89 —The 8089 Macro Assembler.

LINK86 —Resolves control transfer references between 8089 object modules, and data references in 8086, 8088, and 8089 modules.

LOC86 —Assigns absolute memory addresses to 8089 object modules.

OH86 —Converts absolute object modules to hexadecimal format.

UPM —The Universal PROM Mapper, which supports PROM programming in all iAPX 86/11 and iAPX 88/11 applications.

ASM89 translates symbolic 8089 macro assembly language instructions into the appropriate machine codes. The ability to refer to both program and data addresses with symbolic names makes it easier to develop and modify programs, and avoids the errors of hand translation.

The powerful macro facility allows frequently used code sequences to be referred to by a single name, so that any changes to that sequence need to be made in only one place in the program. Common code sequences that differ only slightly can also be referred to with a macro call, and the differences can be substituted with macro parameters.

ASM89 provides symbolic debugging information in the object file. The RBF-89 debugger makes use of this information, so the programmer can symbolically debug 8089 programs. ASM89 also provides cycle counts for each instruction in the assembly listing file (see Table 1). These cycle counts help the programmer determine how long a particular routine or code sequence will take to execute on the 8089.

ASM89 provides relocatable object module compatibility with the 8086 and 8088 microprocessors. This object module compatibility, along with the 8086/8088 relocation and linkage utilities, facilitates the designing of iAPX 86/11 and iAPX 88/11 systems.

ASM89 fully supports the based addressing modes of the 8089. A structure facility allows the user to define a template that enables accessing of based data symbolically.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Model 800, Series II, Series III, Series IV)

Support

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

Documentation Package

8089 Macro Assembler User's Guide (9800938)

8089 Macro Assembler Pocket Reference (9800936)

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users (9800639)

Universal PROM Programmer User's Manual (9800819)

Shipping Media

—Single and Double Density Diskettes

ORDERING INFORMATION

Part Number	Description
MDS*-312	8089 IOP Software Support Package

Requires Software License

***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.

8051
SOFTWARE PACKAGES

PL/M51 Software Package Contains the following:

- **PL/M51 Compiler which is designed to support all phases of software implementation**
- **RL51 Linker and Relocator which enables programmers to develop software in a modular fashion**
- **LIB51 Librarian which lets programmers create and maintain libraries of software object modules**

8051 Software Development Package Contains the following:

- **8051 Macro Assembler which gives symbolic access to 8051 hardware features**
- **RL51 Linker and Relocator program which links modules generated by the assembler**
- **CONV51 which enables software written for the MCS® -48 family to be up graded to run on the 8051**
- **LIB51 Librarian which lets programmers create and maintain libraries of software object modules**

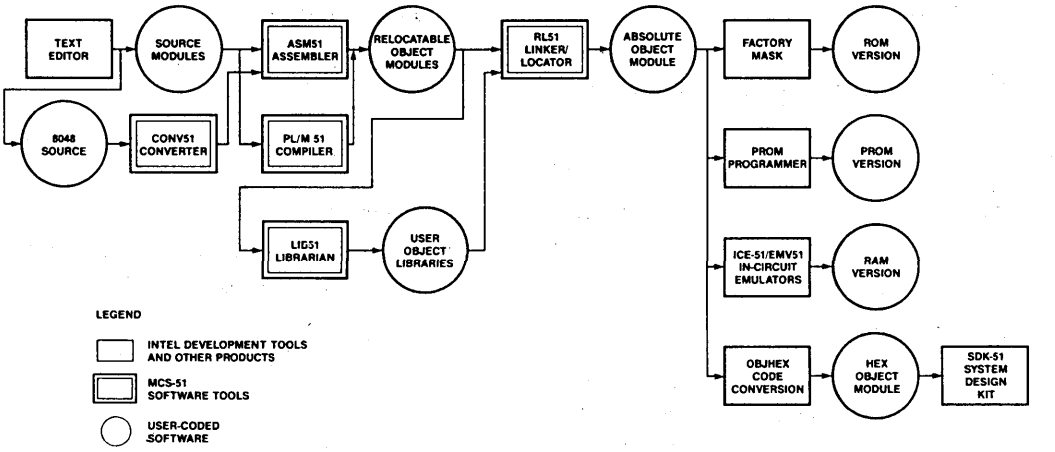


Figure 1. MCS®-51 Program Development Process

MARCH 1984
ORDER NUMBER: 162771-002

PL/M 51 SOFTWARE PACKAGE

- **High-level programming language for the Intel MCS®-51 single-chip microcomputer family**
- **Compatible with PL/M 80 assuring MCS®-80/85 design portability**
- **Enhanced to support boolean processing**
- **Tailored to provide an optimum balance among on-chip RAM usage, code size and code execution time**
- **Allows programmer to have complete control of microcomputer resources**
- **Produces relocatable object code which is linkable to object modules generated by all other 8051 translators**
- **Extends high-level language programming advantages to microcontroller software development**
- **Improved reliability, lower maintenance costs, increased programmer productivity and software portability**
- **Includes the linking and relocating utility and the library manager**
- **Supports all members of the Intel MCS®-51 architecture**

PL/M 51 is a structured, high-level programming language for the Intel MCS-51 family of microcomputers. The PL/M 51 language and compiler have been designed to support the unique software development requirements of the single-chip microcomputer environment. The PL/M language has been enhanced to support Boolean processing and efficient access to the microcomputer functions. New compiler controls allow the programmer complete control over what microcomputer resources are used by PL/M programs.

PL/M 51 is largely compatible with PL/M 80 and PL/M 86. A significant proportion of existing PL/M software can be ported to the MCS-51 with modifications to support the MCS-51 architecture. Existing PL/M programmers can start programming for the MCS-51 with a small relearning effort.

PL/M 51 is the high-level alternative to assembly language programming for the MCS-51. When code size and code execution speed are not critical factors, PL/M 51 is the cost-effective approach to developing reliable, maintainable software.

The PL/M 51 compiler has been designed to support efficiently all phases of software implementation with features like a syntax checker, multiple levels of optimization, cross-reference generation and debug record generation.

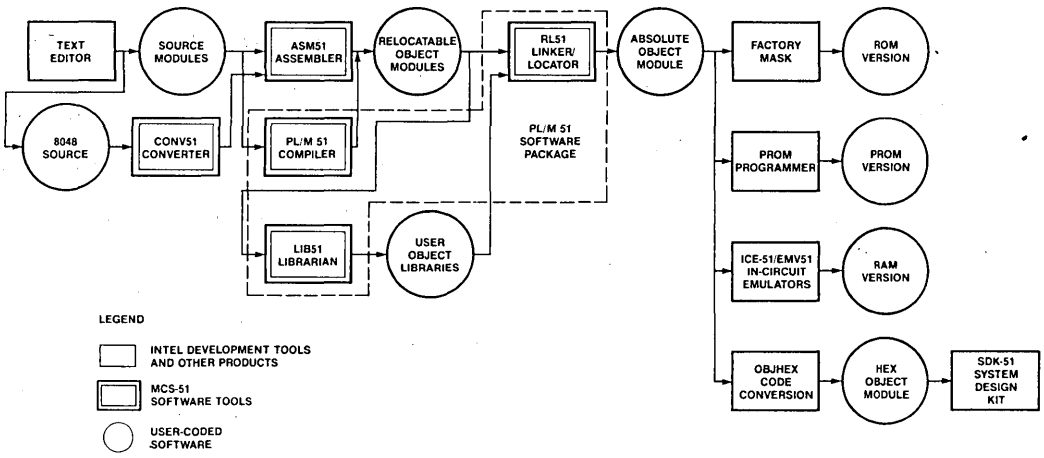


Figure 2. PL/M51 Software Package

PL/M 51 Compiler

FEATURES

Major features of the Intel PL/M 51 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure, for example).

Language Compatiblity

PL/M 51 object modules are compatible with object modules generated by all other MCS-51 translators. This means that PL/M programs may be linked to programs written in any other MCS-51 language.

Object modules are compatible with In-Circuit Emulators and Emulation Vehicles for MCS-51 processors; the DEBUG compiler control provides these tools with symbolic debugging capabilities.

Supports Three Data Types

PL/M makes use of three data types for various applications. These data types range from one to sixteen bits and facilitate various arithmetic, logic, and address functions:

—Bit: a binary digit
—Byte: 8-bit unsigned number or,
—Word: 16-bit unsigned number.

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

PL/M 51 supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

—Array: Indexed list of same type data elements
—Structure: Named collection of same or different type data elements
—Combinations of Both: Arrays of structures or structures of arrays.

Interrupt Handling

A procedure may be defined with the INTERRUPT attribute. The compiler will generate code to save and restore the processor status, for execution of the user-defined interrupt handler routines.

Compiler Controls

The PL/M 51 compiler offers controls that facilitate such features as:

—Including additional PL/M 51 source files from disk
—Cross-reference
—Corresponding assembly language code in the listing file

Program Addressing Control

The PL/M 51 compiler takes full advantage of program addressing with the ROM (SMALL/MEDIUM/LARGE) control. Programs with less than 2 KB code space can use the SMALL or MEDIUM option to generate optimum addressing instructions. Larger programs can address over the full 64 KB range.

Code Optimization

The PL/M 51 compiler offers four levels of optimization for significantly reducing overall program size.

—Combination or "folding" of constant expressions; "Strength reductions" (a shift left rather than multiply by 2)
—Machine code optimizations; elimination of superfluous branches
—Automatic overlaying of on-chip RAM variables
—Register history: an off-chip variable will not be reloaded if its value is available in a register.

Error Checking

The PL/M 51 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation error messages is provided by the compiler and user's guide.

BENEFITS

PL/M 51 is designed to be an efficient, cost-effective solution to the special requirements of MCS-51 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 51 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 51, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliabilty

PL/M 51 is designed to aid in the development of reliable software (PL/M programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

RL51 Linker and Relocator

- **Links modules generated by the assembler and the PL/M compiler**
- **Locates the linked object to absolute memory locations**
- **Enables modular programming of software-efficient program development**
- **Modular programs are easy to understand, maintainable and reliable**

The MCS-51 linker and relocator (RL51) is a utility which enables MCS-51 programmers to develop software in a modular fashion. The utility resolves all references between modules and assigns absolute memory locations to all the relocatable segments, combining relocatable partial segments with the same name.

With this utility, software can be developed more quickly because small functional modules are easier to understand, design and test than large programs.

The total number of allowed symbols in user-developed software is very large because the assembler number of symbols' limit applies only per module, not to the entire program. Therefore programs can be more readable and better documented.

Modules can be saved and used on different programs. Therefore the software investment of the customer is maintained.

RL51 produces two files. The absolute object module file can be directly executed by the MCS-51 family. The listing file shows the results of the link/locate process.

LIB51 Librarian

The LIB51 utility enables MCS-51 programmers to create and maintain libraries of software object modules. With this utility, the customer can develop standard software modules and place them in libraries, which programs can access through a standard interface. When using object libraries, the linker will call only object modules that are required to satisfy external references.

Consequently, the librarian enables the customer to port and reuse software on different projects —thereby maintaining the customer's software investment.

SPECIFICATIONS

Operating Environment

All Intel Microcomputer Development Systems or Intel Personal Development Systems

Documentation Package

PL/M 51 User's Guide
MCS-51 Utilities User's Guide

ORDERING INFORMATION

Part Number	Description
iMDX 352 Requires Software License	PL/M 51 Software Package

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and monthly Technical Newsletters are available.

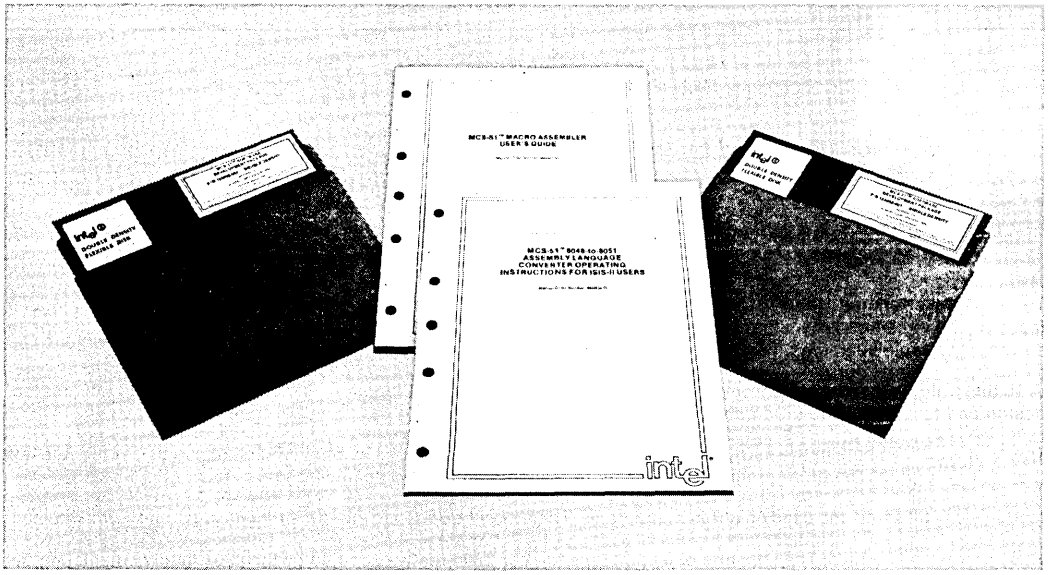
8051 SOFTWARE DEVELOPMENT PACKAGE

- **Symbolic relocatable assembly language programming for 8051 microcontrollers**
- **Extends Intellec® Microcomputer Development System to support 8051 program development**
- **Produces Relocatable Object Code which is linkable to other 8051 Object Modules**
- **Encourage modular program design for maintainability and reliability**
- **Macro Assembler features conditional assembly and macro capabilities**
- **CONV51 Converter for translation of 8048 assembly language source code to 8051 assembly language source code**
- **Provides upward compatibility from the MCS-48™ family of single-chip microcontrollers**

The 8051 software development package provides development system support for the powerful 8051 family of single chip microcomputers. The package contains a symbolic macro assembler and MCS-48 source code converter.

The assembler produces relocatable object modules from 8051 macro assembly language instructions. The object code modules can be linked and located to absolute memory locations. This absolute object code may be used to program the 8751 EPROM version of the chip. The assembler output may also be debugged using the ICE-51™ in-circuit emulator.

The converter translates 8048 assembly language instructions into 8051 source instructions to provide software compatibility between the two families of microcontrollers.



8051 MACRO ASSEMBLER

- **Supports 8051 family program development on Intellec® Microcomputer Development Systems**
- **Gives symbolic access to powerful 8051 hardware features**
- **Produces object file, listing file and error diagnostics**
- **Object files are linkable and locatable**
- **Provides software support for many addressing and data allocation capabilities**
- **Symbolic Assembler supports symbol table, cross-reference, macro capabilities, and conditional assembly**

The 8051 Macro Assembler (ASM51) translates symbolic 8051 macro assembly language modules into linkable and locatable object code modules. Assembly language mnemonics are easier to program and are more readable than binary or hexadecimal machine instructions. By allowing the programmer to give symbolic names to memory locations rather than absolute addresses, software design and debug are performed more quickly and reliably. Furthermore, since modules are linkable and relocatable, the programmer can do his software in modular fashion. This makes programs easy to understand, maintainable and reliable.

The assembler supports macro definitions and calls. This is a convenient way to program a frequently used code sequence only once. The assembler also provides conditional assembly capabilities.

Cross referencing is provided in the symbol table listing, showing the user the lines in which each symbol was defined and referenced.

ASM51 provides symbolic access to the many useful addressing features of the 8051 architecture. These features include referencing for bit and byte locations, and for providing 4-bit operations for BCD arithmetic. The assembler also provides symbolic access to hardware registers, I/O ports, control bits, and RAM addresses. ASM51 can support all members of the 8051 family.

Math routines are enhanced by the MULtiply and DIVide instructions.

If an 8051 program contains errors, the assembler provides a comprehensive set of error diagnostics, which are included in the assembly listing or on another file. Program testing may be performed by using the iUP Universal Programmer and iUP F87/51 personality module to program the 8751 EPROM version of the chip.

ICE51 and EMV51 are available for program debugging.

RL51 LINKER AND RELOCATOR PROGRAM

- **Links modules generated by the assembler**
- **Locates the linked object to absolute memory locations**
- **Enables modular programming of software for efficient program development**
- **Modular programs are easy to understand, maintainable and reliable**

The 8051 linker and relocator (RL51) is a utility which enables 8051 programmers to develop software in a modular fashion. The linker resolves all references between modules and the relocator assigns absolute memory locations to all the relocatable segments, combining relocatable partial segments with the same name.

With this utility, software can be developed more quickly because small functional modules are easier to understand, design and test than large programs.

The number of symbols in the software is very large because the assembler symbol limit applies only per module not the entire program. Therefore programs can be more readable and better documented.

Modules can be saved and used on different programs. Therefore the software investment of the customer is maintained.

RL51 produces two files. The absolute object module file can be directly executed by the 8051 family. The listing file shows the results of the link/locate process.

CONV51
8048 TO 8051 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

- **Enables software written for the MCS-48™ family to be upgraded to run on the 8051**
- **Maps each 8048 instruction to a corresponding 8051 instruction**
- **Preserves comments; translates 8048 macro definitions and calls**
- **Provides diagnostic information and warning messages embedded in the output listing**

The 8048 to 8051 Assembly Language Converter is a utility to help users of the MCS-48 family of microcomputers upgrade their deisgns with the high performance 8051 architecture. By converting 8048 source code to 8051 source code, the software investment developed for the 8048 is maintained when the system is upgraded.

The goal of the converter (CONV51) is to attain functional equivalence with the 8048 code by mapping each 8048 instruction to a corresponding 8051 instruction. In some cases a different instruction is produced because of the enhanced instruction set (e.g., bit CLR instead of ANL).

Although CONV51 tries to attain functional equivalence with each instruction, certain 8048 code sequences cannot be automatically converted. For example, a delay routine which depends on 8048 execution speed would require manual adjustment. A few instructions, in fact, have no 8051 equivalent (such as those involving P4-P7). Finally, there are a few areas of possible intervention such as PSW manipulation and interrupt processing, which at least require the user to confirm proper translation. The converter always warns the user when it cannot guarantee complete conversion.

CONV51 produces two files. The output file contains the ASM51 source program produced from the 8048 instructions. The listing file produces correlated listings of the input and output files, with warning messages in the output file to point out areas that may require users' intervention in the conversion.

LIB51 LIBRARIAN

The LIB51 utility enables MCS-51 programmers to create and maintain libraries of software object modules. With this utility, the customer can develop standard software modules and place them in libraries, which programs can access through a standard interface. When using object libraries, the linker will call only object modules that are required to satisfy external references.

Consequently, the librarian enables the customer to port and reuse software on different projects—thereby maintaining the customer's software investment.

SPECIFICATIONS

OPERATING ENVIRONMENT

All Intel Microcomputer Development Systems or Intel Personal Development System

Documentation Package:

MCS-51 Macro Assembler User's Guide
MCS-51 Utilities User's Guide for 8080/8085 Based Development System
MCS-51 8048-to-8051 Assembly Language Converter Operating Instructions for ISIS-II Users

ORDERING INFORMATION

Part Number	Description
MCI-51-ASM	8051 Software Development Package

*Requires Software License

SUPPORT:

Hotline Telephone Support, Software Performance Reporting (SPR), Software Updates, Technical Reports, Monthly Newsletter available.

MCS® -48 DISKETTE-BASED SOFTWARE SUPPORT PACKAGE

- **Extends Intellec microcomputer development system to support MCS-48 development**
- **MCS-48 assembler provides conditional assembly and macro capability**
- **Takes advantage of powerful ISIS-II file handling and storage capabilities**
- **Provides assembler output in standard Intel hex format**

The MCS-48 assembler translates symbolic 8048 assembly language instructions into the appropriate machine operation codes, and provides both conditional and macroassembler programming. Output may be loaded either to an ICE-49 module for debugging or into the iUP Universal PROM Programmer for 8748 PROM programming. The MCS-48 assembler operates under the ISIS-II operating system on Intel Development systems.

FUNCTIONAL DESCRIPTION

The MCS-48 assembler translates symbolic 8048 assembly language instructions into the appropriate machine operation codes. The ability to refer to program addresses with symbolic names eliminates the errors of hand translation and makes it easier to modify programs when adding or deleting instructions. Conditional assembly permits the programmer to specify which portions of the master source document should be included or deleted in variations on a basic system design, such as the code required to handle optional external devices. Macro capability allows the programmer use of a single label to define a routine. The MCS-48 assembler will assemble the code required by the reserved routine whenever the macro label is inserted in the text. Output from the assembler is in standard Intel hex format. It may be either loaded directly to an in-circuit emulator (ICE-49) module for integrated hardware/software debugging, or loaded into the iUP Universal PROM Programmer for 8748 PROM programming. A sample assembly listing is shown in Table 1.

The MCS 48 assembler supports the 8048, 8049, 8050, 8020, 8021, 8022, 8041 and 8042. The MCS 48 assembler can also support CMOS versions of the 8048 family.

Table 1. Sample MCS-48 Diskette-Based

```
ISIS-II 8048 MACROASSEMBLER, V1.0                                    PAGE 1

LOC  OBJ                    SEQ          SOURCE STATEMENT

                             1   ;DECIMAL ADDITION ROUTINE. ADD BCD NUMBER
                             2   ;AT LOCATION 'BETA' TO BCD NUMBER AT 'ALPHA' WITH
                             3   ;RESULT IN 'ALPHA'. LENGTH OF NUMBER IS 'COUNT' DIGIT
                             4   ;PAIRS. (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
                             5   ;AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
                             6   ;ODD)
                             7   INIT    MACRO   AUGND,ADDND,CNT
                             8           MOV     R0, #AUGND
                             9   L1:     MOV     R1, #ADDND
                            10           MOV     R2, #CNT
                            11           ENDM
                            12
0001E                       13   ALPHA   EQU     30
0028                        14   BETA    EQU     40
0032                        15   COUNT   EQU     5
0100                        16           ORG     100H
                            17           INIT    ALPHA, BETA, COUNT
0100  B81E                  18+          MOV     R0, #ALPHA
0102  B928                  19+L1:       MOV     R1, #BETA
0104  BA32                  20+          MOV     R2, #COUNT
0106  97                    21           CLR     C
0107  F0                    22   LP:     MOV     A, @R0
0108  71                    23           ADDC    A, @R1
0109  57                    24           DA      A
010A  A1                    25           MOV     @R0, A
010B  18                    26           INC     R0
010C  19                    27           INC     R1
010D  EA07                  28           DJNZ    R2, LP
                                         END

USER SYMBOLS
ALPHA  0001E    BETA 0028    COUNT 0005    LP 0107
L1     0102

ASSEMBLY COMPLETE, NO ERRORS


ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE, V1.0                       PAGE 1

SYMBOL CROSS REFERENCE

ALPHA  13#  17
BETA   14#  17
COUNT  15#  17
INIT    7#  17
L1     19#
LP     22#  28
```

SPECIFICATIONS

Operating Environment

(All) Intel Microcomputer Development Systems
(Series II, Series III/Series IV)
Intel Personal Development System

Documentation Package

Titles of: User Guides
Operating Instructions
Reference Manuals

Ordering Information

Part Number	Description
MDS-D48*	MCS-48 Disk Based Assembler Requires Software License

SUPPORT:

Hotline Telephone Support, Software Performance Reports (SPR), Software Updates, Technical Reports, Monthly Newsletters are available.

*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.

MCS®-96 SOFTWARE DEVELOPMENT PACKAGES

- **MCS®-96 Software Support Package**
- **PL/M-96 Software Package**

MCS®-96 SOFTWARE SUPPORT PACKAGE

- **Symbolic relocatable assembly language programming for the 8096 microcontroller family**
- **System Utilities for Program Linking and Relocation**
- **Extends Intellec® Microcomputer Development System to support MCS-96 program development**
- **Encourages modular program design for maintainability and reliability**

The MCS®-96 Software Support Package provides development system support for the MCS-96 family of 16-bit single chip microcomputers. The support package includes a macro assembler and system utilities.

The assembler produces relocatable object modules from MCS-96 macro assembly language instructions. The object modules then are linked and located to absolute memory locations.

The assembler and utilities run on the Intellec® Series III or equivalent Microcomputer Development System.

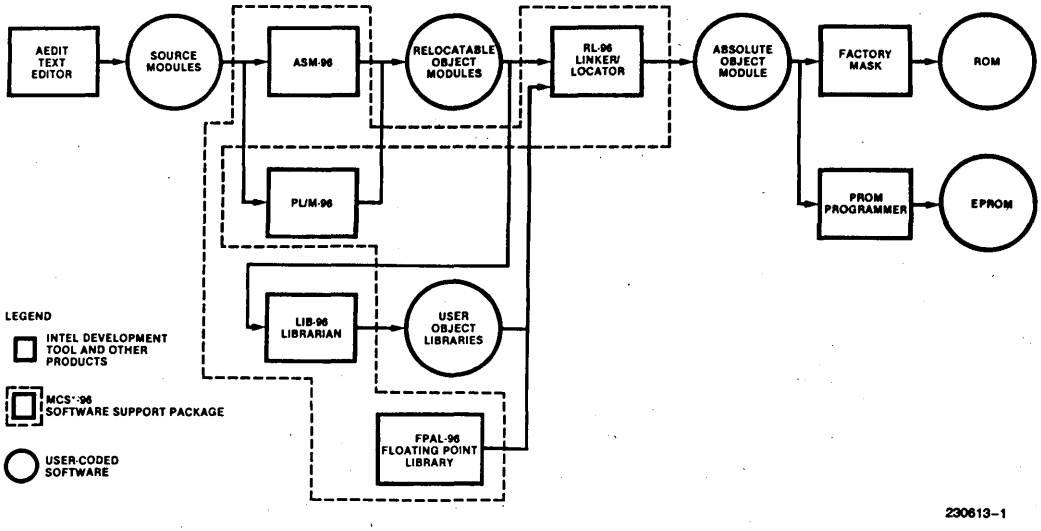


Figure 1. MCS®-96 Software Development Process

Order Number: 230613-003

8096 MACRO ASSEMBLER

- **Supports 8096 family program development on Intellec® Microcomputer Development System**
- **Gives symbolic access to powerful 8096 hardware features**
- **Object files are linkable and locatable**
- **Symbolic Assembler supports macro capabilities, cross reference, symbol table and conditional assembly**

ASM-96 is the macro assembler for the MCS family of microcontrollers. ASM-96 translates symbolic assembly language mnemonics into relocatable object code. Since the object modules are linkable and locatable, ASM-96 encourages modular programming practices.

The macro facility in ASM-96 allows programmers to save development and maintenance time since common code sequences only have to be done once. The assembler also provides conditional assembly capabilities.

ASM-96 supports symbolic access to the many features of the 8096 architecture. An "include" file is provided with all of the 8096 hardware registers defined. Alternatively, the user can define any subset of the 8096 hardware register set.

Math routines are supported with mnemonics for 16 × 16-bit multiply or 32/16-bit divide instructions.

The assembler runs on a Series III/Series IV Intellec Development Systems for high performance.

RL96 LINKER AND RELOCATOR PROGRAM

- **Links modules generated by ASM-96 and PL/M-96**
- **Locates the linked object module to absolute memory locations**
- **Encourages modular programming for faster program development**
- **Automated selection of required modules from Libraries to satisfy symbolic references**

RL96 is a utility that performs two functions useful in MCS-96 software development:

— The link function which combines a number of MCS-96 object modules into a single program.

— The locate functions which assigns an absolute address to all relocatable addresses in the MCS-96 object module.

RL96 resolves all external symbol references between modules and will select object modules from library files if necessary.

RL96 creates two files:

— The program or absolute object module file that can be executed by the targeted member of the MCS-96 family.

— The listing file that shows the results of link/locate, including a memory map symbol table and an optional cross reference listing.

The relocator allows programmers to concentrate on software functionally and not worry about the absolute addresses of the object code. RL96 promotes modular programming. The application can be broken down into separate modules that are easier to design, test and maintain. Standard modules can be developed and used in different applications thus saving software development time.

230613-003

FPAL96 FLOATING POINT ARITHMETIC LIBRARY

- **Implements IEEE Floating Point Arithmetic**
- **Basic Arithmetic Operations +, −, ×, /, Mod Plus Square Root**
- **Supports Single Precision 32 Bit Floating Point Variables**
- **Includes an Error Handler Library**

FPAL96 is a library of single precision 32-bit floating point arithmetic functions. All math adheres to the proposed IEEE floating point standard for accuracy and reliability. An error handler to handle exceptions (for example, divide by zero) is included.

The following functions are included:

ADD	NEGATE
SUBTRACT	ABSOLUTE
MULTIPLY	SQUARE ROOT
DIVIDE	INTEGER
COMPARE	REMAINDER

LIB 96

The LIB 96 utility creates and maintains libraries of software object modules. The customer can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces.

LIB 96 uses the following set of commands:

—CREATE: Creates an empty library file.
—ADD: Adds object modules to a library file.
—DELETE: Deletes object modules from a library file.
—LIST: Lists the modules in the library file.
—EXIT: Terminates LIB 96

When using object libraries, RL96 will include only those object modules that are required to satisfy external references, thus saving memory space.

SPECIFICATIONS

Operating Environment

Required Hardware:
Intellec Microcomputer Development System
— Series III/Series IV

Documentation Package:

MCS-96 Macro Assembler User's Guide
MCS-96 Utilities User's Guide
MCS-96 Assembler and Utilities Pocket Reference Card
8096 Floating Point Arithmetic Library

ORDERING INFORMATION

Part Number

iMDX-355
Requires Software License

Description

MCS-96 Software Support Package

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

230613-003

PL/M-96 SOFTWARE PACKAGE

- **High level programming language for the Intel MCS®-96 microcontroller family**
- **Block structured language design encourages module programming**
- **Provides access to MCS®-96 on chip resources**
- **Produces relocatable object code which is linkable to object modules generated by other MCS®-96 translators**
- **Resident on iAPX-86 Intel microcomputer development systems for higher performance**
- **Includes a linking and relocating utility and the library manager**
- **IEEE Floating Point Library included for numeric support**
- **Compatible with PL/M-86 assuring design portability**

PL/M-96 is a structured, high-level programming language useful for developing software for the Intel MCS-96 family of microcontrollers. PL/M-96 was designed to support the software requirements of advanced 16 bit microcontrollers. Access to the on chip resources of the MCS-96 has been provided in PL/M-96.

PL/M-96 is compatible with PL/M-86. Programmers familiar with PL/M will find they can program in PL/M-96 with little relearning effort.

The PL/M-96 compiler translates PL/M-96 high level language statements into MCS-96 machine instructions. By programming in PL/M an engineer can be more productive in the initial software development cycle of the project. PL/M can also reduce future maintenance and support cost because PL/M programs are easier to understand. PL/M-96 was designed to complement Intel's ASM-96.

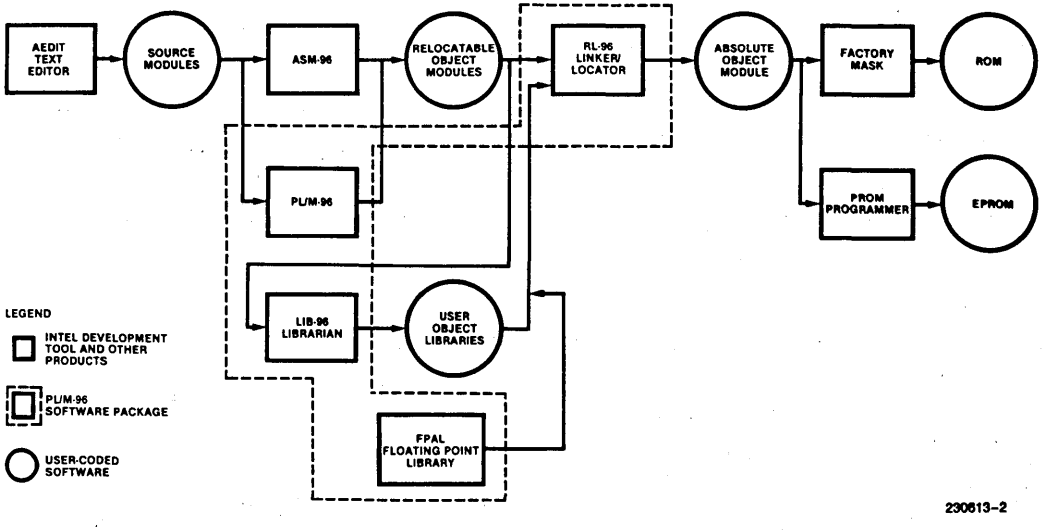


Figure 2. MCS®-96 Software Development Process

PL/M-96 COMPILER

FEATURES

Major features of the PL/M-96 compiler and programming language include:

Structured Programming

Programs written in PL/M-96 are developed as a collection of procedures, modules and blocks. Structured programs are easier to understand, maintain and debug. PL/M-96 programs can be made more reliable by clearly defining the scope of user variables (for example, local variables in a procedure). REENTRANT procedures are also supported by PL/M-96.

Language Compatibility

PL/M-96 object modules are compatible with all other object modules generated by Intel MCS-96 translators. Programmers may choose to link ASM-96 and PL/M-96 object modules together.

PL/M-96 object modules were designed to work with other Intel support tools for the MCS-96. The DEBUG compiler control provides these tools with symbolic information.

Data Types Supported

PL/M-96 supports seven data types for programmer flexibility in various logical, arithmetic and addressing functions. The seven data types include:

—BYTE:	8-bit unsigned number
—WORD:	16-bit unsigned number
—DWORD:	32-bit unsigned number
—SHORTINT:	8-bit signed number
—INTEGER:	16-bit signed number
—LONGINT:	32-bit signed number
—REAL:	32-bit floating point number

Another powerful feature are BASED variables. BASED variables allow the user to map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Data Structures Supported

Two data structuring facilities are supported by PL/M-96. The user can organize data into logical groups. This adds flexibility in referencing data.

— Array: Indexed list of same type data elements

— Structure: Named collection of same or different type data elements

— Combinations of Both: Arrays of structures or structures of arrays

Interrupt Handling

Interrupts are supported in PL/M-96 by defining a procedure with the INTERRUPT attribute. The compiler will generate code to save and restore the program status word when handling hardware interrupts of the MCS-96.

Compiler Controls

Compile time options increase the flexibility of the PL/M-96 compiler. These controls include:

— Optimization

— Conditional compilation

— The inclusion of common PL/M-96 source files from disk

— Cross reference of symbols

— Optional assembly language code in the listing file

230613-003

Code Optimizations

The PL/M-96 compilers has four levels of optimization for reducing program size.

— Combination of constant expressions; "Strength reductions" (e.g.: a shift left rather than multiply by two)

— Machine code optimizations; elimination of superfluous branches; reuse of duplicate code, removal of unreachable code

— Overlaying of on chip RAM variables

— Optimization of based variable operations

— Use of short jumps where possible

Built in Functions

An extensive list of built in functions has been supplied as part of the PL/M-96 language. Besides TYPE CONVERSION functions, there are built in functions for STRING manipulations. Functions are provided for interrogating the MCS-96 hardware flags such as CARRY and OVERFLOW.

Error Checking

If the PL/M-96 compiler detects a programming or compilation error, a fully detailed error message is provided by the compiler. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This powerful PL/M-96 feature can yield a two times increase in throughput when a user is in the initial program development cycle.

BENEFITS

PLM-96 is designed to be an efficient, cost-effective solution to the special requirements of MCS-96 Microcontroller Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M-96 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M-96, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M-96 is designed to aid in the development of reliable software (PL/M programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status. The more simply the program is stated, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

RL96 LINKER AND RELOCATOR PROGRAM

- **Links modules generated by ASM-96 and PL/M-96**
- **Locates the linked object module to absolute memory locations**
- **Encourages modular programming for faster program development**
- **Automated selection of required modules from Libraries to satisfy symbolic references**

RL96 is a utility that performs two functions useful in MCS software development:

— The link function which combines a number of MCS object modules into a single program.

— The locate function which assigns an obsolute address to all relocatable addresses in the MCS-96 object module.

RL96 resolves all external symbol references between modules and will select object modules from library files if necessary.

RL96 creates two files:

— The program or absolute object module file that can be executed by the targeted member of the MCS family.

— The listing file that shows the results of link/locate, including a memory map symbol table and an optional cross reference listing.

The relocator allows programmers to concentrate on software functionality and not worry about the absolute addresses of the object code. RL96 promotes modular programming. The application can be broken down into separate modules that are easier to design, test and maintain. Standard modules can be developed and used in different applications thus saving software development time.

FPAL96 FLOATING POINT ARITHMETIC LIBRARY

- **Implements IEEE Floating Point Arithmetic**
- **Basic Arithmetic Operations +, −, ×, /, Mod Plus Square Root**
- **Supports Single Precision 32 Bit Floating Point Variables**
- **Includes an Error Handler Library**

FPAL96 is a library of single precision 32-bit floating point arithmetic functions. All math adheres to the proposed IEEE floating point standard for accuracy and reliability. An error handler to handle exceptions (for example, divide by zero) is included.

The following functions are included:

ADD	NEGATE
SUBTRACT	ABSOLUTE
MULTIPLY	SQUARE ROOT
DIVIDE	INTEGER
COMPARE	REMAINDER

LIB 96

The LIB 96 utility creates and maintains libraries of software object modules. The customer can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces.

LIB 96 uses the following set of commands:

—CREATE: Creates an empty library file
—ADD: Adds object modules to a library file
—DELETE: Deletes object modules from a library file
—LIST: Lists the modules in the library file
—EXIT: Terminates LIB 96

When using object libraries, RL96 will include only those object modules that are required to satisfy external references, thus saving memory space.

SPECIFICATIONS

Operating Environment

Required Hardware:
Intellec Microcomputer Development System
— Series III/Series IV

Documentation Package:

PL/M-96 User's Guide
MCS-96 Utilities User's Guide
MCS-96 Assembler and Utilities Pocket Reference Card
8096 Floating Point Arithmetic Library

ORDERING INFORMATION

Part Number

iMDX-356
Requires Software License

Description

PL/M-96 Software Package

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

VAX*/VMS* RESIDENT iAPX-86/88/186 SOFTWARE DEVELOPMENT PACKAGES

- **Executes on DEC VAX* Minicomputer under VMS* Operating System to translate PL/M-86, Pascal-86 and ASM-86 Programs for iAPX-86, 88 and 186 Microprocessors.**
- **Packages include Pascal-86; PL/M-86; ASM-86; Link and Relocation Utilities; OH-86 Absolute Object Module to Hexadecimal Format Converter; and Library Manager Program.**
- **Output linkable with Code Generated on Intellec® Development Systems.**

The VAX/VMS Resident Software Development Packages contain software development tools for the iAPX-86, 88, and 186 microprocessors. The package lets the user develop, compile, maintain libraries, and link and locate programs on a VAX running the VMS operating system. The translator output is object module compatible with programs translated by the corresponding version of the translator on an Intellec Development System.

Three packages are available:

1. An ASM-86 Assembler Package which includes the Assembler, the Link Utility, the Locate Utility, the absolute object to hexadecimal format conversion utility and the Library Manager Program.
2. A PL/M-86 Compiler Package which contains the PL/M-86 Compiler and Runtime Support Libraries.
3. A Pascal-86 Compiler Package which contains the Pascal-86 Compiler and Runtime Support Libraries.

The VAX/VMS resident development packages and the Intellec Development System development packages are built from the same technology base. Therefore, the VAX/VMS resident development packages and the Intellec Development System development packages are very similar.

Version numbers can be used to identify features correspondence. The VAX/VMS resident development packages will have the same features as the Intellec Development System product with the same version number.

Support for the iAPX-186 processor will be provided as an update to the iAPX-86, 88 software.

The object modules produced by the translators contain symbol and type information for programming debugging using ICE™ translators and/or the PSCOPE debugger. For final production version, the compiler can remove this extra information and code.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

JUNE 1984
ORDER NUMBER: 210643-002

VAX*-PL/M-86/88/186 SOFTWARE PACKAGE

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-To-Learn Block-Structured Language Encourages Program Modularity**
- **Produces Relocatable Object Code Which is Linkable to All Other Intel 8086 Object Modules, Generated on Either a VAX* or Intellec® Development Systems**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**
- **Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors**
- **Source Input/Object Output Compatible with PL/M-86 Hosted on an Intellec® Development System**
- **ICE™, PSCOPE Symbolic Debugging Fully Supported**

Like its counterpart for MCS®-80/85 program development, and Intellec® hosted iAPX-86 program development, VAX-PL/M-86 is an advanced, structured high-level programming language. The VAX-PL/M-86 compiler was created specifically for performing software development for the Intel iAPX-86, 88, and 186 Microprocessors.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The VAX-PL/M-86 compiler efficiently converts free-form PL/M language statements into equivalent iAPX-86/88/186 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-PASCAL-86/88 SOFTWARE PACKAGE

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **Produces Relocatable Object Code Which Is Linkable to All Other Intel 8086 Object Modules, Generated on Either a VAX* or Intellec® Development Systems**
- **ICE™, PSCOPE Symbolic Debugging Fully Supported**
- **Implements REALMATH for Consistent and Reliable Results**
- **Supports iAPX-86/20, 88/20 Numeric Data Processors**
- **Strict Implementation of ISO Standard Pascal**
- **Useful Extensions Essential for Microcomputer Applications**
- **Separate Compilation with Type-Checking Enforced Between Pascal Modules**
- **Compiler Option to Support Full Run-Time Range-Checking**
- **Source Input/Object Output Compatible with Pascal-86 Hosted on a Intellec Development System**

VAX-PASCAL-86 conforms to and implements the ISO Pascal standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. Other extensions include additional data types not required by the standard and miscellaneous enhancements such as an allowed underscore in names, an OTHERWISE clause in CASE construction and so forth. To assist the development of portable software, the compiler can be directed to flag all non-standard features.

The VAX-PASCAL-86 compiler runs on the Digital Equipment Corporation VAX under the VMS Operating System. A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs on the target system as an alternate to the development system environment. Program modules compiled under PASCAL-86 are compatilble and linkable with modules written in PL/M-86, and ASM-86. With a complete family of compatible programming languages for the iAPX-86, 88, and 186 one can implement each module in the language most appropriate to the task at hand.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-iAPX-86/88/186 MACRO ASSEMBLER

- **Executes on VAX* Minicomputer Under The VMS* Operating System**
- **Produces Relocatable Object Code Which Is Linkable to All Other Intel iAPX-86/88/186 Object Modules, Generated on Either a VAX* or Intellec® Development Systems**
- **Powerful and Flexible Text Macro Facility with Three Macro Listing Options to Aid Debugging**
- **Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions**
- **"Strongly Typed" Assembler Helps Detect Errors at Assembly Time**
- **High-Level Data Structuring Facilities Such as "STRUCTURES" and "RECORDS"**
- **Over 120 Detailed and Fully Documented Error Messages**
- **Produces Relocatable and Linkable Object Code**
- **Source Input/Object Output Compatible with ASM-86 hosted on an Intellec Development System**

VAX-ASM-86 is the "high-level" macro assembler for the iAPX-86/88/186 assembly language. VAX-ASM-86 translates symbolic iAPX-86/88/186 assembly language mnemonics into iAPX-86/88/186 relocatable object code.

VAX-ASM-86 should be used where maximum code efficiency and hardware control is needed. The iAPX-86/88/186 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible iAPX-86/88/186 machine instructions. VAX-ASM-86 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

VAX-ASM-86 offers many features normally found only in high-level languages. The iAPX-86/88/186 assembly language is strongly typed. The assembler performs extensive checks on the usage of variable and labels. The assembler uses the attributes which are derived explicity when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-LIB-86

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **VAX*-LIB-86 is a Library Manager Program which Allows You to:**
  Create Specifically Formatted Files to Contain Libraries of Object Modules
  Maintain These Libraries by Adding or Deleting Modules
  Print a Listing of the Modules and Public Symbols in a Library File
- **Libraries Can be Used as Input to VAX*-LINK-86 Which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program VAX-LIB-86 creates and maintains files containing object modules. The operation of VAX-LIB-86 is controlled by commands to indicate which operation VAX-LIB-86 is to perform. The commands are:

CREATE:	creates an empty library file
ADD:	adds object modules to a library file
DELETE:	deletes modules from a library file
LIST:	lists the module directory of library files
EXIT:	terminates the LIB-86 program and returns control to VMS

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

VAX-OH-86

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **Converts an iAPX 86/88/186 Absolute Object Module to Symbolic Hexadecimal Format**
- **Facilitates Preparing a file for Loading by Symbolic Hexadecimal Loader (e.g. iSBC™ Monitor SDK-86 Loader), or Universal PROM Mapper**
- **Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging**

The VAX-OH-86 utility converts an 86/88 absolute object module to the hexadecimal format. This conversion may be necessary for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or the Universal PROM Mapper. The conversion may also be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute form; the output from VAX-LOC-86 is in absolute format.

*VAX, VMS are trademarks of Digital Equipment Corporation.

VAX*-LINK-86

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **Automatic Combination of Separately Compiled or Assembled 86/88/186 Programs Into a Relocatable Module, Generated on Either a VAX or an Intellec® Development System**
- **Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References**
- **Extensive Debug Symbol Manipulation, allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**
- **Automatic Generation of a Summary Map Giving Results of the LINK-86 Process**
- **Abbreviated Control Syntax**
- **Relocatable modules may be Merged into a Single Module Suitable for Inclusion in a Library**
- **Supports "Incremental" Linking**
- **Supports Type Checking of Public and External Symbols**

VAX-LINK-86 combines object modules specified in the VAX-LINK-86 input list into a single output module. VAX-LINK-86 combines segments from the input modules according to the order in which the modules are listed.

VAX-LINK-86 will accept libraries and object modules built from VAX-PL/M-86, VAX-PASCAL-86, VAX-ASM-86, or any other Intel translator generating 8086 Relocatable Object Modules, such as the Series III resident translators.

Support for incremental linking is provided since an output module produced by VAX-LINK-86 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

VAX-LINK-86 supports type checking of PUBLIC and EXTERNAL symbols reporting a warning if their types are not consistent.

VAX-LINK-86 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the VAX-LINK-86 process and to control the content of the output module.

VAX-LINK-86 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using VAX-LOC-86 and enter final testing with much of the work accomplished.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-LOC-86

- **Executes on the VAX* Minicomputer Under the VMS* Operating System**
- **Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Length, and Debug Symbols and their Addresses**
- **Extensive Capability to Manipulate the Order and Placement of Segments in 8086/8088 Memory**
- **Abbreviated Control Syntax**
- **Automatic and Independent Relocation of Independent Relocation of Segments. Segments May be Relocated to Best Match Users Memory Configuration**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

VAX-LOC-86 converts relative addresses in an input module in iAPX-86/88/186 object module format to absolute addresses. VAX-LOC-86 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

VAX-LOC-86 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the VAX-LOC-86 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program and then simply relocate the object code to suit your application.

SPECIFICATIONS

Operating Environment

Required Hardware

VAX* 11/780, 11/782, 11/750, or 11/730
9 Track Magnetic Tape Drive, 1600 BPI

Required Software

VMS Operating System V3.0 or Later. All of the development packages are delivered as unlinked VAX object code which can be linked to VMS as designed for the system where the development package is to be used. VMS command files to perform the link are provided.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

Documentation Package

iAPX-86, 88 Development Software Installation Manual and User's Guide for VAX/VMS, Order number 121950-001

Shipping Media

9 Track Magnetic Tape 1600 bpi

ORDERING INFORMATION

Part Number	Description
iMDX-341VX	VAX-ASM-86, VAX-LINK-86, VAX-LOC-86, VAX-LIB-86, VAX-OH-86, Package
iMDX-343VX	VAX-PLM-86 Package
IMDX-344VX	VAX-PASCAL-86 Package

REQUIRES SOFTWARE LICENSE

VAX*/VMS* RESIDENT SOFTWARE DEVELOPMENT PACKAGES FOR iAPX 286

- **Hosted on DEC VAX* Minicomputer Under the VMS* Operating System**
- **Allows Development of System and Application Software for the Protected Virtual Address Mode of the iAPX-286**
- **Packages include PL/M-286, BUILD-286, BIND-286, LIB-286 and MAP-286**
- **Compatible with Corresponding Intel Development System Resident Products**

These packages provide the capability of developing software on a VAX*/VMS* host for the iAPX-286 in protected virtual address mode. With these packages a user can assemble and compile 286 programs, configure system and application software and create and manage 286 object libraries. Figure 1 illustrates the process of 286 software development on VAX*/VMS* hosts.

Two packages are available:

1. A PL/M-286 package which contains the PL/M-286 compiler and run time support libraries.
2. An ASM-286 package which contains the iAPX-286 Assembler (ASM-286) and programming utilities. These utilities include the iAPX-286 System Builder (BLD-286), the System Binder (BND-286), a Library Utility (LIB-286) and an Object Map Utility (MAP-286).

These packages are compatible with corresponding products which are hosted on Intel development systems. Correspondence can be established via version numbers. For example, BND-286 V2.0 offers the same set of features on VAX/VMS and Intel development systems.

Owing to this compatibility, iAPX-286 software developed on VAX/VMS can be linked to iAPX-286 software from development systems. Moreover, iAPX-286 programs developed on the VAX can then be downloaded to development systems and debugged using 286 debuggers like the I^2ICE^{TM}-286 system.

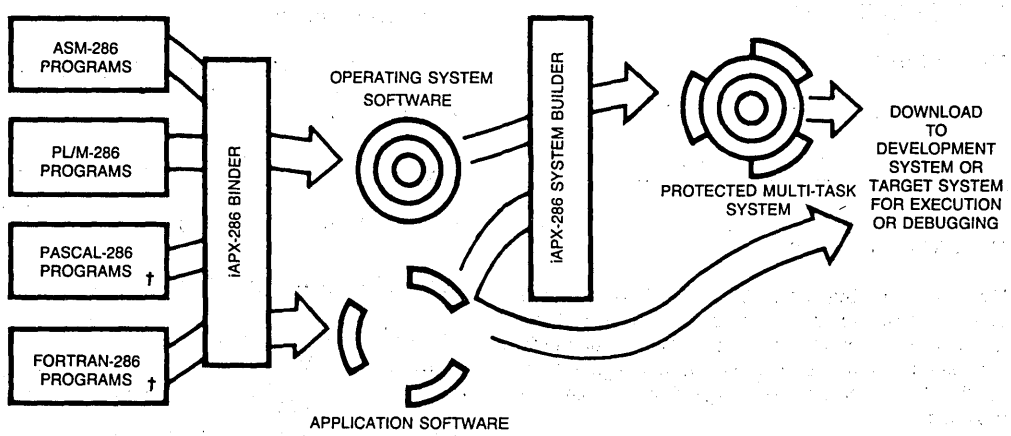


Figure 1: 286 Software Development on VAX*/VMS*

*VAX, VMS are trademarks of Digital Equipment Corporation †Currently Available on Intel Development Systems Only

MARCH 1984
ORDER NUMBER: 231038-001

VAX*/VMS* RESIDENT PL/M-286

- **Systems Programming Language for the protected virtual address mode iAPX-286**
- **Produces relocatable object code linkable to object modules generated by other Intel 286 language translators**
- **Enhanced to support design of protected, multi-user, multi-tasking, virtual memory operating system software**
- **Upward compatible with PL/M-86 and PL/M-80 to allow software portability**
- **Provides multiple levels of optimization to produce efficient code**
- **Compatible with development system resident PL/M-286**

PL/M-286 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode iAPX-286. PL/M-286 has been enchanced to utilize iAPX-286 features–memory management and protection–for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M-286 is upward compatible with PL/M-86 and PL/M-80. Existing systems software can be re-compiled with PL/M-286 to execute in protected virtual address mode on the iAPX-286.

PL/M-286 is the high-level alternative to assembly language programming on the iAPX-286. For the majority of iAPX-286 system programs, PL/M-286 provides the features needed to access and to control efficiently the underlying iAPX-286 hardware, and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M-286 compiler has been designed to efficiently support all phases of software development. Features such as built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed. The compiler also provides complete symbolic debug capability to the various 286 debuggers and emulators.

VAX/VMS resident PL/M-286 is completely feature compatible with development system resident PL/M-286 with the same version number.

231038-001

VAX*/VMS* RESIDENT iAPX-286 MACRO ASSEMBLER

- **Supports full Instruction Set of the iAPX-286 including memory protection and numerics (with 80287)**
- **Structures and RECORDS provide powerful data representation**
- **Type checking at assembly time helps reduce errors at run-time**
- **Powerful and flexible Text Macro facility**
- **Upward compatible with ASM-86/88/186**
- **Compatible with development system resident iAPX-286 Macro Assembler**

ASM-286 is the "high-level" macro assembler for the iAPX-286 assembly language. ASM-286 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM-86/88 mnemonics; new ones have also been added to support the new iAPX-286 instructions. The segmentation directives have been greatly simplified.

The iAPX-286 assembly language includes approximately 150 instruction mnemonics. From these few mnemonics the assembler can generate over 4,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 150 mnemonics to generate all possible machine instructions. ASM-286 generates the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM-286 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM-286 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This means that many programming errors will be detected when the program is assembled, long before it is being debugged.

ASM-286 object modules conform to a thorough, well-defined format used by 286 high-level languages and utilities. This makes it easy to call (and be called from) HLL object modules.

ASM-286 also provides support for the 80287 numerics co-processor. The complete instruction set of the 80287 is available through high-level mnemonics.

VAX/VMS resident ASM-286 is completely feature compatible with development system resident ASM-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 SYSTEM BUILDER

- **A tool for configuring multi-tasking protected, virtual memory systems software for the iAPX-286**
- **Links separately compiled modules. Resolves EXTERNAL/PUBLIC definitions**
- **Creates a memory image of a 286 system for cold start execution**
- **Target system may be bootloadable, programmed into ROM or loaded from mass storage**
- **Generates print file with command listing and system map**
- **Compatible with development system resident iAPX-286 System Builder**

BLD-286 is the iAPX-286 System Builder. It allows systems programmers to configure multi-tasking and memory protected iAPX-286 software. The configuration is specified by the user in a "Build file" using a symbolic meta-language. BLD-286 thus provides the programmer a high-level symbolic interface to the multi-tasking and memory protection features of the iAPX-286 architecture.

BLD-286 accepts as inputs object modules from the iAPX-286 translators, the iAPX-286 Binder and itself (for incremental building). Using the programmer's specifications in the Build File, it produces a bootloadable or loadable module as well as a print file with a map of the configured module.

Using the builders command language, system programmers may perform the following functions:

— Assign physical addresses to segments; also set segment access rights and limits.
— Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.
— Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.
— Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.
— Create Task State Segments and Task Gates for multi-tasking applications.
— Resolve inter-module and inter-level references, and perform type-checking.
— Automatically select required modules from libraries.
— Configure the memory image into partitions in the address space.
— Selectively generate an object file and various sections of the print file.

VAX/VMS BLD-286 is completely feature compatible with development system resident BLD-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 BINDER

- **Links separately compiled program modules into an executable task**
- **Makes the iAPX-286 protection mechanism invisible to application programmers**
- **Assigns virtual addresses to tasks**
- **Performs incremental linking with output of Binder and Builder**
- **Resolves PUPLIC/EXTERNAL code and data references, and performs intermodule type-checking**
- **Provides print file showing segment map, errors and warnings**
- **Generates linkable or loadable module for debugging**
- **Compatible with development system resident iAPX-286 Binder**

BND-286 is a utility that combines iAPX-286 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules, produced by the 286 translators, and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND-286 generates a print file displaying a segment map, and error messages.

The Binder is useful for system as well as application programmers. Since application programmers need to develop software independent of any system architecture, the 286 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged, as well.) System protection features are specified later in the development cycle, using the 286 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of use of the 286 utilities.

VAX/VMS resident BND-286 is completely feature compatible with development system resident BND-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 LIBRARIAN

- **Allows creation and management of iAPX-286 object libraries**
- **Library functions include Create, Delete, Add, Replace, Copy, Save, Backup and Display**
- **Only required modules linked in when using Binder or Builder**
- **Compatible with development system resident iAPX-286 Librarian**

LIB-286 is the iAPX-286 Librarian. It can be used to create and manage iAPX-286 Object Libraries. By placing often used object modules into libraries, the administrative overhead of managing software modules can be reduced.

VAX/VMS based LIB-286 is completely feature compatible with development system resident LIB-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 MAPPER

- **Flexible Utility to display object file information in symbolic form**
- **Compatible with development system resident iAPX-286 Mapper**

MAP-286 is a cross reference utility for iAPX-286 object modules. It provides a symbolic listing of the EXTERNAL and PUBLIC symbols in the specified object modules.

VAX/VMS resident MAP-286 is completely feature compatible with development system resident MAP-286 with the same version number.

SPECIFICATIONS

Operating Environment

DEC VAX* 11/780 or compatible model running VMS* operating system V3.4 (or upward compatible versions)

Documentation

Installation guide and user's manuals for the software are supplied with the products.

SUPPORT

Hotline Telephone Support, Software, Performance Report (SPR) Software Updates, Technical Reports and Monthly Newsletters are available.

ORDERING INFORMATION

Product Code	Description
iMDX-371VX	ASM-286, BLD-286, BND-286, LIB-286, MAP-286
iMDX-373VX	PL/M-286

*VAX,/VMS are trademarks of Digital Equipment Corporation

2920 SOFTWARE SUPPORT PACKAGE

- **Complete software design and development support for the 2920**
- **Extends Intellec® Microcomputer Development System to support 2920 software development**

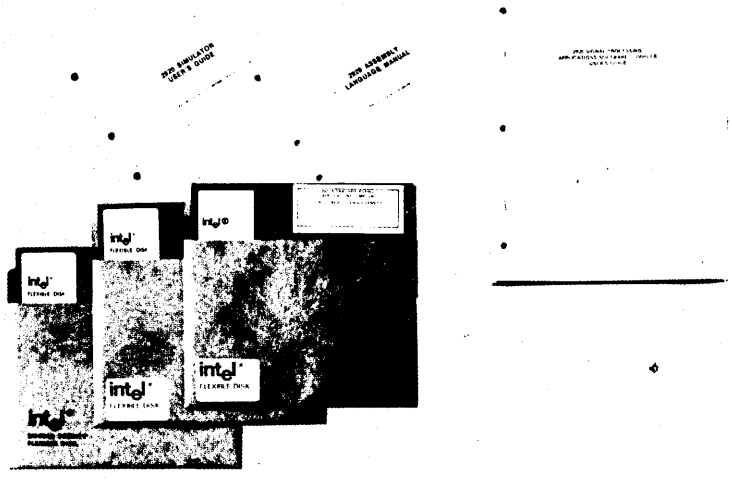
The 2920 Software Support Package furnishes a 2920 Signal Processing Applications Software/Compiler, 2920 Assembler, and 2920 Software Simulator. These three software design and development tools run on the Intellec® Microcomputer Development System.

The 2920 Signal Processing Application Software/Compiler is an interactive tool for designing software to be executed on the 2920 Signal Processor. The compiler accepts English-like statements from the user and generates 2920 assembly language code.

The assembler translates symbolic 2920 assembly language programs into the machine operation code. The user can load the code into the simulator for 2920 simulation or to the Universal PROM Programmer for 2920 EPROM programming.

The simulator, operating entirely in software, allows the user to test and symbolically debug 2920 programs. The user can specify input signals, simulate program execution, set up breakpoints, display input and output, and display and alter the contents of the 2920 registers and memory locations. The simulator can also stop or trace the program and constructively give the user access to the key elements inside a 2920 for analyzing his program.

The compiler, assembler, and simulator enable the designer to develop and test an entire program without a complete prototype design. The 2920 designer works on the Intellec® Microcomputer Development System rather than on a breadboard. The development system can program, store and recall programs or routines and aid in 2920 program design.



2920 Software Support Package

Sept 1980
1662208

2920 SIGNAL PROCESSING APPLICATIONS SOFTWARE/COMPILER

- **Compiler generates 2920 Assembly Language Code**
- **Extensive command set for designing electrical filters**
- **Graphics capability enhances analysis of filter response or piecewise linear function approximations**
- **Powerful MACRO capability for executing frequently used routines**
- **Interactive software support tool for 2920 Signal Processor**
- **Extends Intellec® Microcomputer Development System support of the 2920**
- **Contains MACRO library for several standard filters and signal processing functions**

The 2920 Signal Processing Applications Software/Compiler (SPAS20) is an interactive tool for designing software to execute on the 2920 Signal Processor.

The SPAS20 package can be visualized as being comprised of four inter-related sections: A compiler section, a filter design section, a curve fitting section, and a MACRO section.

Among the abilities of SPAS20 are: ability to generate 2920 assembly language code directly from specifications of signal processing building blocks such as filters and waveform generators; ability to generate 2920 assembly language code for several classes of algebraic equations such as $Y = C * X$, $Y = C * Y$, and $Y = C * X + Y$ where X, Y are variables and C is a constant; ability to generate 2920 assembly language code for one variable function $Y(X) = F(X)$; ability to examine time and frequency responses of filter sections specified by continuous or sampled poles and zeroes; ability to examine piecewise linear approximation of specific function; ability for users to implement more complex commands by grouping sets of commonly used commands into a MACRO.

The SPAS20 package runs under ISIS-II on any Intellec® Microcomputer Development System with 64K RAM. The output of SPAS20 can be assembled with the 2920 assembler, tested with the 2920 Simulator, and programmed into the 2920 chip with the Universal PROM Programmer for prototyping.

FUNCTIONAL DESCRIPTION

The 2920 Signal Processing Applications Software/ Compiler gives the analog designer a "high level language" for his 2920 applications—it decreases the need to code 2920 assembly language. Furthermore, the compiler is interactive. This feature enables the designer to define a filter, or transfer function, graph their response, and change their parameters many times, without having to program and test in an actual 2920 implementation.

Once a filter is realized by moving poles and zeros in the continuous and sampled planes, the filter may be coded and written onto an ISIS file. Similarly, after a function Y = F(X) has been defined, the code for a piecewise linear approximation can be stored onto an ISIS file. Several other file commands are available to store and retrieve command sequences for SPAS20 sessions.

SPAS20 Command Language

DEFINE	This command defines a pole or zero by associating it with a number (i.e., POLE 3), and with real and imaginary coordinates in the continuous or sampled plane. This command also defines a symbol by associating a name with a numeric value, or a MACRO by providing a pointer to a specified command sequence.
GRAPH/ OGRAPH	This command graphically displays the values of object(s) specified. For example, GRAPH GAIN and GRAPH PHASE are used to display filter response. The OGRAPH command will "overgraph" the new response over the old response, after any changes have been made. (You may also graph Group Delay, Step, and Impulse.)
MOVE	Allows the definition of a pole or zero to be changed—its coordinates, its plane, or both.
REMOVE	Deletes the definition of a pole, zero, symbol, or macro.
HELP	Types an explanatory message on the console, pertaining to a command or its attributes.
FIT	This command performs curve fitting, i.e. it approximates an arbitrary user supplied function with a piecewise linear function.
DATA	This command allows for specification of a set of vertices (i.e. X – Y coordinate pairs) which determine a piecewise linear approximation of some defined function, filter response characteristics, etc.
HOLD	Command to correct attenuation due to sample-and-hold distortion: if ON, it corrects absolute gain by sin(x)/x and phase by adding x, where x=TS*FREQ*π. It corrects group delay by subtracting π*TS.
EVALUATE	Gives the decimal numeric value of any expression.
CODE	Creates 2920 assembly language code for given poles, and zeros, equations, and user defined functions.

The SPAS20 compiler also recognizes the following commands for file handling:

PUT/ APPEND	Writes out objects (commands) to a specified file, either creating a new one or appending an existing one. This enables the user to store all or part of a SPAS20 session on a diskette to be brought back later with the INCLUDE command.
DISPLAY	Copies the contents of a file to the console.
INCLUDE	Executes a sequence of instructions from a diskette file as if they were typed in from the console.
LIST	Creates a file containing all console interactions.

In addition to naming macros for specific command sequences, compound and conditional commands may be formed using all of the above statements. These compound commands are:

IF	Establishes conditional flow of control within a block of commands.
REPEAT	Used for repetition of a block of commands; executes indefinitely or until a condition is met (using WHILE, UNTIL, and END statements).
COUNT	Establishes the number of times a command sequence is to be executed, in a looping fashion.

SPAS20 MACRO Facility

A macro is a sequence of commands that is stored on a temporary diskette file. The command sequence is executed when the macro name is entered as a command. This saves repetitive entry of the sequence, and permits alogorithms to be saved on diskette for future use. This SPAS20 facility allows you to do the following:

- Display the text of any macro.
- Define a macro, specifying its name and any parameters that are to be used by the block. This definition is followed by the contents of the macro (commands) and the EM statement to end its definition.
- Invoke a macro by entering its name and appropriate values for any parameters.
- List the names of all defined macros.
- Remove any or all macros.

Intel also supplies several MACRO library files containing the following commonly needed MACROs:

- Filter design MACROS
  - — Butterworth filter
  - — Chebyshev filter
  - — Bilinear transform
  - — Evaluate gain or phase of digital filter in parallel form
  - — Time response simulation
- Function design MACROs
  - — Code and error optimization
  - — Calculate instertitial error
- MACROs for generation of 2920 code
  - — Code for all-POLE filter
  - — Input and A/D conversion
  - — Multiplication
  - — Division
  - — Logarithm functions
  - — Square-root functions
  - — Sinewave oscillator

SAMPLE SPAS20 FILTER DESIGN SESSION

```
-:FI : SPAS20 . SFT
ISIS-II 2920 SIGNAL PROCESSING APPLICATIONS COMPILER. V2.0
*
*DEFINE POLE 1 = -707,707        ; CREATE A POLE IN CONTINUOUS S-PLANE
*
*PZ        . LIST ALL POLES AND ZEROS
POLE 1 = -707.00000,707.00000,CONTINUOUS
*
*FSCALE = 100,10000        ; ESTABLISHES FREQUENCY RANGE OF INTEREST
*
*YSCALE = -45,1        ; ESTABLISHES MAGNITUDE RESPONSE RANGE OF INTEREST
*
*GRAPH GAIN        ; PLOT MAGNITUDE RESPONSE OF POLE PAIR

GAIN      !.....^...^.....^...^...^....^....^....^....^.....^.....^.....!
  1.0     -------------------------....
 -1.2                              ''--.
 -3.4                                   '-.
 -5.6                                      '.
 -7.8                                       '-
-10.0                                         '-
-12.1                                          '.
-14.3                                           '.
-16.5                                            '.
-18.7                                             '-.
-20.9                                               '-
-23.1                                                '.
-25.3                                                 '.
-27.5                                                  '.
-29.7                                                   '-.
-31.9                                                     '-
-34.0                                                      '-
-36.2                                                       '.
-38.4                                                        '.
-40.6
-42.8
-45.0
    DBIHZ !.....^...^.....^...^...^....^....^... ^ ...^.....^.....^.....!
          100 150 200   300 400 500  700 1000 1400 2000  3000    5000     10000
**
*
*: THE UNITS USED IN GRAPHING GAIN ARE SHOWN IN THE LOWER LEFT CORNER.
*: GAIN IN DECIBELS IS GRAPHED VERSES FREQUENCY IN HERTZ
*
*: PREPARE TO MOVE TO THE DIGITAL DOMAIN.
*: SAMPLE RATE MUST BE SPECIFIED.
*
*TS = 1/13020        ; RATE FOR 192 INSTRUCTION PROGRAM AND 10MHZ CLOCK
TS = 7.6805004/10**5
*
```

SAMPLE SPAS20 FILTER DESIGN SESSION (Cont'd.)

```
*MOVE POLE TO Z          ; CONVERT FILTER TO DIGITAL VIA MATCHED-Z TRANSFORMATION
1 POLES/ZEROES MOVED
*
*P          ; LIST TRANSFORMED POLE
POLE 1 = 0.71092836,0.34118369,Z
*
*; COMPARE RESPONSES OF THE ANALOG AND DIGITAL FILTERS BY GRAPHING THE
*; NEW RESPONSE OVER THE OLD
*
*GRAPH GAIN

GAIN      !....^...^....^...^...^....^....^....^....^....^....^.....!
   1.0    ----------------------------...
  -1.2                                  ''--.
  -3.4                                      '-.
  -5.6                                        '-.
  -7.9                                         +-.
-10.0                                            +-.
-12.1                                             +'.
-14.3                                               +'.
-16.5                                                +'-
-18.7                                                  +'-.
-20.9                                                    + -.
-23.1                                                     ++ '--..
-25.3                                                        ++    '''.-
-27.5                                                          ++
-29.7                                                            ++
-31.9                                                              ++
-34.0                                                               +
-36.2                                                                ++
-38.4                                                                  ++
-40.6                                                                    +
-42.8
-45.0
    DB!HZ '....^...^....^...^...^....^....^....^....^....^....^.....!
          100 150 200   300 400 500  700 1000 1400 2000  3000    5000     10000
C*
*
*; PLUS SIGNS INDICATE OLD CURVE
*; NOTE THAT THE DIGITAL FILTER RESPONSE BEGINS TO INCREASE AGAIN
*; AT HALF THE SAMPLE RATE ( 6510 HZ ).
*
*; THE PHASE CHARACTERISTICS OF THIS FILTER CAN BE EXAMINED
*
*YSCALE = -PI,PI          ; ESTABLISHES RANGE OF INTEREST
*
*GRAPH PHASE

PHASE     !....^...^....^...^...^....^....^....^....^....^....^.....!
 3.14
 2.84
 2.54
 2.24
 1.94
 1.65
 1.35                                                              -
 1.05                                                            .'
 0.75                                                          .'
 0.45                                                        .'
 0.15                                                      .'
-0.15     ''''''--------....                            .'
-0.45                    ''''--..                    .'
-0.75                          ''--..              .'
-1.05                                '-..        .'
-1.35                                    ''--.....--'''
-1.65
-1.94
-2.24
-2.54
-2.84
-3.14
   RAD!HZ !....^...^....^...^...^....^....^....^....^....^....^.....!
          100 150 200   300 400 500  700 1000 1400 2000  3000    5000     10000
P*
*
*PUT :F1:POLE PZ          ; SAVE THE POLE LOCATION IN A DISK FILE.BACKUP
*
*CODE POLE 1 INST<11      ; GENERATE 2920 ASSEMBLY CODE FOR THIS FILTER
B:=1 33989990  B2=-0.50541914
```

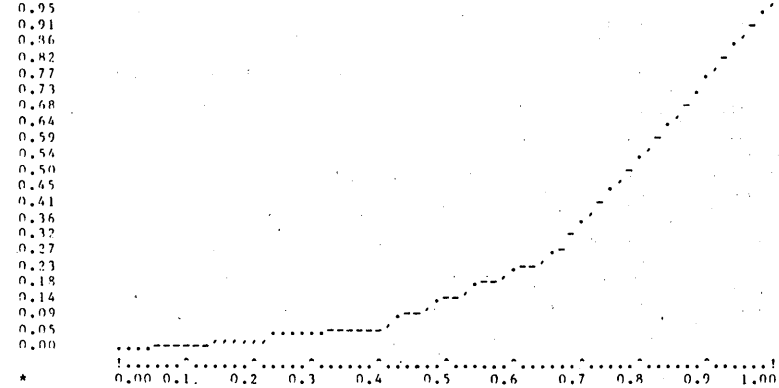
SAMPLE SPAS20 FILTER DESIGN SESSION (Cont'd.)

```
      OPTIMIZED 2920 CODE IS NOW GENERATED  TO SAVE SPACE, SOME
    OF THE SCREEN OUTPUT HAS BEEN DELETED  NORMALLY ALL ATTEMPTS
    BY THE COMPILER TO GENERATE CODE ARE ECHOED ON THE SCREEN. )

INST=10
 POLE 1 = 0.71089458,0.34116779,Z
  BEST: PERROR = 3.3795874/10**5,1.58846567/10**5

; NOTE: MAKE SURE SIGNAL IS <0 74635571
LDA OUT2_P1,OUT1_P1,R00
    ; OUT2_P1=1.00000000*OUT1_P1
LDA OUT1_P1,OUT0_P1,R00
    ; OUT1_P1=1.00000000*OUT0_P1
SUB OUT0_P1,OUT1_P1,R05
    ; OUT0_P1=1.00000000*OUT0_P1-0.031250000*OUT1_P1
ADD OUT0_P1,OUT0_P1,R03
    ; OUT0_P1=1.12500000*OUT0_P1-0.035156250*OUT1_P1
ADD OUT0_P1,OUT1_P1,R02
    ; OUT0_P1=1.12500000*OUT0_P1+0.21484375*OUT1_P1
SUB OUT0_P1,OUT2_P1,R01
    ; OUT0_P1=1.12500000*OUT0_P1+0.21484375*OUT1_P1-0.50000000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R08
    ; OUT0_P1=1.12500000*OUT0_P1+0.21484375*OUT1_P1-0.50390625*OUT2_P1
ADD OUT0_P1,OUT2_P1,R11
    ; OUT0_P1=1.12500000*OUT0_P1+0.21484375*OUT1_P1-0.50341796*OUT2_P1
SUB OUT0_P1,OUT2_P1,R09
    ; OUT0_P1=1.12500000*OUT0_P1+0.21484375*OUT1_P1-0.50537109*OUT2_P1
ADD OUT0_P1,INO_P1,R00
    ; OUT0_P1=1.12500000*OUT0_P1+0.21484375*OUT1_P1-0.50537109*OUT2_P1+1.00000000*INO_P1
*
*; THE CODE COMMAND SPECIFIED THAT THE POLE PAIR BE CODED IN LESS THAN 11
*; INSTRUCTIONS, SO 10 INSTRUCTIONS WERE GENERATED, WITH COMMENTS.
*; THE FINAL ERROR IN RADIUS AND ANGLE FOR THE POLE PAIR WAS OF THE
*; ORDER OF 1/10**5 AS INDICATED ABOVE IN PERROR.
*; THIS OPTIMIZED 2920 ASSEMBLY CODE CAN NOW BE APPENDED TO A FILE
*; WHICH MAY CONTAIN OTHER CODED FUNCTIONAL BLOCKS OF A 2920 PROGRAM
*
*EXIT
```

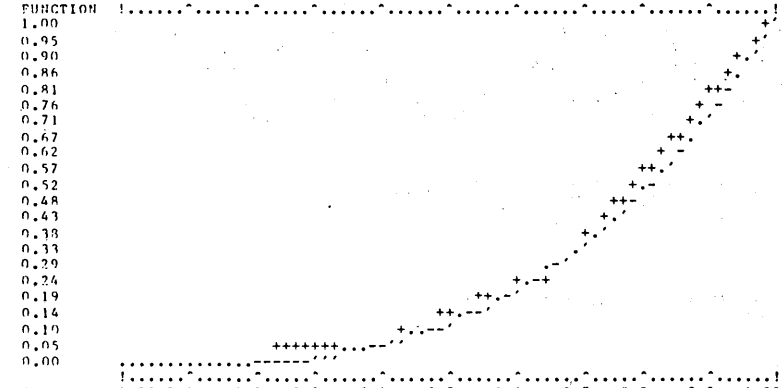
SAMPLE SPAS20 CURVE FITTING SESSION

```
-;        DEMONSTRATION OF THE SPAS20 CURVE-FITTING PACKAGE
-
-
-SPAS20.SFT

ISIS-II 2920 SIGNAL PROCESSING APPLICATIONS SOFTWARE/COMPILER, V2.0
*LIST XCUBED.829
*
*                                                                    3
*;  THE CURVE FITTING COMMANDS IN SPAS20 WILL GENERATE 2920 CODE TO CALCULATE
*;  SOME FUNCTION SUCH AS  Y**3.   Y**3 COULD BE COMPUTED ON THE 2920 CHIP
*;  WITH TWO MULTIPLIES USING ABOUT 18 INSTRUCTIONS AND THE DAR.  HOWEVER IT
*;  WOULD TIE UP THE DAR TOO LONG.  THE CODE GENERATED BY THE CURVE FITTING
*;  COMMANDS DOES NOT USE THE DAR.
*
*CODE FIT XSCUBED(X) = X**3 ERROR<.05   ;ERROR BOUND OF .05
*
*CODE     ; HERE IS THE CODE GENERATED.
LDA TEMP,X,R00
    ; TEMP=1.00000000*X
LDA XCUBED,X,R01
    ; XCUBED=0.50000000*X
ADD XCUBED,X,R06
    ; XCUBED=0.51562500*X
ADD TEMP,X,R01
    ; TEMP=0.50000000*X+1.00000000*TEMP
ADD XCUBED,TEMP,R05
    ; XCUBED=1.00000000*XCUBED+0.031250000*TEMP
SUB XCUBED,TEMP,R02
    ; XCUBED=1.00000000*XCUBED-0.21875000*TEMP
ADD TEMP,X,R00
    ; TEMP=1.00000000*X+1.00000000*TEMP
ADD XCUBED,TEMP,R08
    ; XCUBED=1.00000000*XCUBED+0.003906250*TEMP
SUB XCUBED,TEMP,R04
    ; XCUBED=1.00000000*XCUBED-0.058593750*TEMP
LDA XCUBED,XCUBED,L02
    ; XCUBED=4.0000000*XCUBED-0.234375000*TEMP
  *
  *INST     ; THE FUNCTION WAS CODED IN THIS MANY INSTRUCTIONS;
  INST = 10.0000000
  *
```

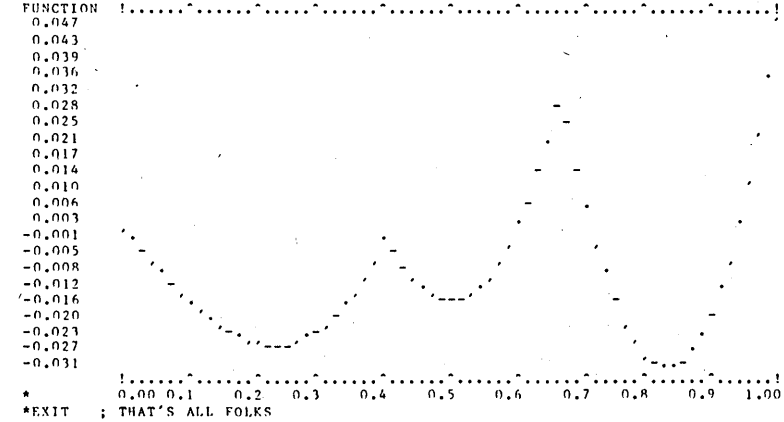
```
*ERROR    ; THE CODE APPROXIMATES X**3 WITHIN THIS ERROR;
ERROR = 0.046875000
*
*DATA 0 THRU 1   ;EXAMINE THE PIECEWISE LINEAR FUNCTIONS VERTICES.
DATA 0.00000000 THRU 1.00000000 = 0.00000000 AT 0.00000000,&
0.06562512 AT 0.40000000,&
0.29562500 AT 0.66666669,&
0.95312500 AT 1.00000000
*
*GRAPH DATA(X)   ;THE DATA ARRAY APPROXIMATES THE FUNCTION AND CAN BE GRAPHED.
```




```
*OGRAPH X**3    ; THE DIFFERANCE BETWEEN THE CODED AND THE ACTUAL APPEARS AS "+".
```




```
*GRA (X**3)-DATA(X)    ; THE ERROR WILL BE GRAPHED.
```




```
*EXIT    ; THAT'S ALL FOLKS
-
```

2920 ASSEMBLER

2920 program development on Intellec® Microcomputer Development Systems

Translates symbolic assembly language instructions into 2920 machine code

Produces Assembly Listing, Object Code File, and Error Diagnostics

Output used for 2920 programming with the Intellec PROM Programmer or the 2920 Simulator for program debug

The 2920 Assembler translates symbolic 2920 Assembly Language instructions into the appropriate machine operation codes. Through this facility, the programmer is able to symbolically program 2920 hardware operations. Compared to machine code, these symbolic references provide faster programming, easier debugging, and greater reliability.

The Assembler produces an object code file (executable machine code), a complete assembly listing, and error diagnostics. The object code output from the Assembler may be loaded directly into the Intel Universal PROM Programmer for programming the 2920 EPROM. The object code may also be loaded to the 2920 Simulator for 2920 system design and debug.

The 2920 Assembler runs under the ISIS-II Operating System on the Intellec Microcomputer Development Systems.

Sample 2920 Assembly Listing

```
ISIS-II 2920 ASSEMBLER X102                                    PAGE   1

ASSEMBLER INVOKED BY: AS2920 SAW ASM DEBUG

SAWTOOTH WAVE GENERATOR

  LINE  LOC OBJECT SOURCE STATEMENT

     1             $TITLE('SAWTOOTH WAVE GENERATOR')
     2             ;
     3             ;
     4     0 0000EF      IN0                ; SAMPLE INPUT CHANNEL 0
     5     1 0000EF      IN0
     6     2 0000EF      IN0
     7     3 008AEB      SUB Y,KP1,IN0      ; SIMULTANEOUSLY CALCULATE SAWTOOTH
     8     4 008A0A      SUB Y,KP1,R1,IN0   ; BY SUBTRACTING 3/16 FROM Y
     9     5 0044EF      LDA DAR,Y,IN0      ; ALSO CHECK SIGN BIT OF Y
    10     6 7A8AED      ADD Y,KP7,CMDS     ; IF Y NEGATIVE START NEXT TOOTH
    11     7 6000EF      CYTS               ; CONVERT SAMPLED INPUT TO DIGITAL (SIGN BIT)
    12     8 7082EF      LDA SAWTOOTH,CMDS  ; SUPPRESS SAWTOOTH IF INPUT WAS < 0
    13     9 4044EF      LDA DAR,Y          ; PREPARE TO OUTPUT SAWTOOTH
    14    10 4000EF      NOP                ; ANALOG LEVEL MUST SETTLE
    15    11 4000EF      NOP
    16    12 4000EF      NOP
    17    13 8000EF      OUT0               ; OUTPUT SAWTOOTH
    18    14 8000EF      OUT0
    19    15 8000EF      OUT0
    20    16 5000EF      EOP                ; PROGRAM WILL END IN THREE MORE INSTRUCTIONS
    21    17 8000EF      OUT0
    22    18 8000EF      OUT0
    23    19 8000EF      OUT0
    24
    25                   END

SYMBOL:                          VALUE:

Y                                    0


ASSEMBLY COMPLETE
ERRORS     =     0
WARNINGS   =     0
RAMSIZE    =     1
ROMSIZE    =    20
```

2920 SIMULATOR

Speeds test and debug of 2920 programs

Simulates 2920 internal operation

Operates on Intellec® Microcomputer Development Systems

Allows users to specify 2920 input signals, and display or alter ROM, RAM, and system variables

Output and internal data can be saved on disk for further analysis.

Provides ability to set breakpoints and to collect trace information

Easy-to-learn commands

The 2920 Simulator is a software facility that provides testing and symbolic debugging of 2920 programs in an Intellec Microcomputer Development Systems environment. The 2920 designers have the capability to specify the 2920 input signals, to set breakpoints, to collect and display 2920 input, output, system variables, and ROM and RAM data values during simulation. The 2920 Simulator accepts the hex format object files produced by the 2920 assembler. Output values and internal trace data may be saved on ISIS-II disk files for further analysis.

Functional Description

2920 Input Signal Specification

The four analog signal inputs to the 2920 processor can be specified as algebraic combinations of basic functions of time. The basic functions are SIN, COS, EXP, LOG, SQR, SAW, SQW, ABS.

2920 Simulation

The simulation of 2920 machine instructions is performed in software. All 2920 internal registers, memory, input values, output values, and other system variables can be examined and modified. The internal processing of the 2920 is simulated. Time constants for the sample and hold capacitators are assumed to be zero. Calculation of input signals is performed in single precision floating point. The speed of simulation varies with the complexity of the input signal, breakpoint setting, and trace condition. Exclusive of I/O time requirements, 2920 instructions will be simulated at a rate of approximately several hundred instructions per second.

Breakpoint Capabilities

After each instruction is simulated, the breakpoint is evaluated to determine whether to stop or continue simulation. Conditional breakpoints are also provided for debugging purposes. Simulation can be manually stopped at any time by pressing the ESC key on the Intellec console.

Trace Capabilities

Based on the qualifier's condition, trace data records can be collected during simulation. The trace data records are stored in Intellec resident memory and are optionally written to the console for display or to a disk file for record.

Symbolic Debugging Capabilities

The 2920 Simulator allows the user to refer to program addresses symbolically. The user can load or save the symbols generated from the hex format object files or created during the debugging session. 2920 program memory in ROM can be disassembled, or filled with assembled instructions.

The 2920 Simulator is designed to provide users with powerful, easy-to-use commands. The user interfaces to the Simulator by entering commands to the Intellec console. The commands consist of one command line, terminated by one of the two line terminators — carriage return or line feed.

The 2920 Simulator offers two types of commands:

Simulation and Control Commands

Command	Operation
Simulate	Starts simulation of the input signals and the 2920 program in simulated ROM memory. Initial setting is "FOREVER."
Trace	Controls the trace selection. Initial setting is "TIME."
Qualifier	Sets qualifier condition during trace. Initial setting is "ALWAYS."
Breakpoint	Sets breakpoint condition during simulation. Initial setting is "NEVER."

Interrogation and Utility Commands

Command	Operation
Display	Displays the values of symbols, RAM, ROM, input, output, registers and system variables.
Change	Alters the values of symbols, RAM, ROM, input, register and system variables.
Base	Establishes the mode of display for output data.
Suffix	Establishes the mode of display for input data.
Load	Fetches user symbol table and object code from input device.
Save	Sends user symbol table and object code to output device.
Define	Enters symbol name and value to user symbol table.
Console	Controls the console on/off display.
List	Defines list device.
Exit	Returns program control to ISIS-II.
Evaluate	Converts expression to equivalent values in binary, decimal, and hex.
Remove	Deletes symbols from symbol table.
Help	Provides a brief summary of the syntax for the command languages.
Graphics On/Off	Switches output mode between list and graphics.
X Size	Enters horizontal display size.

Keyword References

The 2920 Simulator provides users with keyword references to gain access to all of the numeric valued system variables including simulated 2920's memory, register, status flags and input/output. These keyword references can function as the evaluation command, display command, and change command.

- **2920 Processor Keyword References**

IN0	Analog input 0 in volts
IN1	Analog input 1 in volts
IN2	Analog input 2 in volts
IN3	Analog input 3 in volts
OUT0	Analog output 0 in volts (read only)
OUT1	Analog output 1 in volts (read only)
OUT2	Analog output 2 in volts (read only)
OUT3	Analog output 3 in volts (read only)
OUT4	Analog output 4 in volts (read only)
OUT5	Analog output 5 in volts (read only)
OUT6	Analog output 6 in volts (read only)
OUT7	Analog output 7 in volts (read only)
IN	Sampled and held analog input signal in volts
DAR	Digital to analog register (RAM location 40)
PC	Program counter (integer 1 to 192)
CY	Carry (integer 0 or 1)
OVF	Overflow (integer 0 or 1, read only)
OVE	Overflow enable (integer 0 or 1)

- **Software Simulator Keyword References**

TIME	Elapsed simulated time in seconds (read only)
TQUAL	Time when the qualifier last matched in seconds (read only)
COUNT	Number of instructions simulated since last SIMULATE command (integer, read only)
BUFFERSIZE	Number of trace data records (integer, read only)
TINST	Time between successive instructions in seconds (read only)
SIZE	Number of instructions in program disregarding actual EOP placement
TPROG	Time between successive program passes in seconds
VREF	Reference analog level voltage in volts

The above keyword references are designed to aid 2920 program debugging.

ISIS Compatibilities

The 2920 software simulator runs under the ISIS "submit" facility. The 2920 software simulator uses the ISIS-II line editing capabilities to correct errors in an input line on the Intellec console.

Sample 2920 Simulation Session

```
-
-SM2920.SFT

ISIS-II 2920 SIMULATOR, V1.1
*
*;   THIS IS THE SIMULATION OF THE 'SAWTOOTH GENERATOR'
*
*LIST SRG.LOG        ; LISTS THE SIMULATION SESSION TO AN ISIS FILE
*LOAD SRG.HEX        ; LOAD THE OBJECT CODE INTO THE 2920 SIMUALTOR
*ROM 0 TO 5          ; DISPLAY SRG PROGRAN
ROM 000 = LDA .K,KP5,R00,NOP
ROM 001 = ADD .K,KP1,R05,NOP
ROM 002 = LDA .K,.K,R02,NOP
ROM 003 = SUB .OSC,.K,R00,NOP
ROM 004 = LDA DAR,.OSC,R00,NOP
ROM 005 = ADD .OSC,KP4,L01,CNDS
*TPROG=1/10000       ; SET THE SAMPLE RATE
*TRA=PC,RAM .K       ; SET THE ITEMS TO BE TRACED
*BASE=B              ; DISPLAY THE RESULTS IN BINARY
*SIMULATE FROM 0 TILL COUNT=3 ; SIMULATE THREE INSTRUCTIONS
                               TO VERIFY CONSTANT


   PC                           RAM 0
SIMULATION BEGUN
    1.0000000000000000000000     0.1010000000000000000000000
    2.0000000E+0                 0.1010000100000000000000000
    3.0000000E+0                 0.0010100001000000000000000
SIMULATION TERMINATED
*QUALIFIER=PC=0      ; TRACE EVERY PROGRAM PASS
*TRACE=T,DAR,RAM .OSC ; SET THE ITEMS TO BE TRACED
*RAM .OSC=ONE        ; INITIALIZE THE RAM LOCATION
*BREAKPOINT=T>.000132 ; SIMULATE FOR TWO CYCLES
*BASE=D              ; SET THE BASE TO DECIMAL
*SIMULATE FROM 0     ; BEGIN SIMULATION
   T                  DAR                RAM 1
```

```
SIMULATION BEGUN
     0.00010000        0.83984375        0.84277334
     0.00020000        0.68359375        0.68554683
     0.00030000        0.52734375        0.52832026
     0.00040000        0.36718750        0.37109370
     0.00050000        0.21093750        0.21386714
     0.00060000        0.05468750        0.05664056
     0.00070000       -0.10156250        0.89941396
     0.00080000        0.73828125        0.74218745
     0.00090000        0.58203125        0.58496089
     0.00100000        0.42578125        0.42773433
     0.00110000        0.26953125        0.27050776
     0.00120000        0.10937500        0.11328119
     0.00130000       -0.04687500        0.95605459
SIMULATION TERMINATED
*GRAPH ON            ; SWITCHES THE DISPLAY MODE TO GRAPHICS
*TRACE=T,0,DAR,RAM .OSC,-1,-1,1,1  ; SETS ITEMS TO BE TRACED
*RAM .OSC=ONE        ; INITIALIZE THE RAM LOCATION
*SIMULATE FROM 0
   T          0                DAR              RAM I            -1
        -1              1                1
SIMULATION BEGUN
->*                                      1                                *     *
0 *                                      1                          *           *
. *                                      1                     *                *
0 *                                      1                *                     *
0 *                                      1           *                          *
0 *                                      1 *                                    *
1 *                                    2 1                                3     *
0 *                                      1                          *           *
0 *                                      1                     *                *
0 *                                      1                *                     *
0 *                                      1           *                          *
  *                                      1      *                           *
  *                                    2 1                                3 *
SIMULATION TERMINATED
*EXIT
-
```

SPECIFICATIONS

Operating Equipment

Required Hardware

Intellec® Microcomputer Development System
RUNNING ISIS

Required Software

ISIS-II Diskette Operating System

Optional Hardware

Line Printer
Universal PROM Programmer

Optional Software

FORTRAN-80 (Product Code MDS-301)

Documentation Package

2920 Assembly User's Guide (9800987)
2920 Simulator User's Guide (9800988)
2920 Signal Processing Application Compiler User's Guide (121529)

Shipping Media

Flexible Diskettes

ORDERING INFORMATION

Product Code	Description
MCI-20-SPS	2920 Software Support Package Includes 2920 Signal Processing Application Software/Compiler and 2920 Assembler/Similator Software

intel®

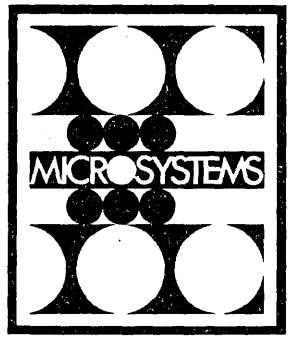
ARTICLE REPRINT

AR-59

June, 1978

Modular Programming in PL/M

William Brown
Microcomputer Systems Division



Modular Programming in PL/M*

William Brown
Intel Corporation

Various methodologies have been used to control the high—and rising—cost of developing software products. Among these, one technique that has proved effective entails constructing programs from small, well-defined modules. This technique, called modular programming, can be used in any programming language; however, without language support to enforce module boundaries, errors often occur.

The PL/M language and compiler are designed to bring the advantages of modular programming to microprocessor software systems. Since the fundamental PL/M language facility for organizing a program is the module, software systems can be partitioned into manageable units. The PL/M module can hold data and procedures and, if properly used, provide encapsulation of programming abstractions. In this way it is related to several other language mechanisms that provide for grouping operations logically related to a single data structure—for example, the Simula class,[1] the Alphard form,[2] the CLU cluster,[3] and the Mesa module.[4]

Modularity

The basic motivation for modularizing a software system is to divide the system into partitions understandable to the implementer. There are many techniques for designing a partitioning. The oldest one applies a functional decomposition of the system into subroutines or procedures. However, in truly large systems, such decomposition usually results in a large number of procedures which, though easily understood, have complex interdependencies.

Encapsulation. Another technique, suggested by Parnas,[5] is based on encapsulation of information. A software system is partitioned in terms of the abstractions which make it most understandable. Thus, a text editor might be expressed as manipulations of strings or a logic simulation as a structure of logic cells. By encapsulating, or hiding, the implementation details of the abstraction, interdependencies are limited to the properties of the abstraction (for example, concatenate, find, etc., for strings, or inputs and outputs for logic cells). Thus, the system is more understandable.

Hiding information also enhances the long-term utility of the system by making programs easier to maintain and modify. First, the source text is encapsulated so that any program changes are localized. Second, if the engineering requirements of the system change, the implementation of the abstraction can be replaced without affecting any other part of the system. For example, the implementation of logic cells might initially be optimized for minimum memory-space requirements. Later, if speed becomes important, the implementation can be replaced by one optimized for speed.

PL/M modules share two aspects of encapsulation with the facilities of Alphard, CLU, and Mesa. First, the module localizes the source text which implements the abstraction. Second, the module hides implementation details. It thereby provides a certain amount of protection.

The PL/M system

This description of the PL/M language and the software development environment concentrates on those features important to modular programming. It is intended to provide enough background so that someone familiar with similar languages and systems can understand the examples. For further information, Intel's *PL/M-80 Programming Manual*[6] provides a complete description of the language, and McCracken[7] provides a tutorial introduction to

* Adapted from a paper presented at COMPSAC 77, Chicago.

0018-9162/78/0300-0040$00.75 © 1978 IEEE

PL/M and the ISIS-II diskette operating system. Intel's *ISIS-II System User's Guide*[8] describes the file management services and general facilities of this operating system.

PL/M is a block-structured procedural language. It is intended as a system implementation language for the Intel 8080 microprocessor. Syntactically, it closely resembles XPL[9] or PL/I.[10] However, the statement structure should be understandable to anyone familiar with a block-structured language.

The data types which PL/M manipulates are probably not familiar to some readers. PL/M has only two basic data types: BYTE and ADDRESS. A BYTE is an 8-bit unsigned value. An ADDRESS is a 16-bit unsigned value. In addition to these data types, PL/M allows singly dimensioned arrays and single-level data structures.

An example declaration for a BYTE variable (CH) and two ADDRESS variables (*B*1 and *B*2) is given below:

```
DECLARE CH BYTE,
        (B1, B2) ADDRESS;
```

PL/M takes a primitive approach to the problems presented by references to objects. A reference to an object is simply the memory address of the object. PL/M uses a dot to denote the operation "address of." Thus, ".CH" yields the address of "CH."

PL/M also allows for accessing variables by their references. This is provided by the BASED notation in declarations. For example, with the declaration

```
DECLARE B ADDRESS,
        CH BASED B BYTE,
        N          BYTE;
```

and the assignments

```
B = .N;
CH = 5;
```

the value of *N* is 5.

The BASED variable concept is important to the procedure mechanism. Only objects of type BYTE or ADDRESS may be passed to a procedure and all parameters are passed by value. Therefore, to pass a large object like an array or to implement a return parameter requires a BASED declaration. In this fashion, PL/M implements call by reference.

The last facility to be discussed is the LITERALLY declaration. A LITERALLY defines a parameterless macro or string substitution in the source text. Thus, with the declaration

```
DECLARE ZERO LITERALLY '0';
```

the appearance of the identifier ZERO is equivalent to writing the constant 0.

PL/M modules. A module is a labeled block which is not enclosed in any other block. Data objects and procedures can be declared in the module, and in one distinguished module (the main program module) an executable statement sequence may appear. Since a module is a block, names declared in it are normally limited to the extent of the block. Thus, all objects are *a priori* hidden inside the module. However, PL/M's PUBLIC and EXTERNAL attributes provide mechanisms to make names in one module explicitly visible in another. (This formulation parallels the Mesa facilities.)

A procedure or data object in a module may be given the PUBLIC attribute. This makes the name of the object visible outside the module. Only objects declared at the first nesting level may be declared PUBLIC. This restriction, and the fact that modules are statically allocated, assures that PUBLIC procedures have a consistent environment for efficient execution.

A module may access PUBLIC information in another module by including a matching EXTERNAL declaration. For a procedure, the EXTERNAL declaration appears as a procedure with only parameter declarations in the body. The attribute EXTERNAL appears as the last item in the procedure head. For data, PUBLIC or EXTERNAL appears as an attribute in the declarations. For example, the declaration

```
DECLARE NAMEREC STRUCTURE(
                LAST    (25) BYTE,
                FIRST   (25) BYTE,
                MI           BYTE) PUBLIC;
```

declares a structure variable, NAMEREC, which has three fields. The fields LAST and FIRST are arrays of 25 BYTES. The field MI is a single BYTE. The matching EXTERNAL declaration is

```
DECLARE NAMEREC STRUCTURE(
                LAST    (25) BYTE,
                FIRST   (25) BYTE,
                MI           BYTE) EXTERNAL;
```

The names of structure fields and procedure parameters in EXTERNAL declarations need not match those in the PUBLIC declaration. Only the types and order must match.

The compiler and linkage system. The current PL/M compiler has two features which are important to implementing modular abstractions. First, the module is the natural unit of compilation. Thus, an implementation of an abstraction can be compiled once and then used for many applications. Second, the compiler supports a textual inclusion facility. This facility is provided by a compiler control having the following general form

```
$INCLUDE (filename)
```

The compiler will read the file given by the filename. The text read will be inserted into the source program, replacing the INCLUDE control. The

EXTERNAL and LITERALLY declarations for a module may be included this way. Thus, an abstraction may be referenced by a single name. Textual inclusion is the mechanism used by Mesa for static binding of implementations of an abstraction to users of the abstraction.

The linkage system is responsible for binding modules together. It matches all EXTERNAL declarations to the appropriate PUBLIC declarations. Unfortunately, this matching is done by name only. No type checking is performed.

Example abstraction—strings. The abstraction to be implemented is that of variable-length character strings. The abstraction has the following operations: LENGTH, COPY, CONCAT, FRONT, REST, FIND, BLANKS, PUT, and GET. It is possible to define each of these operations in precise mathematical terms. However, for the purpose of this example, only informal descriptions with a minimum of formal notation are given. Where a functional notation is necessary, S will represent a string and N will represent a non-negative integer.

LENGTH returns the number of characters in the argument string. The empty string has a length of zero.

COPY returns a duplicate of the argument string.

CONCAT returns a string which is a concatenation of its arguments. The order of concatenation is the first argument string followed by the second. The two argument strings are not affected.

FRONT returns a string which is a copy of the first N characters of the argument string. The value of N must be in the inclusive range from 0 to the length of the string. If N is zero an empty string is returned.

REST returns a string such that CONCAT (FRONT(S,N), REST(S,N)) is a copy of the string S.

FIND locates a character in the argument string and returns the length of the substring ended by that character. If the character is not in the string, zero is returned.

BLANKS returns a string of blanks of a specified length. BLANKS (0) returns an empty string.

PUT outputs a string as a line on a specified file.

GET inputs a line from a specified file and converts it to a string.

```
Declare Ref$String Literally 'Address'
        Character  Literally 'Byte';

Length:
  Procedure (Ref) Address External;
    Declare Ref Ref$String;
  End Length;

Blanks:
  Procedure (N) Ref$String External;
    Declare N Address;
  End Blanks;

Copy:
  Procedure (Ref) Ref$String External;
    Declare Ref Ref$String;
  End Copy;

Concat:
  Procedure (Ref1, Ref2) Ref$String External;
    Declare (Ref1, Ref2) Ref$String;
  End Concat;

Front:
  Procedure (Ref, Ind) Ref$String External;
    Declare Ref Ref$String,
            Ind Address;
  End Front;

Rest:
  Procedure (Ref, Ind) Ref$String External;
    Declare Ref Ref$String,
            Ind Address;
  End Rest;

Find:
  Procedure (Ref, Ch) Address External;
    Declare Ref Ref$String,
            Ch  Character;
  End Find;

Put:
  Procedure (Ref, Fl) External;
    Declare Ref Ref$String,
            Fl Address;
  End Put;

Get:
  Procedure (Fl) Ref$String External;
    Declare Fl Address;
  End Get;

Delete:
  Procedure (Ref) External;
    Declare Ref Ref$String;
  End Delete;
```

Figure 1. The user's view of strings defined by external declarations.

The implementation

Before implementing the string abstraction, concrete PL/M interfaces for the abstract operations must be specified. Figure 1 contains the EXTERNAL and LITERALLY declarations which define strings to the user. These declarations correspond to a definition module in Mesa or the specification part of an Alphard form. To produce these declarations two implementation details had to be fixed.

First, since PL/M allows only scalar parameters, the concept of "references to a string" has been introduced. The LITERALLY declaration defines REF$STRING as ADDRESS. This does not imply, however, that a reference to a string is necessarily the memory address of the representation. The actual representation of the object is hidden by the module structure. This LITERALLY provides for visually distinguishing declarations of string references from other variables of type ADDRESS. However, the language does not enforce any distinction.

Second, an additional operation, DELETE, has been specified. The abstraction was not concerned with the problem of dynamic storage management. It is possible to implement strings with implicit

storage management. However, that would complicate the representation. Therefore, the user is responsible for deleting unused strings.

Representation. The user's view of strings is defined by the declarations in Figure 1. These declarations do not imply anything about the represeentations of strings or string references; the module structure is used to hide these details. Several alternatives are possible. A string might be represented as a linked list of characters or as a dynamically allocated BYTE array. String references might be the address of the string representation or an index into a hidden array maintained by the module.

The representation chosen implements a string reference as the address of a dynamically allocated BYTE array. However, to illustrate encapsulation and the effect of engineering decisions on an implementation, two forms of this representation are supported. For strings of less than 255 characters, the first entry in the dynamic array is the length of the string. Thus, short strings are handled efficiently in minimum space. For strings of 255 or more characters, the first entry in the dynamic array is 255 and the end of the string is indicated by another 255. Thus, long strings pay a slight penalty in both space and time. If a more efficient representation for long strings is required, the representations can be changed without impacting the user of the abstraction.

Completed module. The source text for the completed module to implement strings is in the appendix. This module corresponds to a program module in Mesa or the representation and implementation parts of an Alphard form. The implementation is not completely representative of good software development in that the source text is not adequately documented and it has been validated only to the extent necessary to run the example.

Notice that the STRINGS module accesses two other abstractions by INCLUDE. The first of these provides EXTERNAL declarations for the ISIS-II input/output facilities, described in the user's guide.* The second abstraction, referenced by the file name MEMMAN.DEF, provides for dynamic storage management. This module contains two operations, ALLOC and DEALLOC, which allocate and deallocate contiguous blocks of memory.

The module contains several useful LITERALLY declarations. In addition to REF$STRING and CHARACTER declarations, the type STRING is declared literally. Since this type is always applied to BASED items, the array length specifier of 1 is only a formality.

The procedure NEW is hidden inside the STRINGS module. It takes as a parameter the length of a string to be created and allocates space for the appropriate representation type. It also initializes the length or boundary markers.

The PUBLIC procedure LENGTH defines the length operation. It is typical of the procedures implementing the operations. The first line names the procedure and formal parameter, and the word ADDRESS indicates this is a function returning an ADDRESS value. The word PUBLIC indicates the procedure is to be accessible outside the module. Next comes the declaration of the parameter and two local variables. The first is a STRING based on the reference parameter. The second is a counter for a loop. The body of the LENGTH procedure follows.

The remaining procedures follow the same pattern. However, two points should be mentioned. First, several procedures call MOVE, a built-in PL/M procedure for moving bytes from one memory area to another. Second, the DELETE procedure does not free all the storage for unused strings. The length of the string is set to zero and the remaining storage is freed. This action helps avoid problems arising from inadvertently referencing a deleted string. It is, of course, hidden from the user of the abstraction.

Example program. Figure 2 shows a program using the string abstraction. The input to this program is a text file, TEST.SRC, containing tab characters. Tabs are represented in the text by the character '/'. The program processes the file and outputs the text file TEST.OUT. The output has the tab characters replaced by enough blanks to implement tab stops at columns 8, 16, 24, 32, etc.

The INCLUDES of the files IO.DEF and STRING.DEF at the beginning of the program supply the EXTERNAL declarations for the abstractions. The text of STRING.DEF is exactly that given in Figure 1. The text of IO.DEF is described in the discussion of the module STRINGS.

Next is the procedure declaration for CONCATD. This declaration provides a local extension to the string abstraction. It implements a concatenation operation which deletes the argument strings. Note that this extension is defined in terms of the operations of the string abstraction, and not in terms of the actual representation. Thus, the encapsulation of the implementation is preserved.

Following the procedure declaration are the declarations for the variables used by the program. The variables LINE and OUTLINE are references to the input string and output string, respectively. The rest of the variables are various temporaries and counters.

The body of the algorithm is an iteration which terminates when a null string is encountered. Each LINE is processed in turn until all tabs have been found. When a tab is found (by FIND), all the characters in the line in front of the tab are concatenated to the output string (referenced by OUTLINE). Next, the length of this new string is determined and the proper number of blanks to be inserted is calculated (as LB). This number of blanks is concatenated to the output string. Finally, the original string LINE is replaced by the REST of the string and a new tab is located.

When no more tabs are found, the remaining part of the input string is concatenated to the output string. This string is output. A new LINE is input and the outer iteration is repeated.

```
Tabs:Do;

$Include (String.Def)
$Include (Io.Def)

  Concatd:
    Procedure (Ref1,Ref2) Ref$String;
      Declare (Ref1,Ref2,Retref) Ref$String;
        Retref = Concat (Ref1,Ref2);
        Call Delete (Ref1);
        Call Delete (Ref2);
        Return Retref;
    End Concatd;

  Declare (Line,Outline,Tmp) Ref$String,
          (I,L,Lb) Address,
          (Infile,Outfile,Status) Address;

  Declare Tab Literally ''''/'''';

  Call Open
    (.Infile,.('TEST.SRC '),1,256,.Status);
  Call Open
    (.Outfile,.('TEST.OUT '),2,0,.Status);

  Line = Get(Infile);
  Do While Length(Line) <> 0;
    Outline = Blanks(0);
    I = Find(Line,Tab);
    Do While I <> 0;
      Outline =
        Concatd(Outline,Front(Line,I-1));
      L = Length(Outline);
      Lb = (((L/8)+1)*8)-(L+1);
      Outline =
        Concatd(Outline,Blanks(Lb));
      Tmp = Line;
      Line = Rest(Line,I);
      Call Delete(Tmp);
      I = Find(Line,Tab);
    End;
    Outline = Concatd(Outline,Line);
    Call Put(Outline,Outfile);
    Call Delete(Outline);
    Line = Get(Infile);
  End;

  Call Exit;
End Tabs;
```

Figure 2. Example program using the string abstraction.

Figure 3 shows an input file and the corresponding output file. The output was obtained by supplying a reasonable implementation of the memory management module and executing the TABS program.

```
count/amount/total
25/$.25/$6.25
5/$.42/$2.10
7/$3.20/$22.40

count    amount    total
25       $.25      $6.25
5        $.42      $2.10
7        $3.20     $22.40
```

Figure 3. Input file with the corresponding output file.

Conclusion

As the example program shows, the PL/M module is a simple, efficient encapsulation mechanism that can emulate many of the abstraction facilities of Alphard, Mesa, and CLU. Thus, a number of benefits inherent in such languages, including better readibility and maintainability, are available to the PL/M programmer. Discipline is required, however, since existing implementations of PL/M—unlike those of the other languages—do not check for consistent use of abstractions.

The language facilities and methodology exemplified by the STRINGS module can be successfully applied to real software products. They have been used, for example, in constructing the foundation of Intel's RMX-80 real-time operating system which coordinates programs performing real-time control functions.[11] ■

Acknowledgments

I wish to thank Kevin Kahn and John Doerr for their many comments and suggestions during the writing of this article.

Appendix. Source text for the completed module which implements the strings example.

```
Strings:Do;

$Include (Io.Def)
$Include (Memman.Def)

  Declare  Ref$String  Literally 'Address',
           String      Literally '(1) Byte',
           Character   Literally 'Byte',
           Cr          Literally '13',
           Lf          Literally '10';

  New:
    Procedure (Ln) Ref$String;
      Declare Ln Address,
              Retref Ref$String,
              Str Based Retref String;

      If Ln >= 255 Then Do;
        Retref = Alloc(Ln+2);
        Str (0),Str(Ln+1) = 255;
      End; Else Do;
        Retref = Alloc(Ln+1);
        Str (0) = Ln;
      End;
      Return Retref;
    End New;

  Length:
    Procedure (Ref) Address Public;
      Declare Ref Ref$String,
              Str Based Ref String,
              I Address;

      If Str (0) < 255 Then Return Str (0);
      I = 1;
      Do While Str (I) <> 255;
        I = I+1;
      End;
      Return (I-1);
  End Length;
```

```
Blanks:
  Procedure (N) Ref$String Public;
    Declare (N,I) Address,
            Retref Ref$String,
            Str Based Retref String;

      Retref = New(N);
      If N <> 0 Then
        Do I = 1 To N;
          Str(I) = ' ';
        End;
      Return Retref;
End Blanks;

Copy:
  Procedure (Ref) Ref$String Public;
    Declare (Ref,Retref) Ref$String,
            Ln Address;

      Ln = Length(Ref);
      Retref = New(Ln);
      If LN <> 0 Then
        Call Move(Ln,Ref+1,Retref+1);
      Return Retref;
  End Copy;

Concat:
  Procedure (Ref1,Ref2) Ref$String Public;
    Declare (Ref1,Ref2,Retref) Ref$String,
            (Ln1,Ln2) Address;

      Ln1 = Length(Ref1);
      Ln2 = Length(Ref2);
      Retref = New(Ln1+Ln2);
      If Ln1 <> 0 Then
        Call Move(Ln1,Ref1+1,Retref+1);
      If Ln2 <> 0 Then
        Call Move(Ln2,Ref2+1,Retref+Ln1+1);
        Return Retref;
  End Concat;

Front:
  Procedure (Ref,Ind) Ref$String Public;
    Declare (Ref,Retref) Ref$String,
            Ind Address;

      Retref = New(Ind);
      If Ind <> 0 Then
          Call Move(Ind,Ref+1,Retref+1);
      Return Retref;
  End Front;

Rest:
Procedure (Ref,Ind) Ref$String Public;
    Declare (Ref,Retref) Ref$String,
            (Ln,RestIn,Ind) Address;

      Ln = Length(Ref);
      RestIn = Ln-Ind;
      Retref = New(RestIn);
      If RestIn <> 0 Then
        Call Move(RestIn,Ref + Ind + 1,Retref +1);
      Return Retref;
  End Rest;

Find:
  Procedure (Ref,Ch) Address Public;
    Declare Ref Ref$String,
            Str Based Ref String,
            Ch Character,
            (Ln,I) Address;

      Ln = Length(Ref);
      If Ln = 0 Then Return 0;
      I = 1;
      Do While I <= Ln and Str (I) <> Ch;
        I = I + 1;
      End;
      If Str (I) = Ch Then Return I;
      Return 0;
  End Find;

 Put:
    Procedure (Ref,Fl) Public;
      Declare Ref Ref$String,
              (Fl,Ln,Status) Address;

      Ln = Length(Ref);
      If Ln <> 0 Then
        Call Write(Fl,Ref + 1,Ln,.Status);
      Call Write(Fl, (Cr,Lf),2,.Status);
    End Put;
```



```
Get:
  Procedure (Fl) Ref$String Public;
    Declare Retref Ref$String,
            (Fl,Actual,Status) Address,
            Buffer(128) Byte;

      Call Read
        (Fl, .Buffer,128,.Actual,.Status);
      If Actual = 0 then Return New(0);
      Retref = New(Actual-2);
      Call Move(Actual-2,.Buffer,Retref + 1);
      Return Retref;
  End Get;

  Delete:
    Procedure (Ref) Public;
      Declare Ref Ref$String,
              Str Based Ref$String;

        Call Dealloc(Ref + 1,Length(Ref));
        Str (0) = 0;
    End Delete;

End Strings;
```

References

1. O. J. Dahl, B. Myhrhaug, and K. Nygaard, *The SIMULA 67 Common Base Language*, Publication S-22, Norwegian Computing Center, Oslo, 1970.

2. A. Wulf, "ALPHARD: Toward a Language to Support Structured Programming," Carnegie-Mellon University Tech Report AD-785417, April 1974.

3. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices*, Vol. 9, No. 4, April 1974, pp. 50-59.

4. C. M. Geschke, J. H. Morris, Jr., and E. H. Satterthwaite, "Early Experience with Mesa," *CACM*, Vol. 20, No. 8, August 1977, pp. 540-552.

5. D. Parnas, "A Technique for Software Module Specification," *CACM*, Vol. 15, No. 5, May 1972, pp. 330-336.

6. Intel Corp., *PL/M-80 Programming Manual*, Document No. 98-268B, 1977.

7. D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*, Addison-Wesley Publishing Co., Reading, Mass., 1978.

8. Intel Corporation, *ISIS-II System User's Guide*, Document No. 98-306A, 1976.

9. W. M. McKeeman, J. J. Horning, and D. B. Wortmann, *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

10. ANS Committee X3, *Draft Proposed Standard Programming Language PL/I*, February 1975.

11. Kevin Kahn, "A Small-Scale Operating System Foundation for Microprocessor Applications," *Proc. IEEE*, Vol. 66, No. 2, February 1978, pp. 75-89.

William L. Brown is a senior software engineer in Intel's Microcomputer Systems Division in Aloha, Oregon. His past work includes the revision and enhancement of the PL/M language and the development support software for Intel's bit slice processor. He received his MEE from Rice University in 1974. He is currently an active member of the ACM and the IEEE Computer Society.

intel

ARTICLE
REPRINT

AR-136

June 1980

PL/M-86 Combines Hardware Access With High-Level Language Features

Mary Jane Elmore
Electronic Design, April 26, 1980

ORDER NUMBER:451145-001

PL/M-86 combines hardware access with high-level language features

PL/M-86, a systems-implementation language, is the first high-level language (HLL) designed specifically for the special requirements of microcomputers. The user gets not only high-level access to the μP hardware, and thus control over the processor and its peripheral components, but also such HLL advantages as the ability to write code in English-like statements, more efficient software design and easier debugging and maintenance. Major features include:

- High-level constructs for machine control, especially interrupt handling, direct-port I/O and access to absolute memory locations
- Pointers and based variables
- String manipulation
- LOCKSET, a procedure for multiprocessing environments.

Designed to be executed by Intel's 16-bit 8086 (ELECTRONIC DESIGN, March 1, 1980, p. 97), PL/M-86 is upward-compatible with PL/M-80. Except for interrupts, hardware flags and time-critical code sequences, PL/M-80 programs may be recompiled under PL/M-86 with little or no conversion.

Block-structured language

Both versions are block-structured, encouraging a structured approach to programing with well-structured branching and control statements. They provide a DO-END construct for simple block structures, as well as DO WHILE, DO CASE, an iterative DO, binary decision mechanisms IF-THEN-ELSE and nested IF-THEN-ELSE.

PL/M-86 procedures isolate well-defined tasks where local variables, valid only within their procedure, can be used to avoid unwanted interactions between procedures (Fig. 1). By making it easy to divide the programming tasks into subtasks, PL/M-86 encourages top-down design and permits several software designers to work in parallel. Since programs under development tend to keep changing, modularity also simplifies program maintenance. With PL/M-86, programs can be designed in such a way that one program function can be modified without unexpected repercussions elsewhere in the program.

In addition, as an SIL, PL/M-86 includes special features for writing systems software: I/O handlers, device drivers, system monitors—in short, any executive program that directly controls hardware, even if imbedded in application software (for instance, in machine or instrument control).

An SIL like PL/M-86 allows the system designer to control hardware with HLL constructs rather than error-prone assembly language. Specifically, the system designer can write interrupt-handling routines and routines to input or output data directly to CPU ports. PL/M-86 also allows the programmer to access memory locations directly and provides a flexible means of manipulating data and procedure pointers. Built-in procedures give access to the hardware stack pointer and CPU flags.

Unlike application-oriented languages, PL/M-86

```
M:    DO;/*Beginning of module*/
      DECLARE RECORD (128) STRUCTURE (KEY BYTE, INFO WORD);
      DECLARE CURRENT STRUCTURE (KEY BYTE, INFO WORD);
      DECLARE (J, I) INTEGER;
      /*Data are read in to initialize the records */
SORT:     DO J = 1 TO 127;
          CURRENT.KEY = RECORD(K).KEY;
          CURRENT.INFO = RECORD(J).INFO;
          I = J;
FIND:     DO WHILE I>0 AND RECORD(I-1).KEY>CURRENT.KEY;
              RECORD(I).KEY = RECORD(I-1).KEY;
              RECORD(I).INFO = RECORD(I-1).INFO;
              I = I-1;
          END FIND;
          RECORD(I).KEY = CURRENT.KEY;
          RECORD(I).INFO = CURRENT.INFO;
      END SORT;
      /*Data are written out from the records */
      END M;          /*End of module*/
```

1. Three nested blocks illustrate block hierarchy: Block M includes the whole screened area; block Sort includes all the code with medium and light screen; block Find is outlined by the white area only.

lets the programmer interface directly with the system hardware, without having to bring additional modules in at execution time to interface with the hardware. While Pascal or Fortran requires an operating system or run-time support to perform system-level functions, PL/M-86's "bare-machine" programming saves memory, as the code overhead for an operating or run-time system is eliminated. This SIL thus offers the best of two worlds—the memory efficiency of system-level code and the programming efficiency of an HLL.

```
                    .
                    .
                    .
HITEMP: PROCEDURE  INTERRUPT(5);
        DECLARE  INTERRUPT$ID        BYTE,
                 INDEX, OUTDEX       BYTE,
                 CURRENT$STATUS      WORD;
                    .
                    .
        INTERRUPT$ID = INPUT(INDEX);

        IF INTERRUPT$ID = 00000001B  THEN
        DO;
           OUTPUT(OUTDEX) = 11000000B  /*ALARM AND SHUTDOWN*/
           OUTDEX = OUTDEX + 1
           GO$FLAG + FALSE
        END;
        IF INTERRUPT$ID + 00001000B  THEN
        DO;
           OUTPUT(OUTDEX) = 10000000B  /*WARNING LIGHT*/
           OUTDEX = OUTDEX +1
        END;
        ELSE DO;
                    .
                    .
                    .
        END;

END HITEMP;
```

2. Although a high-level language, PL/M-86 provides direct access to hardware. In this example, a peripheral signals INTERRUPT(5) whenever a certain temperature exceeds its limit. The shown interrupt procedure activates warning signals and stops the process.

```
SORT:  DO J = 1 TO COUNT-1;
       CALL MOVB (@RECORD(J*RECSIZE),@CURRENT,RECSIZE);
       I=J;

FIND:       DO WHILE 1     0
                                AND RECORD (1-1)*RECSIZE + KEY
                                  CURRENT(KEY);
                CALLMOVB(@RECORD(1-1)*RECSIZE),
                                @RECORD(1*RECSIZE),
                                RECSIZE;
                I=1-1;
            END FIND;

            CALL MOVB(@CURRENT,@RECORD(1*RECSIZE),RECSIZE);
    END SORT;
```

3. In this fragment from a SORT routine, the predefined procedure MOVB is called several times. Being a built-in procedure, it does not have to be declared. In the first call (highlighted), the parameter @RECORD(J*RECSIZE) specifies the starting address of the byte sequence to be copied; @CURRENT is the location to which the first byte will be copied; RECSIZE is the number of bytes in the stream of data to be transferred.

Interrupts make it possible to break into the execution sequence of a running program to carry out other tasks and then resume execution of the interrupted program. Sometimes, the external event is repetitive —for instance, a clock pulse that only needs to be counted before other processing resumes. At other times, the external event can be a signal indicating that data are ready to be input or that some process has exceeded allowable limits.

Since μC applications involve processing of interrupts to some degree, an SIL must include provisions for interrupt-handling routines. In an 8086-based system, an interrupt may be generated by some peripheral device that sends an interrupt signal and number to the 8086 CPU (Fig. 2).

The CPU processes an interrupt by:

- Completing the machine instruction currently under execution
- Disabling the interrupt mechanism
- Activating an interrupt procedure corresponding to the number sent by the peripheral device.

After executing a RETURN or END statement, the interrupt procedure automatically reenables the interrupt mechanism and returns control to the point where the interrupt occurred.

For I/O operations, PL/M-86 provides built-in procedures that let the programmer access the CPU's I/O ports directly. This includes support for byte or word I/O and constant or variable port numbers. To input a byte from an 8086 I/O port, use

INPUT (expression)

The value of "expression" specifies one of the input ports of the 8086 CPU. The value returned by INPUT is the byte value found in the specified input port (see Fig. 2).

To access specific memory locations, PL/M-86 provides the AT attribute:

AT (location)

where "location" may be either a whole-number constant in the range of 0 through 1,048,575 or a location reference. The latter uses the "@ operator" to indicate where a specific variable will reside at execution time. For example, @RESULT represents the run-time location of the variable RESULT. The statement

DECLARE (CHAR$A, CHAR$B) BYTE AT (4096);

causes the BYTE variable CHAR$A to be stored at location 4096. The variable CHAR$B follows in the next two bytes.

On the other hand, the construct

```
DECLARE DATUM WORD
DECLARE ITEM BYTE AT (@DATUM)
```

causes ITEM to be declared a BYTE variable, located at the location of DATUM. PL/M-86's ability to access absolute memory locations is especially important for memory-mapped I/O or other hard-wired memory locations.

What are based variables?

Sometimes a direct reference to a variable is either impossible or inconvenient—for example, when the location of a data element remains unknown until it is computed at run time. It may then be necessary to manipulate the locations of data elements rather than the data elements themselves. PL/M-86 provides this indirect form of reference with "based variables." The base of a based variable is another variable pointing to the based variable. Both must be declared separately, with the base coming first. For instance, in

```
DECLARE ITEM$PTR POINTER;
DECLARE ITEM BASED ITEM$PTR BYTE;
```

ITEM$PTR is base and ITEM is the based variable. The construct

```
ITEM$PTR=34AH;
ITEM     =  77H;
```

loads the value 77 (hex) into the memory location 34A (hex).

One variable name can refer to many different data

```
                    DECLARE  B BYTE, C CBYTE,
                      TEST  BYTE,
                      A WORD;
                    IF  TEST  THEN
                    DO;
                      OUTWARD (0F6H)-0FFFFH;
                      A=B
                    END;
                    ELSE A=C
```

```
1.          MOV     AL,TEST
2.          RCR     AL,1
3.          JB      @1
4.          JMP     @2

5.    @1:   MOV     AX,0FFFFH
6.          OUTW    0F6H

7.          MOV     AL,B
8.          MOV     AH,0H
9.          MOV     A,AX
10.         JMP     @3

11.   @2:   MOV     AL,C
12.         MOV     AH,0H
13.         MOV     A,AX

14.   @3:
```

(a)

```
1.          MOV     AL,TEST
2.          RCR     AL,1
3.          JB      @1
4.          JMP     @2

5.    @1:   MOV     AX,0FFFFH
6.          OUTW    0F6H
7.          MOV     AL,B
8.          JMP     @4

9.    @2:   MOV     AL,C
10.   @4:   MOV     AH,0H
11.         MOV     A,AX

12.   @3:
```

(c)

```
1.          MOV     AL,TEST
2.          RCR     AL,1
3.          JB      @1
4.          JMP     @2

5.    @1:   MOV     AX,0FFFFH
6.          OUTW    0F6H
7           MOV     AL,B
8.          JMP     @4
9.          MOV     AH,0H
10.         MOV     A,AX
11.         JMP     @3

12.   @2:   MOV     AL,C
13.   @4:   MOV     AH,0H
14.         MOV     A,AX

15.   @3:
```

(b)

```
1.          MOV     AL,TEST
2.          RCR     AL,1
3.          JNB     @2

4.          MOV     AX,0FFFFH
5.          OUTW    0F6H
6           MOV     AL,B
7.          JMP     @4

8.    @2:   MOV     AL,C
9.    @4:   MOV     AH,0H
10.         MOV     A,AX

11.   @3:
```

(d)

4. An ASM86 program—before optimization (a), after cross-jumping (b), after elimination of unreachable code (c) and after reversing a branch condition (d).

items depending on the value of the base. For instance, the loop

```
.
.
TOTAL = 0;
DO ITEM$PTR = 2100H to 2199H;
TOTAL = TOTAL + ITEM
END;
.
.
```

places in TOTAL the sum of the 256 bytes found in memory locations 2100H through 2199H.

Based variables are even more powerful when the "@ operator" is used to supply values for bases. For example, suppose there are three different real variables, A$ERROR, B$ERROR, and C$ERROR, which should be accessible at different times via the single identifier ERROR. This can be done as follows:

```
DECLARE (A$ERROR, B$ERROR, C$ERROR) REAL;
DECLARE ERROR$PTR POINTER;
DECLARE ERROR BASED ERROR$PTR REAL;
ERROR$PTR = @A$ERROR;
```

At this point, the value of ERROR$PTR is the location of address A$ERROR. A reference to ERROR is, in effect, a reference to A$ERROR. Later in the program, the statement ERROR$PTR = @C$ERROR; turns a reference to ERROR into a reference to C$ERROR. This technique is useful not only for manipulating complicated data structures but also for passing locations to procedures as parameters.

With strings attached

One of the key features built into the 8086 is the ability to handle large-scale string-manipulation assignments far more easily than the 8080 and the 8085. PL/M-86 exploits this feature, with very powerful string-handling procedures to scan, translate or move blocks of bytes or words in ascending or descending order. The system designer thus has access to the 8086's string capabilities without having to worry about absolute memory locations and register contents, as an assembly-language programmer would (Fig. 3).

Another feature designed into the 8086 architecture is multiprocessing capability, accessible via the LOCKSET procedure. Through it, the system designer gains control over shared resources by locking other processors out while, for instance, a memory block is being updated. In a system where an 8086 processor offloads its I/O control tasks to an 8089 I/O processor, some memory locations may be used by both processors.

While the 8086 is accessing and updating that memory location, the 8086 should not be outputting data from that location or writing new data into that location. So a flag is set or reset depending on whether or not the processor seeking access to the critical resource can obtain that access.

An optimizer saves memory

Memory may be cheap, but in a large production run every byte still counts. So, an optimizing compiler will soon pay for itself. PL/M-86 uses a number of optimization techniques:

Folding of constant expressions

Calculating the value of constants in expressions at compile time rather than generating code to calculate it at run time saves both time and memory. In the expression

A = 6 + 3 + A;

the compiler will add 6 and 3 first and produce code to add 9 to A.

Strength reduction

This term applies to the replacement of certain instructions with faster, shorter ones. For example, performing a left-shift of one bit replaces a multiplication by two; n left-shifts correspond to a multiplication with 2^n.

Elimination of common expressions

If an expression appears more than once in the same block, its value is saved rather than recomputed each time. For example, in

A = B + C*D/3
C = E + C*D/3

the value of C*D/3 need not be computed a second time.

Short-jump optimization

When there's a choice of different jump-instruction types, the compiler selects the smallest one possible.

Branch optimization

Branch chaining reduces a branch to another branch to a single branch instruction:

```
BEFORE                AFTER
  JMP  LAB1             JMP  LAB2
   .                     .
   .                     .
   .                     .
LAB1: JMP  LAB2       LAB1: JMP  LAB2
LAB2:                 LAB2:
```

Having defined a BYTE variable (called LOCK, for example), the LOCKSET instruction sets that variable to a value that denies memory access.

If LOCK=1 means "access not available" and LOCK=0 means "access allowed," and if all processors in the system have been programmed to recognize that convention, the following code segment gives access to a *critical* memory location while preventing other processors from doing so until the operation is finished:

```
/*BEGIN CRITICAL REGION*/
DO WHILE LOCKSET (@LOCK, 1);
END;

.
.
.

LOCK=0;
/*END CRITICAL REGION*/
```

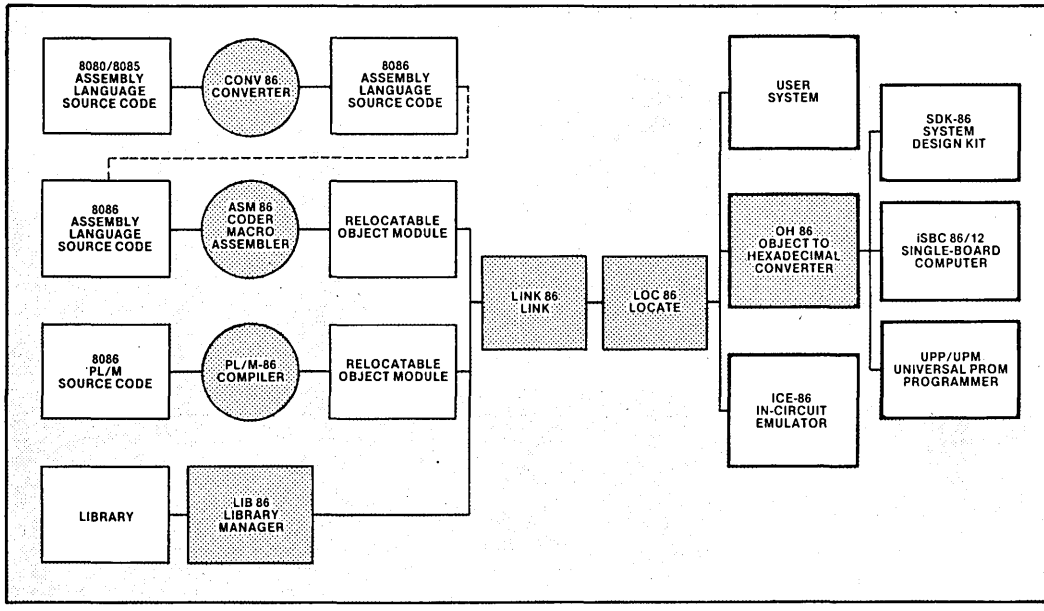
In this segment, the processor loops until memory location LOCK is reset by another processor—i.e., LOCKSET returns ZERO until that processor sets LOCK to prevent other processors from accessing the memory area. The processor carries out its program, then unlocks the memory area (LOCK=0). The first executable line of the program segment (DO WHILE...) references the variable LOCK and assigns the value 1 to that location.

If the value returned is 0, LOCK had not already been set and the current processor has now set it. But if the value returned is 1, the LOCK had already been set and the processor must wait until the busy processor releases the memory lock. Since the locking mechanism uses a simple BYTE variable, there is no practical limit to the number of locks available.

A language isn't enough

PL/M-86 is implemented as a compiler, not as an interpreter, because in the normal μC design process a debugged program is loaded into PROMs for the prototype system. A compiler produces object modules in a form that can be directly executed by the CPU.

The PL/M-86 compiler boasts many compile-time options to help with coding and debugging. Most important is *conditional compilation*, which permits the compiler to skip over selected portions of the source code if certain conditions are met. This feature enables the designer to produce different object modules for different applications of the program. An INCLUDE command, on the other hand, allows the user



5. The PL/M-86 package (screen) contains, in addition to the compiler, an 8086 assembler and many important utilities. The final machine code can be loaded into a number of optional hardware items.

to include routines from a different source file as well.

Another compiler option, CODE/NOCODE, provides listings of the generated object code in assembly-language format, interleaved with the PL/M statements for easier debugging. The PL/M-86 compiler also provides a flexible cross-reference of program symbols between PL/M-86 modules.

The PL/M-86 compiler also includes sophisticated code-optimization techniques to produce efficient object modules. A compile-time OPTIMIZE control provides three levels of optimization: Level 0 skips optimization for a quick compilation. Level-1 optimization is the PL/M-86 default and provides constant-folding, strength reduction and elimination of common expressions. Level 2 adds jump optimization, branch chaining, cross-jumping and deletion of unreachable code (see "An Optimizer Saves Memory").

An example incorporating several optimization techniques is shown in Fig. 4. The program determines whether the byte variable TEST is true (i.e., the least significant bit is 1). If it is, the hex value 0FFFF will be output to port 0F6H and the value of the BYTE variable B will be assigned to the WORD variable A. If the variable TEST is not true, variable A will be assigned the value C.

The assembly code produced by the short PL/M-86 module contains 57 bytes (Fig. 4a). Cross-jumping inserts a JUMP (line 8, Fig. 4b) to combine the identical code at the end of two converging paths (lines 8 and 9 and 12 and 13 in Fig. 4a) and diverts the program flow to the second occurrence of the two lines. The first occurrence is now unreachable and can be deleted (Fig. 4c). Another line of code is saved by reversing a branch condition, which produces line 3 of Fig. 4d.

The PL/M-86 compiler, which runs on Intel's Intellec μC-development system, is not a "stand-alone" design tool but part of an integrated set of design-aid tools for the 8086 or 8088. These tools include an assembler for ASM86, a high-level assembly language that produces object modules compatible with those from PL/M-86 (both can be combined using the 8086/8088 relocation and linkage tools).

ASM86 complements PL/M-86 since it lets the programmer choose the language most appropriate for a task and then combine the modules. Commonly used PL/M-86 and ASM86 object modules can be stored and managed using LIB86, the 8086 object-module librarian. PL/M-86 or ASM86 object modules may be loaded by the ICE-86 in-circuit emulator, and the software may then be debugged and integrated with the hardware. After hex conversion, Intellec's PROM programmer allows the debugged object modules to be stored in EPROMs (Fig. 5).■■

SYSTEM DESIGN/SOFTWARE

COMPILER OPTIMIZATION TECHNIQUES

Techniques used within the PL/M-86 compiler make the programmer's job easier while supplying highly efficient code

by Armond Inselberg and Stan Mazor

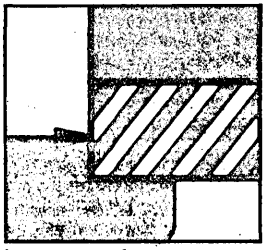
Increasing demands for software development have combined with continuing shortages of programming personnel to create a crisis situation. Shortages of skilled programmers can be partially relieved by careful choice among available programming languages and their compilers. High level languages can make programming easier. Compilers can reduce time spent coding and make up for a shortage of experience by providing the techniques needed to optimize both size and execution speed of machine level code.

What is a compiler?

Software implementation environments can be divided into two levels, as shown in Fig 1: the program machine level and the hardware machine level. Although actual code execution takes place at the hardware machine level, a software engineer cannot efficiently communicate directly with this level. Instead, a programming language, such as PL/M-86, is used as the communication link with the programming machine. The compiler is responsible for translating language input to the programming machine into the language of the hardware machine. In this regard, the maturity of the PL/M-86 compiler as a powerful tool for 8086 software development is revealed.

The compilation process

During the compilation process, the compiler closely binds the input program, determines its syntactic correctness, and generates efficient hardware machine code. Closely binding a program means to fix the types of variables, the forms of the expressions, and the program's structure. To generate efficient hardware machine code, various optimization techniques are used.



The two major steps of the compilation process are the parsing of the input source program and the generation of the output object code. (See Fig 2.) Parsing is achieved by a lexical and syntactic analysis. Lexical analysis separates individual components or tokens making up the program's symbols. These symbols include variable names, key words, and operators. Syntactic analysis checks the program for any syntax errors by determining the structure of the source program in terms of its blocks, statements, and expressions. Results of the parsing are an intermediate text string and a dictionary of variables used in the program.

Generation of the dictionary, or symbol table, is central to the compilation process as it provides a reference for the variable names and their properties. Built during examination of the data declarations, the symbol table is continually referenced during the remainder of the compilation.

The second step of the compilation first performs optimization over the intermediate text, independent of the target hardware. Final object code is then generated, with consideration for hardware machine dependent optimization.

Stanley Mazor is with Intel Corp, 1350 Bordeaux Dr, Sunnyvale, CA 94086, where he has participated in the designs of the MCS-4, MCS-8, 8080, and several other microcomputers. Prior to joining Intel in 1969, he was assistant manager of the computer center at San Francisco State College and a principal designer of the Symbol computer at Fairchild. Mr Mazor has published over 30 articles and papers on microcomputers and shares patents on the 8080 and MCS-4. He is a senior member of the IEEE.

Armond Inselberg is a senior consultant at the Institute for Software Engineering, Suite 200, 535 Middlefield Rd, Menlo Park, CA 94025. He is involved in data processing capacity management for workload analysis and forecasting. Previously, he worked at Intel, Stanford University, and IBM. He has a PhD in computer science from Washington University and an MBA from the University of Santa Clara.

MARCH 1982
ORDER NUMBER: 210397-001

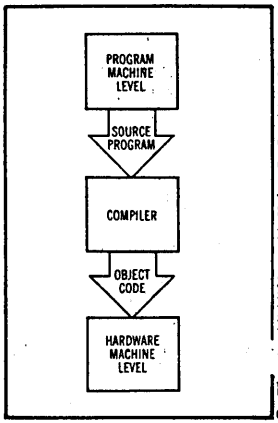


Fig 1 Software implementation environment. Programmer communicates with program machine level, while actual code execution occurs at hardware machine level. Compiler serves as interface between two levels

Optimization philosophy

Efficiency of the generated object code is a primary objective of the compiler. Providing correct code is, of course, the primary objective, but it is never stated explicitly. As with most compilers, the PL/M-86 compiler is geared toward optimizing programs written using good programming practices.

There is a tradeoff between the speed of compilation and the optimization of the resulting object code. Although optimized code is most desirable for finalized production software, the preference during development is for fast compilation. Since a conflict exists between the speed of compilation and code optimization, a compromise must be made.

With PL/M-86, the user can select the level of optimization. Level 0 is the most basic, and level 3, the most advanced. Each successive level provides all optimization techniques of the lower levels, while adding further techniques. If an optimization level is not specified at compile time, the system defaults to level 1.

Specific techniques used within the PL/M-86 compiler serve to optimize the amount of code generated, the execution time of the code, or both the amount of code and the execution time. Hardware machine independent and machine dependent optimization techniques make up a secondary classification of the techniques. Machine independent techniques optimize object code, independent of the target processor. Machine dependent optimization takes advantage of the architecture of the target processor. A third classification is based on whether the techniques optimize over a single program statement or over a range of statements. Table 1 summarizes PL/M-86 optimization techniques for these three classifications.

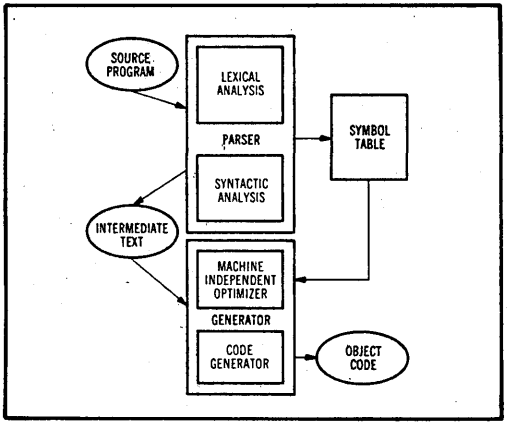


Fig 2 Compilation process. Parsing of source program produces symbol table and intermediate text string. Text string is then optimized, resulting in generation of object code

Amount of code generated

When only a limited amount of memory is available to hold the program, optimizing the amount of code is particularly relevant. Three techniques within the compiler work to reduce the amount of generated code.

Branching to duplicate code—Removing code which occurs more than once, this technique can be used when the paths through duplicate copies of code have the same termination point in the program. In this case, as shown in Fig 3, the second copy of code is replaced with a jump to the original copy.

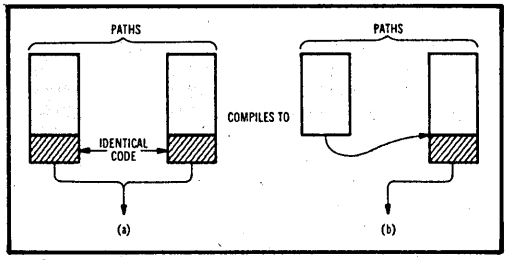


Fig 3 Branching to duplicate code optimization. Both copies of code have same termination point (a); during compilation, second copy of code is replaced by jump to first copy (b)

An example of two program paths that have portions of identical code and terminate at the same point can be found in an IF-THEN-ELSE statement.

```
IF X > Y                                    MOV  AL,Y
   THEN DO;                                 CMP  X,AL
        X=Y;          compiles to           JBE  @1
        X=X+1;                              MOV  X,AL
        END;                          @1:   INC  X
   ELSE X=X+1;
```

In the example, the common program statement X = X + 1; is compiled to INC X and is used by both paths through the compiled IF statement. If X is less than or equal to Y, the JBE (jump below or equal) instruction is executed, causing a jump to the INC instruction. If X is greater than Y, INC is reached even though the JBE is not executed.

Removal of unreachable code—This technique causes the compiler to skip those parts of the program that will never be executed. For example, unlabeled program statements that follow a GOTO statement cannot be reached, and therefore will never be executed. Thus,

```
        GOTO LABELZ;
not      IF X > Y          compiles to   JMP    LABELZ
compiled    THEN X=X+1;                 LABELZ: INC Y
         LABELZ: Y=Y+1;
```

TABLE 1

PL/M-86 Optimization Techniques

	Amount of Code		Execution Speed		Both Amount of Code and Execution Speed	
	Hardware Independent	Hardware Dependent	Hardware Independent	Hardware Dependent	Hardware Independent	Hardware Dependent
Single statement		Instruction size	Strength reduction		Folding of constants Expression arrangement Short circuit of Boolean expressions Function evaluation	
Range of statements	Branching to duplicate code Removal of unreachable code			Address pointer comparison	Elimination of common subexpressions Elimination of superfluous branches	Peephole Indeterminant storage operations

Although this optimization technique reduces the amount of code generated, it is needed only when the programmer is careless.

Instruction size—The compiler in this case selects the shortest encoding of the instruction. Instructions involving a hardware register can be shortened by one byte if the register is the accumulator. In addition, jumps to locations within 127 bytes require shorter instructions because the increment rather than the target address is specified. For example, if a JA conditional jump instruction jumps to a label @2 that is 14 bytes away, the distance of 14 bytes is stored in the instruction. Thus, the instruction uses one byte to specify an offset rather than four bytes to indicate the target address of @2.

```
JA @2          encoded as      77 14
                                opcode  offset
```

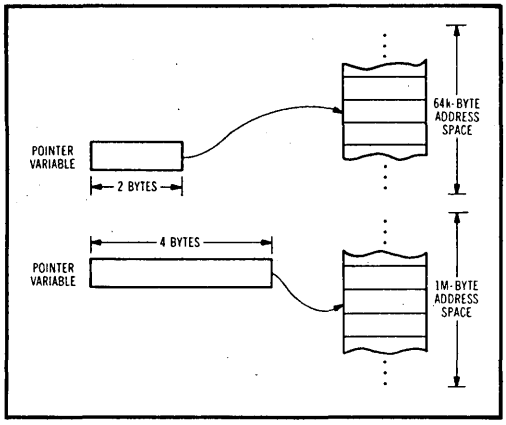


Fig 4 Instruction size optimization. If address space is restricted to 64k (top), compiler allocates 2 bytes for type pointer variable; otherwise, variables require 4 bytes (bottom)

Another aspect of this optimization technique is that the compiler will allocate two bytes to variables declared to be of type pointer, if the address spaces for code and data are restricted to 64k bytes each. Otherwise, as shown in Fig 4, variables of type pointer require four bytes. The programmer indicates the size of the address space to the compiler through a compiler control switch.

Execution speed

Optimizing the execution speed can be critical for time-dependent processing. Two optimization techniques available for improving execution speed are strength reduction and address pointer comparison.

Strength reduction—Execution is optimized by replacing certain operations with faster executing operations. For example, the compiler replaces "multiply a variable Y by two" with a shift left operation. The result is the same, but a shift left executes faster than a multiply.

```
X = Y*2;        compiles to     MOV  AL,Y
                                SHL  AL,1
                                MOV  X,AL
```

Address pointer comparison—This optimization technique generates code to compare two 32-bit pointer variables. Physical addresses are actually 20 bits, but are stored as a 16-bit base and a 16-bit offset field. When the base is shifted left by 4 bits and added to the offset, it yields a 20-bit address (Fig 5). Execution speed is improved because, instead of calculating the 20-bit address to compare pointers, code is generated to first compare the base parts. Only if the base parts are equal is it necessary to compare the offset parts.

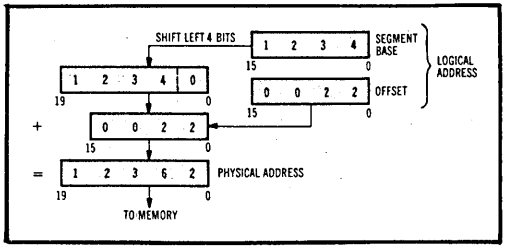


Fig 5 Address pointer comparison. 32-bit pointer variables are stored as 16-bit base and 16-bit offset. Shifting base left 4 bits and adding it to offset results in 20-bit address

For example, two variables, PTR1 and PTR2, are declared to be of type pointer. If PTR1 is greater than PTR2, then X is set equal to 0.

```
DECLARE (PTR1,PTR2) POINTER;
IF PTR1 > PTR2
 THEN X=0;          compiles to     LES   AX,PTR1
                                    PUSH  ES
                                    LES   DX,PTR2
                                    MOV   DI,ES
                                    POP   SI
                                    CMP   SI,DI
                                    JNE   $+4H
                                    CMP   AX,DX
                                    JBE   @1
                                    MOV   X,0H
                          @1:
```

In this example, the LES instruction loads the AX register with the offset of PTR1. The base is loaded into the ES register, then moved to the SI register by means of the stack. The offset of PTR2 is loaded into the DX register and the base is moved to the DI register. The two base values in the SI and DI registers are compared by the CMP instruction. If the results are not equal, the JNE instruction (jump not equal) is executed, skipping the code used to compare the offsets, and jumping to the instruction that sets X to 0.

When only a limited amount of memory is available to hold the program, optimizing the amount of code is particularly relevant.

Optimizing both amount of code and execution speed

Most optimization techniques reduce the amount of generated code and improve execution speed. Eight techniques accomplish this within the PL/M-86 compiler.

Folding of constants—This technique causes the compiler to perform arithmetic operations at compile time rather than at execution time. For example, a statement with the expression 6 + 3 + W would be coded as 9 + W. Thus,

```
V = 6+3+W;          compiles to     MOV   AL,W
                                    ADD   AL,9H
                                    MOV   V,AL
```

Expression arrangement—Code for expression evaluation is generated such that the operations are performed in that order which produces the most efficient code. If expressions I times J and K times L are to be calculated, and their results subtracted, then

```
Z = (I*J) - (K*L);      compiles to     MOV   AL,J
                                        MUL   I
                                        PUSH  AX
                                        MOV   AL,L
                                        MUL   K
                                        POP   CX
                                        SUB   CX,AX
                                        MOV   Z,CL
```

In this example, the result of I * J is pushed onto the stack, freeing the accumulator for a second multiply. After K * L is evaluated, the result of I * J is popped into the CX register. The registers are then subtracted. This process is much more efficient than having the compiler first save the two multiplication results in temporary variables, then move these results to registers, and finally subtract the registers.

Short circuit of Boolean expressions—Generated code terminates the evaluation of a Boolean expression as soon as its outcome is established. For example, consider the expression (V>X AND I>J). If V is not greater than X, the expression will be false, regardless of the results of the rest of the expression; therefore, the remainder of the expression need not be evaluated. Thus,

```
IF (V > X AND I > J)    compiles to     MOV   AL,V
    THEN B=1;                           CMP   AL,X
                                        JBE   @1
                                        MOV   AL,I
                                        CMP   AL,J
                                        JBE   @1
                                        MOV   B,1H
                                @1:
```

In this example, the generated code tests V for greater than X. If this comparison is false, the JBE (jump on below or equal) to label @1 is executed. This label is generated by the compiler to go around the IF statement without executing the remaining code of the Boolean expression. This technique not only saves execution time but reduces the number of generated instructions required to evaluate the expression.

Function evaluation—The compiler evaluates several specific functions as they are encountered in the source program at compile time. For example, for a 10-element array named W, the LAST function obtains the value 9, the last subscript of the array. Arrays are indexed starting with 0.

```
DECLARE W(10) BYTE;
I = LAST (W);           compiles to     MOV   I,9H
```

By evaluating such functions, the compiler saves execution time and storage space, and makes the programmer's job easier by permitting the functions to be referenced.

Elimination of common subexpressions—The compiler recognizes multiple occurrences of an expression and saves the value of the expression in a register or stack so that it need not be recalculated. For example, the expression J + I or I + J may occur several times but will be evaluated only once.

```
X = J + I;          compiles to     MOV  AL,J
Y = I + J;                          ADD  AL,I
                                    MOV  X,AL
                                    MOV  Y,AL
```

By saving the result of J + I in the AL register, rather than recalculating each time it is encountered, generated object code and execution time are greatly reduced.

Optimizing the execution speed can be critical for time-dependent processing.

Elimination of superfluous branches—Optimization using this technique reduces the number of jumps that must be executed. In the first example, jumping to a LABELX that contains a jump to LABELZ transforms the first jump into a branch directly to LABELZ.

```
IF X > Y                    compiles to   MOV AL,X
   THEN GOTO LABELX;                      CMP AL,Y
                                          JA  LABELZ
          .                                 .
          .                                 .
          .                                 .
LABELX: GOTO LABELZ;                LABELX: JMP LABELZ
          .                                 .
          .                                 .
          .                                 .
LABELZ:                             LABELZ:
```

Another example is the selection of a single conditional jump instruction based on the result of a comparison. This optimization can occur frequently, eliminating an unconditional JMP instruction each time through the selection of the appropriate conditional jump. Consider the IF statement that executes some code only if X > Y.

```
IF X > Y            compiles to         MOV AL,X
   THEN DO;                             CMP AL,Y
        Z=R;                            JBE @1   ]
        R=R+1;                          MOV AL,R
      END;                              MOV Z,AL
                                        INC R
                                  @1
                                        MOV AL,X
                                        CMP AL,Y
                    compiles to         JA  $+5H ]
                    without use of      JMP @1   ]
                    optimization        MOV AL,R
                    technique           MOV Z,AL
                                        INC R
                                  @1:
```

In this example, the JA (jump above) and JMP (unconditional jump) instructions are replaced by a single JBE (jump below or equal) instruction.

Peephole—This optimization attempts to discard redundant instructions. One such action might be loading a register with a value that it contains already. For example, if Y is set equal to X + 1, the value of Y is currently in the accumulators since it was last used to calculate X + 1. If Y is again used in the next statement, there is no need to fetch the value of Y. Thus,

```
Y=X+1;          compiles to     MOV  AL,X
Z=W+Y;                          INC  AL
                                MOV  Y,AL
                                ADD  AL,W
                                MOV  Z,AL
```

Since the value of Y is currently in the accumulator as a result of the calculation of X + 1, it need not be reloaded into the accumulator for the calculation of W + Y.

Indeterminant storage operation—The compiler does not reload the starting point of a based data structure each time that it is referenced. For example, consider PART to be an array of structure elements based by the pointer variable PARTPTR.

```
DECLARE PART BASED PARTPTR (10)
  STRUCTURE (PARTNO WORD,
             AMT    BYTE,
             COST   WORD);

PART(2).PARTNO=6C4H;  compiles to   MOV  BX,PARTPTR
PART(6).AMT=79H;                    MOV  PART[BX,0AH],6C4H
                                    MOV  [BX+20H],79H
```

The first reference to the array structure places the base of the array, contained in PARTPTR, in the BX register. Further references to the array structure do not require that the BX register be reloaded.

Evaluation examples

PL/M-86 offers four levels of optimization. Optimization techniques provided at each of these levels are classified in Table 2. To indicate how much storage is actually

TABLE 2

Optimization Techniques Provided In Each Compiler Level

Optimization Technique	Optimization Level			
	0	1	2	3
Folding of constants	X	X	X	X
Expression arrangement	X	X	X	X
Short circuit of Boolean expression	X	X	X	X
Function evaluation	X	X	X	X
Strength reduction		X	X	X
Elimination of common subexpressions		X	X	X
Elimination of superfluous branches			X	X
Removal of unreachable code			X	X
Branching to duplicate code			X	X
Instruction size			X	X
Peephole			X	X
Indeterminant storage operations				X
Address pointer comparisons				X

TABLE 3

Object Code (bytes) Generated For Each Optimization Level

	Level 0	Level 1	Level 2	Level 3
Program A: Mastermind	1688	1559	1450	1450
Program B: General sort	1953	1789	1503	1503
Program C: Frequency count	849	765	694	694
Program D: Process simulation	7955	7951	7083	7083
Program E: Service queue	289	250	212	185
Average % size reduction from previous level		7.9%	12.28%	2.55%

saved by these techniques, five sample programs were compiled at each level using version 2.1 of the compiler; Table 3 provides the size in bytes of resulting compilations. The reduction in size obtained in going from one level to the next higher level is due to the additional optimization techniques used at the higher level.

Programs used in this study demonstrate the compiler's ability to optimize various types of instructions. Program A plays the game of mastermind with the operator performing a large amount of input/output with the cathode ray tube. Program B performs a sort on an array of 1000 records, making extensive use of structures and pointers. Performing a frequency word count on an arbitrary text file, Program C uses string move instructions and pointers. Program D uses simple coding with no structures or pointer addressing to perform a process simulation. Service queue simulation using linked data structures is done in Program E.

For each successive level of optimization, the individual percentages in size reduction of the programs were averaged. From Table 3, it becomes apparent that Level 3 optimization provides nearly a 25% reduction in storage requirements.

Conclusion

As the demand for microprocessor software increases, the selection of the implementation language will receive more attention. In choosing a language, users must consider not only high level constructs of the language itself, but also the capabilities of available compilers to translate the resulting programs.

November 1983

PL/M-51: A High-Level Language for the 8051 Microcontroller Family

Rajen Jaswa
Wescon/82

Originally prepared for and presented at Wescon/82

ORDER NUMBER:210806-001

PL/M-51: A HIGH-LEVEL LANGUAGE FOR THE 8051 MICROCONTROLLER FAMILY

High-level language advantages are fairly well recognized now. Developing software for embedded microcontrollers using assembly language is labor intensive and therefore an expensive task. It is not easy to come up with a sequence of well-defined stages to go from the system design stage to the system implementation software. The transformation of an algorithm flowchart to the actual assembly-language code requires considerable intuitive guesses and inventiveness on the part of the programmer. Also, assembly language is difficult to read and inspect. Because assembly language projects are difficult to manage, there has been a widespread movement towards using high-level languages. High-level languages provide, in general, improved programmer productivity, and reliable, maintainable, portable software.

In the microcontroller environment, the major considerations for a high-level language are efficient code, close control over hardware resources and optimum use of scarce on-chip data memory (RAM is very expensive in terms of silicon real estate). Intel developed PL/M-51 for the 8051 single-chip microcontrollers with the specific goal of trying to meet these criteria with minimal impact on the traditional high-level language benefits of reliability and maintainability.

OVERVIEW OF THE 8051 ARCHITECTURE

The 8051 is a stand-alone high-performance single-chip computer intended for use in sophisticated real-time applications such as instrumentation, industrial control and intelligent computer peripherals. It provides the hardware features, architectural enhancements and new instructions that make it a powerful and cost effective controller for applications requiring up to 64K-bytes of program memory and/or up to 64K-bytes of data storage. Figure 1 shows the 8051 Functional Block Diagram.

The 8051 microcomputer integrates on a single chip the CPU, 4K x 8 read-only program memory, 128 x 8 read/write data memory, 32 I/O lines, two 16-bit timer/event counters, a five-source, two-priority level, nested interrupt structure, serial I/O port for either multi-processor communications, I/O expansion, or full duplex UART, and on-chip oscillator and clock circuits.

The 8051 has four address spaces tailored to support a wide range of control applications efficiently—program memory, on-chip and external data memory, and the bit memory space. This complex (but sophisticated) memory architecture is supported by a rich (but unorthogonal) set of addressing modes for efficient memory access—register addressing, direct, indirect, immediate and base-register plus index-register indirect addressing. To support this complex memory architecture, a high-level language's syntax must mirror the underlying microcontroller architecture. The challenge is to implement this without compromising the language's readability and maintainability.

The popular 8051 architecture forms the core of the MCS-51™ microcontroller family. The need to base processors on a popular, industry-standard architecture is dictated by the cost of developing processor support hardware and software tools, as well as a desire to maintain the customer's investment in engineering resources and capital equipment. The upgradeability requirement has to be traded off against providing optimum functionality in the processor for the target market segment. Consequently, the 8051 family consists of straight-line enhancements—RAM, ROM memories and clock rates—as well as microcontrollers like the 8044 remote universal peripheral interface processor (RUPI), which has the 8051 core architecture but supports an interrupt structure and I/O functions tailored to the distributed processing environment. The cost of developing a new support environment for processors targeted to specific (and small) market niches would make the processor an unviable product. Consequently, software tools for proliferation processors should be configurable from the core processor support products.

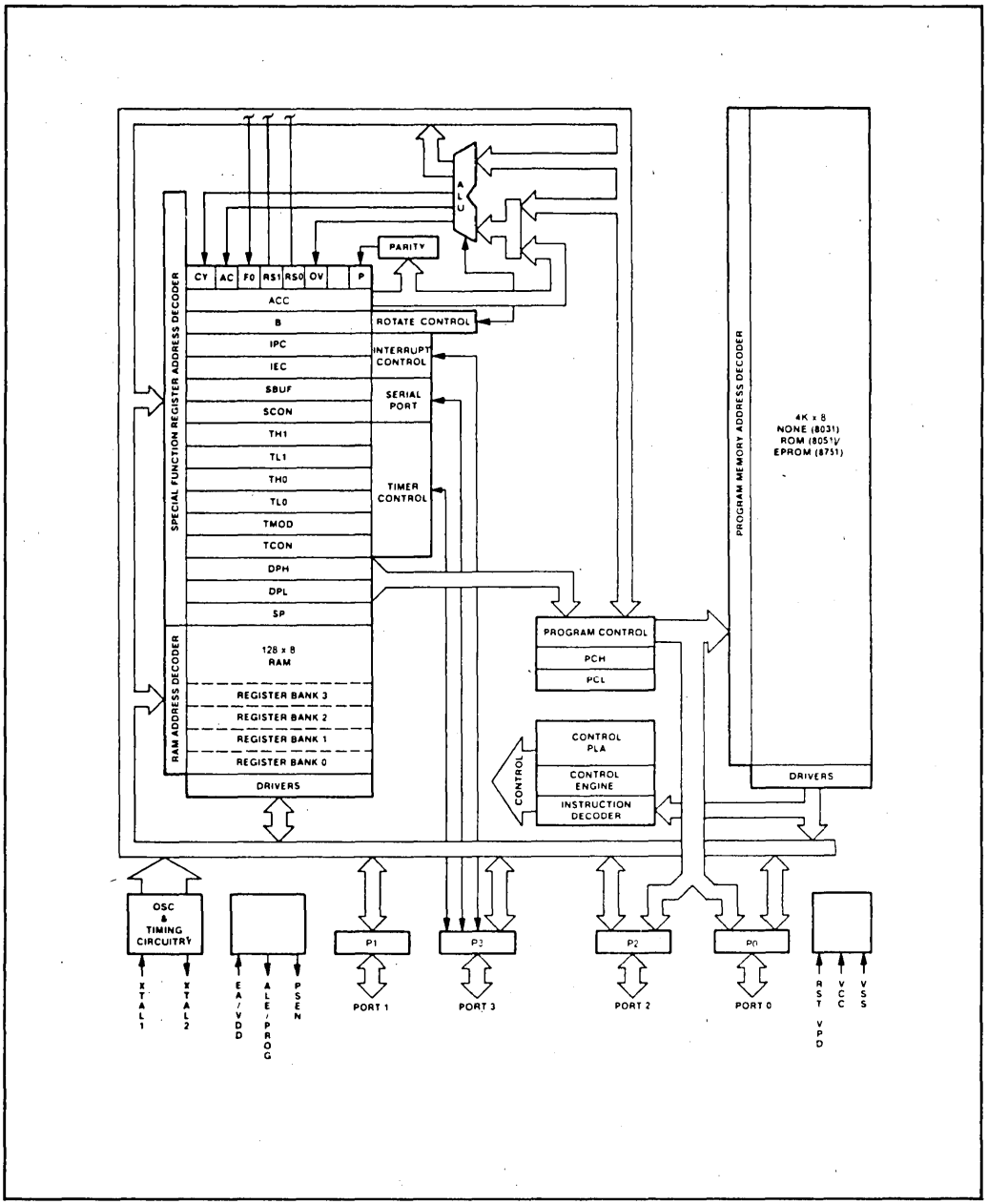


Figure 1. 8051 Functional Block Diagram

PL/M-51

PL/M-51 was developed to facilitate the design of reliable, maintainable microcontroller systems. This goal translates into a programming language which encourages and enforces good software engineering practices such as structured programming, top-down design and implementation, step-wise refinement and software walk-throughs. However, this goal has to be traded off against the exigencies of the microcontroller environment—high performance requirements, scarce memory resources and control over the hardware facilities. PL/M-51 tries to satisfy these conflicting requirements by enforcing block structured software design, providing control-flow statements for structures programming (if-then-else, do case, do while, . . .) as well as by supporting 8051 architecture specific attributes at the language level, for example—the REGISTER and AUXILIARY variable attributes, and the specifics of interrupt handling.

SOFTWARE ORGANIZATION WITH PL/M-51

Most applications are decomposed into logically related functions which can be programmed more or less independently of other functions. Interactions between functions are via a few well-defined data parameters and system level status blocks which are globally accessible to all functions at all times. PL/M-51 program structure maps very well into this structured software organization. PL/M-51 programs consist of one "main" module and several functional modules which are independently compilable units and consequently can be independently developed and debugged. Each module consists of one or more procedures. A procedure contains variable declarations and a sequence of executable statements. Variables have restricted scope to the block they are defined in, unless the scope has been extended by the PUBLIC/EXTERNAL attribute. The advantage of block scoping of variables is that programming errors of duplicate variable use are quickly identified. Figures 2 and 3 show the organization of PL/M-51 programs for heirarchical tree-structured real-time software systems. PL/M-51 does not enforce a tree-structured organization, but it provides a modular organization facility for implementing it.

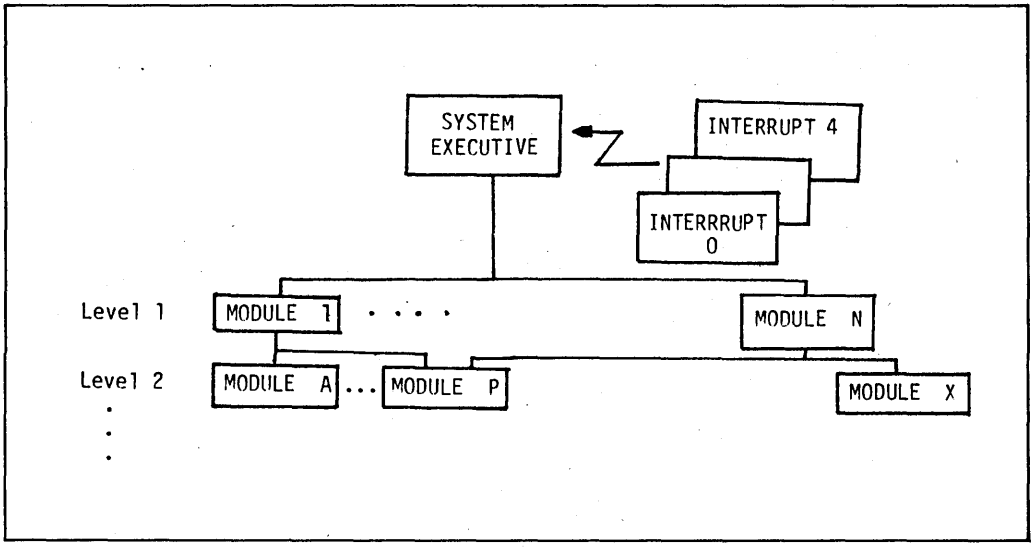


Figure 2. Hierarchical Real-time Software Systems

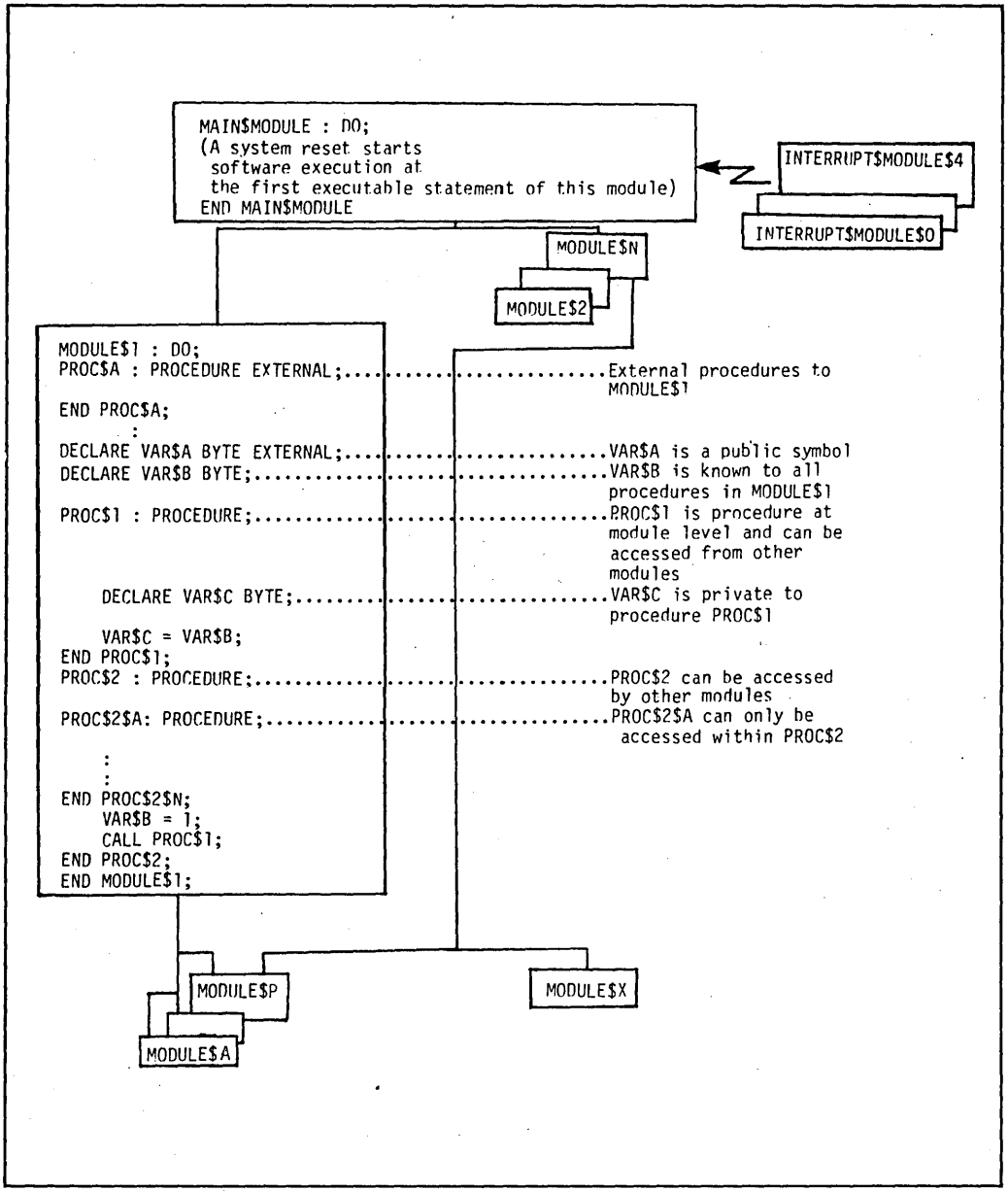


Figure 3. Organization of PL/M-51 Programs

DATA TYPES

8051 microcontroller software requires intimate knowledge of the machine representation of data variables because a significant amount of processing is done at the bit level. Consequently, the basic types of data in PL/M-51 are BIT, BYTE and WORD—as opposed to INTEGER, REAL . . . COMPLEX machine-independent data types in other high-level languages. With the three basic data types of PL/M-51, the state of each variable is known to the programmer—at the bit level. This is important, if PL/M-51 programs are to take advantage of the powerful boolean instructions on the 8051.

BUILT-IN FUNCTIONS

The PL/M-51 language has been enhanced with a number of useful standard functions which provide information about data representation at run-time to programs, do type conversions and provide machine level functions at a high-level language.

The LENGTH and index of the LAST element in an array and the SIZE of a variable in bytes can be obtained by a program at run-time. This facility permits the development of program libraries which can be reused on other projects.

System programs require the ability to manipulate data at the machine representation level as well as at the logical level. Consequently, PL/M-51 provides type conversions BIT to BYTE to WORD as well as machine level instructions like rotate and shift for variable manipulation.

The 8051 architecture has a powerful instruction repertoire for conditional execution on bit states. PL/M-51 provides a TESTCLEAR function to support process synchronization primitives—for example, semaphores require uninterruptible test-set atomic operations.

8051 ARCHITECTURE SPECIFIC ATTRIBUTES

The 8051 architecture is designed to provide optimum performance over a wide range of control applications. Consequently, it has a sophisticated (and complex) memory organization, and four register banks in the central processing unit (CPU) for rapid task switching during interrupts. PL/M-51 supports programming for this environment by embracing architecture specific attributes within the language syntax.

Memory mapping of variables is done by specifying a suffix attribute during data declaration. The possible attributes are CONSTANT, AUXILIARY, REGISTER AT (128-255), MAIN and IDATA. CONSTANT variables reside within the code memory, while AUXILIARY variables are assigned to off-chip data memory. The default memory assignment or MAIN variables reside within the directly-addressable on-chip data memory. IDATA variables are indirectly-addressable over the entire on-chip data memory (0–255). The REGISTER attribute maps the variable to the pre-defined mapped registers, I/O ports and functions on-chip. The compiler generates the appropriate addressing instructions to access these variables. The key benefit of letting the compiler generate addresses (mechanically) is that when decisions to move variables from one memory space to another are made, only the declaration attribute has to be modified, and the module recompiled. The impact of such an action is an assembly language program would require identifying all references to the affected variable and changes in its code an error-prone and laborious job.

Rapid response to events are key to high performance in control applications. The 8051 architecture provides four register banks and task-switching requires only the program counter, program status word, A, B and DPTR registers to be saved. PL/M-51 allows procedures to be associated with a particular register bank. Only the program counter, not the RO-R7 register bank, needs to be saved on the stack during a subroutine call, since they use the same register bank. Task switching and the associated register bank switching is supported by the interrupt mechanism for external and internal events.

Interrupt service routines are identified by associating the hardware INTERRUPT number attribute to a procedure. The register bank too should be identified for the interrupt service routine. To prevent data corruption, interrupt service routines should use different register banks than non-interrupt code. Also, low and high priority interrupts should not use the same register bank. Since it is illegal to call procedures using different register banks, communication of information from interrupt events have to be handled via shared global data areas.

A GENERIC COMPILER

The rapid development of silicon technology allows semiconductor houses to optimize processors to specific market segments. For example, the 8044 slave processors provide intelligent peripheral control and are based on the 8051 CPU architecture. PL/M-51 can be configured to support the 8044 by inputting to the compiler a processor definition file which has information about register names and memory mapping of I/O functions and bits. Configurable compilers provide an optimum approach to managing the costs of maintaining system software, as well as supporting proliferation processors based on successful CPU architectures.

CONCLUSIONS

Software development for microcontroller applications can be executed in a planned methodical manner. PL/M-51 provides software engineers with a tool for promoting structured software design for the 8051 microcontroller family. PL/M-51 provides an environment for controlled system development.

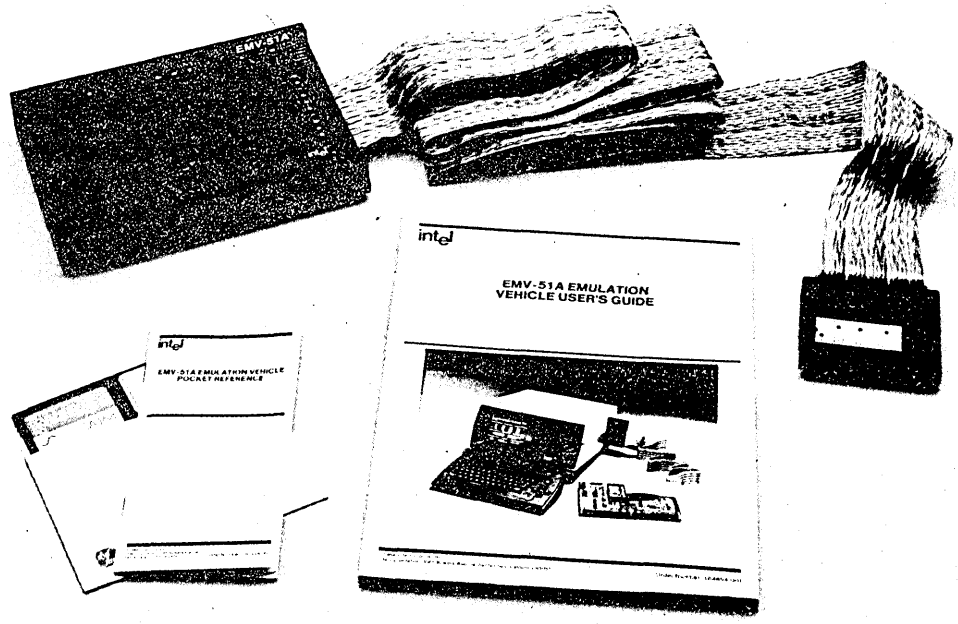
In-Circuit Emulators

5

EMV-51A
8051A EMULATION VEHICLE

- **Precise, full-speed, real-time 8051 emulation**
  - **-Load, drive, timing characteristics**
  - **-Full-speed program RAM**
  - **-Serial and parallel ports**
- **Breakpoints/trace**
  - **-4 execution address breakpoints**
  - **-1 range breakpoint**
  - **-Branch and value breakpoints**
- **Full symbolic debugging**
- **Software debugging with or without user system**
- **Advanced, easy-to-use features**
  - **-Programmable function keys**
  - **-Macros**
- **Help facility: EMV-51A command reference at console**
- **Hosted on Intel's Personal Development System**

The EMV-51A system interfaces to any user-designed 8051 or 8052 system and assists in the debugging and development of that system. The EMV-51A consists of an emulator plug, serving as the direct communication link to the user system, an 80-inch cable, and a module hosted by an Intel Personal Development System (iPDS™). The electrical and timing characteristics of the user's 8051 are accurately emulated when using the EMV-51A system. A friendly human interface presents commands in a menu display, and organizes commands in an easy-to-learn fashion. The EMV-51A system allows the designer to emulate the system's 8051 in real-time or single-step mode. Breakpoints allow the user to stop emulation at user-specified conditions, and trace qualifiers allow for conditional display of trace information. Program memory can be displayed and altered using ASM51 mnemonics and symbolic references. Advanced capabilities allow for programmable keys, macros, and control constructs. The EMV-51A system may also be used in the debugging and development of 8052 systems through its ability to debug all of the 8052 features that are shared with the 8051 and the internal 8K ROM space provided in the 8052.



MAY 1984
Order Number: 280017-001

FUNCTIONAL DESCRIPTION

EMV-51A hardware consists of three parts: the controller, the emulator module, and the cable assembly. The controller contains all the logic to support break, trace, emulation, and communication with the host and the emulator module. The emulator module contains the hardware used to execute 8051 code and supplies all MCS®-51 signals to the user's system. This module connects to the controller via a six-foot cable, and the controller connects to an iPDS host through the EMV/PROM programming adapter board. This adapter board is required to use the EMV-51A on the iPDS.

EMV-51A software contains all the control for user interaction. The software programs the controller, implements all emulator functions, and displays information to the user. This software is run on the iPDS host, and is packaged on a 5-¼ inch diskette. An additional software diagnostic routine, included on the disk, thoroughly checks the EMV-51A hardware.

EMV-51A software will accept and interpret commands entered by the user. These commands will be communicated as a set of micro-commands via a host interface to the controller. Command registers in the controller direct micro-operations to various sections of the break, map, or trace circuitry. Some commands control the emulator board, others determine whether the emulator will emulate the user system, while others interrogate the user system. When appropriate, the controller will pass information back to the host where the information will be processed and displayed to the user. See Figure 1 for a block diagram of the EMV-51A hardware.

The EMV-51A package includes the 8051 Macro Assembler and the 8051 Linker and Relocater (RL51). This assembler provides full macro capabilities, supports symbolic development for both code development and debugging, and supports modular code development with relocation features. RL51 will relocate, link, and generate loadable object files from the relocatable

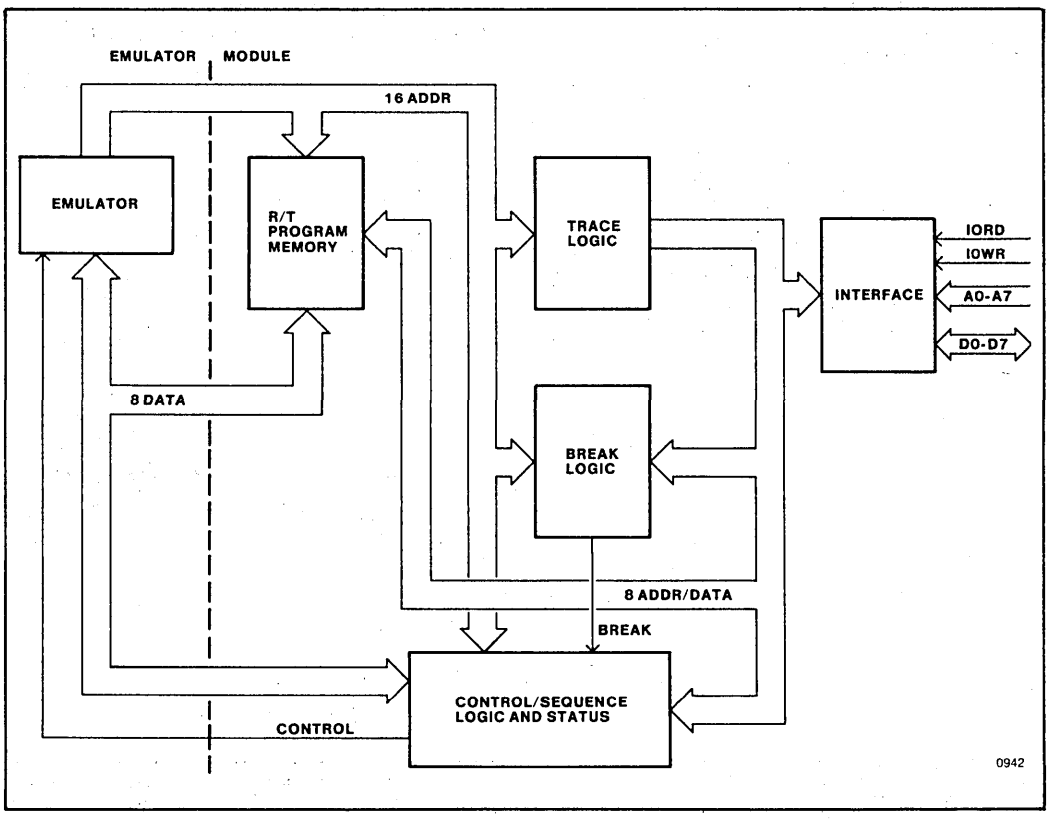


Figure 1. EMV-51A Block Diagram

modules produced by the assembler. EMV-51A fully supports all mnemonics, object file formats, and symbolic references generated by ASM51 and RL51.

EMV-51A documentation includes a comprehensive user's manual and a command dictionary reference guide.

FEATURE SET

The EMV-51A system provides fundamental capabilities for debugging an 8051 or an 8052 microprocessor system. These basic and general capabilities are described in the following sections.

Real-Time Breakpoints

The EMV-51A system allows the user system to execute user code at full clock speed, until a predefined condition occurs. The breakpoints may be a combination of four execution addresses or a combination of an execution address range and a single execution address. These break capabilities allow the user to stop the user system at various states in the normal processing cycle and to interrogate the state of the system.

Real-Time Memory

The EMV-51A system supplies 8K of high speed RAM memory. The RAM can be used to execute the user program and allows easy changes to the user code. This memory can be used either in place of the user's memory before the memory exists in the user system or used in lieu of the user's memory to ease the debugging effort.

Real-Time Trace

The EMV-51A system maintains an active real-time trace buffer that tracks the last two executed addresses from the user's system. This trace is collected in real-time during execution of the user system. This information can be used to discover where the user's program has been before breaking.

Software Break

During step mode, the EMV-51A system iteratively single steps, then executes a short software interrogation routine. This slow-down mode of operation continues until a register is set to a specific value, or any branch instruction occurs, or until a specified number of instructions have been executed. These additional break features provide users added execution control and microprocessor state information in exchange for real-time emulation.

Software Trace

The EMV-51A system will automatically query the 8051 or 8052 processor and optionally display up to 4 lines of information. This display can show execution address, disassembled code, current register values, or processor status information.

COMMANDS

The EMV-51A system has a friendly and easy-to-use human interface, and commands that are well organized and easy-to-learn. Menu displays prompt the user and assist in learning the different commands. Sample EMV-51A displays are shown in Figure 2. Commands fall into four categories: utility commands, display/modify commands, emulation commands, and advanced commands. Once these basic command categories are understood, locating any command becomes simple. Table 1 lists a summary of EMV-51A commands and the command categories.

The EMV-51A system is a full symbolic emulator, and hence all commands and displays allow for symbolic entry. Thus the EMV-51A system and users communicate by referring explicitly to symbols defined in the user's source program or symbols defined during the debugging session.

Utility Commands

Utility commands perform functions not directly related to the task of emulation and debugging. These commands access the iPDS resources and display information about the emulator. Some examples of utility commands are RESET, LOAD, HELP, and EVALUATE.

Display/Modify Commands

Display/modify commands change or display any register, port, or memory addressable by the 8051 processor chip, plus the internal 8K of ROM memory addressable by the 8052. Exam-

ples of display/modify commands include REGISTER, ASM/DASM, CBYTE, DBYTE, RBYTE, and PBYTE. A sample display using the REGISTER command is shown in Figure (3a).

Emulation Commands

All commands causing execution displays, or execution initiation, fall into the emulation category. Thus, the GO, BREAK, and TRACE commands are in this category along with BR0,1,2,3, BV, TR0,1,2,3, TS, and STEP.

Advanced Commands

The advanced commands offer the user an easy way to increase the power of the EMV-51A and thus increase the debugging capability of this product. These advanced features allow EMV-51A command sequences to be combined, executed, and stored. Examples of advanced commands include MACRO, FUNCTION, and control constructs. Figure (3b) shows a display with a macro.

EMULATION MODES

The EMV-51A system combines two approaches to emulation, real-time emulation and software emulation. Programs with time-critical sections of code or critical interrupt routines can be emulated, traced, and debugged in real time. Real-time emulation supports specific execution breakpoints or range breakpoints. The real-time trace will display up to two instruction addresses last executed. Real-time emulation mode is entered by initiating emulation with the GO command. All break and trace commands associated with the GO command act in real-time emulation mode.

Table 1. Summary of EMV-51A Commands and Command Categories

Emulation Commands

BREAK - Display breakpoint menu
BR0, 1, 2, 3 - Breakpoint register for execution address
BRR - Breakpoint register for execution range
BRB - Break on branch
BV - Break on value
BC - Clear all breaks
DTRACE - Display trace menu
TB0, 1, 2, 3 - Enable/disable display by bit value
TR0, 1, 2, 3 - Enable/disable display by execution address
TV - Enable/disable display by register value
TR - Enable/disable display of registers
TS - Enable/disable display of PSW
TD - Enable/disable display of code disassembly
STEP - Enter slow-down emulation mode
GO - Enter real-time emulation mode

Advanced Commands

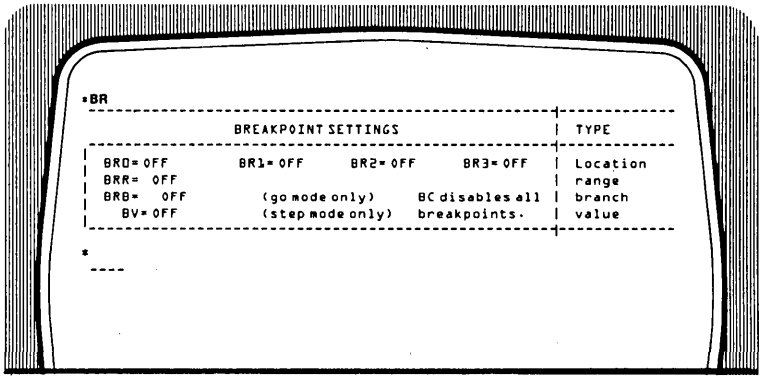
MACRO - Define, and display macro
IF THEN, COUNT, REPEAT, WHILE, UNTIL - Control constructs
FUNCTION - Invoke macro assigned to function key

Utility Commands

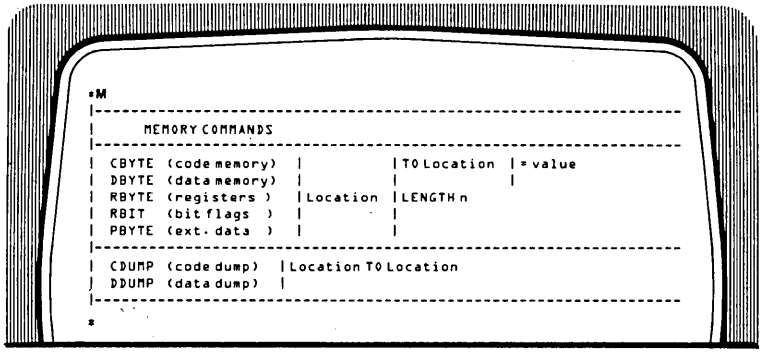
HELP - Display command syntax
LOAD - Load object file in mapped memory
LIST - Generate copy of emulation work session
DEFINE - Define symbol or macro
SYMBOL - Display symbols
REMOVE - Delete symbol or macro
ENABLE/DISABLE - Control for expanded display
EVALUATE - Evaluate any expression
SUFFIX/BASE - Set input and display numeric base
SAVE - Save code memory to file
RESET - Reset emulation processor
EXIT - Terminate EMV-51A session

Display / Modify Commands

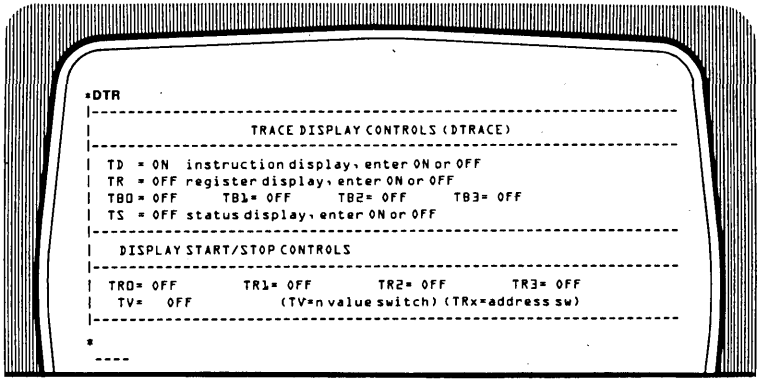
REGISTER - Change/display 8051 registers
INTERRUPT - Change/display interrupt status
MEMORY - Display menu
CBYTE, DBYTE, PBYTE, RBYTE - Change/display memory
RBIT - Change/display bit memory
CDUMP, DDUMP - Display memory as ASCII and hexadecimal
ASM/DASM - Change/display code memory as assembly language mnemonics



a) Menu Display For Accessing Memory



b) Menu Display For Accessing Memory



c) Menu Display For Setting Trace

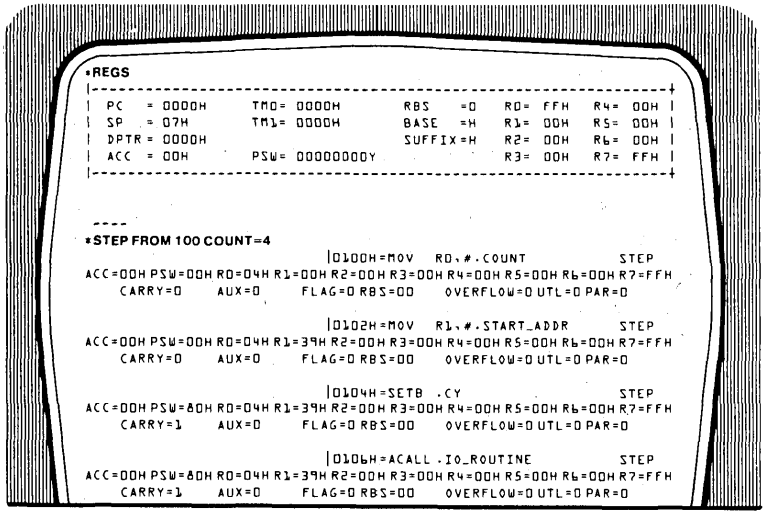
1909

Figure 2. Typical EMV-51A Menu Displays

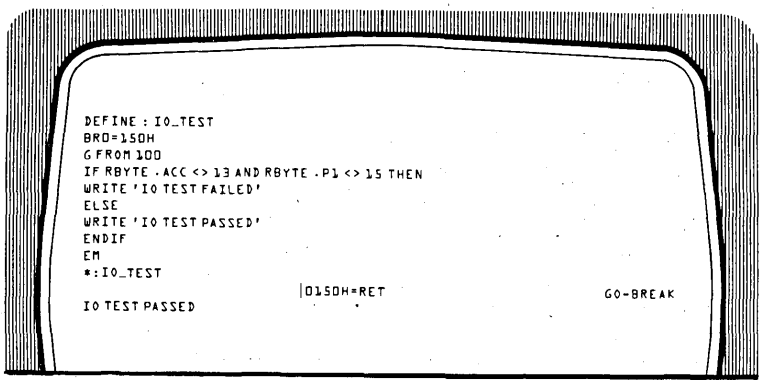
When full-speed emulation is not critical to the debuggging effort, the EMV-51A system will emulate one instruction, check for a variety of breakpoint and trace point conditions, display trace information, and continue with another instruction. This slow-down mode of operation provides enhanced break and trace facilities at the expense of non-real-time execution. Slow-down-mode emulation is entered by initiating emulation with the STEP command. Figure (3a) shows a display for the single-stepping mode.

INTENDED USE

The EMV-51A system is particularly well suited to assist in debugging small- to medium-sized



a) Display of (1) Registers and (2) Single Stepping through a Portion of a User's Program (Using Symbolics with Selective Trace of Processor and Register Status Information)



b) Display Showing Macro Capability for Debugging System Hardware and Software

1929

Figure 3. Sample Emulation Displays

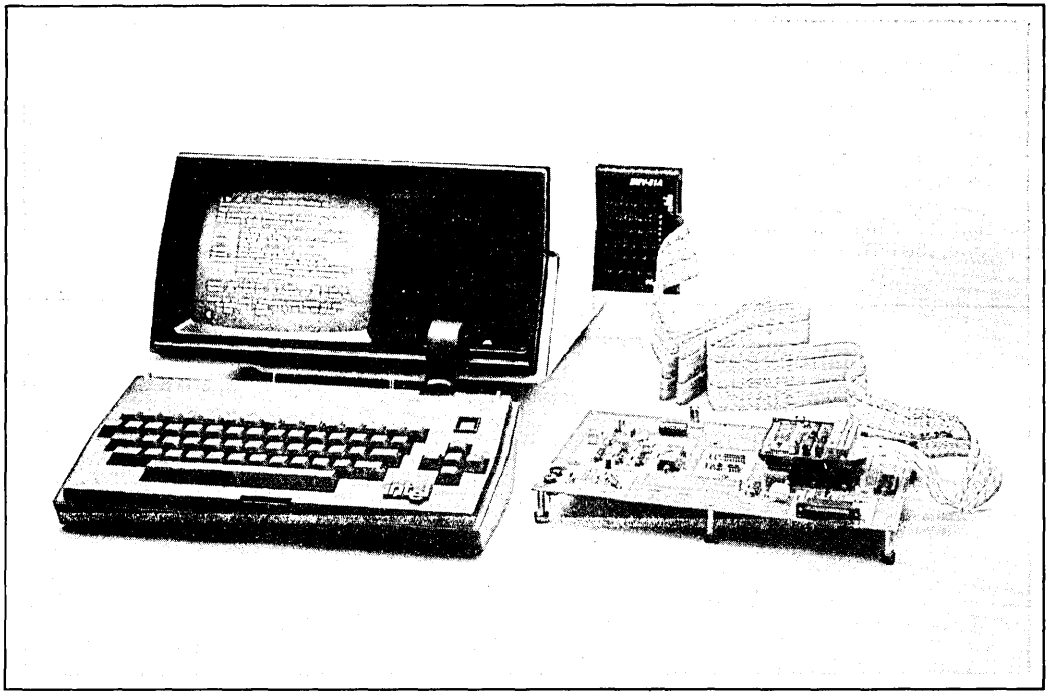


Figure 4. EMV-51A in iPDS™ Debugging Environment

programs whose program complexity is low to moderate in terms of interrupts, program nesting, and execution flow.

8051 and 8052 Debugging

The EMV-51A system can debug both the internal 8K of ROM space provided by the 8052 and the space provided by the features that the 8052 shares with the 8051. (The extra timer and extra data RAM of the 8052 are not emulated by the EMV-51A system.)

SPECIFICATIONS

EMV-51A System Operating Requirements

The EMV-51A system operates with an iPDS system. The iPDS system must be configured with the EMV/iUP adapter option, iPDS-140.

Equipment Supplied

- EMV-51A emulator
- User's manual
- Pocket reference
- EMV-51A software and diagnostic diskette
- 8051 software development package

Emulation Clock Rate

User's system: 1.2 to 12 MHz
EMV-supplied crystal: 12 MHz

Environmental Characteristics

Operating temperature: 0-40° C
Operating humidity: 50-90% RH, non-condensing

Physical Characteristics

Controller: 7.8 in. x 1.5 in. x 5.8 in. (19.8 cm. x 3.8 cm. x 14.7 cm.)

Emulator: 3.3 in. x 3.3 in. x 1.5 in. (8.4 cm. x 8.4 cm. x 3.8 cm.)

Total Weight: 1 lb. 7 oz. (0.65 kg.)

Electrical Characteristics

Power requirements from iPDS: +5V ± 5%
@ 1.9A
*Power requirements from user system: +5V
± 5% @ 200 ma MAX
Characteristics of user socket: Same as 8031,
8051, 8052, or 8751

*The emulator can be strapped to draw its power from either the iPDS unit or the user system.

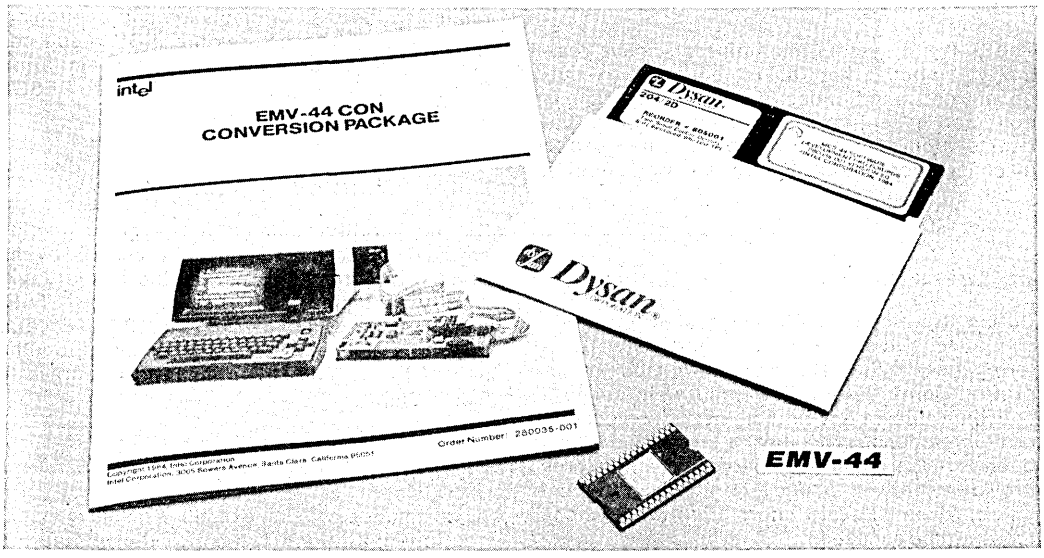
Ordering Information

Part Number	Description
iPDS-EMV-51A	Emulation vehicle for 8051 microcontroller with diskette and documentation

EMV-44 CON
8044 EMULATION VEHICLE
CONVERSION PACKAGE

- **All materials needed to add 8044 support to an EMV-51/51A system**
- **Full-speed, real-time 8044 emulation**
  - **Load, drive, timing characteristics**
  - **Full-speed program RAM**
  - **Serial and parallel ports**
  - **SDLC communications port**
- **Breakpoints/trace**
  - **Four execution address breakpoints**
  - **Range, branch, and value breakpoints**
- **Full symbolic debugging, including support for 8044 expanded symbols**
- **Software debugging with or without user system**
- **Advanced ease-of-use features**
  - **Programmable function keys**
  - **Macros**
- **Help facility tailored for 8044 emulation**
- **Hosted on the Intel Personal Development System (iPDS™)**
- **Use to troubleshoot individual 8044-based designs and complete BITBUS™ system**

The EMV-44 conversion package converts an EMV-51 or EMV-51A emulation vehicle to an EMV-44 emulation vehicle. The resulting EMV-44 system interfaces to any user-designed 8044 system and assists in the debugging and development of the system. (The EMV-44 system cannot be purchased as a separate item; to obtain an EMV-44 system, this EMV-44 conversion package must be used to convert either an EMV-51 or EMV-51A system.) The EMV-44 conversion package consists of a special 8044 component, new development software, and new documentation. To create an EMV-44 system, one needs only replace the special 8051 component in the EMV-51 or EMV-51A system with the new 8044 component, and then install the new software. The EMV-44 accurately emulates the electrical and timing characteristics of the user's 8044. A friendly human interface presents commands in a menu display and organizes commands in an easy-to-learn fashion. The EMV-44 system allows the designer to emulate the specified system's 8044 in real-time or single-step mode. Breakpoints allow the user to stop emulation at user-specified conditions, and trace qualifiers allow for conditional display of trace information. Program memory can be displayed and altered using ASM51 mnemonics and symbolic references. Advanced capabilities allow for programmable keys, macros, and control constructs.



JUNE 1984
Order Number:280035-001

FUNCTIONAL DESCRIPTION

The EMV-44 system provides fundamental capabilities for debugging an 8044. These capabilities are described in the following sections.

Real-Time Breakpoint

The EMV-44 system allows the user system to execute user code at full clock speed, until a predefined condition occurs. The breakpoints may be a combination of four execution addresses or a combination of an execution address range and a single execution address. These break capabilities allow the user to stop the user system at various states in the normal processing cycle and to interrogate the state of the system.

Real-Time Memory

The EMV-44 system uses either the EMV-51 system's 4K or the EMV-51A system's 8K of high speed RAM memory.The RAM can be used to execute the user program and allows easy changes to the user code. The RAM memory can be used either in place of the user's memory before the memory exists in the user system or used in lieu of the user's memory to ease the debugging effort.

Real-Time Trace

The EMV-44 system maintains an active real-time trace buffer that tracks the last two executed addresses in the user's system. The trace is collected in real-time during execution of the user system. This information can be used to discover where the user's program was before it broke.

Software Break

During step mode, the EMV-44 system iteratively single steps, then executes a short software interrogation routine. This slow-down mode of operation continues until a register is set to a specific value, or any branch instruction occurs, or until a specified number of instructions have been executed. These additional break features provide users with added execution control and microprocessor state information in exchange for real-time emulation.

Software Trace

The EMV-44 system will, during interrogation, automatically query the 8044 processor and optionally display up to 4 lines of information. This display can show the execution address, disassembled code, current register values, or processor status information.

COMMANDS

The EMV-44 system has a friendly and easy-to-use human interface, and commands that are well organized and easy-to-learn. Menu displays prompt the user and assist in learning the different commands. Sample EMV-44 displays are shown in Figure 1. Commands fall into four categories: utility commands, display/modify commands, emulation commands, and advanced commands. Once these basic command categories are understood, locating any command becomes simple. Table 1 gives a summary of EMV-44 commands and command categories.

The EMV-44 system is a full symbolic emulator, and hence all commands and displays allow for symbolic entry. Thus the EMV-44 system and users communicate by referring explicitly to symbols defined in the user's source program or symbols defined during the debugging session.

Utility Commands

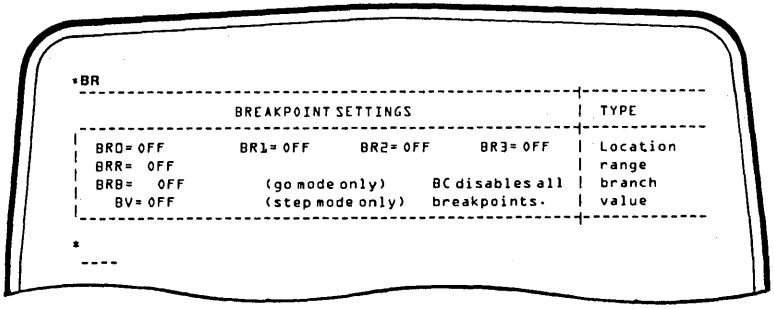
Utility commands perform functions not directly related to the task of emulation and debugging. These commands access the iPDS resources and display information about the emulator. Some examples of utility commands are RESET, LOAD, HELP, and EVALUATE.

Display/Modify Commands

Display/modify commands change or display any register, port, or memory addressable by the 8044 processor chip. Examples of display/-modify commands include REGISTER, ASM/DASM, CBYTE, DBYTE, RBYTE, and PBYTE. A sample of a display resulting from the use of the REGISTER command is shown in Figure 2(a).

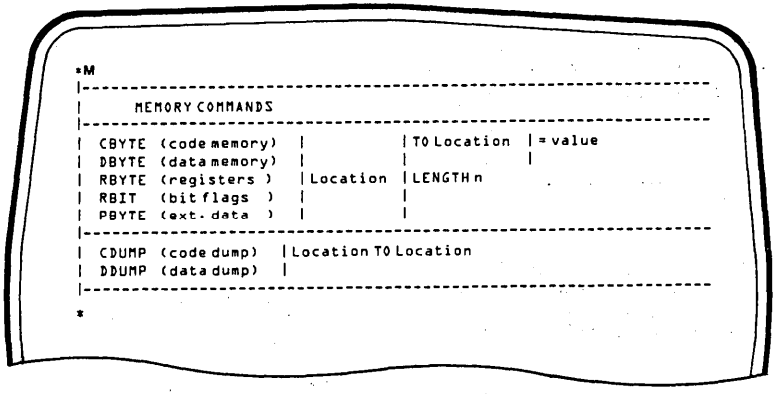
Emulation Commands

All commands causing execution displays, or execution initiation, fall into the emulation



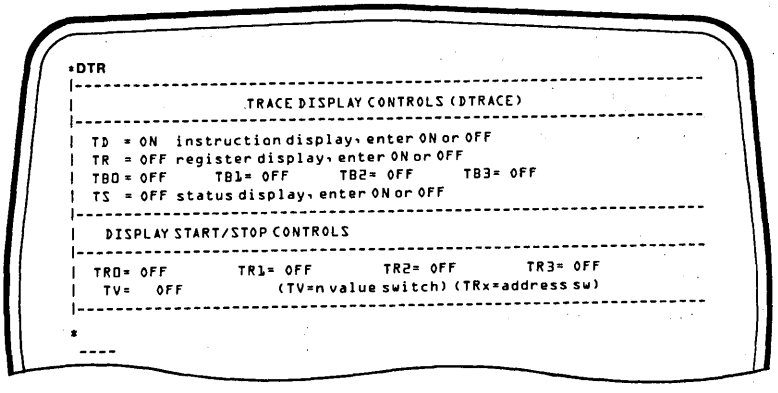
a) Menu Display for Setting Breakpoint

0344



b) Menu Display For Accessing Memory

0330



c) Menu Display For Setting Trace

0348

Figure 1. Typical EMV-44 Menu Displays

280035-001

category. Thus, the GO, BREAK, and TRACE commands are in this category along with BR0,1,2,3, BV, TR0,1,2,3, TS, and STEP.

Advanced Commands

The advanced commands offer the user an easy way to increase the power of the EMV-44 and thus increase the debugging capability of this product. These advanced features allow EMV-44 command sequences to be combined, executed, and stored. Examples of advanced commands include MACRO, FUNCTION, and control constructs. Figure 2(b) shows a display of a macro.

EMULATION MODES

The EMV-44 system combines two approaches to emulation, real-time emulation and software emulation. Programs with time-critical sections of code or critical interrupt routines can be emulated, traced, and debugged in real time.

Real-time emulation supports specific execution breakpoints or range breakpoints. The real-time trace will display up to two instruction addresses last executed. Real-time emulation mode is entered by initiating emulation with the GO command. All break and trace commands associated with the GO command act in real-time emulation mode.

When full-speed emulation is not critical to the debugging effort, the EMV-44 system will emulate one instruction, check for a variety of breakpoint and trace point conditions, display trace information, and continue with another instruction. This slow-down mode of operation provides enhanced break and trace facilities at the expense of non-real-time execution. Slow-down-mode emulation is entered by initiating emulation with the STEP command. Figure 2(a) shows a display for the single-stepping mode.

Table 1. Summary of EMV-44 Commands and Command Categories

Emulation Commands

BREAK - Display breakpoint menu
BR0, 1, 2, 3 - Breakpoint register for execution address
BRR - Breakpoint register for execution range
BRB - Break on branch
BV - Break on value
BC - Clear all breaks
DTRACE - Display trace menu
TB0, 1, 2, 3 - Enable/disable display by bit value
TR0, 1, 2, 3 - Enable/disable display by execution address
TV - Enable/disable display by register value
TR - Enable/disable display of registers
TS - Enable/disable display of PSW
TD - Enable/disable display of code disassembly
STEP - Enter slow-down emulation mode
GO - Enter real-time emulation mode

Advanced Commands

MACRO - Define, and display macro
IF THEN, COUNT, REPEAT, WHILE, UNTIL } Control constructs
FUNCTION - Invoke macro assigned to function key

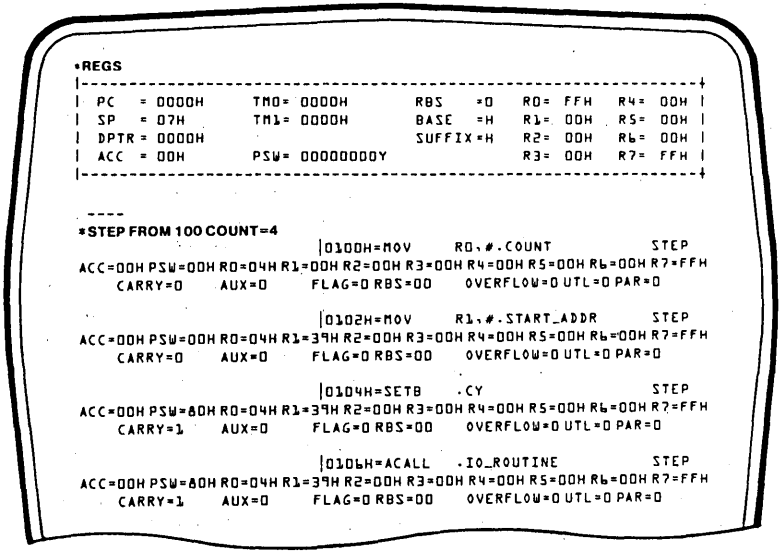
Utility Commands

HELP - Display command syntax
LOAD - Load object file in mapped memory
LIST - Generate copy of emulation work session
DEFINE - Define symbol or macro
SYMBOL - Display symbols
ENABLE/DISABLE - Control for expanded display
EVALUATE - Evaluate any expression
SUFFIX/BASE - Set input and display numeric base
SAVE - Save code memory to file
RESET - Reset emulation processor
EXIT - Terminate EMV-44 session

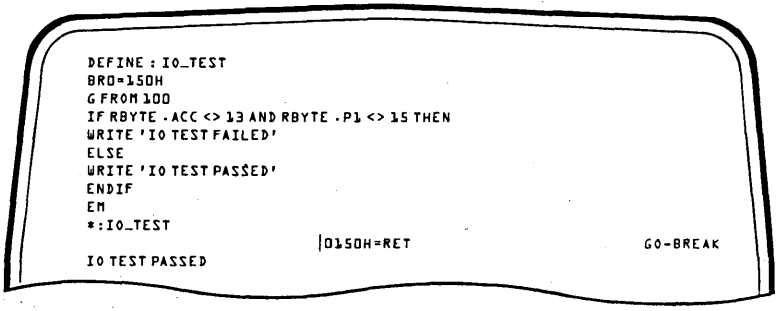
Display/Modify Commands

REGISTER - Change/display 8044 registers
INTERRUPT - Change/display interrupt status
MEMORY - Display menu
CBYTE, DBYTE, PBYTE, RBYTE } Change/display memory
RBIT - Change/display bit memory
CDUMP, DDUMP } Display memory as ASCII and hexadecimal
ASM/DASM - Change/display code memory as assembly language mnemonics

280035-001



a) Display of (1) Registers and (2) Single Stepping through a Portion of a User's Program (Using Symbolics with Selective Trace of Processor and Register Status Information)



b) Display Showing Macro Capability for Debugging System Hardware and Software

1949

Figure 2. Sample Emulation Displays

INTENDED USE

The EMV-44 system is particularly well suited for debugging 8044 designs that include small- to medium-size programs with program complexity that is low to moderate in terms of interrupts, program nesting, and execution flow. In addition to product development, the EMV-44 system is well suited to product testing and servicing. Designs using the BITBUS can be debugged, tested, and serviced while connected to the BITBUS.

FUNCTIONAL DESCRIPTION

The EMV-44 conversion package consists of a special 8044 component, new development software, and new documentation. To create an EMV-44 system, one needs only replace the special 8051 component in the EMV-51 or EMV-51A system with the new 8044 component, and then install the new software. The resulting EMV-44 system has three parts: the controller, the emulator module, and the cable assembly. The controller contains all the logic to support break,

280035-001

trace, emulation, and communication with the host and the emulator module. The emulator module contains the hardware used to execute 8044 code and supplies all MCS®-44 signals to the user's system. This module connects to the controller via a six foot cable, and the controller connects to an iPDS host through the EMV/PROM programming adapter board. This iPDS board is required to use the EMV-44 with the iPDS system.

EMV-44 software contains all the control for user interaction. The software programs the controller, implements all emulator functions, and displays information to the user. This software is run on the iPDS host, and is packaged on a 5-¼ inch diskette. An additional software diagnostic routine, included on the disk, thoroughly checks the EMV-44 hardware.

EMV-44 software will accept and interpret commands entered by the user. These commands will be communicated as a set of micro-commands via a host interface to the controller. Command registers in the controller direct micro-operations to various sections of the break, map, or trace circuitry. Some commands control the emulator board, others determine whether the emulator will emulate the user system, while others interrogate the user system. When appropriate, the controller will pass information back to the host where the information will be processed and displayed to the user. See Fig. 3 for a block diagram of the EMV-44 hardware.

The original EMV-51 or EMV-51A system includes the 8051 Relocating Macro Assembler (ASM51) and the 8051 Linker and Relocater (RL51). The assembler provides full macro capabilities, supports symbolic development for both code development and debugging, and supports modular code development with relocation features. The RL51 utility will relocate, link, and generate loadable object files from the relocatable modules produced by the assembler. EMV-44 fully supports all mnemonics, object file formats, and symbolic references generated by the ASM51 and RL51 programs.

EMV-44 documentation includes a comprehensive user's manual and a command dictionary reference guide.

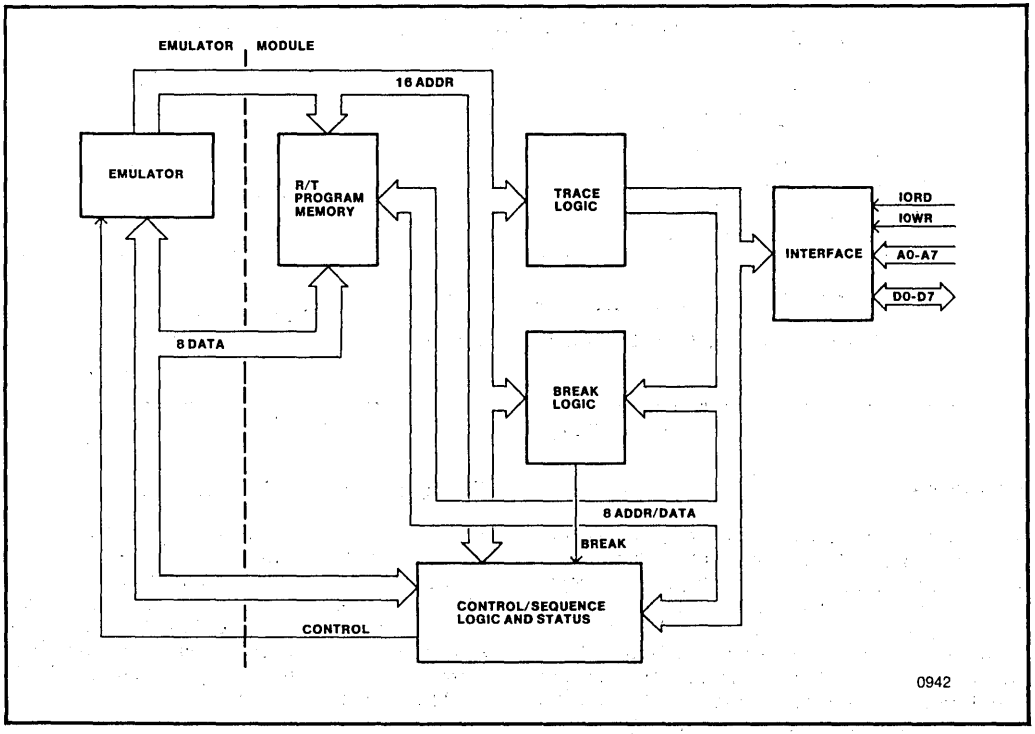


Figure 3. EMV-44 Block Diagram

280035-001

SPECIFICATIONS

EMV-44 Operating Requirements

The EMV-44 system operates with an iPDS system (see Figure 4). The iPDS system must be configured with the EMV/iUP adapter option, iPDS-140.

Equipment Supplied

- 8044 "bondout" microcontroller
- EMV-44 conversion package manual
- EMV-44 software and diagnostic diskette
- EMV-44 label

EMV-44 Emulation Clock Rate

User's system: 1.2 to 12 MHz*
EMV-supplied crystal: 12 MHz

*Note that the bondout 8044 microcontroller supplied with the EMV-44 conversion package has a limitation when serial clock mode 0 is used: the external SCLK signal must be synchronized with the XTAL clock. A simple two flip-flop external circuit can be constructed to provide this synchronization.

EMV-44 Environmental Characteristics

Operating temperature: 0-40° C
Operating humidity: 50-90% RH, non-condensing

EMV-44 Physical Characteristics

Controller: 7.8in. x 1.5 in. x 5.8 in. (19.8 cm. x 3.8 cm. x 14.7 cm.)
Emulator: 3.3 in. x 3.3 in. x 1.5 in. (18.4 cm. x 18.4 cm. x 3.8 cm.)
Total Weight: 1 lb. 7 oz. (0.65 kg.)

EMV-44 Electrical Characteristics

Power requirements from iPDS: +5 V ± 5% @ 1.9A
*Power requirements from user system: +5 V ± 5% @ 200 ma MAX
Characteristics of user socket: Same as 8044, 8344, or 8744

*The emulator can be strapped to draw its power from either the iPDS or the user system.

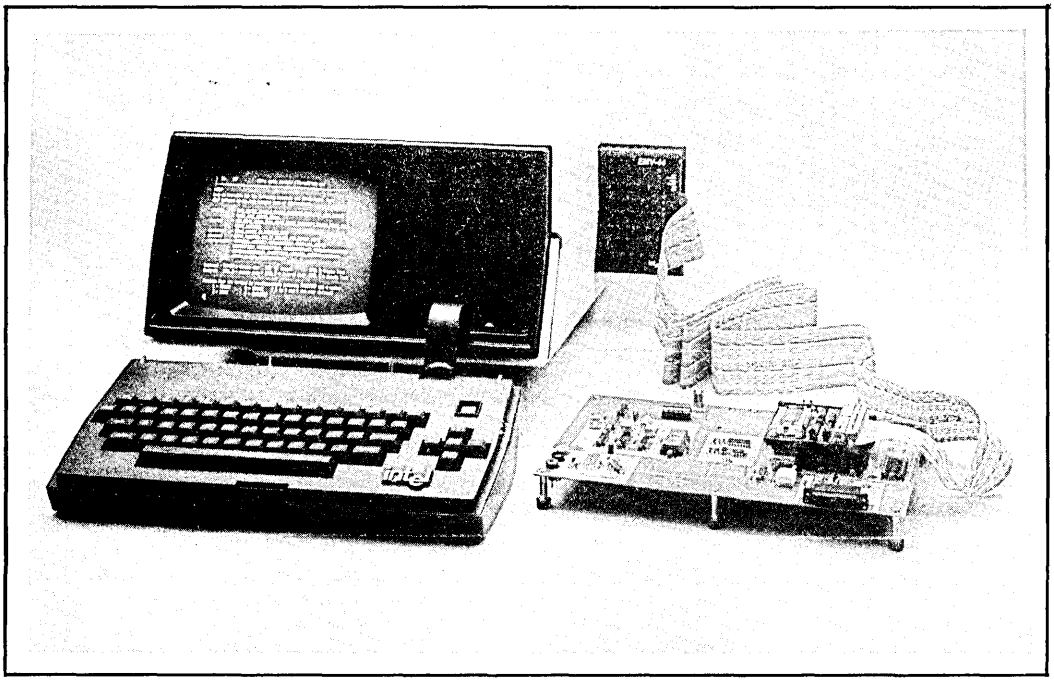


Figure 4. EMV-44 in iPDS™ Debugging Environment

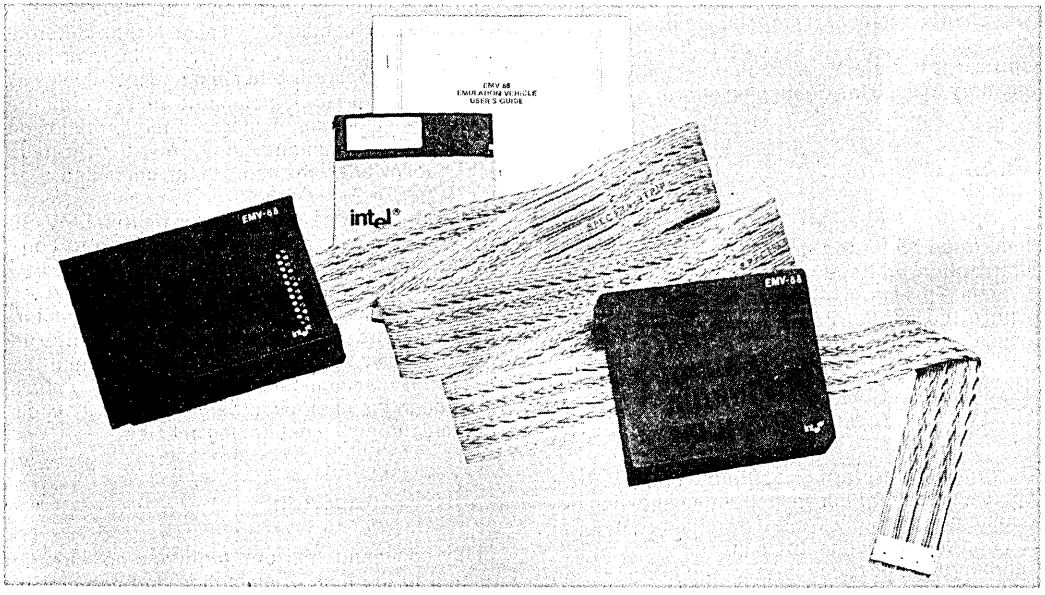
EMV-44 CONVERSION PACKAGE ORDERING INFORMATION

Part Number	Description
iPDS-EMV-44 CON	EMV-44 conversion package with 8044 "bondout" microcontroller, diskette, and documentation

EMV-88
iAPX 8088 EMULATION VEHICLE

- 8 MHz in-circuit emulation
- Hosted on Intel's Personal Development System (iPDS™)
- Advanced easy-to-use features
  - -Programmable function keys
  - -Macros
  - -Loop-control constructs
  - -Instruction disassembly
- Help facility: EMV-88 command syntax reference at console
- Breakpoints
  - -Three modes: execution, data access, or I/O access
  - -One range breakpoint
  - -Externally controlled breakpoints
  - -Break-on-branch capability
- Full symbolic debugging
- 1K byte real-time execution trace
- 4K bytes of on-board zero-wait-state mapped memory
- Software debugging with or without user system
- Includes Macro Assembler ASM 86/88, 8080/8085 to 8086/8088 assembly language source code conversion, LINK 86/88, and EMV-88 control software packages
- Fully supports 8088 RQ/GT, NMI, and Min and Max modes
- Supports PL/M 86/88

The EMV-88 system contains all the hardware, software, and documentation needed to interface to a user-designed iAPX-88 system and assists in the debugging and development of that system; in addition, the system can be used for testing in a manufacturing and/or service environment. The EMV-88 system consists of a buffer box and a controller that is hosted by an Intel Personal Development System (iPDS). The electrical and timing characteristics of the user's 8088 are emulated by the EMV-88 (see page 10 for timing comparisons). Software for the EMV-88 system allows the designer to emulate the user system's 8088 in real-time or single-step mode. Execution breakpoints stop emulation at user-specified conditions, and trace qualifiers control display of trace information. Program memory can be displayed and altered using ASM 86/87/88 mnemonics and symbolic references. Advanced emulator capabilities allow for programmable keys, command macros, and control constructs.



JUNE 1984
ORDER NUMBER:280021-001

FUNCTIONAL DESCRIPTION

The EMV-88 system provides fundamental capabilities for debugging an iAPX-88-based microsystem. These basic capabilities are described in the following sections.

Real-Time Breakpoint

The EMV-88 system allows a user system to execute user code at full clock speed (2 to 8 MHz) until a predefined breakpoint condition occurs. Breakpoints may be specified as a combination of four addresses or a combination of an address range and a single address. Breaks occur on execution addresses, read or write data addresses, read or write I/O port addresses, or on branch. Additionally, an externally supplied signal can cause a break. These break capabilities allow the user to stop the target system during the normal processing cycle and interrogate the state of the target system.

Real-Time Memory

The EMV-88 system supplies 4 Kbytes of high-speed RAM memory mappable on any even 4K-boundary within the 1 megabyte address space of the iAPX-88 microsystem. The RAM can be used to store the user program and make possible changes to user code. The memory can also be used as the user's memory before it exists in the target system, or in place of the user's memory to ease the debugging effort.

Real-Time Trace

The EMV-88 system maintains an active real-time trace buffer that tracks the last 1K byte of instruction addresses executed by the target system. This information can be used to discover where the user's program was before it broke emulation.

Software Break

During single-step execution, the EMV-88 system steps through an instruction and then executes a short software interrogation routine; at the end of the routine, the emulator stops or advances to the next single-step and interrogation cycle. This slow-down mode of emulation continues for a single instruction until a break condition is reached or a specified number of instructions has been executed. This type of emulation provides added execution control and microprocessor state information in exchange for real-time emulation.

Software Trace

Between single steps or after a real-time breakpoint, the EMV-88 system can automatically query the 8088 processor and optionally display up to four lines of information. This display can show execution address, disassembled code, current register values, or processor status information. Users can direct their display screens to present only desired information.

COMMANDS

The EMV-88 system has a friendly, easy-to-use human interface and commands that are well organized and easy-to-learn. Menu displays prompt and assist the user in learning the different commands. Figure 1 shows sample menu displays.

EMV-88 commands fall into four categories: utility commands, display/modify commands, emulation commands, and advanced commands. Once users understand the basic command categories, locating any command becomes simple. These categories, and many of the specific commands, are similar for the different emulation vehicles, and thus the learning time required to operate other emulation vehicles is reduced. The HELP command displays all the EMV-88 commands; if users want information on a particular command, they only need to type HELP followed by the name of the command.

Table 1 provides a summary of the EMV-88 system commands arranged according to command categories.

The EMV-88 system is a full symbolic emulator: all commands and displays can be entered symbolically. Thus the EMV-88 system and the user can communicate by referring to symbols defined in the user's source program or symbols defined during the debugging session.

Utility Commands

Utility commands perform functions not directly related to the task of emulation and debugging. These commands gain access to the IPDS system resources and display information about the emulator.

280021-001

Display/Modify Commands

Display/modify commands change or display any register, port, or memory location addressable by the iAPX-88 target system. These commands provide access to specific areas of the processor or target system and thus minimize extraneous display information.

Emulation Commands

Commands that control program execution or initiate emulation fall into this category; for example, GO, BREAK, and DETRACE.

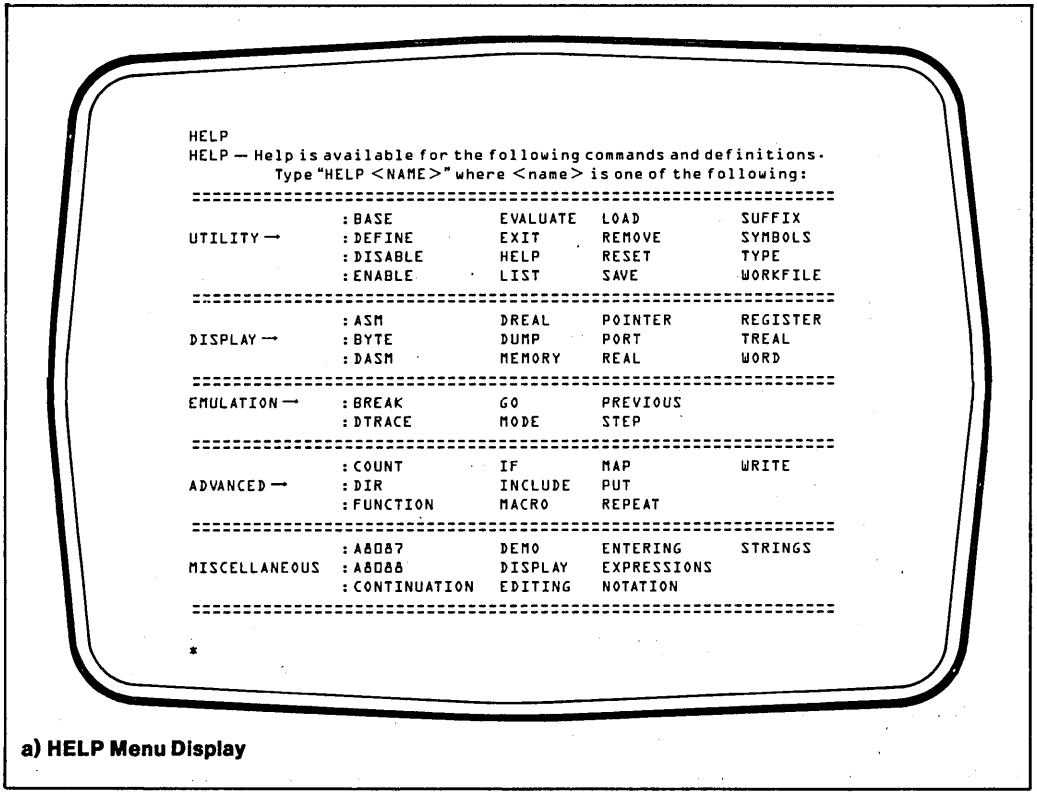
Advanced Commands

The advanced commands offer an easy way to increase debugging capability and create automated test sequences using the EMV-88. The advanced commands permit commands to be combined and saved as test routines. Tests can be developed (with their own messages) that guide the user through a series of diagnostic and troubleshooting paths, making possible easy-to-use fault location down to the system or component level.

EMULATION MODES

The EMV-88 system offers three approaches to emulation: real-time emulation, single-step emulation, and branch-break emulation.

Programs with time-critical sections of code or critical interrupt routines can be emulated, traced, and debugged in real time. Real-time emulation supports specific execution breakpoints or range breakpoints. The real-time trace displays up to 1K byte of the most recently executed instruction addresses. The real-time emulation mode is entered by initiating emulation with the GO command. All break and trace commands that are associated with the GO command act in the EMV-88's real-time emulation mode.



```
HELP
HELP — Help is available for the following commands and definitions.
       Type "HELP <NAME>" where <name> is one of the following:
=============================================================
                :BASE          EVALUATE  LOAD          SUFFIX
UTILITY →       :DEFINE        EXIT      REMOVE        SYMBOLS
                :DISABLE       HELP      RESET         TYPE
                :ENABLE        LIST      SAVE          WORKFILE
=============================================================
                :ASM           DREAL     POINTER       REGISTER
DISPLAY →       :BYTE          DUMP      PORT          TREAL
                :DASM          MEMORY    REAL          WORD
=============================================================
EMULATION →     :BREAK         GO        PREVIOUS
                :DTRACE        MODE      STEP
=============================================================
                :COUNT         IF        MAP           WRITE
ADVANCED →      :DIR           INCLUDE   PUT
                :FUNCTION      MACRO     REPEAT
=============================================================
                :A8087         DEMO      ENTERING      STRINGS
MISCELLANEOUS   :A8088         DISPLAY   EXPRESSIONS
                :CONTINUATION  EDITING   NOTATION
=============================================================

*
```

a) HELP Menu Display

Figure 1. Typical EMU-88 Menu Displays

280021-001

```
*REG
-----------------------------------------------------------------

                        *REGISTER DISPLAY*

     RAX=0031H          RAH=00H          RAL=31H
     RBX=0000H          RBH=00H          RBL=00H
     RCX=1234H          RCH=12H          RCL=34H
     RDX=0000H          RDH=00H          RDL=00H

     SP=0FFFH       DP=1000H       SI=0000H       DI=0000H
     CS=FFFFH       DS=0000H       SS=0000H       ES=0000H

     IP=0000H

     RF=F002H          OF=0   DF=0   IFF=0   TF=0
                       SF=0   ZF=0    AF=0   PF=0    CF=0

-----------------------------------------------------------------

--------
*
```

b) REGISTER Display

```
*DTR
-----------------------------------------------------------------
                      *TRACE DISPLAY CONTROLS*
-----------------------------------------------------------------

TD= OFF                (instruction display,  ON/OFF)
TR= OFF                (register    display,  ON/OFF)
TS= ON                 (status      display,  ON/OFF)

-----------------------------------------------------------------
                    *DISPLAY START/STOP CONTROL*

TR0=00034H ON       TR1=00490H OFF    TR2= OFF        TR3=20035H ON
                      ( enter both location and ON/OFF switch )
 TV= OFF
                      ( enter REGISTER VALUE SW )


-----------------------------------------------------------------

--------
*
```

c) Menu Display for Setting Trace

Figure 1. Typical EMU-88 Menu Displays (continued)

280021-001

Table 1. Summary of EMV-88 Commands and Command Categories

Commands	Description
Utility Commands	
DEFINE	Defines symbol or macro.
DOMAIN	Establishes default module.
ENABLE/DISABLE	Control for expanded display.
EVALUATE	Evaluates any expression (numerical or logical).
EXIT	Terminates EMV-88 session.
HELP	Displays command syntax.
INCLUDE	Loads a macro definition or a command file.
LINE	Displays statement numbers and associated absolute addresses.
LIST	Generates copy of emulation work session.
LOAD	Loads object file in mapped memory.
MODULE	Displays module names in EMV-88 module table.
REMOVE	Deletes symbol or macro.
RESET	Resets emulation processor.
SAVE	Saves memory to file.
SYMBOLS	Displays symbols.
SUFFIX/BASE	Sets input and displays numeric base.
TYPE	Sets/displays data type to symbol name.
Emulation Commands	
BR	Displays breakpoint menu.
BR0, 1, 2, 3	Breakpoint register for execution address.
BRB	Breaks on branch.
BRR	Breakpoint register for execution range.
BV	Breaks on value.
BC	Clears all breaks.
DTRACE	Displays trace menu.
GO	Enters real-time emulation mode.
MO	Break qualifier.
PREVIOUS	Displays execution trace.
STEP	Enters slow-down emulation mode.
TD	Enables/disables display of code disassembly.
TR	Enables/disables display of registers.
TR0, 1, 2, 3	Enables/disables display by execution address.
TS	Enables/disables display of PSW.
TV	Enables/disables display by register value.
Display/Modify Commands	
ASM/DASM	Changes/displays code memory in assembly language mnemonics.
DUMP	Displays memory as ASCII and hexadecimal.
MEMORY	Displays menu for memory access.
PORT	Changes/displays ports.
REGISTER	Displays 8088 registers menu.
BYTE, WORD, POINTER, SINTEGER, INTEGER	Change/display memory.
REAL, TREAL, DREA	8087 commands.

280021-001

Table 1. Summary of EMV-88 Commands and Command Categories (continued)

Commands	Description
Advanced Commands	
DIR	Displays names of all available macros.
FUNCTION	Invokes macro assigned to function key.
MACRO	Displays macro text.
MAP	Sets/displays memory map.
PUT	Stores macro definitions.
WRITE	Evaluates and displays expressions and strings.

IF THEN
COUNT
REPEAT
WHILE
UNTIL
	Loop control constructs.

When full-speed emulation is not critical to the debugging effort, the EMV-88 system emulates one instruction, checks for a variety of break-point and trace-point conditions, displays trace information, and continues with another instruction. This slow-down mode of operation permits an enhanced non-real-time execution break and trace facility. The STEP command is used to enter the slow-down emulation mode.

A third mode, branch break, bridges the gap between real-time emulation and single-step emulation. During branch break, the EMV-88 system emulates in either real-time mode or single-step mode until any branch instruction is executed. After the branch has executed, the emulator breaks emulation. Thus this mode makes possible a fast and convenient mechanism for observing program flow.

INTENDED USE

The EMV-88 system is designed for debugging certain kinds of programs: it is particularly well suited to assist in debugging small- to medium-sized programs whose complexity is low to moderate in terms of interrupts, program nesting, and random execution flow. It is also designed for testing and troubleshooting in manufacturing and service environments.

Figure 2 shows the EMV-88 system and the IPDS in a debugging environment.

PHYSICAL DESCRIPTION

Hardware Components

The EMV-88 hardware consists of two parts: the controller and the buffer box. The controller contains all the logic to support mapped memory, break, trace, emulation, and communication with the host. The controller module is inserted in the side panel of the iPDS host development system and connects to the EMV/PROM programming adapter board, an option that must be present for EMV use. A five-foot cable connects the controller module to the buffer box. The buffer box buffers all iAPX-88 signals and contains the emulating 8088-2 processor and all logic to support both Min and Max modes of operation. The buffer box plugs into the user's 8088 socket by means of a short cable and supplies all iAPX-8088 signals to the target system.

Software Components

The EMV-88 software offers extensive emulation control of the target CPU. The software programs the controller, implements all emulator functions, displays information, and offers advanced features to further support debugging activities. Software is run on the iPDS host, and is packaged on a 5-¼ inch diskette. An additional software diagnostic routine, included on the disk, thoroughly exercises the EMV-88 hardware.

280021-001

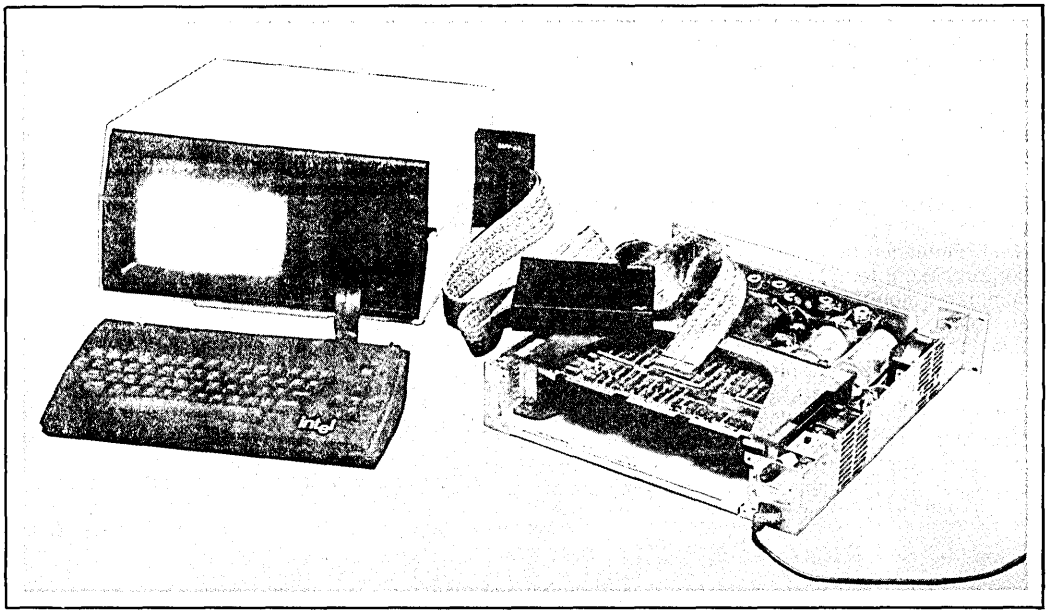


Figure 2. The EMV-88 System and the iPDS™ System in a Debugging Environment

The EMV-88 also includes the 8086/8088 Macro Assembler (ASM 86/88) and a software utility package. The macro assembler provides full macro capabilities, supports symbolic reference during code development and debugging, and supports modular code development and relocation features. The utility package includes the following software:

- CONV 86/88 to convert ASM 80/85 programs to ASM 86/88 programs
- LINK 86/88 to combine several ASM 86/88 object modules into a single object module
- LIB 86/88 to manage libraries of 8086/8088 object modules
- LOC 86/88 to locate relocatable modules to absolute executable addresses
- OH 86/88 to convert executable modules from object module format to hexadecimal format

Documentation

The EMV-88 system includes a documentation kit containing a comprehensive *EMV-88 Emulation Vehicle User's Guide* and an *EMV-88 Emulation Vehicle Pocket Reference*. In addition, extensive documentation is included for the Macro Assembler and the utilities software.

System Operation

A multitude of hardware and software interactions allow the EMV-88 to provide break, trace, map, and interrogation features. The iPDS host maintains symbol tables (from information passed to it from Intel language translators) and user symbols (defined during the debugging session). The symbol tables allow the EMV-88 to provide full symbolic debugging capabilities.

For the command monitor, the EMV-88 system requires 1K byte of user processor address space and six bytes of the user stack. The EMV-88 system supplies this 1K byte of memory and the necessary logic to map this memory anywhere in the user's decoded memory space on any 1K-byte boundary. During initialization, the EMV-88 system loads the command memory with the monitor software. This monitor routine has responsibility for interrogating the 8088 processor, changing any register or memory associated with the target system, and executing user code.

For example, during a memory display operation, the EMV-88 monitor causes the desired data in the memory to be passed to the iPDS host. The iPDS host takes the data, formats the data, associates symbolic information with the data, and displays the information for the user.

The address of the first byte of every executed target system instruction is stored in a 1K-byte circular trace buffer. During interrogation, this buffer can be read and displayed by the user. Trace information is passed back to the host where it is symbolically expanded, disassembled, and displayed.

Breakpoints are implemented by setting bits (associated with addresses) in a breakpoint RAM. Emulation begins by directing the emulation processor to execute user code rather than command-monitor code. Execution addresses are tracked through a queue emulator and, when a match between an execution address and a set bit in the breakpoint RAM is found, the EMV-88 system asserts the NMI line (after the execution of the instruction) to cause a break. Control is then passed to the software command monitor, which notifies the iPDS host of the break event. See Figure 3 for a block diagram of the EMV-88 hardware.

SPECIFICATIONS

EMV-88 Operating Requirements

The EMV-88 system operates with an IPDS system. The iPDS system must be equipped with the EMV/iUP adapter option (the iPDS-140).

EMV-88 System Specifications

Total access to 1M byte of iAPX-88 address space (except for 1K byte of EMV command memory)

Compatible with all Intel Series-II-based iAPX software

Can load a 4K-byte object file with symbols in less than one minute

Step mode operates at 5000:1 speed slow-down

Equipment Supplied

EMV-88 emulator

EMV-88 software and diagnostic diskette

8086/8088 Macro Assembler and utility package

Documentation Supplied

EMV-88 Emulation Vehicle User's Guide

EMV-88 Emulation Vehicle Pocket Reference

Documentation for the 8086/8088 Macro Assembler and utilities software

Emulation Clock Rate

EMV-88-resident clock: 5 MHz
User-supplied clock: 2 to 8 MHz

DC Characteristics

1. Output Low Voltage [V_{OL}(Max) = 0.5 V]

	I_{OL} (Min)
AD0-AD7, A8-15	24 mA
A16/S3-A10/S6, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, IO/$\overline{M}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$	24 mA
HLDA, '$\overline{RQ}$/$\overline{GT}$	0.5 mA

2. Output High Voltage [V_{OH}(Min) = 2.0 V]

	I_{OH} (Min)
AD0-AD7, A8-15	−15 mA
A16/S3-A19/S6, $\overline{SSO}$, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, IO/$\overline{M}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$, HLDA	−15 mA
$\overline{RQ}$/ $\overline{GT}$	1.03 mA

3. Input Low Voltage [V_{IL}(Max) = 0.8 V]

	I_{IL} (Max)
AD0-AD7	−0.1 mA
NMI, CLK	−0.1 mA
READY	−0.4 mA
INTR, HOLD, $\overline{TEST}$, RESET	−2.0 mA
MN/ $\overline{MX}$ (0.1 μf to GND)	−0.1 mA

280021-001

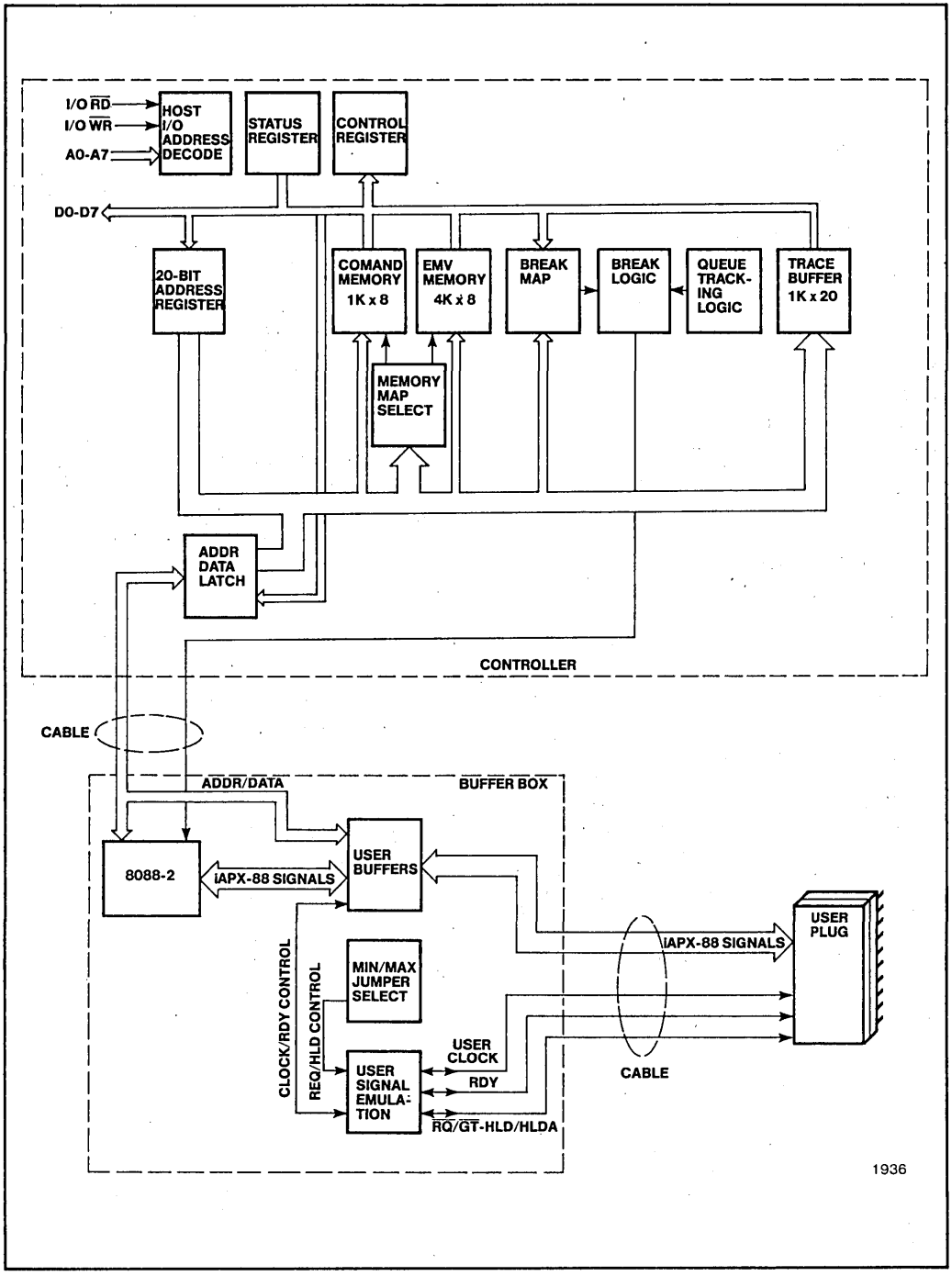


Figure 3. EMV-88 Block Diagram

280021-001

4. Input High Voltage [V_{IH}(Min) = 2.0 V]

	I_{IH} (Max)
AD0-AD7	.02 mA
NMI, CLK	.02 mA
READY	.02 mA
INTR, HOLD, $\overline{TEST}$, RESET	0.4 mA
MN/$\overline{MX}$	.02 mA

Output Drive Capacity

The EMV-88 module has a greater output drive capacity than the 8088 chip, except for the $\overline{RQ}$/$\overline{GT}$ output when $\overline{GT}$ is active.

EMV-88 User Cable

Capacitance: 22 pf/ft

Impedance: 105 ohms

Capacitive Loading

The EMV-88 module presents the user system with a maximum load of 34 pf and 0.8 mA.

All EMV-88 outputs are capable of driving a minimum of 15 pf and 20 mA while meeting all the probe's timing specifications. The EMV-88 module will drive larger capacitive loads but with possible performance degradation.

Timing Differences Between the 8088-2 Microprocessor and EMV-88 Emulation

MIN Mode

Symbol	Parameter	8088-2		EMV-88	
		Min ns	Max ns	Min ns	Max ns
TDVCL	Data in setup time	20		30	
TRYHCH	Ready setup time into 8088	68		86	
THVCH	HOLD setup time	20		7.5	
TINVCH	Setup time for recognition				
INTR		15		19	
NMI		15		78	
$\overline{TEST}$		15		77	
TCLAV	Address valid delay	10	60	24	74
TCHLAV	HLDA valid delay	10	100	18	118
TCHCTV	Control active delay	10	60	29	85
TCVCTV	Control active delay 1	10	70	22	80

MAX Mode

Symbol	Parameter	8088-2		EMV-88	
		Min ns	Max ns	Min ns	Max ns
TDVCL	Data in setup time	20		30	
TRYHCH	Ready setup time into 8088	68		86	
TINVCH	Setup time for recognition				
INTR		15	19		
NMI		15	78		
$\overline{TEST}$		15	77		
TCLAV	Address valid delay	10	60	24	74
TCLDV	Data valid delay	10	60	24	74
TCLRL	RD active delay	10	100	24	116
TGVCH	RQ/GT setup time	15		25	
TCHSV	Status active delay	10	60	24	70

280021-001

Environmental Characteristics

Operating temperature: 50°-95° F (10°-35° C)

Operating humidity: 0-90% relative humidity, non-condensing

Physical Characteristics

Controller: 7.75 in. x 1.5 in. x 5.75 in. (19.7 cm. x 3.8 cm. x 14.6 cm.)

Buffer box: 7.25 in. x 1.5 in. x 5.75 in. (18.4 cm. x 3.8 cm. x 14.7 cm.)

Total weight: 2 lb. 6 oz. (1018 grams)

Electrical Characteristics

Power required from the iPDS system: +5 V ±2.5% @ 2.5A (includes emulator requirements)

ORDERING INFORMATION

Part Number	Description
iPDS-EMV-88	Emulation vehicle for the iAPX-88 microsystem (includes diskette and documentation)

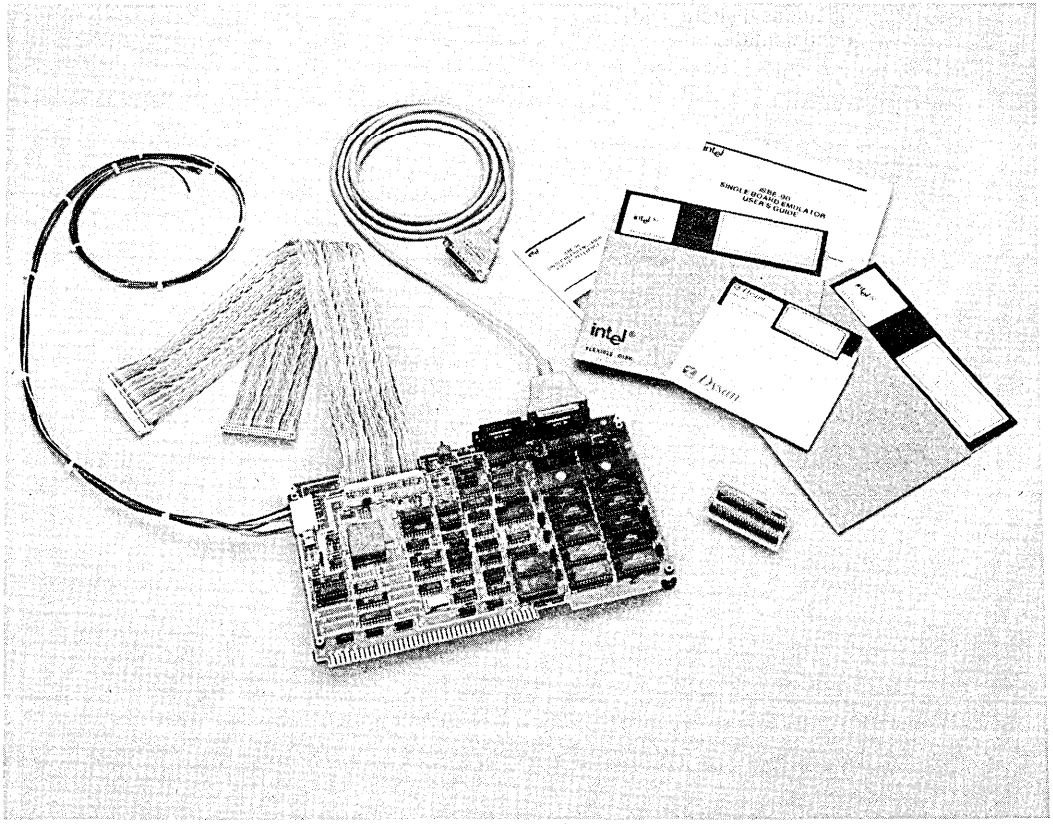
280021-001

iSBE-96
SINGLE BOARD EMULATOR FOR THE MCS®-96 FAMILY OF MICROCONTROLLERS

- **Eight software execution breakpoints that can selectively be turned on and off**
- **12-MHz emulation speed**
- **Configurable serial I/O**
- **17.75K of on-board user memory**
- **Optionally expandable to 64K of on-board user memory**

The iSBE-96 emulator supports the execution and debugging of programs for the MCS®-96 family of microcontrollers at speeds up to 12 MHz. The MCS-96 family configurations are shown in Table 1. The iSBE-96 emulator consists of an 8097 microcontroller, a serial port and cable, and an EPROM-based monitor that controls emulator operation and the user interface.

The iSBE-96 emulator is a combination of hardware and software that permits programs written for the MCS-96 family of microcontrollers to be run and debugged in the emulator's artificial environment or in the user's prototype system. As a result, development time can be reduced by the early integration of hardware and software.



SEPTEMBER 1984
ORDER NUMBER: 231015-002

FUNCTIONAL DESCRIPTION

Integrated Hardware and Software Development

The iSBE-96 emulator allows hardware and software development to proceed simultaneously. This approach is more time- and cost-effective than the alternate method: independent hardware and software development followed by system integration. With the iSBE-96 emulator, prototype hardware can be added to the system as it is designed; software and hardware integration occurs while the product is being developed. The emulator aids in the recognition of hardware and software problems.

Emulation is the controlled execution of the prototype software in the prototype hardware or in an artificial hardware environment that duplicates the microcontroller of the prototype system. The iSBE-96 emulator permits reading and writing of system memory, and control of program execution. The emulator also allows interactive debugging of the prototype software and can externally control program execution while operating in the prototype system. When the prototype system memory is not yet available, the iSBE-96 emulator's on-board memory permits software debugging.

Table 1. Configurations of the MCS®-96 Family of Microcontrollers

		68 Pin	48 Pin
Digital I/O	ROMLESS	8096	8094
	ROM	8396	8394
Analog and Digital I/O	ROMLESS	8097	8095
	ROM	8397	8395

iSBE-96 Software

The iSBE-96 emulator is shipped with three software disks: an 8 in. double-density and an 8 in. single-density disk for use with an Intellec® Series II/III, and a 5-1/4 in. disk for use with the Series IV development system.

The iSBE-96 emulator is supplied with an ISIS driver routine that communicates with the monitor software on the iSBE-96 emulator board. The driver interrupts the 8097 using the non-maskable interrupt (NMI) line for incoming keyboard input. The commands associated with the driver and the monitor are described in the following sections.

ISIS DRIVER

The iSBE-96 emulator is shipped with the ISIS driver software for use on the Series II, III, or IV development systems. The driver software provides a few easy-to-use commands. These commands are described in Table 2.

Table 2. ISIS Driver Commands

Driver Command	Function
EXIT	Exits the ISIS driver and returns to the ISIS operating system.
<CONTROL> C	Same as for the EXIT command.
HELP	Displays the syntax of all commands.
INCLUDE	Specifies a command file.
<CONTROL> I	Turns the command file on and off.
<TAB>	Same as <CONTROL> I (turns the command file on and off).
LIST	Specifies a list file.
<CONTROL> L	Turns list file on and off.
<CONTROL> S	Stops scrolling of the screen display.
<CONTROL> Q	Resumes scrolling of the screen display.
<CONTROL> X	eletes the line being entered.
<ESCAPE>	Aborts the command executing.

iSBE-96 MONITOR

The iSBE-96 monitor performs the following functions:

- Loads and saves user programs.
- Independently emulates user programs.
- Examines and changes memory contents.
- Examines registers.
- Maps the file capabilities of the serial ports (DS/DT).
- Maps different memory configurations.

The monitor commands are described in Table 3.

231015-002

Table 3. iSBE-96 Monitor Commands

Monitor Command	Function
BAUD	Sets up the baud rate.
BR	Permits display and setting of up to eight software breakpoints.
BYTE	Permits display and changing of a single byte or range of bytes of memory or a single byte of the 8097 internal registers.
CHANGE	Permits display and changing of a series of memory words or bytes.
<CONTROL> S	Stops scrolling of the screen display.
<CONTROL> Q	Resumes scrolling of the screen display.
<CONTROL> X	Deletes the line being entered.
<ESCAPE>	Aborts the command executing.
GO	Begins emulation and continues until an enabled breakpoint is reached or the escape key is pressed.
LOAD	Loads programs and data from disks.
MAP	Permits mapping of several preprogrammed memory maps; also permits configurable serial I/O and selective servicing of the watchdog timer.
PC	Displays and changes the program counter.
PSW	Displays and changes the program status word.
RESET CHIP	Resets the 8096 to power-up conditions.
SAVE	Saves programs and data to disks.
SP	Displays and changes the stack pointer.
STEP	Provides single-step emulation with selective display formats.
VERSION	Displays the monitor version number.
WORD	Permits display and changing of a single word or range of words of memory or a single word of the 8097 internal registers.

Integrating Hardware and Software

When the prototype system hardware is developed, the iSBE-96 emulator interfaces to the prototype through two 50-pin ribbon cables. The emulator can then execute code from the iSBE-96 on-board RAM (or from user-provided memory) and exercise the prototype system hardware.

BLOCK DIAGRAM

Figure 1 is a block diagram showing the iSBE-96 emulator. The following sections describe each block.

The Processor

The 68-pin processor of the iSBE-96 emulator is used only in the 8097 external-access mode.

No adapter board is provided for the 68-pin versions of the 8096 and 8097 microcontrollers.

When debugging a 68-pin package, the two 50-pin ribbon cables are available to plug directly into 50-pin connectors on the user's prototype system.

When debugging a 48-pin package, the two 50-pin cables plug into the 48-pin adapter board, which is then plugged into a 48-pin socket in the prototype system.

iSBE-96 Emulator I/O

The iSBE-96 emulator's memory-mapped I/O devices are used to manage the system. These I/O devices are mapped into memory between locations 01F00H and 01FFFH.

Included as part of the I/O are two serial ports. One is configured as data set (DS) and the other as data terminal (DT). When operating with an Intellec development system, the data set port is used as the system console and the link for exchanging files.

231015-002

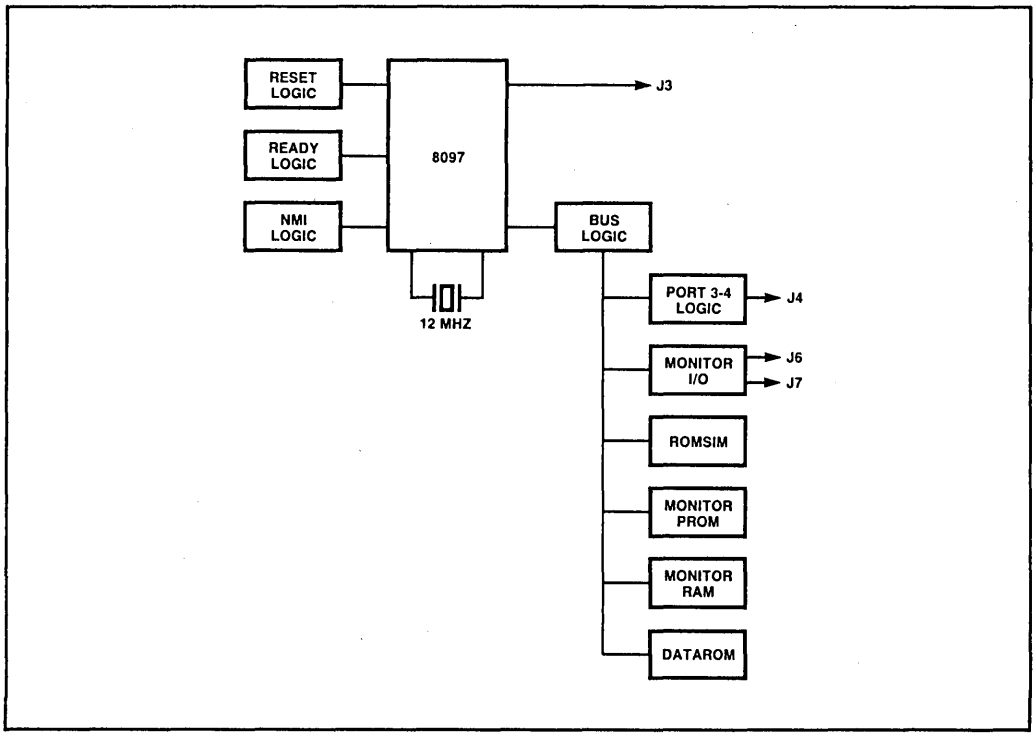


Figure 1. Block Diagram for the ISBE-96 Single Board Emulator

The serial ports are serviced under control of the NMI interrupt. The NMI interrupt has highest priority on the microcontroller and interrupts the user program when characters are entered from the keyboard. When in emulation, the monitor will still service inputs from the keyboard and execute certain monitor commands. Monitor activity is not totally transparent to the user.

Simulated ROM (ROMSIM)

There are eight 28-pin JEDEC byte-wide sockets with 2K-by-8 static RAMS present on the board. The partition on the user's prototype system that will be ROM is simulated by RAM on the iSBE-96 emulator board. This RAM facilitates easy program development, allowing users to correct and test problems in their programs.

ROMSIM can be expanded by replacing the iSBE-96 RAMs with 8K-by-8 static RAMs.

Port 3-4 Logic

The port 3-4 logic has two functions: to provide bus expansion and to provide I/O ports. The port 3-4 logic is controlled by a software switch available with the MAP command.

The iSBE-96 emulator reconstructs ports 3 and 4 of the 8394, 8395, 8396, and 8397 microcontrollers when the logic is defined by the MAP command as port 3-4. This port function should be selected when one of these four microcontrollers is intended as the target microcontroller.

When the BUS switch of the MAP command is specified, the iSBE-96 address/data expansion bus is available to the prototype system.

THE iSBE-96 EMULATOR MEMORY MAP

The target system should be designed with a memory map that is compatible with one of the *iSBE-96 memory maps. Figure 2 shows the* default address mapping. The following sections describe the areas of memory.

231015-002

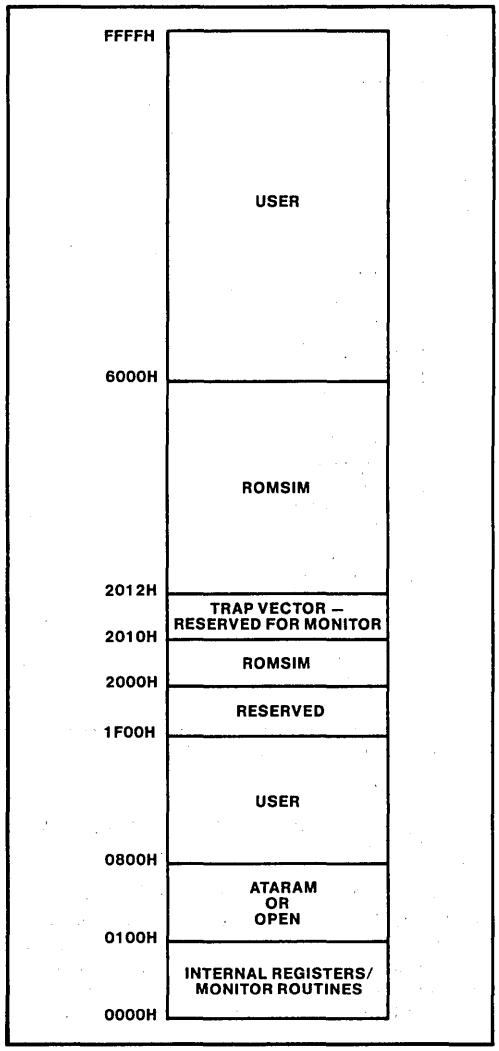


Figure 2. iSBE-96 Emulator Default Mapping

Internal Registers/Monitor Routines

Normally locations 000H through 0FFH contain the internal register space of the 8097. However, instruction fetches from these locations access external memory. This memory space contains the monitor's non-maskable interrupt service routine and utility routines.

For the monitor to access the user memory, the address and data is passed to the interrupt or utility routines. The routines then modify the mode register to enable user memory, disable all of the monitor's memory (except page zero), and perform the appropriate operation. After an operation is complete, the service and utility routines restore the mode register to its previous state and return to the main monitor code. The NMI service routine is used to handle the keyboard input on the serial port.

DATARAM

Locations 100H to 7FFH are mapped as the DATARAM space. This RAM is for general purpose use and is optionally enabled by using the MAP command. When the DATARAM buffer is not enabled, any access to this partition results in an access to user prototype system memory.

User Area

Locations 800H to 1EFFH are a user area. If an access is made to this partition, it is directed to the user's prototype system. Any memory mapped as I/O in the user system should be placed in this partition. With 8K-by-8 static RAMs, this area is located and available on the iSBE-96 board.

Reserved Area

Locations 1F00H to 1FFFH are reserved by the monitor for on-board I/O devices.

ROMSIM

Because some of the MCS-96 family of microcontrollers are ROMless parts, a user program can be loaded for execution into the on-board RAMs of the iSBE-96 emulator. Locations 2000H to 5FFFH are mapped to this RAM space; the space is called ROMSIM.

Trap Vector

Locations 2000H to 2010H are the interrupt vector locations. Vector address location 2010H is used by the iSBE-96 monitor for NMI.

User Area

The partition 6000H to 0FFFFH is mapped to the user proptotype area. During emulation any access to this partition is directed to the user's prototype system.

231015-002

EXPANDING ON-BOARD MEMORY

On-board memory can be expanded to a full 64K bytes by replacing the supplied 2K-by-8 static RAMs with 8K-by-8 static RAMs or PROMs. The user may also replace on-board ROMSIM memory with 2K-by-8 PROMs or even locate all 64K bytes of memory on the prototype system.

DESIGN CONSIDERATIONS

Designers should note the following considerations for designing with the iSBE-96 emulator:

- The iSBE-96 software uses 6 bytes of user stack space.
- Analog signal accuracy is impaired when driven over the emulator cable (up to ±50 mv loss of A/D conversion accuracy).
- The iSBE-96 emulator has some ac/dc parametric differences from the 8097 chip.
- The NMI vector is used for console service (Intel reserved interrupt).
- Keyboard activity during emulation affects real-time emulation because a 50 to 100 microsecond interrupt service routine is executed for every keystroke.
- The only hardware reset available for the iSBE-96 emulator is the system reset momentary switch (switch 1 on the emulator board).
- The prototype system interface cable is designed to support only the 48-pin package directly. Support for the 68-pin package is accomplished through the two 50-pin ribbon cables provided.
- User system memory should be configured to the iSBE-96 memory map (see Figure 2).
- The iSBE-96 emulator will not operate from a user system crystal.
- The iSBE-96 driver software is not compatible with the Intellec Model 800 development system.

SPECIFICATIONS

Equipment Supplied

Standard MULTIBUS®-size board assembly

EPROM-based monitor

Auxiliary power cable

RS-232 serial cable

Two standard, 18 in., 50-pin ribbon cables for connection to the user's prototype system

An adapter board for the 48-pin version of the MCS-96 microcontroller

One 8 in. single-density software disk for the Series II and III

One 8 in. double-density software disk for the Series II and III

One 5-1/4 in. software disk for the Series IV

Documentation

iSBE-96 User's Guide (Order number 164116)
iSBE-96 Pocket Reference (Order number 164157)

Emulation Clock

12 MHz supplied crystal

Physical Characteristics

Width: 6.75 in. (17.15 cm)
Length: 12 in. (30.48 cm)
Height: 0.75 in. (1.91 cm)

DC Electrical Requirements

Voltage	Current
+5v±5%	3.5a max
+12v±5%	0.06a max
−12v±5%	0.05a max

Environmental Characteristics

Operating Temperature: 10° to 40° C

Operating Humidity: 10% to 85% relative humidity, without condensation

231015-002

ORDERING INFORMATION

Part Number	Description
iSBE-96	Single board emulator for the MCS®-96 family of microcontrollers; with disks and documentation.

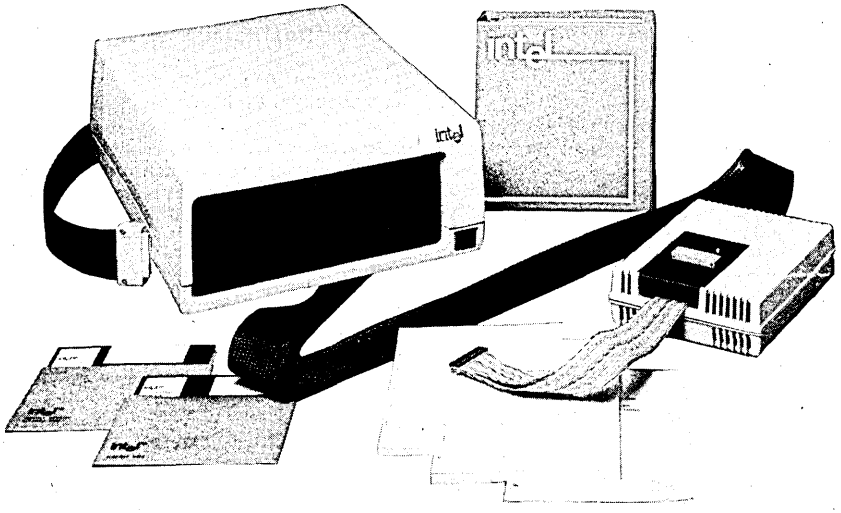
I²ICE™ INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM

PRELIMINARY

- Provides real-time in-circuit emulation.
  - Full speed, real-time emulation for Intel iAPX microprocessors.
- Offers symbolic debugging capabilities.
  - Accesses memory locations and program variables (including dynamic variables and high-level language data structures) using program-defined names.
  - Maintains a virtual symbol table for program variables.
- Breaks emulation when a specified event or combination of events occurs.
- Maintains a 1023-frame trace buffer.
  - Collects trace data in real-time.
- Provides low cost conversions among iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286 microprocessors.
- Simultaneously controls up to four iAPX microprocessors for debugging multiprocessor systems.
- Provides an integrated 16-channel 100 MHz logic timing analyzer.
- Maps user program memory into zero-wait-state RAM.
  - Wait-states are programmable from zero to 15 machine cycles.
  - Memory is expandable from 32K bytes to 288K bytes.
  - I²ICE software does not intrude into user space.
- Provides disassembly and single-line assembler to help with on-line code patching.

The Intel Integrated Instrumentation and In-Circuit Emulation (I²ICE™) system aids the design of systems that use the iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286 microprocessors. The I²ICE system combines symbolic software debugging, in-circuit emulation, and the optional Intel Logic Timing Analyzer (ILTA). The I²ICE system supports programs written in PL/M-86, FORTRAN-86, Pascal-86, and assembly language.

One of Intel's Intellec® microcomputer development systems (for example, the Series IV Development System) hosts the I²ICE system.



OCTOBER, 1983
ORDER NUMBER: 210469-004

PHYSICAL DESCRIPTION

I²ICE hardware consists of the host interface board, the I²ICE instrumentation chassis, the emulation base module, the emulation personality module, a host/chassis cable, inter-chassis cables (for multiple chassis systems), a user cable, optional high-speed memory, and an optional logic timing analyzer. I²ICE software consists of I²ICE host software, I²ICE probe software, confidence tests, PSCOPE-86, and optional iLTA software (see Table 1).

The host interface board resides in the host development system. A cable connects the host interface board to the I²ICE instrumentation chassis. Another cable connects the I²ICE instrumentation chassis to the buffer box.

The instrumentation chassis contains high-speed zero-wait-state emulation memory, break-and-trace logic, memory and I/O maps, and the emulation clips assembly.

The chassis may also contain the optional logic timing analyzer and optional high-speed memory. High-speed memory is expandable from 32K bytes to 160K bytes.

The buffer box contains the emulation personality module. This module configures the I²ICE system for a particular iAPX microprocessor. The user cable connects the buffer box to user prototype hardware.

The host development system may host up to four I²ICE instrumentation chassis. Each chassis may have its own buffer box, user cable, emulation clips, and logic timing analyzer.

FUNCTIONAL DESCRIPTION

The I²ICE™ Memory Map

The I²ICE system can direct (map) an emulated microprocessor's memory space (the user program memory) to any combination of the following:

- High-speed I²ICE memory. This consists of 32K bytes of programmable wait-state memory (programmable from 0 to 15). This memory resides in the I²ICE chassis on the map-I/O board.
- Optional high-speed I²ICE memory. This consists of 128K bytes of programmable wait-state memory. This memory resides in the I²ICE chassis on an optional high-speed memory board.
- MULTIBUS® memory (host system memory). This resides in the host development system itself.
- User memory. This resides in the user prototype hardware.

When a user program runs in I²ICE memory or user memory, the I²ICE system emulates in real time. A memory access to MULTIBUS memory, however, inserts approximately 25 wait-states into the memory cycle.

Resource Borrowing

The I²ICE memory map allows the prototype system to borrow memory resources from the I²ICE system.

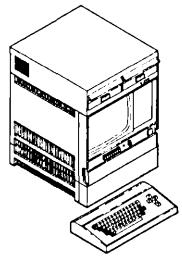
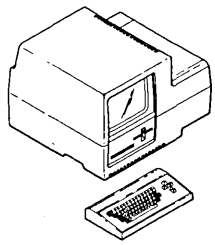
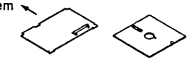
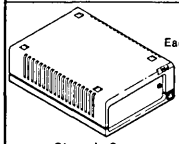
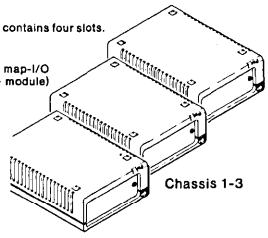
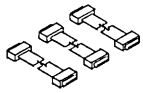
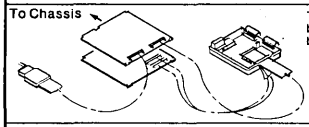
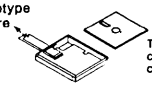
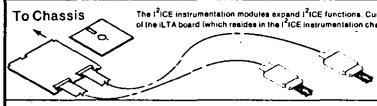
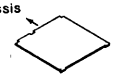
If prototype memory is not yet available, the user program may reside in I²ICE memory. Because this memory is RAM, you can make changes quickly and easily. For example, if your prototype contains EPROM, you need not erase and reprogram that EPROM during development.

Later, as prototype memory becomes available, you can reassign the verified user program, memory block by memory block, to prototype memory.

Access Restrictions

In addition to directing memory accesses, you can specify the following access restrictions.

- Read-only. The I²ICE system displays an error message if a user program attempts to write to an area of memory designated as read-only. You can, however, write to a read-only area with I²ICE commands.
- Read/write, no verify. Normally, the I²ICE system performs a read-after-write verification after program loads and after writing to memory with an I²ICE command. The I²ICE system can suppress this verification. For example, if a prototype has memory-mapped I/O, you do not want a verifying read that may change the state of the I/O device.
- Guarded. Initially, the I²ICE system puts all memory in a guarded state. Neither the user program nor the I²ICE user can access guarded memory.

	NAME	REQUIRED FOR
 Series III or Series IV 	Host development system	All applications
To Development System  The host interface board occupies a slot in the host development system.  A 10 or 42 ft. host/chassis cable connects the host interface board to the I²ICE instrumentation chassis.	Host interface board and I²ICE software (includes PSCOPE) Host/chassis cable	Communication between the host and the I²ICE system
 Chassis 0 Each I²ICE instrumentation chassis contains four slots. The break/trace board and the map-I/O board (from the emulation base module) reside in two slots. The other two slots provide for an optional high-speed memory board and the optional iLTA board.  Chassis 1-3  I²ICE instrumentation chassis are linked by 2 or 10 ft. inter-chassis cable sets	Instrumentation chassis Inter-chassis cable sets	Real-time multiprocessor emulation Breaking and tracing Memory and I/O mapping
To Chassis  The emulation base module consists of the break/trace board, the map-I/O board,the emulation clips pod, the buffer box base, and cables.	Emulation base module	Real-time microprocessor emulation Breaking and tracing Memory and I/O mapping
To prototype hardware  The emulation personality module personalizes the I²ICE instrumentation chassis for a specific probe. It consists of a personality board, a buffer box cover, a user cable, and I²ICE software.	Emulation personality modules	Specific processor emulation
To Chassis  The I²ICE instrumentation modules expand I²ICE functions. Currently available is the iLTA system. The iLTA system consists of the iLTA board (which resides in the I²ICE instrumentation chassis), iLTA probe cables, and the iLTA probe pods.	Intel logic timing analyzer (iLTA)	Test/measurement
To Chassis  The I²ICE™ high speed memory module supplies 128K bytes of programmable wait-state (zero to 15) memory. It resides in the instrumentation chassis.	Optional High Speed memory module	Memory expansion

The I²ICE™ I/O Map

The I²ICE system can direct (map) an emulated microprocessor's I/O space to the host development system's console, to the prototype system, to debugging procedures, or to a combination of these.

Simulating I/O with the Host Development Console

Suppose a user program requires input from an I/O device that is not yet a part of the prototype. Map the input port range assigned to that device to the host development system's console.

Then, when the user program requires input, it halts and the I²ICE system console displays a message requesting the data. When you enter the required data at the keyboard, the user program continues.

Simulating I/O with I²ICE™ Debugging Procedures

You could also write a procedure in the I²ICE command language that supplies the needed input data. When you set up the I/O map, specify that this I/O procedure is invoked when certain I/O ports are accessed.

I/O ports are mapped in blocks of 64 byte-wide ports or 32 word-wide ports. You can map a total of 64K byte-wide ports or 32K word-wide ports.

Symbolic Debugging

With symbolic debugging, you can reference a memory location by specifying its symbolic reference. A symbolic reference is a procedure name, line number, or label in the user program that corresponds to a location in the user program's memory space.

Typical Symbolic Functions

Symbolic functions include the following:

- Changing or inspecting the value and type of a program variable by using its program-defined name, rather than the address of the memory location where the variable and a hexadecimal value for the data are stored.
- Defining break and trace events using source-code symbols.

With symbolic debugging, you can reference static variables, dynamic (stack-resident) variables, based variables, and record structures combining primitive data types. The primitive data types are ADDRESS, BOOLEAN, BYTE, BCD, CHAR, WORD, DWORD, SELECTOR, POINTER, three INTEGER types, and four REAL types.

The Virtual Symbol Table

The I²ICE system maintains a virtual symbol table for program symbols; that is, the entire symbol table need not fit into memory at the same time.

The I²ICE system divides the symbol table into pages. If a program's symbol table is large, the I²ICE system reads only some of the symbol table pages into memory. When you reference a variable whose symbol is not currently defined in memory, the I²ICE system reads the needed symbol table page from disk into memory.

Breakpoint, Trace, and Arm Specifications

With I²ICE commands, you can define breakpoint, trace, and arm specifications.

Breakpoints allow you to halt a user program and examine the effect of the program's execution on the prototype. With the I²ICE system, you can set a breakpoint at a particular memory location or at a particular statement in a user program (including high-level language programs). You can also have a break occur when the user program enters or accesses a specified memory partition or reads or writes a user program variable. When you command the user program to resume execution, it picks up from where it left off.

Normally, the I²ICE system traces while the user program executes. With a trace specification, however, you can choose to have tracing occur only when specific conditions are met.

An arm specification describes an event or combination of events that must occur before the I²ICE system can recognize certain breakpoint and trace specifications. Typical events are the execution of an instruction or the modification of a data value.

The I²ICE command language allows you to specify complex, multilevel events. For example, you can specify that a break occurs when a variable is written, but only if that write occurs within a certain procedure. The execution of the procedure is the arm condition; the variable modification is the break condition.

Coprocessor Support

The 86/88 emulation personality module provides transparent RQ/GT and MIN/MAX pin emulation to support real-time prototype systems that use the 8087 or 8089 as coprocessors. The 86/88 emulation personality module also provides debugging features specific to the 8087. I²ICE commands provide access to the 8087's stack, status registers, and flags. The I²ICE system's disassembly and trace features extend to 8087 instructions and data types.

The 186 and 286 emulation personality modules also allow the prototype hardware to contain coprocessors. The 186 probe can qualify break points and collect trace information when the coprocessor drives the status lines ($\overline{S0}$-$\overline{S2}$) in the prescribed manner. The 286 personality module allows the hardware to contain the 80287 processor extension and provides special debugging features — you can enable and disable the 80287 and change and examine its registers.

DEBUGGING WITH THE I²ICE™ SYSTEM

The I²ICE system allows both hardware and software debugging (see Figure 1).

- Software debugging. I²ICE commands permit symbolic debugging of user programs written in high-level languages as well as assembly language. By looping the user cable back into the buffer box, you can debug a user program even if no prototype hardware is present.
- Hardware debugging. The I²ICE system is a real-time, in-circuit emulator. Trace data are collected in real-time, and I²ICE software does not intrude into user program space. The optional iLTA adds the high-speed timing and data acquisition of a logic timing analyzer.

The usefulness of an I²ICE system extends throughout the development cycle, beginning with the symbolic debugging of prototype software and ending with the final integration of debugged software and prototype hardware.

PSCOPE-86

PSCOPE-86 is a high-level language, symbolic debugger designed for use with Pascal-86, PL/M-86, and FORTRAN-86. It is a separate product included with the I²ICE system; it runs in the host development system. PSCOPE-86 is field-proven, familiar to Intel customers, and suited for the debugging of applications software when the hardware capabilities of the I²ICE system are not needed. The PSCOPE-86 and I²ICE command languages are similar.

Designing a product that contains a microcomputer requires close coordination of hardware and software development. A typical design process takes advantage of both the I²ICE system and PSCOPE-86. Use the I²ICE

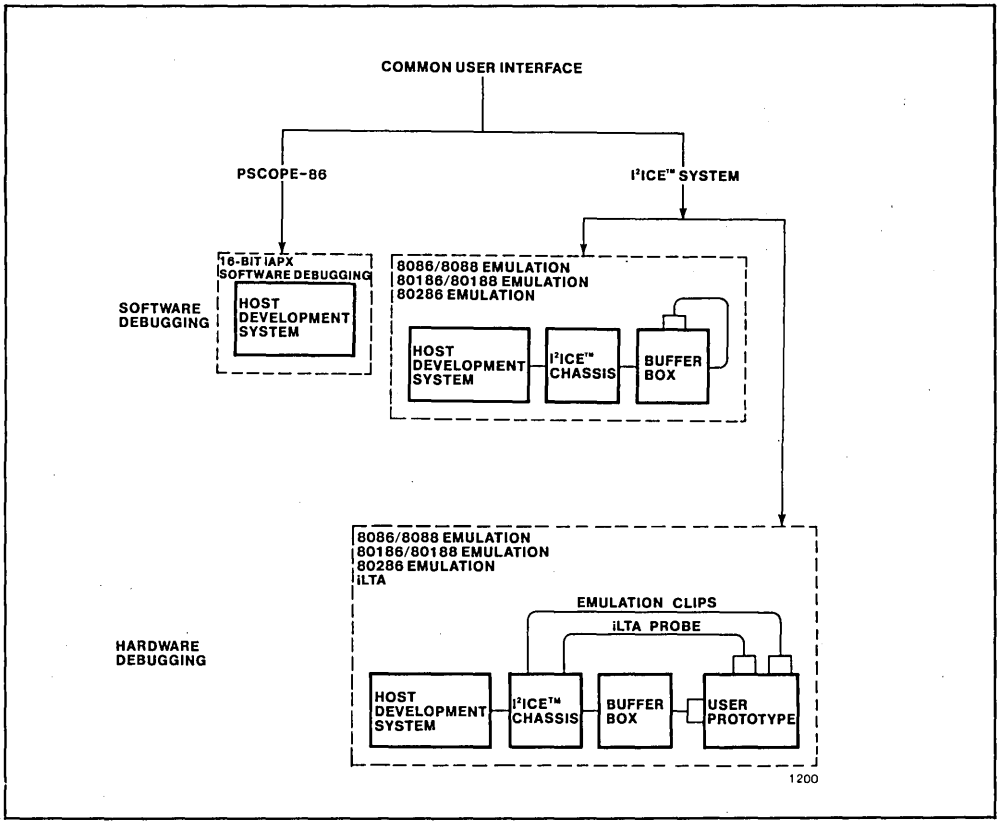


Figure 1 I²ICE™ Debugging Capabilities

system for real-time debugging and emulation, and throughout the design process, use PSCOPE-86 for the debugging of applications software.

THE I²ICE™ COMMAND LANGUAGE

The syntax of I²ICE commands resembles that of a high-level language. The I²ICE command language is versatile and powerful while remaining easy to learn and use. On-line help is available with the HELP command.

The I²ICE command language deals with user-created debugging objects. By manipulating debugging objects, you can streamline complex debugging sessions.

The I²ICE command language deals with user-created debugging objects. By manipulating debugging objects,you can streamline complex debugging sessions.

Debugging objects are uniquely named, user-created, software constructs that the I²ICE system uses to manage the debugging environment. The four types of debugging objects are debugging procedures, LITERALLY definitions, debugging registers, and debugging variables. In the following examples, I²ICE keywords are shown in all caps.

- Debugging procedures (named groups of I²ICE commands) can simulate missing software or hardware, collect debugging information, and make troubleshooting decisions. For example, here is the definition of a debugging procedure called *init* that simulates input from I/O ports 2 and 4.

```
*DEFINE PROCEDURE init = DO
.*IF %0==2 THEN
..*PORTDATA=100T
..*ELSE IF %0==4 THEN
...*PORTDATA=65T
...*END
..*END
.*END
```

- LITERALLY definitions are shorthand names for previously defined character strings. LITERALLY definitions save keystrokes or improve clarity. For example, here is the definition of a LITERALLY that saves keystrokes. This LITERALLY allows you to type DEF for DEFINE.

```
*DEFINE LITERALLY DEF = 'DEFINE'
```

Note that these definitions may be saved to a disk and auto-reloaded.

- Debugging registers are user-created software registers that hold arm, breakpoint, and trace specifications. You can order the I²ICE system to emulate the user program and specify one or more debugging registers. There is no need to reenter the specification for each emulation. For example, here is the definition of a debugging register called *pay* that contains a trace specification. This example takes advantage of the previous LITERALLY definition.

```
*DEF TRCREG pay=:cmaker.payment
```

To emulate a user program and trace only during the procedure *payment*, specify the debugging register *pay* as part of the GO command.

```
*GO USING pay
```

- Debugging variables are user-created variables used with I²ICE commands. For example, here is the definition of a debugging variable called *begin*. Its type is POINTER.

```
*DEFINE POINTER begin=0020H:0006H
```

During a debugging session, you could set the execution point to this pointer value by typing the following:

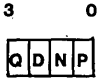
```
*$=begin
```

The I²ICE pseudo-variable $ represents the current execution point.

Example of a Debugging Session

Figures 2, 3, and 4 illustrate some of the key capabilities of the I²ICE system. The user program is written in Pascal-86. It was compiled, linked, and located on an Intellec Series III Development System. The resulting file consists of absolute code and is called CMAKER.86.

This program controls an automatic changemaker. The program reads the amount tendered (the variable *paid*) and the amount of the purchase (the variable *purchase*). It calculates the coins needed for change and asserts control signals to a change release mechanism by writing an output port. Each of the lower four bits of the output port controls the release of a different coin denomination.



Q = quarters
D = dimes
N = nickels
P = pennies

```
SERIES-III Pascal-86, V2.0
Source File: CMAKER.SRC
Object File: CMAKER.OBJ
Controls Specified: XREF, DEBUG, TYPE

 STMT  LINE  NESTING        SOURCE TEXT: MAKER.SRC
    1     1   0  0          PROGRAM cmaker;
    2     2   0  0          VAR change,coins                    :integer;
    3     3   0  0              quarters,nickles,dimes,pennies  :integer;
    4     4   0  0              paid,purchase                   :word;

    5     6   0  0          PROCEDURE payment;
    6     7   1  0              VAR numberofcoins               :integer;
    7     8   1  0                  release                     :word;
    8     9   1  0              BEGIN       (*payment*)
    8    10   1  1              numberofcoins:=quarters+dimes+nickles+pennies;
    9    11   1  1                  while numberofcoins<>0 do
   10    12   1  1                  BEGIN
   10    13   1  2                  release:=0;
   11    14   1  2                  if quarters<>0 then
   12    15   1  2                    BEGIN
   12    16   1  3                    release:=release+8;
   13    17   1  3                    quarters:=quarters-1
                                      END;
   15    19   1  2                  if dimes<>0 then
   16    20   1  2                    BEGIN
   16    21   1  3                    release:=release+4;
   17    22   1  3                    dimes:=dimes-1
                                      END;
   19    24   1  2                  if nickles<>0 then
   20    25   1  2                    BEGIN
   20    26   1  3                    release:=release+2;
   21    27   1  3                    nickles:=nickles-1
                                      END;
   23    29   1  2                  if pennies<>0 then
   24    30   1  2                    BEGIN
   24    31   1  3                    release:=release+1;
   25    32   1  3                    pennies:=pennies-1
                                      END;
   27    34   1  2                  numberofcoins:=quarters+dimes+nickles+pennies;
   28    35   1  2                  OUTWRD(130,release);
   29    36   1  2                  END;
   31    37   1  1              END;     (*payment*)

   32    39   0  0          BEGIN (*main*)
   32    40   0  1          INWRD(2,paid);
   33    41   0  1          INWRD(70,purchase);
   34    42   0  1          change  :=paid-purchase;
   35    43   0  1          coins   :=change mod 100;
   36    44   0  1          quarters :=coins div 25;
   37    45   0  1          coins   :=coins mod 25;
   38    46   0  1          dimes   :=coins div 10;
   39    47   0  1          coins   :=coins mod 10;
   40    48   0  1          nickles  :=coins div 5;
   41    49   0  1          pennies  :=coins mod 5;
   42    50   0  1          payment;
   43    51   0  1          END.    (*main*)
```

Figure 2 Listing of the Program Used in the Debugging Session

```
(1) *BASE
    DECIMAL

(2) *MAP 0K LENGTH 32K HS
    *MAPIO 0T LENGTH 192T ICE
    *MAP
    MAP       0K LENGTH     32K    HS
    MAP     32K LENGTH    992K    GUARDED
    *MAPIO
    MAPIO    00000H LENGTH  000C0H    ICE
    MAPIO    000C0H LENGTH  0FF40H    USER

(3) *LOAD :F1:CMAKER.86

(4) *DEFINE POINTER begin = $
    *DEFINE BRKREG pay = :cmaker#9
    *DEFINE PROC display = DO
    .*WRITE USING ('"quarters =",T,0,>')quarters
    .*WRITE USING ('"dimes    =",T,0')dimes
    .*WRITE USING ('" nickles  =",T,0,>')nickles
    .*WRITE USING ('"pennies  =",T,0')pennies
    .*RETURN TRUE
    .*END

(5) *GO USING pay
    ?UNIT 0 PORT 2H REQUESTS WORD INPUT (ENTER VALUE)*100
    ?UNIT 0 PORT 46H REQUESTS WORD INPUT (ENTER VALUE)*65
    *Probe 0 stopped at :CMAKER#9 + 4 because of execute break
     Break register is PAY   Trace Buffer Overflow

(6) *quarters;dimes;numberofcoins
    +1
    +1
    +2

(7) *DEFINE SYSREG wr_number = WRITE AT .:cmaker.payment.numberofcoins &
    **CALL display
    *GO USING wr_number
    *quarters  = +1      dimes      = +1
     nickles   = +0      pennies    = +0
    Probe 0 stopped at :CMAKER#28 + 3 because of bus break
     Break register is WR_NUMBER

(8) *numberofcoins
    +0
    *EVAL release
    1100Y 12T CH '..'

(9) *CLIPSOUT = 11Y

(10) *GO FOREVER
     ?UNIT 0 PORT 82H OUTPUT WORD 0C
     ?Probe 0 stopped at location 0033:00AEH because of bus not active
      Bus address = 0203DE
     *$=begin
     *
```

Figure 3 Sample Debugging Session

(1) Checking to see that the default radix is decimal.

(2) Mapping user program memory to I²ICE high-speed memory and user I/O ports to the I²ICE system console.

(3) Loading the user program.

(4) Defining debugging objects.

The debugging variable *begin* is set to $, an I²ICE pseudo-variable representing the current execution point. At this point in the debugging session, $ is the beginning of the user program.

The break register *pay* specifies a breakpoint at statement 9 in the user program.

The debugging procedure *display* displays the value of some user program variables on the console.

(5) Beginning emulation with the debugging register *pay*. The console requests the two input values, *paid* and *purchase*. Then, the break occurs.

(6) Displaying three user program variables.

(7) Defining another debugging register. The specified event is the writing of the user program variable *numberofcoins*. When that event occurs, the I²ICE system calls the debugging procedure *display*. In addition to displaying some user program variables, this debugging procedure returns a Boolean value. Because this value is TRUE, the break occurs; if the value were FALSE, emulation would continue.

(8) Displaying the two user program variables, *numberofcoins* and *release*. The EVAL command displays *release* in binary, decimal, hexadecimal, and ASCII. Unprintable ASCII characters appear as periods (.).

(9) Asserting both output lines on the emulation clips. These lines are input to the prototype hardware and control a change release mechanism.

(10) Resuming emulation. The console displays the write of *release* to the output port. The user program finishes executing, and the probe stops emulating because of bus inactivity. $ is set back to the beginning of the user program in preparation for another emulation.

Figure 3 Explanation of Sample Debugging Session

I²ICE™ Command Functions

The I²ICE command language contains the following functional categories.

- Emulation commands. The GO command instructs the I²ICE system to begin emulation. You can also specify that the I²ICE system break or trace under certain specified conditions.
- Utility commands. These are general purpose commands for use in a debugging environment. For example, one use of the EVAL command is to calculate the nearest source-code line number that corresponds to the address of an assembly language instruction. The HELP command provides on-line assistance. The EDIT command invokes a menu-driven text editor, allowing you to update debugging object definitions. A command line editor is also provided.
- Environment commands. These are commands that set up the debugging environment. For example, the MAP command sets up the memory map. Another environment command (WAITSTATE) inserts wait-states into memory accesses, allowing the simulation of slow memories.

- File handling commands. These are commands that access disk files. You can save debugging object definitions in a disk file and load them in later debugging sessions. You can also record your debugging session in a disk file for later analysis.
- Probe-specific commands. These are commands whose effects are different for different probes. For example, the PINS command displays the state of selected signal lines on the current probe.
- Option-specific commands. These are commands that control an optional test/measurement device, such as the logic timing analyzer.

I²ICE™ INSTRUMENTATION SUPPORT

I²ICE™ Emulation Clips

Eight external input lines are sampled in real-time at each processor bus cycle. The I²ICE system records the values of these lines in its trace buffer. The I²ICE system can use these values when defining events.

Four additional output lines synchronize I²ICE events with external hardware. Two lines are active and programmable with I²ICE commands. Two others allow the I²ICE chassis to be linked with previously released Intel emulators or with other external test hardware.

Intel Logic Timing Analyzer

The logic timing analyzer is the first in a series of chassis-resident, test/measurement modules designed to extend the capability of the I²ICE system to recognize events and collect data. The iLTA and the I²ICE emulator work together. They can trigger and/or arm/disarm each other. In addition, waveforms acquired by the iLTA can be time-aligned with I²ICE traces.

The iLTA brings the flexibility of a logic timing analyzer's high-speed triggering and glitch detection to the I²ICE system. The iLTA is a general purpose logic timing analyzer, supplemented with special features for microsystem debugging and I²ICE integration. Following are some of iLTAs features.

- 16 channel, 100 MHz asynchronous operation.
- 16 channel, 50 MHz synchronous operation.
- Single- or double-height timing waveforms presented with data scrolling, magnification, and delta-time read-out features.
- Minimum 3 nanosecond glitch detection (3 ns + 1 ns/volt for signal swings greater than 3 volts).
- A dual-threshold acquisition mode, with programmable logic level thresholds.
- A burst acquisition mode with window boundary indicators.
- User-defined channel labels and state display radixes.
- Disk storage for preservation and restoration of analyzer setups and acquired waveforms.
- Logic waveform comparison features (compares current acquisitions with previous traces stored in auxiliary memory or on disk).
- Menu-driven operation and user-friendly display. The display takes advantage of screen highlighting, blinking characters, and reverse video.
- Powerful post-processing data analysis commands that are part of the I²ICE command language.
- Multiple emulator break/trace and iLTA trigger/trace conditions may be shared with as many as four emulators and four iLTAs.

SPECIFICATIONS

Host Requirements

512K bytes in the memory space of the host processor.
Two double-density diskette drives.
System console.

For the iLTA to run on a Series III Development System, the III-820 board must be installed. The iLTA option also requires additional memory.

I²ICE™ Software

I²ICE host software
I²ICE probe software
I²ICE confidence tests
PSCOPE-86
Optional iLTA software and iLTA confidence tests

System Performance

Mappable zero wait-state memory	Minimum 32K bytes Maximum 160K bytes
Trace buffer	1023 X 48 bits
Virtual symbol table	The number of user program symbols is limited only by available disk space.

Physical Characteristics

Instrumentation chassis

width	17.0 in.	(43.2 cm.)
height	8.25 in.	(21.0 cm.)
depth	24.13 in.	(61.3 cm.)
weight	48 lbs.	(21.9 kg.)

Host/chassis cable
10 ft. (3.0 m) and 42 ft. (12.8 m) options

Inter-chassis cable set
2 ft. (61 cm) and 10 ft. (3.0 m) options

Buffer box

width	8.5 in.	(21.6 cm.)
height	3.0 in.	(7.6 cm.)
depth	10.0 in.	(25.4 cm.)
weight	8 lbs.	(3.7 kg.)

Electrical Characteristics

90-132 V or 180-264 V (selectable)
47-63 Hz
12 amps (AC)

Environmental Requirements

Operating temperature	0° to 40°C (32° to 104°F)
Operating humidity	Maximum of 85% relative humidity, non-condensing

Emulation Clips

Emulation clipsin lines are sampled once every bus cycle when the address bits become valid on the address bus. During emulation, the I²ICE system records the value of the clipsins lines in the trace buffer once every execution cycle.

I²ICE™ Emulation Clips — DC Characteristics

Signal	Input Voltage		Input Current		Output Current	
	Low V_{IL} V	High V_{IH} V	Low I_{IL} μA	High I_{IH} μA	Low I_{OL} mA	High I_{OH} mA
clipsout lines					33 at 0.7 V	4.8 at 2.0 V
SYSBREAK SYSTRACE					38 at 0.7 V	1.0 at 2.0 V
clipsin lines	1.05	2.5	50	50		

I²ICE™ 86/88 User Interface — DC Characteristics

Pin Name	Input Voltage		Output Voltage		Input Current		Output Current	
	Low V_{IL} V	High V_{IH} V	Low V_{OL} V	High V_{OH} V	Low I_{IL} mA	High I_{IH} mA	Low I_{OL} mA	High I_{OH} mA
AD0-AD15	0.8	2.0	0.5	2.0	0.2	0.02	24	12
A16-A19 $\overline{BHE}$/S7	0.8	2.0	0.55	2.0	0.4	0.05	64	15
RD			0.55	2.0			64	15
$\overline{DEN}$ ($\overline{S0}$) DT/$\overline{R}$ ($\overline{S1}$) M/$\overline{IO}$ ($\overline{S2}$) $\overline{WR}$ ($\overline{LOCK}$) $\overline{INTA}$ (QS1) ALE (QS0)			0.5	2.4			20	6.5
NMI	0.8	2.0			0.4	0.05		
READY	0.8	2.0			0.4	0.04		
INTR	0.8	2.0			2.0	0.05		
TEST	0.8	2.0			0.6	0.04		
RESET	0.8	2.0			2.2	0.07		
HOLD ($\overline{RQ}$/$\overline{GT0}$) HOLDA ($\overline{RG}$/$\overline{GT1}$)	0.8	2.0	0.45	2.4	1.0	0.02	23	2.5

The 86/88 probe has a greater output drive capacity than the 8086 or 8088 chip.

I²ICE 86/88 User Cable —Capacitance 21 pf/ft.
—Impedance 95 ohms

Capacitive Loading — 86/88 Probe

The 86/88 probe presents the user system with a maximum load of 34 pf and 0.8 mA.

All 86/88 probe outputs are capable of driving a minimum of 20 mA and 15 pf while meeting all the probe's timing specifications. The 86/88 probe will drive larger capacitive loads, but with possible performance degradation.

Coprocessor Operation — 86/88 Probe

During emulation with external coprocessors, a two-clock delay precedes each RQ, GT, and RLS pulse in MAX mode and each HOLD and HOLDA assertion in MIN mode.

The user can choose to have the coprocessor run only during emulation or all the time. If the coprocessor runs all the time, then during interregation mode the coprocessor may have as much as a one microsecond delay in addition to the two-clock delay mentioned previously.

The I²ICE system ignores a coprocessor when the probe is in the reset state. If a coprocessor asserts RQ during this time, the RQ/GT sequence may get out of synchronization. The probe is reset when the I²ICE host software loads I²ICE probe software.

Timing Differences between I²ICE™ 8086 Emulation and the 8086-1 Microprocessor (10 MHz clock)

MAX Mode Symbol	Parameter	8086-1 Min (ns)	8086-1 Max (ns)	I²ICE™ Min (ns)	I²ICE™ Max (ns)
TCHSV	Status Active Delay	10	45	20	50
TCLAV	Address Valid Delay	10	50	20	60
MIN Mode Symbol	**Parameter**	**8086-1 Min (ns)**	**8086-1 Max (ns)**	**I²ICE™ Min (ns)**	**I²ICE™ Max (ns)**
TCVCTV	Control Active Delay 1	10	50	25	59
TCVCTX	Control Inactive Delay	10	50	25	59
TCHCTV	Control Active Delay 2	10	45	21	54

I²ICE™ 186/188 User Interface — DC Characteristics

Pin Name	Input Voltage		Output Voltage		Input Current		Output Current	
	Low V_{IL} V	High V_{IH} V	Low V_{OL} V	High V_{OH} V	Low I_{IL} mA	High I_{IH} mA	Low I_{OL} mA	High I_{OH} mA
AD0-AD15	0.8	2.0	0.45[1]	2.4[2]	0.4	0.1	60 max	10 max
A16-A19			0.45[1]	2.4[2]			60 max	10 max
$\overline{BHE}$ $\overline{S0}$-$\overline{S2}$			0.45[1]	2.4[2]			60 max	10 max
$\overline{LOCK}$ RESET CLKOUT TMROUT0 TMROUT1 $\overline{INTA0}$ $\overline{INTA1}$ HLDA ALE $\overline{RD}$,$\overline{WR}$ (chip selects) DTR $\overline{DEN}$			0.45[1]	2.4[2]			60 max	10 max
X1,X2 $\overline{RES}$	0.8	3.8			0.1	0.1		
$\overline{TEST}$ TMRIN0, TMRIN1 DRQ0, DRQ1, NMI INT0-INT3 HOLD	0.8	2.0			0.4	0.1		
ARDY SRDY	0.8	2.0			2.0	0.1		

[1] I_{OL} = 18 mA
[2] I_{OH} = 3 mA

The 186/188 probe has a greater output drive capacity than the 80186 or 80188 chip.

186/188 probe outputs are capable of driving a minimum of 20 mA and 150 pf while meeting the probes timing specifications. The 186/188 probe will drive larger capacitive loads but with possible performance degradation.

I²ICE™ 286 HIGHLIGHTS

- Supports real and protected mode (software).
- Includes an object code loader for both 86 and 286 object files.
- Supports multiprocessing (with coprocessors and with the 80287 processor extension).
- Supports local descriptor tables (LDTs).
- Provides full 24-bit address mapping (with optional 16K granularity).
- Provides the capability to read/write normally invisible portions of segment and table registers.
- Supports multitasking.
- Does not slip on breakpoints.

I²ICE™ 286 User Interface — DC Characteristics

Pin Name	Input Voltage		Output Voltage		Input Current		Output Current	
	Low V_{IL} V	High V_{IH} V	Low V_{OL} V	High V_{OH} V	Low I_{IL} mA	High I_{IH} mA	Low I_{OL} mA	High I_{OH} mA
A0-A23			0.4	2			24	2.6
D0-D15	0.8	2.0	0.4	2	0.1	0.02	24	2.6
$\overline{S0}$,$\overline{S1}$			0.55	3.4			64	15
M/$\overline{IO}$			0.55	3.4			64	15
$\overline{LOCK}$			0.55	3.4			64	15
COD/$\overline{INTA}$			0.55	3.4			64	15
$\overline{BHE}$			0.55	3.4			64	15
$\overline{ERROR}$	0.8	2.0			0.8	0.1		
$\overline{BUSY}$	0.8	2.0			0.4	0.05		
$\overline{PEACK}$			0.55	3.4			64	15
HLDA			0.5	3.4			20	1.0
HOLD	0.8	2.0			0.4	0.05		
PEREQ	0.8	2.0			0.4	0.05		
INTR	0.8	2.0			0.4	0.05		
NMI	0.8	2.0			0.4	0.05		
$\overline{CLK}$	0.8	2.0			0.8	0.1		
$\overline{READY}$	0.8	2.0			3.0	0.09		

The 286 probe has a greater output drive capacity than the 80286 chip.

All 286 probe outputs are capable of driving a minimum of 20 mA and 15 pf while meeting all the probe's timing specifications. The 286 probe will drive larger capacitive loads but with possible performance degradation.

Documentation

121790 *PSCOPE-86 High-Level Program Debugger User's Guide*
163250 *I²ICE™ System Overview and Installation*
163251 *Guide to Using the I²ICE™ System*
163252 *I²ICE™ Reference Manual*
163253 *I²ICE™ Command Dictionary*
163256 *iLTA User's Guide*
163257 *iLTA Reference Manual*
163258 *iLTA Learner's Guide*
163259 *Guide to Using the I²ICE™ Probes*
210350 *PSCOPE-86 Data Sheet*
230839 *iLTA Data Sheet*

ORDERING INFORMATION

System hardware may be ordered as basic stand-alone items or as a hardware kit. All software must be ordered individually.

Order Code	Description: Basic System Items
III-514B	I²ICE instrumentation chassis
III-520	I²ICE host interface board
III-530	I²ICE host/chassis cable — 10 ft. (3.0 m)
III-531	I²ICE host/chassis cable — 42 ft. (12.8 m)
III-532	I²ICE inter-chassis cable set — 2 ft. (0.6 m)
III-533	I²ICE inter-chassis cable set — 10 ft. (3.0 m)
III-620	I²ICE emulation base module
III-086A	I²ICE 86/88 emulation personality module
SBC333	iSBC 337 MULTIMODULE™ numeric data processor (needed for use with the 8086/8088 personality module when there will be an internal 8087 co-processor)
III-186A	I²ICE 186/188 emulation personality module
III-286A	I²ICE 286 emulation personality module
III-810	I²ICE logic timing analyzer (iLTA)
III-820	Series III IOC board (must be used with iLTA)
III-813	iLTA terminator set, 16 channel (supplements iLTA-supplied set)
III-814	iLTA terminator set, 8 channel (supplements iLTA-supplied set)
III-815	Microhook set (40 microhooks—supplements iLTA-supplied microhooks)
III-816	Logic probe pod, channels 0-7 (supplements iLTA-supplied pod)
III-817	Logic probe pod, channels 8-F (supplements ILTA-supplied pod)
III-707	Optional high-speed memory board (128K)

Order Code	Description: System Hardware Kits
III-010	I²ICE 86/88 hardware support kit (includes III-514B, II-520, III-530, III-620, III-086A)
III-110	I²ICE 186/188 hardware support kit (includes III-514B, II-520, III-530, III-620, III-186A)
III-210	I²ICE 286 hardware support kit (includes III-514B, II-520, III-530, III-620, III-286A)
III-811	iLTA Series III hardware kit (includes III-810 and III-820)

Order Code	Description: System Software
III-901A	I²ICE 86/88 emulation software — 8 in. single density
III-901B	I²ICE 86/88 emulation software — 8 in. double density
III-901C	I²ICE 86/88 emulation software — 5¼ in. double density
III-911A	I²ICE 186/188 emulation software — 8 in. single density
III-911B	I²ICE 186/188 emulation software — 8 in. double density
III-911C	I²ICE 186/188 emulation software — 5¼ in. double density
III-921A	I²ICE 286 emulation software — 8 in. single density
III-921B	I²ICE 286 emulation software — 8 in. double density
III-921C	I²ICE 286 emulation software — 5¼ in. double density

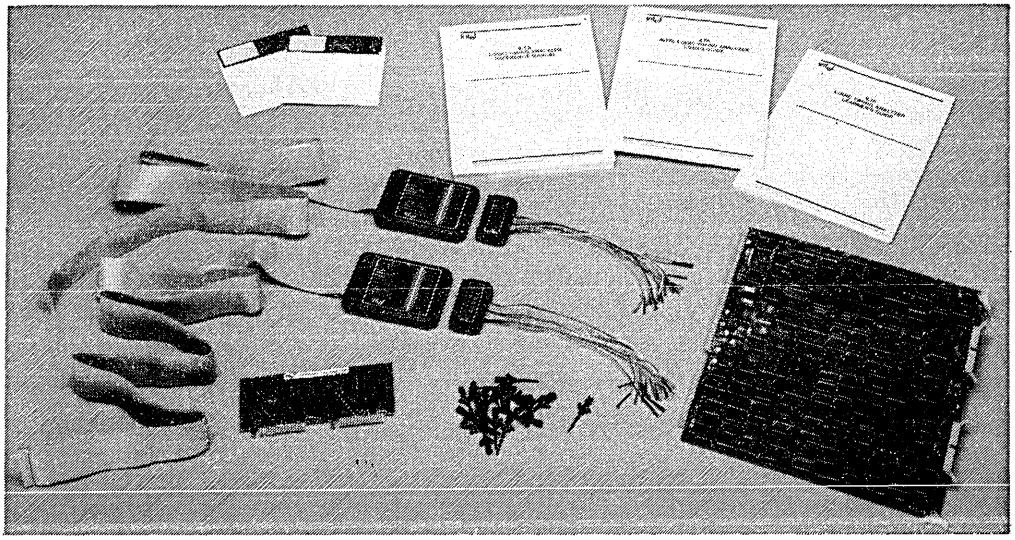
III-951A	I²ICE base software — 8 in. single density	III-981A	iLTA software — 8 in. single density
III-951B	I²ICE base software — 8 in. double density	III-981B	iLTA software — 8 in. double density
III-951C	I²ICE base software — 5¼ in. double density	III-981C	iLTA software — 5¼ in. double density

iLTA
Logic Timing Analyzer

PRELIMINARY

- Integrates the features of a stand-alone logic analyzer with the features of a powerful in-circuit emulation system.
- Provides trigger, arm, disarm, and trace of a state and timing logic analyzer from an in-circuit emulator.
- Provides full state and timing acquisition performance:
  - Up to 100-MHz asynchronous acquisition.
  - Up to 50-MHz synchronous acquisition.
- Features five data acquisition modes: Standard, ICE Sync, Burst, Dual Threshold, and Glitch.
- Provides 16 data acquisition channels.
- Displays data either in logic state form or as timing diagrams.
- Ensures versatile triggering with four word recognizers, nine triggering modes, stop and start data qualification, trigger qualifiers, and I^2ICE^{TM} interaction.
- Offers 3-ns glitch detection.
- Features flexible data analysis software: delta time readout, search word, and auxiliary memory for data comparison applications.
- Allows user-definable mnemonics and labeling.
- Operates from menus or I^2ICE^{TM} command language.
- Stores and recalls acquired data for post-processing or setup information.

The Intel Logic Timing Analyzer (iLTA) is a general purpose logic analyzer, enhanced with special features for microsystem debugging and user-system integration. The iLTA brings the flexibility of a logic timing analyzer's high-speed triggering and glitch detection coupled with a state analyzer's selective acquisition of system data to the I^2ICE^{TM} system.



ORDER NUMBER: 230839-002

PHYSICAL DESCRIPTION

The iLTA hardware consists of the ILTA circuit board, two probe pods, four probe terminators, and a set of grabbers. Also included is the Confidence Test/Demo Board, which is used for running the confidence tests and for learning to use the iLTA

Two probe pods connect the iLTA to the target system using one of two sets of probe terminators. The standard probe terminator set has 16 data acquisition channels. The dual threshold probe terminator set provides eight-channel analysis for dual threshold or glitch acquisition.

The iLTA circuit board resides in the top slot of the I^2ICE instrumentation chassis; an I^2ICE probe can be installed in the same chassis. The user can control four iLTAs (one per chassis) from one host.

FUNCTIONAL DESCRIPTION

The user easily controls the iLTA'S four basic functions: recognizing and qualifying events, acquiring and storing data, displaying collected data, and analyzing collected data.

Event Recognition And Qualification

The iLTA has four programmable word recognizers which can be programmed symbolically using mnemonics from the iLTA's user-definable decode table or programmed in binary, octal, or hexadecimal. Two word recognizers include a digital filter to further qualify data. A clock qualifier defines additional conditions when data can be sampled. Two trigger qualifiers can be used to recognize and/or trigger on data edges.

The trigger modes define how events recognized by the word recognizers affect iLTA data collection. The nine iLTA trigger modes offer different event specifications leading up to triggering and tracing data, allowing the user to choose the mode that collects the most useful data for a specific application. The triggering modes set arm, disarm, trace, and trigger conditions by using the occurrence or non-occurrence of events that match specified word recognizer patterns. The triggering modes also let the user set a delay counter in certain situations and send arm, disarm, break, and trace signals to the I^2ICE system. Figure 1 shows the iLTA trigger setup menu.

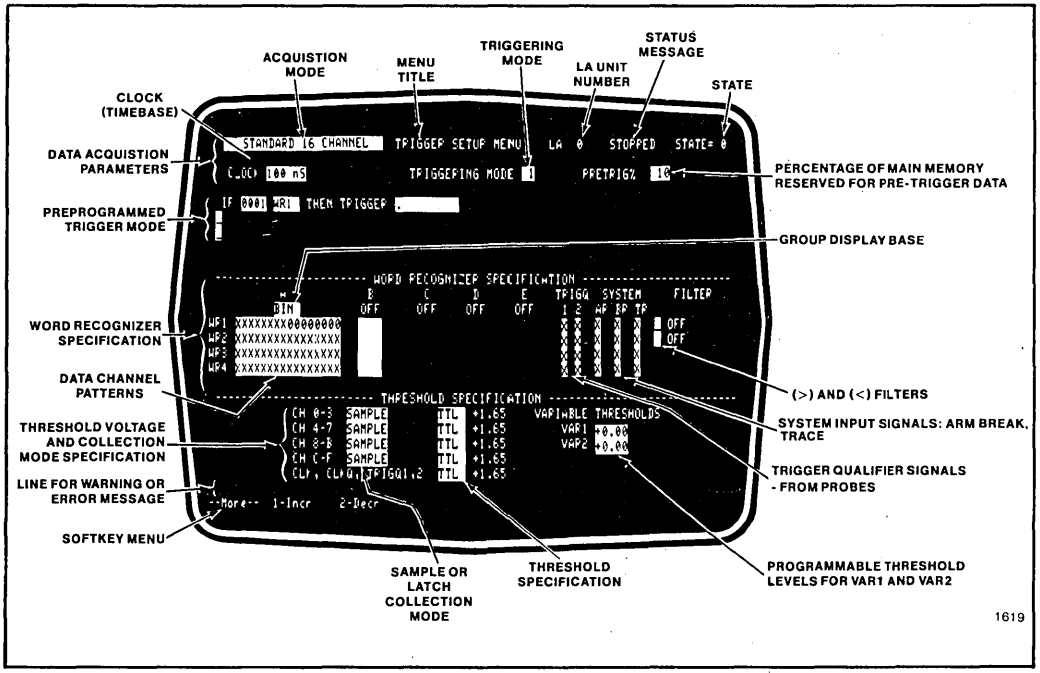


Figure 1. The iLTA Trigger Setup Menu

Data Acquisition And Storage

The user-specified trace qualifications determine the data that the iLTA collects. The user can specify word recognizer events, data acquisition modes, voltage thresholds, and sampling modes.

The iLTA has five data acquisition modes:

- 16-channel Standard Mode has all 16 channels available for data collection.
- 15-channel ICE Sync Mode collects data on 15 channels and uses the 16th channel to receive timing information from the I^2ICE system.
- 15-channel Burst Mode collects discrete bursts of data on 15 channels and uses the 16th channel to mark the time discontinuities.
- 8-channel Dual Threshold Mode collects and displays three-state logic signals and can be used for rise time analysis or any other situation where two thresholds are useful.
- 8-channel Glitch Mode detects and displays glitches as short as three nanoseconds.

The user can choose either asynchronous sampling using the internal clock at speeds to 100 MHz or synchronous sampling using an external clock at system speeds to 50 MHz.

The iLTA has two 512-word memory areas, main memory and auxiliary memory. Data is collected into iLTA main memory. The user can examine main memory contents immediately after collection or store the contents of main memory either in iLTA auxiliary memory or in a file in the development system disk for later processing or comparison.

Data Display

The iLTA offers both state and timing displays of data acquisitions. The state display presents data in logic state form, in the user's choice of channel groupings and of number bases (binary, octal, hexadecimal, ASCII characters, or user-defined mnemonics). Figure 2 shows the state display menu.

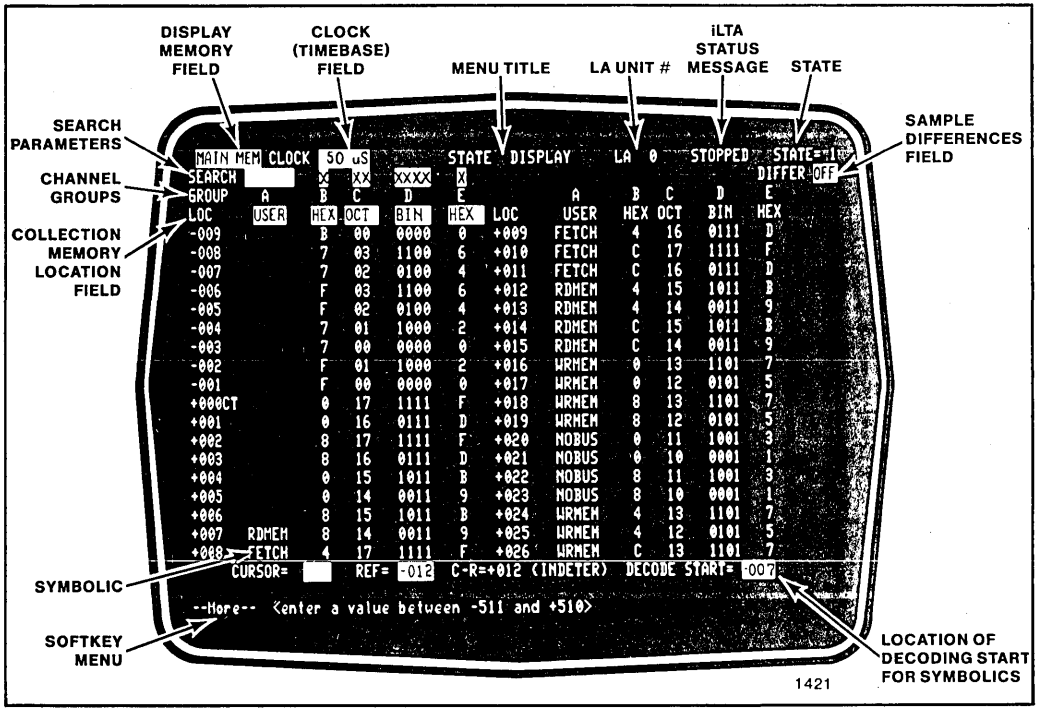


Figure 2. The State Display Menu

Timing diagrams let the user examine the timing relationship between signals. The memory graticule and the first and last memory locations displayed let the user relate the memory locations of the displayed data to the data in memory. The display magnification feature allows examination of detailed timing information. Figure 3 shows timing diagrams.

Analysis

The iLTA allows comparison of the data collection in main memory to the data collection in auxiliary memory and lets you find either the differences or the similarities. The data skew between collections can be masked out. The data in auxiliary memory can be from a previous iLTA data collection or can be modified by direct user input.

Once data is collected, the iLTA search function helps the user find a particular data event in displayed memory quickly and easily. The user can define one data event or multiple data events (which are logically ANDed together).

The three marking cursors help pinpoint events or measure the distance between data events in both time and memory locations.

Because the iLTA commands are a subset of the I^2ICE commands, the user can include iLTA commands in I^2ICE debug procedures and so post-process collected iLTA data.

Easy To Use

The iLTA is an easy-to-use enhancement to the Intellec® Development System.

The iLTA operates in either menu-driven mode or command mode. Because screen menus indicate the setup and display the choices available, the user need not learn the command syntax and

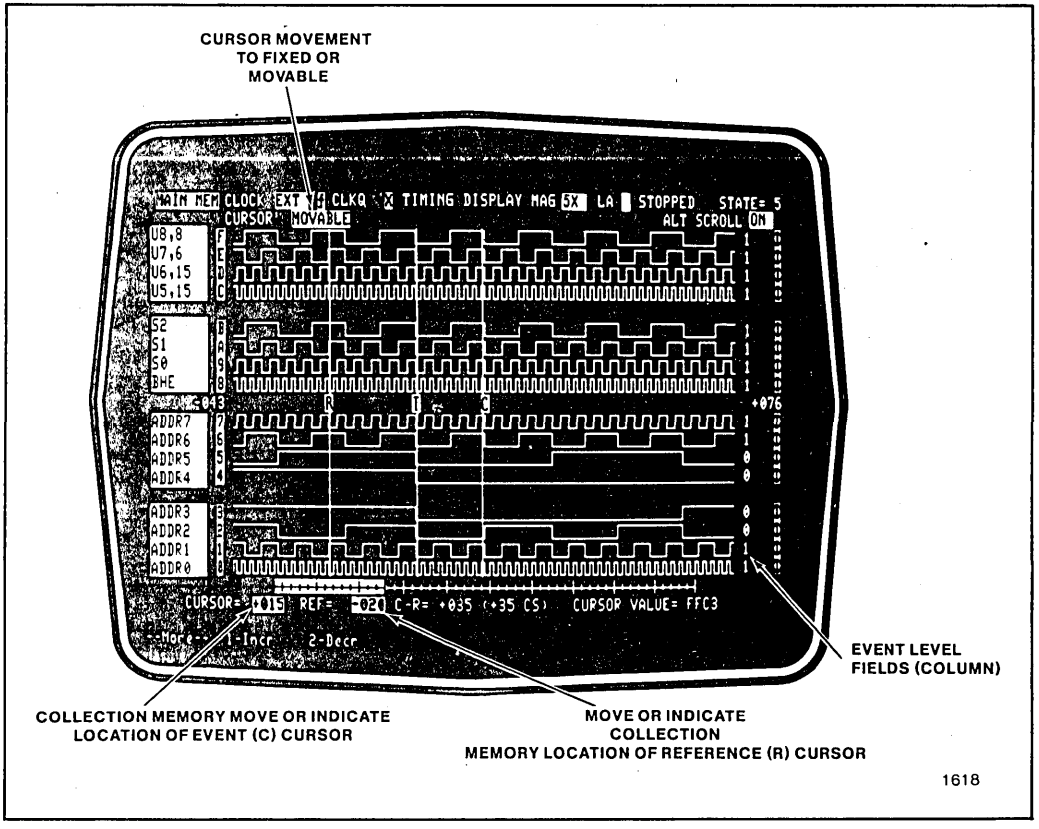


Figure 3. Timing Diagrams

the legal command values before using the iLTA. The iLTA has four main menus:

- The Trigger Setup menu defines the conditions under which the system will store data.
- The Group Setup menu sets up display and compare conditions and user symbols.
- The Timing Display menu displays the data collected as timing diagrams.
- The State Display menu displays the data collected in logic state form.

Sometimes commands are more convenient to use than menus. The iLTA commands are an extension of the I^2ICE commands, so the user can include iLTA commands in I^2ICE procedures and thus control iLTA operation from the I^2ICE system.

After a session, the user can save the contents of main memory, auxiliary memory, and all menu setups in a system file for recall in later sessions. Libraries of test procedures can be created to simplify debug procedures or be passed to other users.

COMBINING LOGIC ANALYSIS AND IN-CIRCUIT EMULATION

The iLTA integrates the features of a stand-alone logic analyzer with the event machines of the I^2ICE system. The iLTA can interact with the I^2ICE system in two ways: an I^2ICE signal can cause an iLTA action or an iLTA action can enable a system signal that can cause an action in the I^2ICE instruments. The iLTA adds two unique functions when used with the I^2ICE emulator:

- The iLTA can send control signals to the I^2ICE system that the user can program to cause the I^2ICE system to arm, disarm, break, or trace.
- The iLTA can recognize I^2ICE system signals that can be used to cause the iLTA to arm, disarm, restart, qualify trace activity, or trigger.

ENHANCED DEBUGGING WITH THE iLTA

The I^2ICE and the iLTA work together to offer enhanced debugging capabilities.

- Hardware debugging
  - Verify control line timing relationships.
  - Isolate glitches and trace conditions.
  - Examine hardware operation.
  - Verify expected hardware performance.
- Software/hardware interaction
  - Troubleshoot interaction between software instructions and hardware operations.
  - Help determine whether bugs are caused by hardware or software.
  - Verify expected hardware/software interaction by combining the I^2ICE system with the iLTA.
- Software debugging
  - Examine software I/O drivers and interaction with the mainline program under the I^2ICE control.

SPECIFICATIONS

Host Requirements

The iLTA runs in the top slot (of four) of an I^2ICE instrumentation chassis connected to an Intellec Series III or Series IV Development System. A Series III or IIIE must have two double-density flexible disk subsystems, an additional SBC-012B, and an III-820 IOC upgrade. For a Series IV, an SBC-012B and at least 2 Mbytes of mass storage are required. The iLTA does not run on the Intellec Model 800.

iLTA Software

iLTA/I^2ICE host software
iLTA Confidence Test

Physical Characteristics

All power and cooling functions are provided by the I²ICE instrumentation chassis.

Maximum Power Requirements
+5.0 VDC at 2.0 A
−5.2 VDC at 15.5 A
+15 VDC at 0.4 A
−15 VDC at 0.4 A

Threshold Voltages
−6.40 VDC to +6.35 VDC in 50 mV steps, with an accuracy of 67 mV + 2% of V_{TH}

Maximum Input Voltage
±50 volts continuous.

Input Impedance
$R_{IN} = (V_{IN} * 2\ Mohm)/(V_{IN} + V_{TH} + 6)$
AC — Maximum of 10pF in parallel with the above DC impedance between any probe input and ground. Typically 5pF.

Maximum capacitance of probe plus terminator is ≤ 20pF per channel. Typically 15pF.

Clock Rates
Internal 10 ns to 50 ms (100 MHz to 20 Hz)
External DC to 50 MHz

Setup time — 5 ns; min typically 0 ns
Hold time — 5 ns; min typically 3.0 ns

Duration:
minimum active pulse width — 7 ns + 1ns/V for signal swings greater than 3 V.
minimum pulse period - 20 ns

Minimum Glitch Capture

3 ns (+ 1 ns/V for signal swings greater than 3 V) at threshold and 25% of total voltage swing over (or below) the threshold or 250 mV overdrive, whichever is greater.

Memory Range 512 words

Filter Ranges 1 or 2 to 255

Channel-to-channel skew 3 ns (maximum)

Operating Temperatures

0° C to 40°C
(Non-operating temperatures -40° C to +75° C)
Probe pods 0° C to 45° C

Operating Humidity

0 to 95% (non-condensing)

Documentation

iLTA User's Guide, order number 163256
iLTA's Reference Manual, order number 164257
iLTA Learner's Guide, order number 163258
I²ICE™ System Overview and Installation, order number 163250
Guide to Using the I²ICE™ System, order number 163251
I²ICE™ Reference Manual, order number 163252
I²ICE™ Command Dictionary, order number 163253
Guide to Using the I²ICE™ Probes, order number 163259

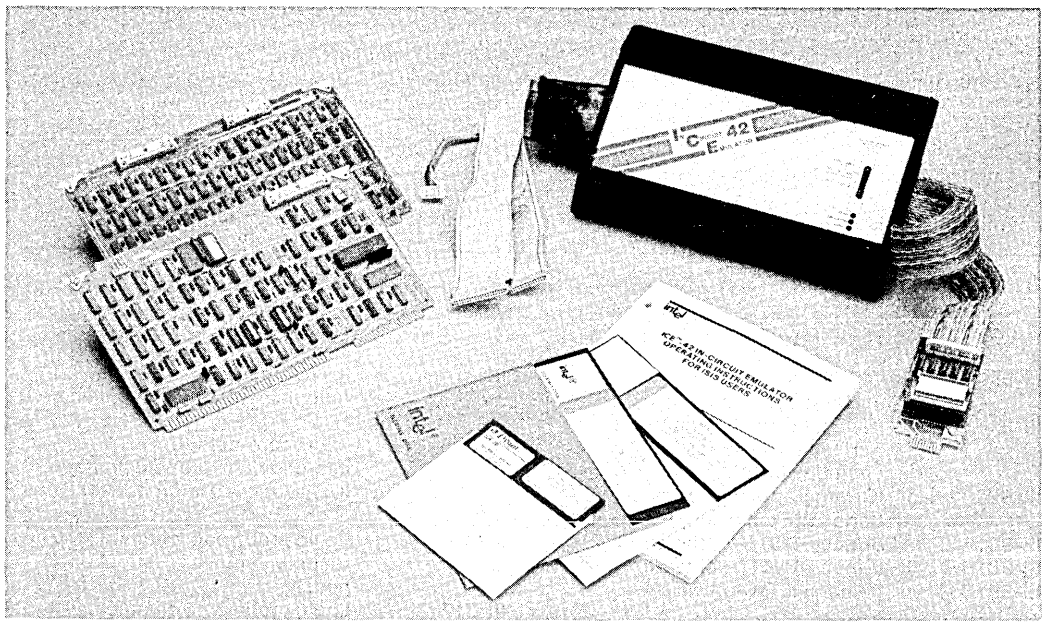
ORDERING INFORMATION

Order Code	Description
III-810	I²ICE logic timing analyzer for Series IV
III-811	I²ICE logic timing analyzer for Series III and Series IIIE (includes III-820)
III-820	IOC upgrade (for Series III and Series IIIE only)
III-981A	8-inch SD iLTA software
III-981B	8-inch DD iLTA software
III-981C	5¼-inch iLTA software
III-813	iLTA terminator set, 16 channel (supplements iLTA-supplied set)
III-814	iLTA terminator set, 8 channel (supplements iLTA-supplied set)
III-815	Microhook set (40 microhooks — supplements iLTA-supplied microhooks)
III-816	Logic probe pod, channels 0-7 (supplements iLTA-supplied pod)
III-817	Logic probe pod, channels 8-F (supplements iLTA-supplied pod)

ICE™-42
8042 IN-CIRCUIT EMULATOR

- **Precise, full-speed, real-time emulation**
  - **Load, drive, timing characteristics**
  - **Full-speed program RAM**
  - **Parallel ports**
  - **Data Bus**
- **User-specified breakpoints**
- **Execution trace**
  - **User-specified qualifier registers**
  - **Conditional trigger**
  - **Symbolic groupings and display**
  - **Instruction and frame modes**
- **Emulation timer**
- **Full symbolic debugging**
- **Single-line assembly and disassembly for program instruction changes**
- **Macro commands and conditional block constructs for automated debugging sessions**
- **HELP facility: ICE™-42 command syntax reference at the console**
- **User confidence test of ICE™-42 hardware**

The ICE™-42 module resides in the Intellec Microcomputer Development System and interfaces to any user-designed 8042 or 8041A system through a cable terminating in an 8042 emulator microprocessor and a pin-compatible plug. The emulator processor, together with 2K bytes of user program RAM located in the ICE-42 buffer box, replaces the 8042 device in the user system while maintaining the 8042 electrical and timing characteristics. Powerful Intellec debugging functions are thus extended into the user system. Using the ICE-42 module, the designer can emulate the system's 8042 chip in real-time or single-step mode. Breakpoints allow the user to stop emulation on user-specified conditions, and a trace qualifier feature allows the conditional collection of 1000 frames of trace data. Using the single-line 8042 assembler the user may alter program memory using the 8042 assembler mnemonics and symbolic references, without leaving the emulator environment. Frequently used command sequences can be combined into compound commands and identified as macros with user-defined names.



MAY 1983
ORDER NUMBER:210818-002

FUNCTIONAL DESCRIPTION

Integrated Hardware and Software Development

The ICE-42 emulator allows hardware and software development to proceed interactively. This approach is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-42 module, prototype hardware can be added to the system as it is designed. Software and hardware integration occurs while the product is being developed. Figure 1 shows the ICE-42 emulator connected to a user prototype.

The ICE-42 emulator assists four stages of development:

SOFTWARE DEBUGGING

This emulator operates without being connected to the user's system before any of the user's hardware is available. In this stage ICE-42 debugging capabilities can be used in conjunction with the Intellec text editor and 8042 macroassembler to facilitate program development.

HARDWARE DEVELOPMENT

The ICE-42 module's precise emulation characteristics and full-speed program RAM make it a valuable tool for debugging hardware.

SYSTEM INTEGRATION

Integration of software and hardware begins when any functional element of the user system hardware is connected to the 8042 socket. As each section of the user's hardware is completed, it is added to the prototype. Thus, each section of the hardware and software is "system" tested in real-time operation as it becomes available.

SYSTEM TEST

When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-42 module is then used for real-time emulation of the 8042 chip to debug the system as a completed unit.

The final product verification test may be performed using the 8742 EPROM version of the 8042 microcomputer. Thus, the ICE-42 module provides the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

Symbolic Debugging

The ICE-42 emulator permits the user to define and to use symbolic, rather than absolute, references to program and data memory addresses. Thus, there is no need to recall or look up the addresses of key locations in the program, or to become involved with machine code.

When a symbol is used for memory reference in an ICE-42 emulator command, the emulator supplies the corresponding location as stored in the ICE-42 emulator symbol table. This table can be loaded with the symbol table produced by the assembler during application program assembly. The user obtains the symbol table during software preparation simply by using the "DEBUG" switch in the 8042 macroassembler. Furthermore, the user interactively modifies the emulator symbol table by adding new symbols or changing or deleting old ones. This feature provides great flexibility in debugging and minimizes the need to work with hexadecimal values.

Through symbolic references in combination with other features of the emulator, the user can easily:

- Interpret the results of emulation activity collected during trace.
- Disassemble program memory to mnemonics, or assemble mnemonic instructions to executable code.
- Reference labels or addresses defined in a user program.

Automated Debugging and Testing

MACRO COMMAND

A macro is a set of commands given a name. A group of commands executed frequently can be defined as a macro. The user executes the group of commands by typing a colon followed by the macro name. Up to ten parameters may be passed to the macro.

Macro commands can be defined at the beginning of a debug session and then used throughout the whole session. One or more macro definitions can be saved on diskette for later use. The Intellec text editor may be used to edit the macro file. The macro definitions are easy to include in any later emulation session.

The power of the development system can be applied to manufacturing testing as well as development by writing test sequences as macros. The macros are stored on diskettes for use during system test.

COMPOUND COMMAND

Compound commands provide conditional execution of commands (IF command) and execution of commands repeatedly until certain conditions are met (COUNT, REPEAT commands).

Compound commands may be nested any number of times, and may be used in macro commands.

Example:

```
*DEFINE .I=0          ; Define symbol .I to 0
*COUNT 100H           ; Repeat the following
                        commands 100H times.
.*IF .I AND 1 THEN    ; Check if .I is odd
..*CBYTE.I=.I         ; Fill the memory at
                        location .I to value .I
..*END
.*.I-.I+1             ; Increment .I by 1.
.*END                 ; Command executes
                        upon carriage-return
                        after END
```

(The asterisks are system prompts; the dots indicate the nesting level of compound commands.)

Operating Modes

The ICE-42 software is an Intellec RAM-based program that provides easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-42 commands are configured with a broad range of modifiers that provide maximum flexibility in describing the operation to be performed.

EMULATION

The ICE-42 module can emulate the operation of prototype 8042 system, at real-time speed (up to 12MHz) or in single steps. Emulation commands to the ICE-42 module control the process of setting up, running, and halting an emulation of the user's 8042-based system. Breakpoints and tracepoints enable the ICE-42 emulator to halt emulation and provide a detailed trace of execution in any part of the user's program. A summary of the emulation commands is shown in Table 1.

Table 1 Major Emulation Commands

Command	Description
GO	Begins real-time emulation and optionally specifies break conditions.
BR0, BR1, BR	Sets or displays either or both Breakpoint Registers used for stopping real-time emulation.
STEP	Performs single-step emulation.
QR0, QR1	Specifies match conditions for qualified trace.
TR	Specifies or displays trace-data collection conditions and optionally sets Qualifier Register (QR0, QR1).
Synchronization Line Commands	Sets and displays status of synchronization line outputs or latched inputs. Used to allow real-time emulation or trace to start and stop synchronously with external events.

Breakpoints

The ICE-42 hardware includes two breakpoint registers that allow halting of emulation when specified conditions are met. The emulator continuously compares the values stored in the breakpoint registers with the status of specified address, opcode, operand, or port values, and halts emulation when this comparison is satisfied. When an instruction initiates a break, that instruction is executed completely before the break takes place. The ICE-42 emulator then regains control of the console and enters the interrogation mode. With the breakpoint feature, the user can request an emulation break when the program:

- Executes an instruction at a specific address or within a range of addresses.

- Executes a particular opcode.
- Receives a specific signal on a port pin.
- Fetches a particular operand from the user program memory.
- Fetches an operand from a specific address in program memory.

Trace and Tracepoints

Tracing is used with real-time and single-step emulation to record diagnostic information in the trace buffer as a program is executed. The information collected includes opcodes executed, port values, and memory addresses. The ICE-42 emulator collects 1000 frames of trace data.

If desired this information can be displayed as assembler instruction mnemonics for analysis during interrogation or single-step mode. The trace-collection facility may be set to run conditionally or unconditionally. Two unique trace qualifier registers, specified in the same way as breakpoint registers, govern conditional trace activity. The qualifiers can be used to condition trace data collection to take place as follows:

- Under all conditions (forever).
- Only while the trace qualifier is satisfied.
- For the frames or instructions preceding the time when a trace qualifier is first satisfied (pre-trigger trace).
- For the frames or instructions after a trace qualifier is first satisfied (post-triggered trace).

Table 2 shows an example of trace display.

INTERROGATION AND UTILITY

Interrogation and utility commands give convenient access to detailed information about the

Table 2 Trace Display (Instruction Mode)

FRAME	LOC	OBJ	INSTRUCTION	P1	P2	T0	T1	DBYIN	YOUT	YSTS	TOVF
0000:	100H	2355	MOV A,#55H	FFH	FFH	0	0	66H	DFH	02H	0
0004:	102H	39	OUTL P1,A	FFH	FFH	0	0	66H	DFH	02H	0
0008:	103H	3A	OUTL P2,A	55H	FFH	0	0	66H	DFH	02H	0
0012:	104H	22	IN A,DBB	55H	55H	0	0	66H		02H	0
0014:	105H	37	CPL A	55H	55H	0	0		DFH	02H	0
0016:	106H	02	OUT DBB,A	55H	55H	0	0	66H		00H	0
0018:	107H	BA03	MOV R2,#03H	55H	55H	0	0	66H	99H	00H	0
0022:	109H	B840	MOV R0,#.TABLE0	55H	55H	0	0	66H	99H	01H	0
0026:	108H	B960	MOV R1,#.TABLE1	55H	55H	0	0	66H	99H	01H	0
.LOOP											
0030:	10DH	F0	MOV A,@R0	55H	55H	0	0		99H	01H	0
0032:	10EH	A1	MOV @R1,A	55H	55H	0	0	66H		01H	0
0034:	10FH	18	INC R0	55H	55H	0	0		99H	01H	0
0036:	110H	19	INC R1	55H	55H	0	0	66H		01H	0
0038:	111H	EA0D	DJNZ R2,.LOOP	55H	55H	0	0	66H	99H	01H	0
.LOOP											
0042:	10DH	F0	MOV A,@R0	55H	55H	0	0		99H	01H	0
0044:	10EH	A1	MOV @R1,A	55H	55H	0	0	66H		01H	0
0046:	10FH	18	INC R0	55H	55H	0	0		99H	01H	0
0048:	110H	19	INC R1	55H	55H	0	0	66H		01H	0
0050:	111H	EA0D	DJNZ R2,.LOOP	55H	55H	0	0	66H	99H	01H	0
.LOOP											
0054:	10DH	F0	MOV A,@R0	55H	55H	0	0		99H	01H	0
0056:	10EH	A1	MOV @R1,A	55H	55H	0	0	66H		01H	0
0058:	10FH	18	INC R0	55H	55H	0	0		99H	01H	0
0060:	110H	19	INC R1	55H	55H	0	0	66H		01H	0
0062:	111H	EA0D	DJNZ R2,.LOOP	55H	55H	0	0	66H	99H	01H	0
0066:	113H	00	NOP	55H	55H	0	0		99H	01H	0

user program and the state of the 8042 that is useful in debugging hardware and software. Changes can be made in memory and in the 8042 registers, flags, and port values. Commands are also provided for various utility operations such as loading and saving program files, defining symbols, displaying trace data, controlling system synchronization and returning control to ISIS-II. A summary of the basic interrogation and utility commands is shown in Table 3. Two additional time-saving emulator features are discussed below.

Single-Line Assembler/Disassembler

The single-line assembler/disassembler (ASM and DASM commands) permits the designer to examine and alter program memory using assembly language mnemonics, without leaving the emulator environment or requiring time-consuming program reassembly. When assembling new mnemonic instructions into program memory, previously defined symbolic references (from the original program assembly, or subsequently defined during the emulation session)

Table 3 Major Interrogation and Utility Commands

Command	Description
HELP	Displays help messages for ICE-42 emulator command-entry assistance.
LOAD	Loads user object program (8042 code) into user-program memory, and user symbols into ICE-42 emulator symbol table.
SAVE	Saves ICE-42 emulator symbol table and/or user object program in ISIS-II hexadecimal file.
LIST	Copies all emulator console input and output to ISIS-II file.
EXIT	Terminates ICE-42 emulator operation.
DEFINE	Defines ICE-42 emulator symbol or macro.
REMOVE	Removes ICE-42 emulator symbol or macro.
ASM	Assembles mnemonic instructions into user-program memory.
DASM	Disassembles and displays user-program memory contents.
Change/Display Commands	Change or display value of symbolic reference in ICE-42 emulator symbol table, contents of key-word references (including registers, I/O ports, and status flags), or memory references.
EVALUATE	Evaluates expression and displays resulting value.
MACRO	Displays ICE-42 macro or macros.
INTERRUPT	Displays contents for the Data Bus and timer interrupt registers.
SECONDS	Displays contents of emulation timer, in microseconds.
Trace Commands	Position trace buffer pointer and select format for trace display.
PRINT	Displays trace data pointed to by trace buffer pointer.
MODE	Sets or displays the emulation mode, 8041A or 8042.

Table 4 HELP Command

```
*HELP
Help is available for the following items. Type HELP followed by the item name.
The help items cannot be abbreviated. (For more information, type HELP HELP or
HELP INFO.)
Emulation:   Trace Collection:          Misc:          <address>
GO GR SYO    TR   QR   QRO  QR1  SY1   BASE           <CPU$keyword>
BR BROBR1                              DISABLE        <expr>
STEP         Trace Display:            ENABLE         <ICE42 #keyword>
             TRACE   MOVE    PRINT     ERROR          <identifier>
             OLDEST NEWEST             EVALUATE       <instruction>
                                       HELP           <masked#constant>
Change/       Display/ Define/ Remove: INFO           <match$cond>
<CHANGE>     REMOVE   CBYTE            <LIGHTS>       <numeric$constant>
<DISPLAY>    SYMBOL   DBYTE   DASM     LIST           <partition>
REGISTER     RESET            ASM      LOAD           <string>
                                       MODE
SECONDS      WRITE                     SAVE           <string$constant>
DEFINE       STACK            SY       SUFFIX         <symbolic$ref>
                                       SYMBOLIC       <mode>
Macro:              Compound                          <trace$reference>
DEFINE       DIR    Commands:                         <unlimited$match$cond>
DISABLE      ENABLE COUNT                             <user$symbols>
INCLUDE      PUT    IF
<MACRO$DISPLAY>      REPEAT
<MACRO$INVOCATION>
*
*
*HELP IF
IF - The conditional command allows conditional execution of one or more commands
based on the values of boolean conditions.
     IF <expr> 'THEN  <cr>              <true$list>::='<command> <cr>  @
      <true$list>                       <false$list>::='<command> <cr>  @
      'ORIF <expr> <cr>                 <command>::=An ICE-42 command.
      <true$list>  @
      'ELSE <cr>
      <false$list>
      END
The <expr>s are evaluated in order as 16-bit unsigned integers. If one is
reached whose value has low-order bit 1 (TRUE), all commands in the <true$list>
following that <expr> are then executed and all commands in the other <true$-
list>s and in the <false$list> are skipped. If all <expr>s have value with low-
order bit 0 (FALSE), then all commands in all <true$list>s are skipped and, if
ELSE is present, all commands in the <false$list> are executed.
     (EX:  IF .LOOP=5 THEN
             STEP
             ELSE
             GO
             END)
*
*
*
*
*EXIT
```

may be used in the instruction operand field. The emulator supplies the absolute address or data values as stored in the emulator symbol table. These features eliminate user time spent translating to and from machine code and searching for absolute addresses, with a corresponding reduction in transcription errors.

HELP

The HELP file allows display of ICE-42 command syntax information at the Intellec console. By typing "HELP", a listing of all items for which help messages are available is displayed. Typing "HELP <Item>" then displays relevant information about the item requested, including typical usage examples. Table 4 shows some sample HELP messages.

EMULATION ACCURACY

The speed and interface demands of a high-performance single-chip microcomputer require extremely accurate emulation, including full-speed, real-time operation with the full function of the microcomputer. The ICE-42 module achieves accurate emulation with an 8042 emulator chip, a special configuration of the 8042 microcomputer family, as its emulation processor.

Each of the 40 pins on the user plug is connected directly to the corresponding 8042 pin on the emulator chip. Thus the user system sees the emulator as an 8042 microcomputer at the 8042 socket. The resulting characteristics provide extremely accurate emulation of the 8042 including speed, timing characteristics, load and drive values, and crystal operation. However, the emulator may draw more power from the user system than a standard 8042 family device.

Additional emulator processor pins provide signals such as internal address, data, clock, and control lines to the emulator buffer box. These signals let static RAM in the buffer box substitute for on-chip program ROM or EPROM. The emulator chip also gives the ICE module "back-door" access to internal chip operation, allowing the emulator to break and trace execution without interfering with the values on the user-system pins.

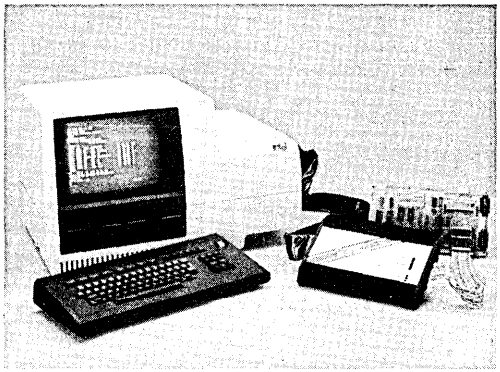


Figure 1 A Typical 8042 Development Configuration. The host system is an Intellec Series IV. The ICE-42 module is connected to a user prototype system.

SPECIFICATIONS

ICE™-42 Operating Requirements

Intellec Model 800, Series II, Series III, or Series IV Microcomputer Development SYstem (64K RAM required)

System console (Model 800 only)

Intellec Diskette Operating System: ISIS (Version 3.4 or later).

Equipment Supplied

- Printed circuit boards (2)
- Emulation buffer box, Intellec interface cables, and user-interface cable with 8042 emulation processor
- Crystal power accessory
- Operating instructions manuals
- Diskette-based ICE-42 software (single and double density)

Emulation Clock

User's system clock (up to 12MHz) or ICE-42 crystal power accessory (12 MHz)

Environmental Characteristics

Operating Temperature — 0° to 40°C

Operating Humidity — Up to 95% relative humidity without condensation.

Physical Characteristics

Printed Circuit Boards

Width: 12.00 in. (30.48 cm)
Height: 6.75 in. (17.15 cm)
Depth: 0.50 in. (1.27 cm)

Buffer Box

Width: 8.00 in. (20.32 cm)
Length: 12.00 in. (30.48 cm)
Depth: 1.75 in. (4.44 cm)
Weight: 4.0 lb. (1.81 kg)

Electrical Characteristics

DC Power Requirements (from Intellec® system)

$V_{CC} = +5V, \pm 5\%$
$I_{CC} = 13.2A$ max; 11.0A typical
$V_{DD} = +12V, \pm 5\%$
$I_{DD} = 0.1A$ max; 0.05A typical
$V_{BB} = -10V, \pm 5\%$
$I_{BB} = 0.05A$ max; 0.01A typical

User plug characteristics at 8042 socket — Same as 8042 or 8742 except that the user system sees an added load of 25 pF capacitance and 50μA leakage from the ICE-42 emulator user plug at ports 1, 2, T0, and T1.

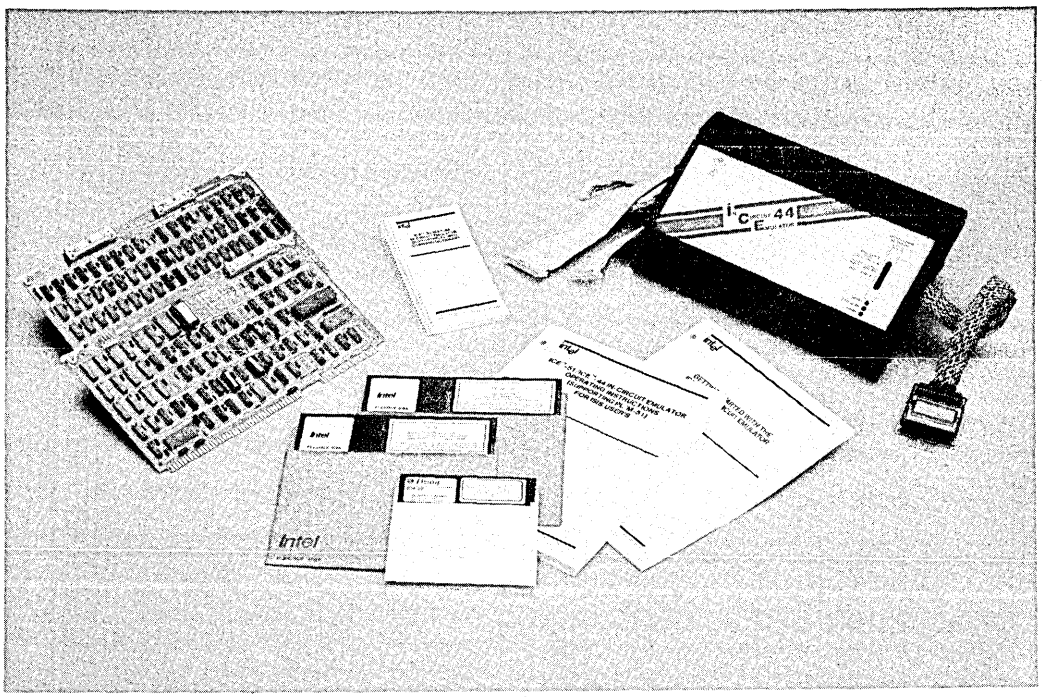
ORDERING INFORMATION

Part Number	Description
ICE-42	8042 Microcontroller In-Circuit Emulator, cable assembly and interactive diskette software

ICE™-44 MODULE
8044 IN-CIRCUIT EMULATOR

- **Precise, full-speed, real-time emulation**
- **8K bytes full-speed RAM**
- **User-specified breakpoints**
- **Execution trace**
  - **-User-specified qualifier registers**
  - **-Conditional trigger**
  - **-Symbolic groupings and display**
  - **-Instruction and frame modes**
- **Emulation timer**
- **Full symbolic debugging**
- **Single-line assembly and disassembly for program instruction changes**
- **Macro commands and conditional block constructs for automated debugging sessions**

The ICE™-44 module resides in the Intellec® Microcomputer Development System and interfaces to any user-designed 8044 system through a cable terminating in an 8044 emulator microprocessor and a pin-compatible plug. The emulator processor, together with 8K bytes of user RAM located in the ICE-44 buffer box, replaces the 8044 device in the user system while maintaining the 8044 electrical and timing characteristics. Powerful Intellec debugging functions are thus extended into the user system. Using the ICE-44 module, the designer can emulate the system's 8044 in real-time or single -step mode. Breakpoints allow the user to stop emulation on user-specified conditions, and a trace qualifier feature allows the conditional collection of 1000 frames of trace data. Using the single-line 8044 assembler, the user may alter program memory using 8044 assembler mnemonics and symbolic references, without leaving the emulator environment. Frequently used command sequences can be combined into compound commands and identified as macros with user-defined names.



SEPTEMBER, 1984
ORDER NUMBER: 210369-002

FUNCTIONAL DESCRIPTION

The ICE-44 emulator aids the design effort in several ways: software and hardware integration and debugging, symbolic debugging, and automated debugging and testing. The following sections describe these features.

Integrated Hardware and Software Development

The ICE-44 emulator allows hardware and software development to proceed interactively. This approach is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-44 module, prototype hardware can be added to the system as it is designed. Software and hardware integration occurs while the product is being developed.

The ICE-44 emulator assists in four stages of development, as described in the following sections.

SOFTWARE DEBUGGING

The ICE-44 can be operated without being connected to the user's system and before any of the user's hardware is available. In this stage, ICE-44 debugging capabilities can be used with the Intellec text editor and the 8044 macro assembler to facilitate program development.

HARDWARE DEVELOPMENT

The ICE-44 module's precise emulation characteristics and full-speed program RAM make it a valuable tool for debugging hardware, including the time-critical synchronous data link control (SDLC) serial port, parallel port, and timer interfaces.

SYSTEM INTEGRATION

Software and hardware integration can begin when any functional element of the user system hardware is connected to the 8044 socket. As each section of the user's hardware is completed, it is added to the prototype. Thus, each section of the hardware and software is system tested in real-time operation as it becomes available.

SYSTEM TEST

When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-44 module can then be used for real-time emulation of the 8044 to debug the system as a completed unit.

The final product verification test may be performed using the 8744 EPROM version of the 8044 microcomputer. Thus, the ICE-44 module provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools. Figure 1 shows an 8044 development configuration.

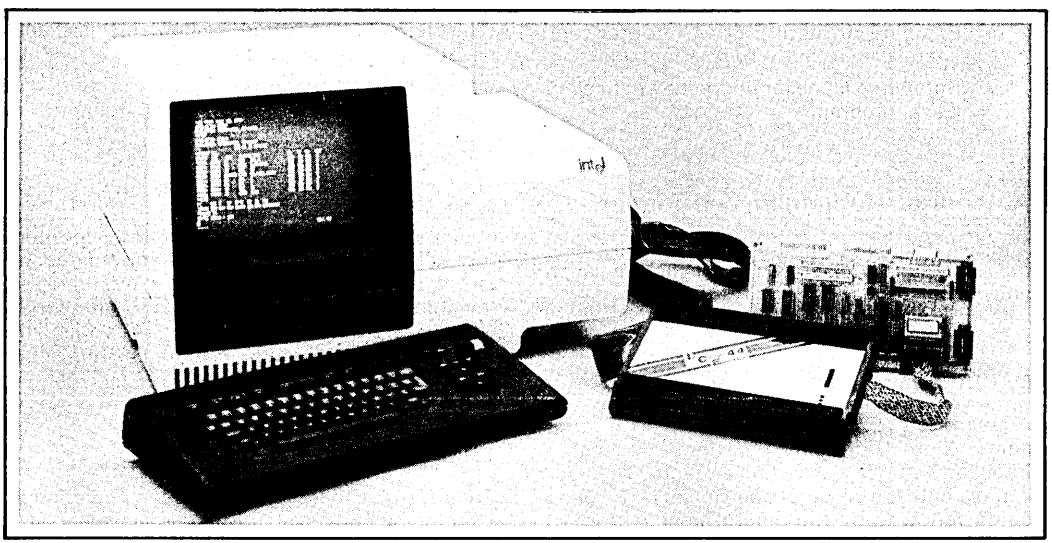


Figure 1. A Typical 8044 Development Configuration. The host system is an Intellec® Series IV. The ICE™-44 module is connected to a user prototype system.

Order Number: 210369-002

Symbolic Debugging

The ICE-44 emulator permits the user to define and use symbolic (rather than absolute) references to program and data memory addresses; additional symbols are predefined by the ICE-44 software for referencing registers, flags, and input/output ports. Thus, the user need not recall or look up the addresses of key locations in a program as they change with each assembly, or become involved with machine code.

When a symbol is used for memory reference in an ICE-44 emulator command, the emulator supplies the corresponding location as stored in the ICE-44 emulator symbol table. This table is loaded with the symbol table produced by the assembler during application program assembly. The user can obtain the symbol table during software preparation simply by using the DEBUG switch in the ASM44 macro assembler. Furthermore, the user can interactively modify the emulator symbol table by adding new symbols, or changing or deleting old ones. This feature provides flexibility in debugging and minimizes the need to work with hexadecimal values.

Through symbolic references in combination with other features of the emulator, the user can easily do the following:

- Interpret the results of emulation activity collected during trace.
- Disassemble program memory to mnemonics, or assemble mnemonic instructions to executable code.
- Examine or modify 8044 internal registers, data memory, or port contents.
- Reference labels or addresses defined in a user program.

Automated Debugging and Testing

The following sections describe ways in which the user can automate some of the emulation and debug commands.

MACRO COMMANDS

A macro is a set of commands that is given a name. A group of commands that is executed frequently can be defined as a macro. The user can execute the group of commands by typing a colon followed by the macro name. Up to 10 parameters may be passed to a macro.

Macro commands can be defined at the beginning of a debug session and then used throughout a session. Macro definitions can be saved on disk for later use. The Intellec text editor may be used to edit the macro file.

The power of the development system can be applied to manufacturing testing as well as development, by writing test sequences as macros. The macros are stored on disk for use during system test.

COMPOUND COMMANDS

There are two kinds of compound commands. The IF command permits conditional execution of commands, and the COUNT and REPEAT commands allow repetitious execution of commands until certain conditions are met.

Compound commands may be nested any number of times, and they may be used in macro commands.

Example:

```
*DEFINE .I=0          ;Define symbol .I as 0
*COUNT 100H           ;Repeat the following
                      ;commands 100H times
.*IF .I AND 1 THEN    :Check if .I is odd
..*BYTE .I=.I         ;Fill the memory at location .I
                      ;to value .I
..*END
.*.I=.I+1             ;Increment by 1
.*END                 ;Command executes upon
                      ;carriage return after END
```

(The characters *, .*, and ..* shown in this example are system prompts that indicate the nesting level of compound commands.)

Operating Commands

The ICE-44 software is an Intellec RAM-based program that provides the user with easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-44 commands are configured with a broad range of modifiers that provide the user with maximum flexibility in describing the operation to be performed.

EMULATION

The ICE-44 module can emulate the operation of a prototype 8044 system, at real-time speed (1.2 to 12 MHz) or in single-steps. Emulation

commands to the ICE-44 module control the process of setting up, running, and halting an emulation of the user's 8044-based system. Breakpoints and tracepoints enable the ICE-44 emulator to halt emulation and provide a detailed trace of execution in any part of the user's program. A summary of the emulation commands is shown in Table 1.

Breakpoints

The ICE-44 hardware includes two breakpoint registers that allow the user to halt emulation when specified conditions are met. The emulator continuously compares the values stored in the breakpoint registers with the status of specified address, opcode, operand, or port values, and halts emulation when this comparison is satisfied. When an instruction initiates a break, that instruction is executed completely before the break takes place. The ICE-44 emulator then regains control of the console and enters the interrogation mode. With the breakpoint feature, the user can specify an emulation break when the program:

- Executes an instruction at a specified address or within a range of addresses.
- Executes a particular opcode.
- Receives a specific signal on a port pin.
- Fetches a particular operand from the user program memory.
- Fetches an operand from a specific address in program memory.

Trace and Tracepoints

Tracing is used with both real-time and single-step emulation to record diagnostic information in the trace buffer as a program is executed. The information collected includes opcodes executed, port values, and memory addresses. The ICE-44 emulator collects up to 1000 frames of trace data.

The trace data can be displayed as assembler instruction mnemonics, if desired, for analysis during interrogation or single-step mode. The trace-collection facility may be set to run conditionally or unconditionally. Two unique trace qualifier registers, specified in the same way as breakpoint registers, govern conditional trace activity. The qualifiers can be used to condition trace data collection to take place as follows:

- Under all conditions (forever).
- Only while the trace qualifier is satisfied.
- For the frames or instructions preceding the time when a trace qualifier is first satisfied (pre-triggered trace).
- For the frames or instructions after a trace qualifier is first satisfied (post-triggered trace).

Figure 2 shows an example of a trace display in instruction mode.

Table 1. Major Emulation Commands

Command	Description
GO	Begins real-time emulation and optionally specifies break conditions.
BR0, BR1, BR	Sets or displays either or both breakpoint registers used for stopping real-time emulation.
STEP	Performs single-step emulation.
QR0, QR1	Specifies match conditions for qualified trace.
TR	Specifies or displays trace-data collection conditions and optionally sets the qualifier register (QR0, QR1).
Synchronization line Commands	Sets and displays the status of synchronization line output or latched input. Used to synchronize the starting and stoping of real-time emulation or trace to occur with external events.

Order Number: 210369-002

```
*P ALL
FRAME  LOC    OBJ      INSTRUCTION      P1   P2   P0   TOVF
0000   0000H  F0       MOVX @DPTR,A     00H  88H  FFH    0
0007:  0001H  F8       MOV  R0, A       00H  88H  FFH    0
0011:  0002H  F4       CPL  A           00H  88H  FFH    0
0015:  0003H  33       RCL  A           00H  88H  FFH    0
0019:  0004H  759044   MOV  .P1, #44H   00H  88H  FFH    0
0027:  0007H  C3       CLR  C           00H  88H  FFH    0
0031:  0008H  98       SUBB A, B0       44H  88H  FFH    0
0035:  0009H  D000     POP  00H         44H  88H  FFH    0
0043:  0008H  753405   MOV  34H, #.FIVE 44H  88H  FFH    0
0044:  000EH  0156     AJMP 0056H       44H  88H  FFH    0
```

Figure 2. Sample Trace Display in Instruction Mode

INTERROGATION AND UTILITY COMMANDS

Interrogation and utility commands allow convenient access to detailed information about the user program and the state of the 8044 that is useful in debugging hardware and software. Changes can be made in memory and in the 8044 registers, flags, and port values. Commands are also provided for various utility operations such as loading and saving program files, defining symbols, displaying trace data, controlling system synchronization, and returning control to ISIS. A summary of the basic interrogation and utility commands is shown in Table 2.

Table 2. Major Interrogation and Utility Commands

Command	Description
HELP	Displays help messages for ICE-44 emulator command-entry assistance.
LOAD	Loads the user object program (8044 code) into user program memory, and the user symbols into the ICE-44 emulator symbol table.
SAVE	Saves the ICE-44 emulator symbol table and the user object program in an ISIS hexadecimal file.
LIST	Copies all emulator console input and output to an ISIS file.
GO	Begins ICE-44 emulation .
EXIT	Terminates ICE-44 emulation operation and returns control to the ISIS operating system.
DEFINE	Defines the ICE-44 emulator symbol or macro.
REMOVE	Deletes user-defined symbols, modules, or macro names from the symbol table or macro table.
ASM	Assembles mnemonic instructions into user program memory.
DASM	Disassembles and displays user program memory contents.
Change/Display Commands	Changes or displays the value of a symbolic reference in the ICE-44 emulator symbol table, the contents of keyword references (including registers, I/O ports, and status flags), or the memory references.
EVALUATE	Evaluates an expression and displays the resulting value.
MACRO	Displays an ICE-44 macro or macros.
INTERRUPT	Displays serial, external, or timer-interrupt register settings.
SECONDS	Displays the contents of the emulation timer in microseconds.
Trace Commands	Positions the trace buffer pointer and selects the format for the trace display.
PRINT	Displays the trace data pointed to by the trace buffer pointer.

Single-line Assembler/Disassembler

The single-line assembler/disassembler (ASM and DASM commands) is a time-saving emulator feature that permits the designer to examine and alter program memory using assembly language mnemonics, without leaving the emulator environment or requiring time-consuming program reassembly. When assembling new mnemonic instructions into program memory, previously defined symbolic references (from the original program assembly, or subsequently defined during the emulation session) may be used in the instruction operand field. The emulator will supply the absolute address or data values as stored in the emulator symbol table. These features eliminate user time spent translating to and from machine code and searching for absolute addresses, with a corresponding reduction in transcription errors.

Help

The HELP file allows the user to display ICE-44 command syntax information at the Intellec console. Typing HELP displays a listing of all items for which help messages are available; typing HELP <item> displays relevant information about the item requested, including typical usage examples. See Figures 3 and 4 for screen displays of a HELP menu and a HELP <item> menu.

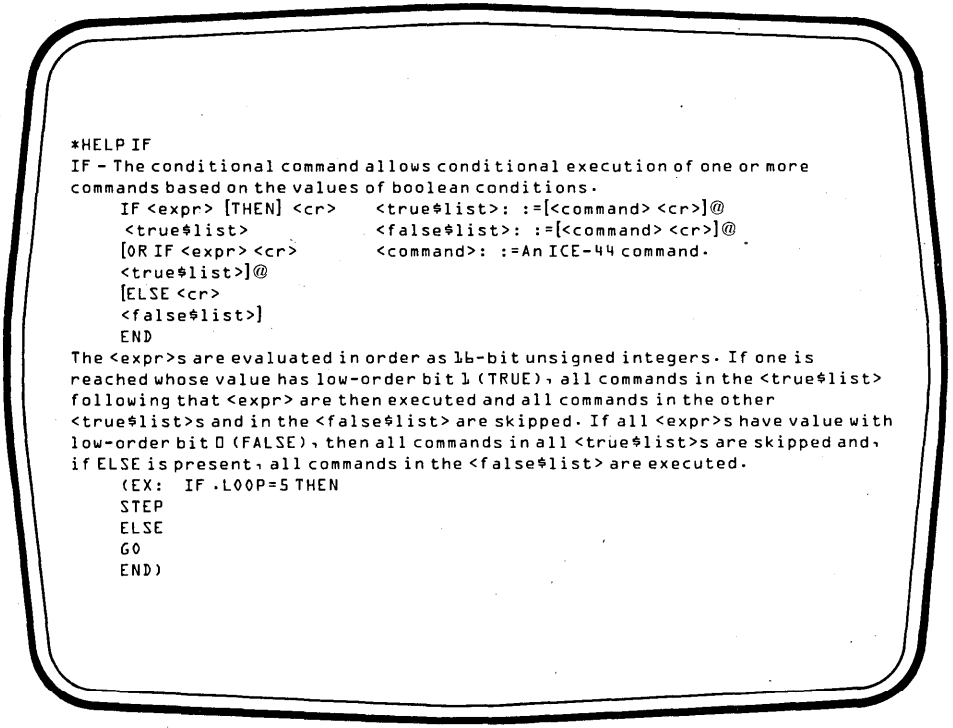
Emulation Accuracy

The speed and interface demands of a high-performance single-chip microcomputer require extremely accurate emulation, including full-speed, real-time operation with the full function of the microcomputer. The ICE-44 emulator achieves accurate emulation with an 8044 bond-out chip, a special configuration of the 8044 microcomputer, as its emulation processor.

Each of the 40 pins on the user plug is connected directly to the corresponding 8044 pin on the bond-out chip. Thus, the user system sees the emulator as an 8044 microcomputer at the 8044 socket. The resulting characteristics provide extremely accurate emulation of the

```
*HELP
Help is available for the following items. Type HELP followed by the item name.
The help items cannot be abbreviated. (for more information, type HELP HELP or
HELP INFO.)
Emulation:      Trace Collection:          Misc:    <address>
GO GR SYO       TR QR QRO QR1 SY1          BASE     <CPU$keyword>
BR BRO BR1                                 DISABLE  <expr>
STEP            Trace Display:             ENABLE   <ICE44$keyword>
                TRACE MOVE PRINT           ERROR    <identifier>
                OLDEST NEWEST              EVALUATE <instruction>
                                           HELP     <masked$constant>
Change/Display/Define/Remove:              INFO     <match$cond>
<CHANGE>        REMOVE   CBYTE   RBIT      <LIGHTS> <numeric$ref>
<DISPLAY>       SYMBOL   DBYTE   DASM      LIST     <partition>
REGISTER        RESET    PBYTE   ASM       LOAD     <string>
SECONDS         WRITE    RBYTE   MAP       SAVE     <string$constant>
DEFINE          STACK    XBYTE   SY        SUFFIX   <symbolic$ref>
                                           SYMBOLIC <system$symbolic>
Macro:                     Compound                 <trace$reference>
DEFINE              DIR    Commands:                <unlimited$match$cond>
DISABLE             ENABLE COUNT                    <user$symbols>
INCLUDE             PUT    IF
<MACRO$DISPLAY>            REPEAT
<MACRO$INVOCATION>
```

Figure 3. Menu Display for HELP



```
*HELP IF
IF - The conditional command allows conditional execution of one or more
commands based on the values of boolean conditions.
     IF <expr> [THEN] <cr>     <true$list>: :=[<command> <cr>]@
     <true$list>               <false$list>: :=[<command> <cr>]@
     [OR IF <expr> <cr>        <command>: :=An ICE-44 command.
     <true$list>]@
     [ELSE <cr>
     <false$list>]
     END
The <expr>s are evaluated in order as 16-bit unsigned integers. If one is
reached whose value has low-order bit 1 (TRUE), all commands in the <true$list>
following that <expr> are then executed and all commands in the other
<true$list>s and in the <false$list> are skipped. If all <expr>s have value with
low-order bit 0 (FALSE), then all commands in all <true$list>s are skipped and,
if ELSE is present, all commands in the <false$list> are executed.
     (EX:  IF .LOOP=5 THEN
     STEP
     ELSE
     GO
     END)
```

Figure 4. Menu Display for HELP IF

8044, including speed, timing characteristics, load and drive values, and crystal operation. The emulator may draw more power from the user system than a standard 8044 family device (see Electrical Characteristics).

Additional bond-out pins provide the emulator box with signals such as internal address, data, clock, and control lines. These signals let static RAM in the buffer box substitute for on-chip program ROM, EPROM, or user supplied external program memory. The 8K bytes of full-speed RAM in the buffer box can be mapped in 4K blocks to anywhere within the 64K program memory space of the 8044. The bond-out chip also gives the emulator "backdoor" access to internal chip operation, so that the emulator can break and trace execution without interfering with the values on the user-system pins.

SPECIFICATIONS

ICE™-44 Operating Requirements

Intellec Model 800, Series II/III, or Series IV development system (64K RAM required)

System Console

One disk drive, single-density or double-density

Intellec disk operating system (single or double density) ISIS v. 3.4 or later

Equipment Supplied

- Printed circuit boards(2)
- Emulation buffer-box, Intellec interface cables, and user-interface cable with an 8044 emulation processor
- Dual auxiliary connector kit for the Model 800, Series II/III, and Series IV development systems
- Crystal power accessory
- Literature kit
  -ICE-44 operating instructions manual
  -ICE-44 command dictionary
  -ICE-44 user's guide
- Disk-based ICE-44 software (5 1/4 inch and 8 inch, single and double density)

Order Number: 210369-002

Emulation Clock

User's system clock (1.2 to 12MHz) or ICE-44 crystal power assembly (12MHz)

Environmental Characteristics

Operating Temperature: 0° to 40° C

Operating Humidity: Up to 95% relative humidity without condensation

Physical Characteristics

Printed Circuit Boards

Width: 12.00 in. (30.48cm)
Height: 6.75 in. (17.15 cm)
Depth: 0.50 in. (1.27 cm)

Buffer Box

Width: 8.00 in. (20.32 cm)
Length: 12.00 in. (30.48 cm)
Depth: 1.75 in. (4.44cm)
Weight: 4.0 lb (1.81 kg)

Interface Cables

Host-emulator interface cable length: 48 in. (121.92 cm)
Emulator-user-system interface cable length: 12.00 in. (30.48 cm)

Electrical Characteristics

DC Power Requirements (from the Intellec system):

V_{CC} = +5V, +5%, -2.5%
I_{CC} = 13.2A max; 11.0A typical
V_{DD} = +12V, + 5%
I_{DD} = 0.1 A max; 0.05A typical
V_{BB} = -10V, + 5%
I_{BB} = 0.05A max; 0.01A typical

User Plug Characteristics at the 8044 Socket:

Same as an 8044 or 8744, except that the user system will see an added load of 25 pf capacitance and 50 uA leakage from the ICE-44 emulator user plug at ports 0, 1, and 2.

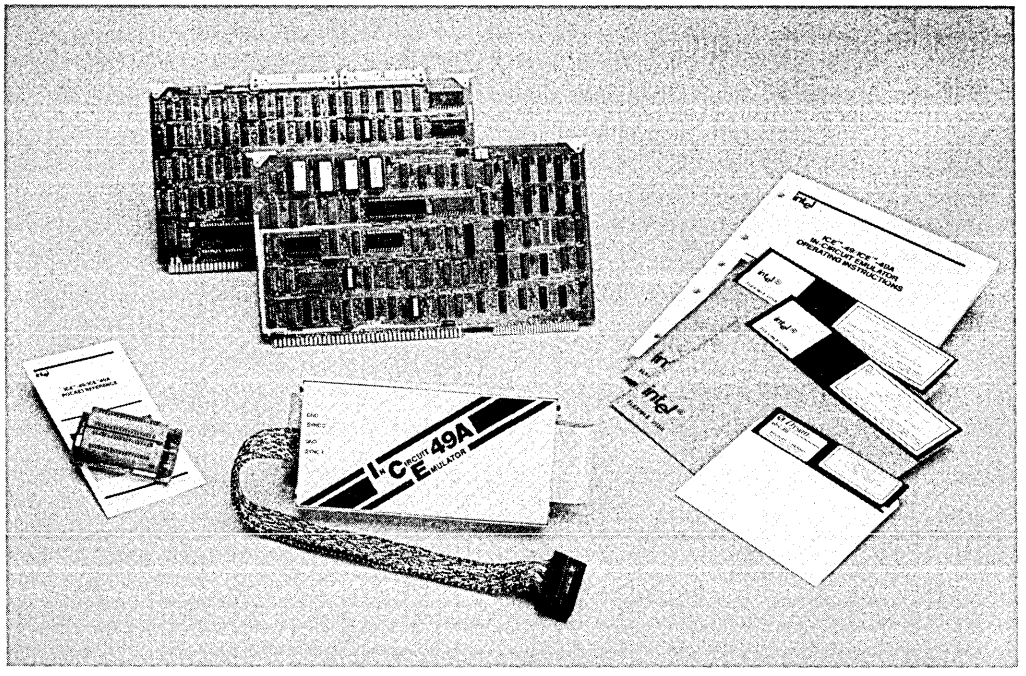
ORDERING INFORMATION

Part Number	Description
ICE-44	8044 microcontroller in-circuit emulator, cable assembly, and interactive disk software

ICE™-49A MCS®-48 IN-CIRCUIT EMULATOR

- **Extends Intellec® microcomputer development system debug power to user configured system via external cable and 40-pin plug, replacing system MCS®-48 device**
- **Emulates user system MCS®-48 device in real-time (up to 11 MHz)**
- **User confidence test of ICE™-49A hardware**
- **Collects bus, register, and MCS®-48 status information on instructions emulated**
- **Provides capability to examine and alter MCS®-48 registers, memory, and flag values, and to examine pin and port values**
- **Integrates hardware and software efforts early to save development time**

The ICE™-49A, MCS®-48 In-Circuit Emulator module is an Intellec system-resident module that interfaces with any MCS-48 system. The MCS-48 family consists of the 8050, 8049, 8048, 8749, 8748, 8040, 8039, 8035, and 8021 microcomputers. The ICE-49A module interfaces with an MCS-48 system through a cable terminating in an MCS-48 pin-compatible plug which replaces the MCS-48 device in the system. With the ICE-49A plug in place, the designer can operate his system in real-time while collecting up to 255 instruction cycles of real-time trace data. In addition, he can single step the system program to monitor more closely the program logic during execution. Static RAM memory is available through the ICE-49A module to emulate MCS-48 program and data memory. The designer can display and alter the contents of data and replacement RAM control memory, internal MCS-48 registers and flags and I/O ports. Powerful debug capability is extended into the MCS-48 system while ICE-49A debug hardware and software remain inside the Intellec system. Symbolic reference capability allows the designer to use meaningful symbols rather than absolute values when examining and modifying memory, registers, flags, and I/O ports in this system.



MAY 1983
ORDER NUMBER:162400-003

FUNCTIONAL DESCRIPTION

Debug Capability Inside User System

The ICE-49A module provides the user with the ability to debug a full prototype or production system without introducing extraneous hardware or software test tools. The module connects to the user system through the socket provided for the MCS-48 device in the user system. Intellec system memory is used for the execution of the ICE-49A software. The Intellec host console and file handling capabilities provide the designer with the ability to communicate with the ICE-49A module and display information on the operation of the prototype system. (The ICE-49A module block diagram is shown in Figure 1.)

Batch Testing

In conjunction with the ISIS diskette operating system, the ICE-49A module can run extensive system diagnostics without operator intervention. The designer or test engineer can define a complete diagnostic exercise, which is stored in a file on the diskette. When activated with an ISIS submit command, this file can instruct the ICE-49A module to execute the diagnostic routine and store the results in another file on the diskette. Results are available to the designer at his convenience. In this way, routine diagnostics and long term testing may be done without tying up valuable manpower.

Integrated Hardware/Software Development

The user prototype need consist of no more than an MCS-48 socket and timing logic to being integration of software and hardware development efforts. Through the ICE-49A module mapping capabilities, Intellec system resources can be accessed to replace prototype memory. Hardware designs can be tested using the system software to drive the final product. Thus, the system integration phase, which can be costly when attempting to mesh completed hardware and software products, becomes a convenient two-way debug tool when begun early in the design cycle.

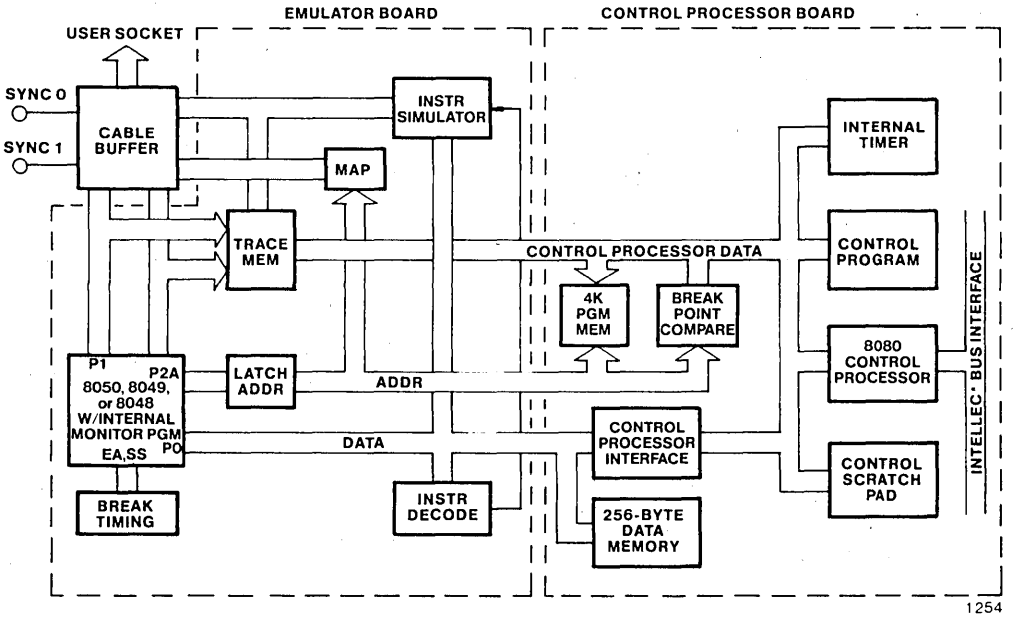


Figure 1 ICE™-49A Module Block Diagram

Real-Time Trace

The ICE-49A module captures trace information while the designer is executing programs in real time. The instructions executed, program counter, port values for bus 0, port 1 and port 2, and the values of selected MCS-48 status lines are stored for the last 255 instruction cycles executed. When retrieved for display, code is disassembled for user convenience. This provides data for determining how the user system was reacting prior to emulation break, and is available whether the break was user initiated or the result of an error condition. For more detailed information on the actions of internal registers, flags, or other system operations, the user may operate in single or multiple step sequences tailored to system debug needs.

Memory Mapping

The 8050, 8749, 8049, 8748, and 8048 contain internal program and data memory. Both program and data memory can be expanded using external memory devices.

Internal Memory

When the MCS-48 microcomputer is replaced by the ICE-49A socket in a system, the ICE-49A module supplies static RAM memory as a replacement for the internal microcomputer memory. The ICE-49A module has enough RAM memory available to emulate up to the total 4K control memory capability of the system. The ICE-49A module also provides for up to 512 bytes of data memory.

External Memory

The ICE-49A module separates replacement control memory into sixteen 256-byte blocks. Replacement external data memory consists of one 256-byte block. Each block of memory can be defined separately as supplied by the user system or supplied by the ICE-49A module. The user may assign ICE-49A equivalent memory to take the place of external memory not yet supplied by his system.

Symbolic Debugging

ICE-49A Emulator software provides symbolic definition of all MCS-48 registers, flags, and selected MCS-48 pins. Symbolically defined pseudo registers provide access to the sense of MCS-48 flip flops which enable time, counter, interrupt, and flag-0/flag-1 options. In addition, the user may reference locations in program and data memory, or their contents, symbolically. The user symbol table generated along with the object file during a program assembly may be loaded to Intellec host memory for access during emulation. The user is encouraged to add to this symbol table any additional symbolic values for memory addresses, constants, or variables he may find useful during system debugging. Symbols may be substituted for numeric values in any of the ICE-49A commands. Symbolic reference is a great advantage to the system designer. He is no longer burdened with the need to recall or look up those addresses of key locations in his program that can change with each assembly. Meaningful symbols from his source program may be used instead. For example, the command:

```
GO FROM .START TILL XDATA. RSLT WRITTEN
```

begins execution of the program at the address referenced by the label START in the designer's assembly program. A breakpoint is set to occur the first time the microprocessor writes to the external data memory location referenced by RSLT. The designer does not have to be concerned with the physical locations of START and RSLT. The ICE-49A software driver supplies them automatically from information stored in the symbol table.

Hardware

The ICE-49A module is a microcomputer system utilizing Intel's 8050, 8749/8049, or 8748/8048 microcomputer as its nucleus. The 8050 provides emulation for the 8040/8050; the 8749 provides emulation for the 8039/8049/8749; the 8748 provides emulation for the 8021/8035/-8048/8748. The ICE-49A module uses an Intel 8080 to communicate with the Intellec host processor via a common memory space. The 8080 also controls an internal ICE-49A bus for intra-module communication. ICE-49A hardware consists of two PC boards, the controller board, and the emulator board, all of which reside in the Intellec chassis. A cable interfaces the ICE-49A boards to the MCS-48 system. The cable terminates in a MCS-48 pin compatible plug which replaces any MCS-48 device in the user system. Figure 2 shows the ICE-49A module used with the Series IV development system and connected to a user prototype board.

Real-Time Trace

Trace Buffer

While the ICE-49A module is executing the user program, it is monitoring port, program counter, data, and status lines. Values for each instruction cycle executed are stored in a 255x44 real-time RAM trace buffer. A resettable timer resident on the controller board counts instruction cycles.

Controller Board

The ICE-49A module talks to the Intellec system as a peripheral device. The controller board receives commands from the Intellec system and responds through the parameter block. Three 15-bit hardware breakpoint registers are available for loading by the user. While in emulation mode, a hardware comparator is constantly monitoring address and status lines for a match to terminate an emulation. The breakpoint registers provide a signal when a match is detected. The user may disable the emulation break capability and use the signal to synchronize other debug tools. The controller board returns real-time trace data, MCS-48 register, flag, and pin values, and ICE-49A status information, to a control block in the Intellec system when emulation is terminated. This information is available to the user through the ICE-49A interrogation commands. Error conditions, when present, are automatically displayed on the Intellec system console. The controller board also contains static RAM memory, which can be used to emulate MCS-48 program and data memory in real time. 4K of memory is available in sixteen 256-byte pages to emulate MCS-48 PROM or PROM program memory. A 256-byte page of data memory is available to access in place of MCS-48 external data memory. The controller board address map directs the ICE-49A module to access either replacement ICE-49A memory or actual user system external memory in 256-byte segments based on information provided by the user.

Emulator Board

The emulator board contains the 8749/8049* and peripheral logic required to emulate the MCS-48 device in the user system. A software selectable 11 MHz or 5.5 MHz clock drives the emulated MCS-48 device. This clock can be disabled and replaced with a user supplied TTL clock in the user system.

Cable Card

The cable card is included for cable driving. It transmits address and data bus information to the user system through a 40-pin connector which plugs into the user system in the socket designed for the MCS-48 device.

Software

The ICE-49A software driver is a RAM-based program which provides the user with an easy to use command language for defining breakpoints, initiating real-time emulation or single step operation, and interrogating and altering user system status recorded during emulation, (See Table 1, Table 2, and Table 3). The ICE-49A command language contains a broad range of modifiers to provide the user with maximum flexibility in defining the operation to be performed. The ICE-49A software driver is available on diskette and operates in 64K of Intellec RAM memory.

Table 1 ICE™-49A Emulation Commands

Command	Operation
Enable	Activates breakpoint and display registers for use with Go and Step commands.
Go	Initiates real-time emulation and allows user to specify breakpoints and data retrieval.
Step	Initiates emulation in single instruction increments. Each step is followed by register dump. User may optionally tailor other diagnostic activity to his needs.
Interrupt	Emulates user system interrupt.

* Use 8748/8048 with internal monitor program when emulating 8748/8048/8035/8021. Use 8050 with internal monitor program when emulating 8050/8040.

Table 2 ICE™-49A Interrogation Commands

Command	Operation
Display	Prints contents of memory, MCS-48 device registers, I/O ports, flags, pins, real-time trace data, symbol table, or other diagnostic data on list device.
Change	Alters contents of memory, register output port, or flag. Sets or alters breakpoints and display registers.
Map	Defines memory status.
Base	Establishes mode of display for output data.
Suffix	Establishes mode of display input data.

Table 3 ICE™-49A Utility Commands

Command	Operation
Load	Fetches user symbol table and object code from input device.
Save	Sends user symbol table and object code to output device.
Define	Enters symbol name and value to user symbol table.
Move	Moves block of memory data to another area of memory.
List	Defines list device.
Exit	Returns program control to ISIS
Evaluate	Converts expression to equivalent values in binary, octal, decimal, and hex.
Remove	Deletes symbols from symbol table.
Reset	Reinitializes ICE-49A hardware.

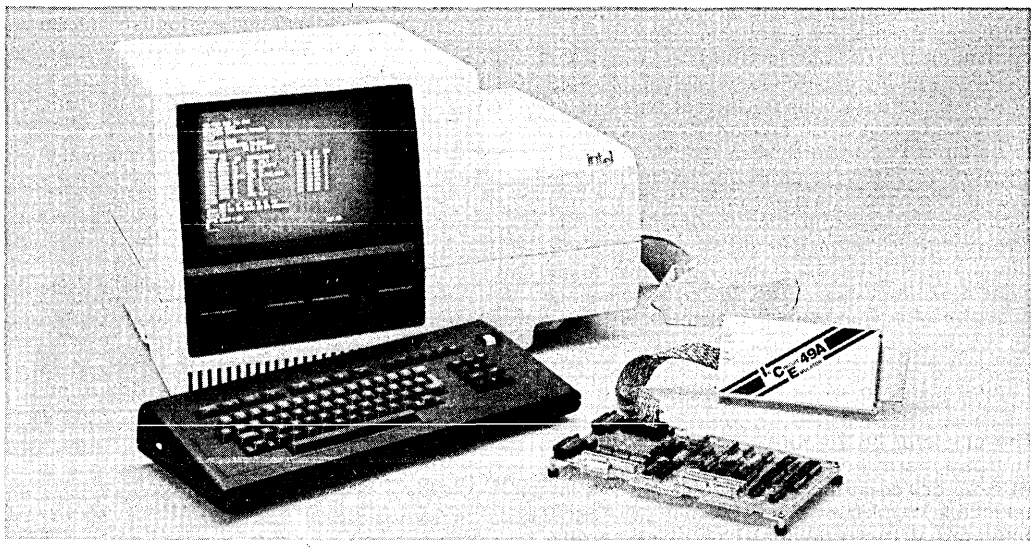


Figure 2. The ICE-49A module hosted by a Series IV development system and connected to a user prototype board.

SPECIFICATIONS

ICE™-49A Operating Environment

Required Hardware

Intellec Model 800 Series II, Series III or Series IV microcomputer development system (64KB RAM required)

System console (Model 800 only)

ICE-49A Module

Required Software

System monitor

ISIS (v 3.4 or later)

ICE-49A diskette based software

Equipment Supplied

Printed circuit boards (control board, emulator board)

Interface cable and buffer module

Diskette-based ICE-49A software:
—8 inch single and double density
—5¼ inch double density

8048/8748 with internal monitor program and 8050 with internal monitor program

CON 49A confidence test software, diskette-based (single and double density)

Diagnostic Loop bulk assembly (for use with CON 49A)

Emulation Clock

Crystal controlled 11 MHz internal, 5.5 MHz internal or user supplied TTL external (1.0 MHz to 8.0 MHz): software selectable.

Physical Characteristics

Width — 12.00 in. (30.48 cm)
Height — 6.75 in. (17.15 cm)
Depth — 0.50 in. (1.27 cm)
Weight — 8.00 lb. (3.64 kg)

Electrical Characteristics

DC Power Requirements

$V_{CC} = +5\% -2\%$
$I_{CC} = 10A$ max; 7.0A typ
$V_{DD} = +12V \pm 5\%$
$I_{DD} = 79$ mA max; 45 mA typ
$V_{BB} = -10V \pm 5\%$
$I_{BB} = 20$ mA max

Input Impedance @ ICE-49A user socket pins:

$V_{IL} = 0.8V$ (max), $I_{IL} = -1.6$ mA,
$V_{IH} = 2.0V$ (min), $I_{IH} = 40\ \mu A$
For Bus:
$V_{IL} = 0.8V$ (max), $I_{IL} = -250\ \mu A$
$V_{IH} = 2.0V$ (min), $I_{IH} = 20\ \mu A$

Output Impedance @ICE-49A user socket pins:

P1, P2:
$V_{OL} = 0.5V$ (max), $I_{OL} = 16$ mA
$V_{OH} = V_{CC}$ (10K pullup)
For Bus:
$V_{OL} = 0.5V$ (max), $I_{OL} = 25$ mA
$V_{OH} = 2.4V$ (min), $I_{OH} -10$ mA
Others:
$V_{OL} = 0.5V$ (max), $I_{OL} = 16$ mA
$V_{OH} = 2.4V$ (min), $I_{OH} -400\ \mu A$

Environmental Characteristics

Operating Temperature — 10°C to 40°C (Room Temperature)

Operating Humidity — 10% to 85% relative humidity without condensation

Reference Manuals

9800632 — *ICE™-49A Operating Instructions* (SUPPLIED)

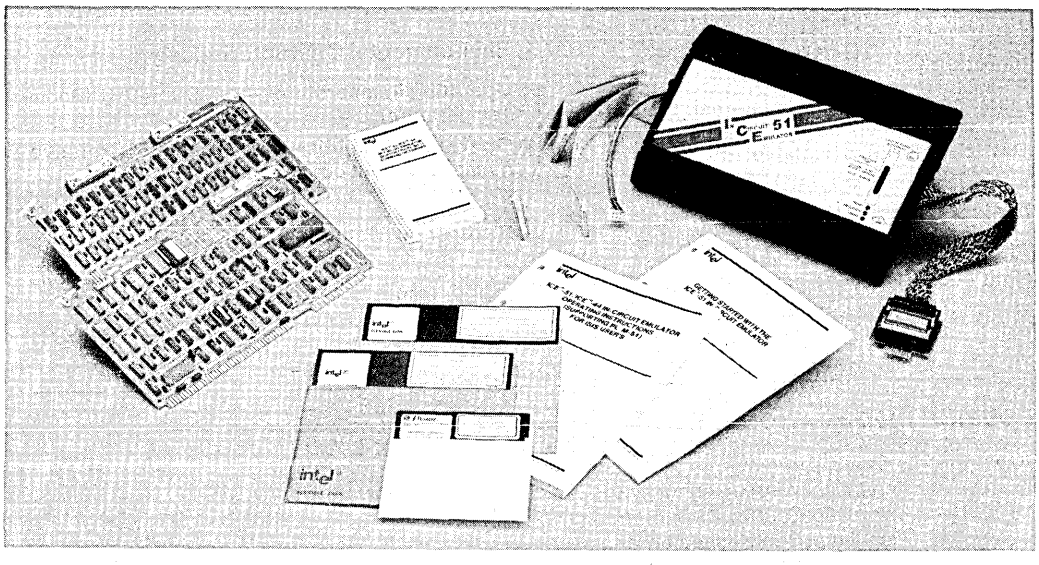
ORDERING INFORMATION

Part Number	Description
ICE-49A	8050, 8049, 8048, 8039, 8749, 8748, 8035, 8021 CPU in-circuit emulator. Cable assembly and interactive diskette software included.

ICE™-51 MODULE
8051 IN-CIRCUIT EMULATOR

- **Precise, full-speed, real-time emulation**
- **8K bytes full-speed RAM**
- **User-specified breakpoints**
- **Execution trace**
  - **-User-specified qualifier registers Conditional trigger**
  - **-Symbolic groupings and display**
  - **-Instruction and frame modes**
  - **-Trace by symbol or line number**
- **Supports 8K bytes ROM**
- **PL/M-51 support**
- **Full symbolic debugging**
- **Single-line assembly and disassembly for program instruction changes**
- **Macro commands and conditional block constructs for automated debugging sessions**
- **Emulation timer**
- **External load option**

The ICE™-51 module resides in the Intellec® Microcomputer Development System and interfaces to any user-designed 8051 system through a cable terminating in an 8051 emulator microprocessor and a pin-compatible plug. The emulator processor, together with 8K bytes of user RAM located in the ICE-51 buffer box, replaces the 8051 device in the user system while maintaining the 8051 electrical and timing characteristics. Powerful Intellec debugging functions are thus extended into the user system. Using the ICE-51 module, the designer can emulate the system's 8051 in real-time or single-step mode. Breakpoints allow the user to stop emulation on user-specified conditions, and a trace qualifier feature allows the conditional collection of 1000 frames of trace data. Using the single-line 8051 assembler, the user may alter program memory using ASM51 mnemonics and symbolic references, without leaving the emulator environment. Frequently used command sequences can be combined into compound commands and identified as macros with user-defined names. The ICE-51 system may also be used in the debugging and development of 8052 systems through its ability to debug all the 8052 features that are shared with the 8051 and the internal 8K ROM.



SEPTEMBER, 1984
ORDER NUMBER: 230783-002

FUNCTIONAL DESCRIPTION

The ICE-51 emulator aids the design effort in several ways: software and hardware integration and debugging, symbolic debugging, PL/M-51 support, and automated debugging and testing. The following sections describe these features.

Integrated Hardware and Software Development

The ICE-51 emulator allows hardware and software development to proceed interactively. This approach is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-51 module, prototype hardware can be added to the system as it is designed. Software and hardware integration occurs while the product is being developed.

The ICE-51 emulator assists in four stages of development, as described in the following sections.

SOFTWARE DEBUGGING

The ICE-51 can be operated without being connected to the user's system and before any of the user's hardware is available. In this stage, ICE-51 debugging capabilities can be used with the Intellec text editor and the 8051 macro assembler to facilitate program development.

HARDWARE DEVELOPMENT

The ICE-51 module's precise emulation characteristics and full-speed program RAM make it a valuable tool for debugging hardware, including the serial and parallel ports, and timer interfaces.

SYSTEM INTEGRATION

Software and hardware integration can begin when any functional element of the user system hardware is connected to the 8051 socket. As each section of the user's hardware is completed, it is added to the prototype. Thus, each section of the hardware and software is system tested in real-time operation as it becomes available.

SYSTEM TEST

When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-51 module can then be used for real-time emulation of the 8051 to debug the system as a completed unit.

The final product verification test may be performed using the 8751 EPROM version of the 8051 microcomputer. Thus, the ICE-51 module provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools. Figure 1 shows an 8051 development configuration.

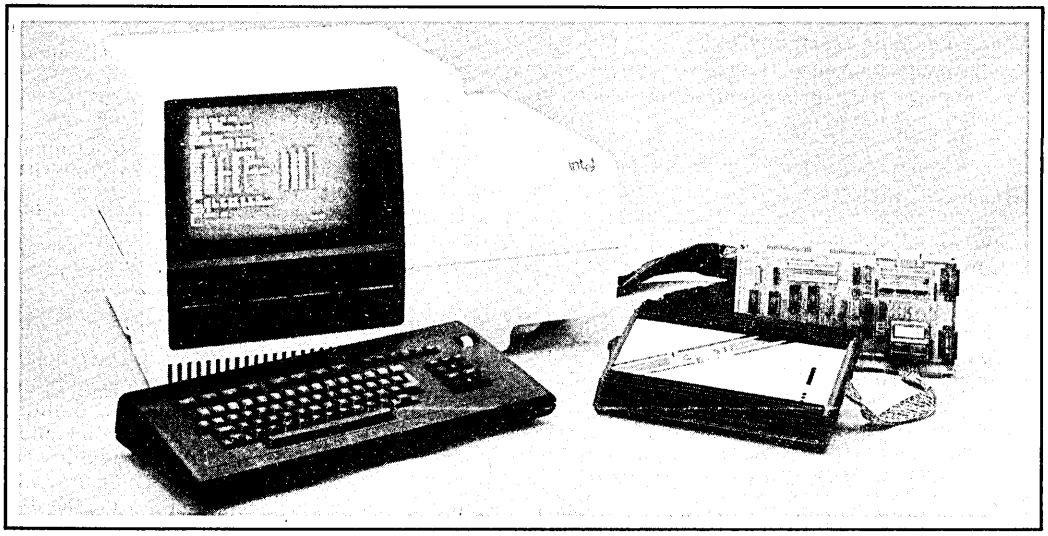


Figure 1. A Typical 8051 Development Configuration. The host system is an Intellec® Series IV. The ICE-51 module is connected to a user prototype system.

Order Number: 230783-002

Symbolic Debugging

The ICE-51 emulator permits the user to define and use symbolic (rather than absolute) references to program and data memory addresses; additional symbols are predefined by the ICE-51 software for referencing registers, flags, and input/output ports. Thus, the user need not recall or look up the addresses of key locations in a program as they change with each assembly, or become involved with machine code.

When a symbol is used for memory reference in an ICE-51 emulator command, the emulator supplies the corresponding location as stored in the ICE-51 emulator symbol table. This table is loaded with the symbol table produced by the assembler during application program assembly. The user can obtain the symbol table during software preparation simply by using the DEBUG switch in the ASM51 macro assembler. Furthermore, the user can interactively modify the emulator symbol table by adding new symbols, or changing or deleting old ones. This feature provides flexibility in debugging and minimizes the need to work with hexadecimal values.

Through symbolic references in combination with other features of the emulator, the user can easily do the following:

- Interpret the results of emulation activity collected during trace.
- Disassemble program memory to mnemonics, or assemble mnemonic instructions to executable code.
- Examine or modify 8051 internal registers, data memory, or port contents.
- Reference labels or addresses defined in a user program.

PL/M Support

The ICE-51 is capable of debugging high-level PL/M programs by module and line numbers. An external load option allows loading user code into user-supplied external memory and selective loading of symbols, lines, or external code. The select option permits the loading of symbols or lines for specific modules or ranges of modules produced by PL/M programs.

Automated Debugging and Testing

The following sections describe ways in which the user can automate some of the emulation and debug commands.

MACRO COMMANDS

A macro is a set of commands that is given a name. A group of commands that is executed frequently can be defined as a macro. The user can execute the group of commands by typing a colon followed by the macro name. Up to 10 parameters may be passed to a macro.

Macro commands can be defined at the beginning of a debug session and then used throughout a session. Macro definitions can be saved on disk for later use. The Intellec text editor may be used to edit the macro file.

The power of the development system can be applied to manufacturing testing as well as development, by writing test sequences as macros. The macros are stored on disk for use during system test.

COMPOUND COMMANDS

There are two kinds of compound commands. The IF command permits conditional execution of commands, and the COUNT and REPEAT commands allow repetitious execution of commands until certain conditions are met.

Compound commands may be nested any number of times, and they may be used in macro commands.

Example:

```
*DEFINE .I=0          ;Define symbol .I as 0
*COUNT 100H           ;Repeat the following
                      ;commands 100H times
.*IF .I AND 1 THEN    :Check if .I is odd
..*BYTE .I=.I         ;Fill the memory at location .I
                      ;to value .I
..*END
.*.I=.I+1             ;Increment by 1
.*END                 ;Command executes upon
                      ;carriage return after END
```

(The characters *, .*, and ..* shown in this example are system prompts that indicate the nesting level of compound commands.)

Operating Commands

The ICE-51 software is an Intellec RAM-based program that provides the user with easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-51 commands are configured with a broad range of modifiers that provide the user with maximum flexibility in describing the operation to be performed.

Order Number: 230783-002

EMULATION

The ICE-51 module can emulate the operation of a prototype 8051 system, at real-time speed (1.2 to 12 MHz) or in single-steps. Emulation commands to the ICE-51 module control the process of setting up, running, and halting an emulation of the user's 8051-based system. Breakpoints and tracepoints enable the ICE-51 emulator to halt emulation and provide a detailed trace of execution in any part of the user's program. In addition, the user can disable the printing of display headers, and display or set the contents of one or more two-byte memory locations. A summary of the emulation commands is shown in Table 1.

Breakpoints

The ICE-51 hardware includes two breakpoint registers that allow the user to halt emulation when specified conditions are met. The emulator continuously compares the values stored in the breakpoint registers with the status of specified address, opcode, operand, or port values, and halts emulation when this comparison is satisfied. When an instruction initiates a break, that instruction is executed completely before the break takes place. The ICE-51 emulator then regains control of the console and enters the interrogation mode. With the breakpoint feature, the user can specify an emulation break when the program:

- Executes an instruction at a specified address or within a range of addresses.
- Executes a particular opcode.
- Receives a specific signal on a port pin.
- Fetches a particular operand from the user program memory.
- Fetches an operand from a specific address in program memory.

Trace and Tracepoints

Tracing is used with both real-time and single-step emulation to record diagnostic information in the trace buffer as a program is executed. The information collected includes opcodes executed, port values, and memory addresses. The ICE-51 emulator collects up to 1000 frames of trace data.

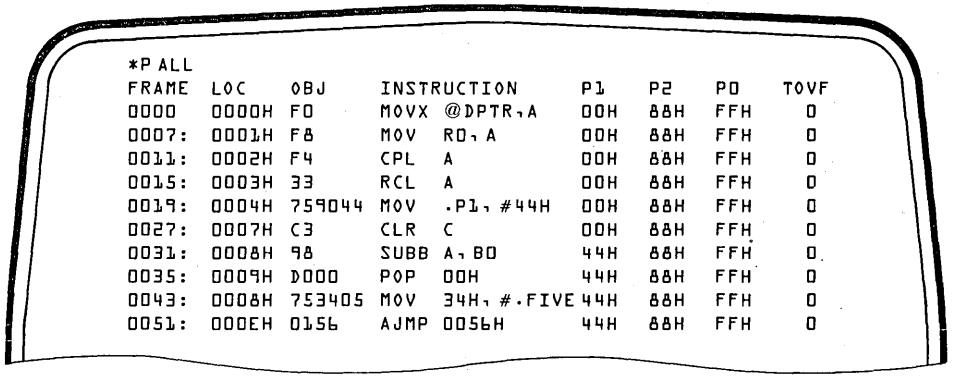
The trace data can be displayed as assembler instruction mnemonics, if desired, for analysis during interrogation or single-step mode. The trace-collection facility may be set to run conditionally or unconditionally. Two unique trace qualifier registers, specified in the same way as breakpoint registers, govern conditional trace activity. The qualifiers can be used to condition trace data collection to take place as follows:

- Under all conditions (forever).
- Only while the trace qualifier is satisfied.
- For the frames or instructions preceding the time when a trace qualifier is first satisfied (pre-triggered trace).
- For the frames or instructions after a trace qualifier is first satisfied (post-triggered trace).

Figure 2 shows an example of a trace display in instruction mode.

Table 1. Major Emulation Commands

Command	Description
GO	Begins real-time emulation and optionally specifies break conditions.
BR0, BR1, BR	Sets or displays either or both breakpoint registers used for stopping real-time emulation.
STEP	Performs single-step emulation.
QR0, QR1	Specifies match conditions for qualified trace.
TR	Specifies or displays trace-data collection conditions and optionally sets the qualifier register (QR0, QR1).
Synchronization Line Commands	Sets and displays the status of synchronization line output or latched input. Used to synchronize the starting and stoping of real-time emulation or trace to occur with external events.



```
*P ALL
FRAME  LOC    OBJ     INSTRUCTION      P1   P2   P0   TOVF
0000   0000H  F0      MOVX @DPTR,A     00H  88H  FFH   0
0007:  0001H  F8      MOV  R0, A       00H  88H  FFH   0
0011:  0002H  F4      CPL  A           00H  88H  FFH   0
0015:  0003H  33      RCL  A           00H  88H  FFH   0
0019:  0004H  759044  MOV  .P1, #44H   00H  88H  FFH   0
0027:  0007H  C3      CLR  C           00H  88H  FFH   0
0031:  0008H  98      SUBB A, B0       44H  88H  FFH   0
0035:  0009H  D000    POP  00H         44H  88H  FFH   0
0043:  000BH  753405  MOV  34H, #.FIVE 44H  88H  FFH   0
0051:  000EH  0156    AJMP 0056H       44H  88H  FFH   0
```

Figure 2. Sample Trace Display in Instruction Mode

INTERROGATION AND UTILITY COMMANDS

Interrogation and utility commands allow convenient access to detailed information about the user program and the state of the 8051 that is useful in debugging hardware and software. Changes can be made in memory and in the 8051 registers, flags, and port values. Commands are also provided for various utility operations such as loading and saving program files, defining symbols, displaying trace data, controlling system synchronization, and returning control to ISIS. A summary of the basic interrogation and utility commands is shown in Table 2.

Single-line Assembler/Disassembler

The single-line assembler/disassembler (ASM and DASM commands) is a time-saving emulator feature that permits the designer to examine and alter program memory using assembly language mnemonics, without leaving the emulator environment or requiring time-consuming program reassembly. When assembling new mnemonic instructions into program memory, previously defined symbolic references (from the original program assembly, or subsequently defined during the emulation session) may be used in the instruction operand field. The emulator will

Table 2. Major Interrogation and Utility Commands

Command	Description
HELP	Displays help messages for ICE-51 emulator command-entry assistance.
LOAD	Loads the user object program (8051 code) into user program memory, and the user symbols into the ICE-51 emulator symbol table.
SAVE	Saves the ICE-51 emulator symbol table and the user object program in an ISIS hexadecimal file.
LIST	Copies all emulator console input and output to an ISIS file.
GO	Begins ICE-51 emulation .
EXIT	Terminates ICE-51 emulation operation and returns control to the ISIS operating system.
DEFINE	Defines the ICE-51 emulator symbol or macro.
REMOVE	Deletes user-defined symbols, modules, or macro names from the symbol table or macro table.
ASM	Assembles mnemonic instructions into user program memory.
DASM	Disassembles and displays user program memory contents.
Change/Display Commands	Changes or displays the value of a symbolic reference in the ICE-51 emulator symbol table, the contents of keyword references (including registers, I/O ports, and status flags), or the memory references.

Table 2. Major Interrogation and Utility Commands Continued

Command	Description
EVALUATE	Evaluates an expression and displays the resulting value.
MACRO	Displays an ICE-51 macro or macros.
INTERRUPT	Displays serial, external, or timer-interrupt register settings.
SECONDS	Displays the contents of the emulation timer in microseconds.
Trace Commands	Positions the trace buffer pointer and selects the format for the trace display.
PRINT	Displays the trace data pointed to by the trace buffer pointer.
Word Commands	Displays or sets the contents of one or more two-byte memory locations.
LINES	Displays all the module names, their associated line numbers, and line number address.

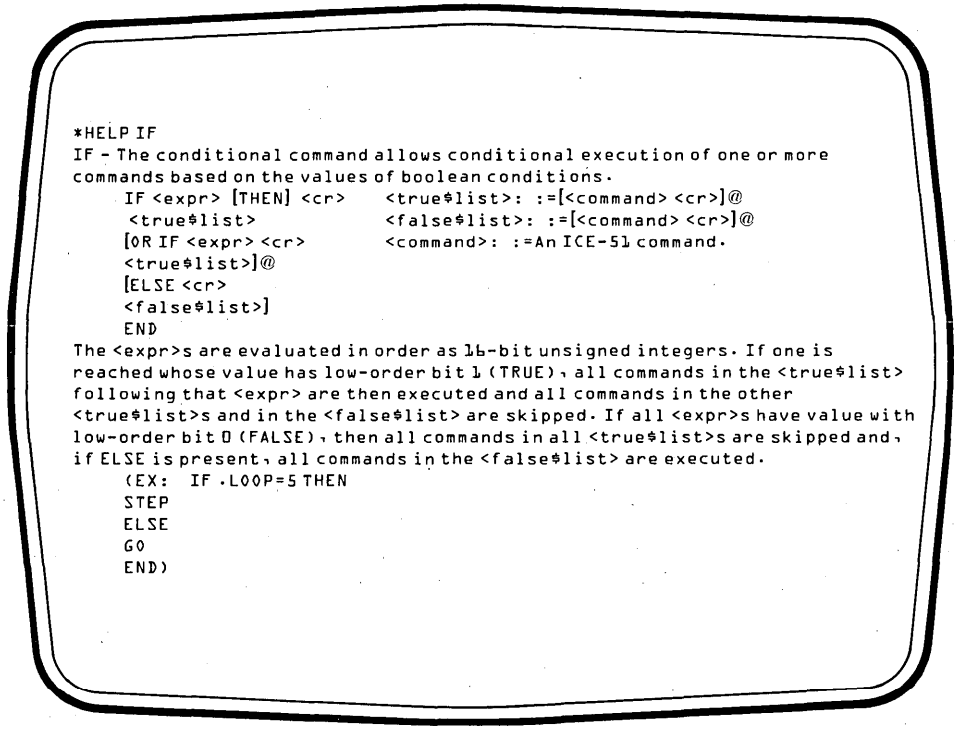
supply the absolute address or data values as stored in the emulator symbol table. These features eliminate user time spent translating to and from machine code and searching for absolute addresses, with a corresponding reduction in transcription errors.

Help

The HELP file allows the user to display ICE-51 command syntax information at the Intellec console. Typing HELP displays a listing of all items for which help messages are available; typing HELP <item> displays relevant information about the item requested, including typical usage examples. See Figures 3 and 4 for screen displays of a HELP menu and a HELP <item> menu.

```
*HELP
Help is available for the following items. Type HELP followed by the item name.
The help items cannot be abbreviated. (for more information, type HELP HELP or
HELP INFO.)
Emulation:      Trace Collection:          Misc:    <address>
GO GR SYO       TR QR QRO QR1 SY1          BASE     <CPU$keyword>
BR BRO BR1                                 DISABLE  <expr>
STEP            Trace Display:             ENABLE   <ICE51$keyword>
                TRACE MOVE PRINT           ERROR    <identifier>
                OLDEST NEWEST              EVALUATE <instruction>
                                           HELP     <masked$constant>
Change/Display/Define/Remove:              INFO     <match$cond>
<CHANGE>        REMOVE   CBYTE   RBIT      <LIGHTS> <numeric$ref>
<DISPLAY>       SYMBOL   DBYTE   DASM      LIST     <partition>
REGISTER        RESET    PBYTE   ASM       LOAD     <string>
SECONDS         WRITE    RBYTE   MAP       SAVE     <string$constant>
DEFINE          STACK    XBYTE   SY        SUFFIX   <symbolic$ref>
                                           SYMBOLIC <system$symbolic>
Macro:                     Compound                 <trace$reference>
DEFINE              DIR    Commands:                <unlimited$match$cond>
DISABLE             ENABLE COUNT                    <user$symbols>
INCLUDE             PUT    IF
<MACRO$DISPLAY>            REPEAT
<MACRO$INVOCATION>
```

Figure 3. Menu Display for HELP



```
*HELP IF
IF - The conditional command allows conditional execution of one or more
commands based on the values of boolean conditions.
     IF <expr> [THEN] <cr>     <true$list>: :=[<command> <cr>]@
      <true$list>              <false$list>: :=[<command> <cr>]@
     [OR IF <expr> <cr>        <command>: :=An ICE-51 command.
     <true$list>]@
     [ELSE <cr>
     <false$list>]
     END
The <expr>s are evaluated in order as 16-bit unsigned integers. If one is
reached whose value has low-order bit 1 (TRUE), all commands in the <true$list>
following that <expr> are then executed and all commands in the other
<true$list>s and in the <false$list> are skipped. If all <expr>s have value with
low-order bit 0 (FALSE), then all commands in all <true$list>s are skipped and,
if ELSE is present, all commands in the <false$list> are executed.
     (EX:  IF .LOOP=5 THEN
     STEP
     ELSE
     GO
     END)
```

Figure 4. Menu Display for HELP IF

Emulation Accuracy

The speed and interface demands of a high-performance single-chip microcomputer require extremely accurate emulation, including full-speed, real-time operation with the full function of the microcomputer. The ICE-51 emulator achieves accurate emulation with an 8051 bond-out chip, a special configuration of the 8051 microcomputer, as its emulation processor.

Each of the 40 pins on the user plug is connected directly to the corresponding 8051 pin on the bond-out chip. Thus, the user system sees the emulator as an 8051 microcomputer at the 8051 socket. The resulting characteristics provide extremely accurate emulation of the 8051, including speed, timing characteristics, load and drive values, and crystal operation. The emulator may draw more power from the user system than a standard 8051 family device (see Electrical Characteristics).

Additional bond-out pins provide the emulator box with signals such as internal address, data, clock, and control lines. These signals let static RAM in the buffer box substitute for on-chip program ROM, EPROM, or user supplied external program memory. The 8K bytes of full-speed RAM in the buffer box can be mapped in 4K blocks to anywhere within the 64K program memory space of the 8051. The bond-out chip also gives the emulator "backdoor" access to internal chip operation, so that the emulator can break and trace execution without interfering with the values on the user-system pins.

8051 AND 80C51 DEBUGGING

The minor differences between the NMOS 8051 and the CMOS Version, the 80C51, do not prohibit the emulation of the 80C51 with the current version of the ICE-51. The specifications of the 8051 and the 80C51 are very similar and can be found in the 8051 and the 80C51 data sheets. The guideleines and limitations for emulation of the 8051 and 80C51 are presented below.

- Maintain V_{CC} at 5V ± 10%. This ensures that V_{IL} and V_{IH} remain within the 8051 specifications and that the voltage limitations are not violated.

- The user's power supply must be able to supply at least 160 mA for the ICE-51 emulation processor, which is not CMOS.
- The 80C51 CPU idle mode is not supported by the ICE-51.
- The power-down mechanism of the 80C51 is not supported by the ICE-51 emulation processor. The power-down bit location has no effect on the chip, and the power-down voltage (V_{DP}) source is not supported.

The ICE-51 is able to provide CMOS support except for the operating limitations outlined above.

SPECIFICATIONS

ICE™-51 Operating Requirements

Intellec Model 800, Series II/III, or Series IV development system (64K RAM required)

System Console

One disk drive, single-density or double-density

Intellec disk operating system (single or double density) ISIS v. 3.4 or later

Equipment Supplied

- Printed circuit boards(2)
- Emulation buffer-box, Intellec interface cables, and user-interface cable with an 8051 emulation processor
- Dual auxiliary connector kit for the Model 800, Series II/III, and Series IV development systems
- Crystal power accessory
- Literature kit
  -ICE-51 operating instructions manual
  -ICE-51 command dictionary
  -ICE-51 user's guide
- Disk-based ICE-51 software (5 1/4 inch and 8 inch, single and double density)

Emulation Clock

User's system clock (1.2 to 12MHz) or ICE-51 crystal power assembly (12MHz)

Environmental Characteristics

Operating Temperature: 0° to 40° C

Operating Humidity: Up to 95% relative humidity without condensation

Physical Characteristics

Printed Circuit Boards

Width: 12.00 in. (30.48cm)
Height: 6.75 in. (17.15 cm)
Depth: 0.50 in. (1.27 cm)

Buffer Box

Width: 8.00 in. (20.32 cm)
Length: 12.00 in. (30.48 cm)
Depth: 1.75 in. (4.44cm)
Weight: 4.0 lb (1.81 kg)

Interface Cables

Host-emulator interface cable length: 48 in. (121.92 cm)
Emulator-user-system interface cable length: 12.00 in. (30.48 cm)

Electrical Characteristics

DC Power Requirements (from the Intellec system):

$V_{CC} = +5V, +5\%, -2.5\%$
$I_{CC} = 13.2A$ max; 11.0A typical
$V_{DD} = +12V, \pm 5\%$
$I_{DD} = 0.1$ A max; 0.05A typical
$V_{BB} = -10V, \pm 5\%$
$I_{BB} = 0.05A$ max; 0.01A typical

User Plug Characteristics at the 8051 Socket:

Same as an 8031, 8051, or 8751, except that the user system will see an added load of 25 pf capacitance and 50 uA leakage from the ICE-51 emulator user plug at ports 0, 1, and 2.

ORDERING INFORMATION

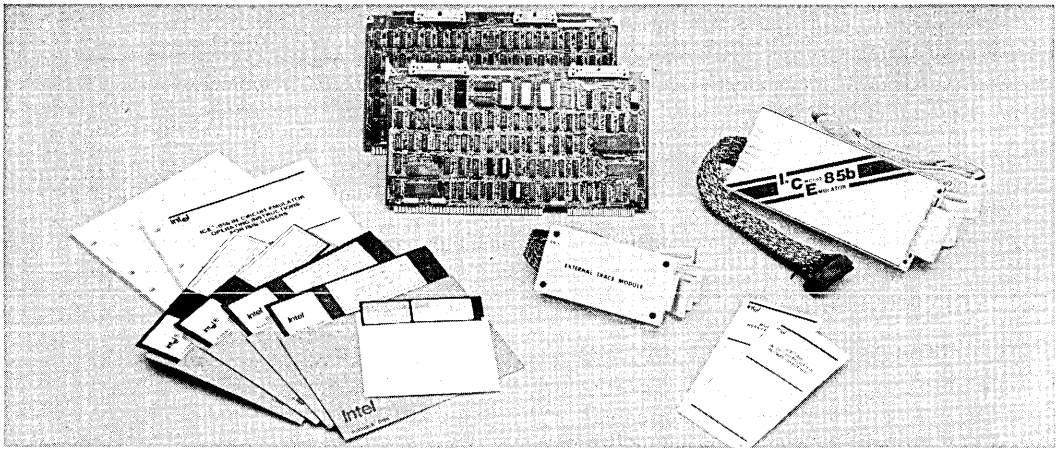
Part Number	Description
ICE-51	8051 microcontroller in-circuit emulator, cable assembly, and interactive disk software

ICE-85B™
MCS-85™ IN-CIRCUIT EMULATOR
WITH MULTI-ICE™ SOFTWARE

- **Connects the Intellec® system resources to the user-configured system via a 40-pin adaptor plug**
- **Executes user system software in real-time (5 MHz clock)**
- **Allows user-configured system to share Intellec® memory and I/O facilities**
- **Provides 1023 states of 8085 trace data**
- **Displays trace data from the user's 8085 in assembler mnemonics and allows personality groupings of data sampled by the external 18-channel trace module**
- **Offers full symbolic debugging capability for both assembly language and Intel's high-level compiler languages PL/M-80 and FORTRAN-80**
- **The Multi-ICE™ software lets two ICE-85 In-Circuit Emulators operate simultaneously in a single Intellec Microcomputer Development System.**
- **Includes enhanced software features: symbolic display of addresses, macro commands, compound commands, software synchronization of processes, and INCLUDE file capability.**

The ICE-85B™ module resides in the Intellec® Microcomputer Development System and interfaces to the user system's 8085. It provides the ability to examine and alter MCS-85™ registers, memory, flag values, interrupt bits and I/O ports. Using the ICE-85B module, the designer can execute prototype software in real-time or single-step mode and can substitute Intellec® system memory and I/O for user system equivalent. ICE capability can be extended to the rest of the user system peripheral circuitry by allowing the user to create and execute a library of user-defined peripheral chip analyzer routines.

Multi-ICE software allows two ICE-85 In-Circuit Emulators to run simultaneously in a single Intellec Microcomputer Development System. Multi-ICE software used in lieu of the standard ICE software gives users full control of the two ICE-85 modules for debugging of multi-processor systems.



MAY 1983

AFN-01557C

SYMBOLIC DEBUGGING CAPABILITY

ICE-85B allows the user to make symbolic references to I/O ports, memory addresses and data in his program. Symbols and PL/M-80 statement number may be substituted for numeric values in any of the ICE-85 commands. The user is relieved from looking up addresses of variables or program subroutines.

The user symbol table generated along with the object file during a PL/M-80 or FORTRAN-80 compilation or by the ISIS-II 8080/8085 Macro Assembler is loaded into the Intellec® System memory along with the user program which is to be emulated. The user may add to this symbol table any additional symbolic values for memory addresses, constants, or variables that are found useful during system debugging. By referring to symbolic memory addresses, the user can examine, change or break at the intended location.

ICE-85B provides symbolic definition of all 8085 registers, interrupt bits and flags. The following symbolic references are also provided for user convenience: TIMER, the low-order 16 bits of a register containing the number of 2 MHz clock pulses elapsed during emulation; HTIMER, the high-order 16 bits of the timer counter; PPC, the address of the last instruction emulated; BUFFERSIZE, the number of frames of valid trace data (between 0 and 1022).

PERSONALITY GROUPED DISPLAYS

Trace data in the 1023 by 42-channel real-time trace memory buffer is displayed in easy to read format. The user has the option to specify trace data displays in actual 8085 assembler instruction mnemonics. The data collected from the External Trace Module can be grouped and symbolically named according to user specifications and displayed in the appropriate number base designation. Simple ICE-85B commands allow the user to select any portion of the 42-bit trace buffer for immediate display.

MEMORY AND I/O MAPPING

Memory and I/O for the user system can be resident in the user system or "borrowed" from the Intellec® System through ICE-85B's mapping capability.

ICE-85B separates user memory into 32 2K blocks. Each block of memory can be defined independently. The user may assign Intellec® System equivalents to take the place of devices not yet designed for the user system during prototyping. In addition, Intellec® System memory or I/O can be accessed in place of suspect user system devices during prototyping or production checkout.

User ready synchronization—resource borrowing from the Intellec System is (at user option) independent of the user system; the user does not need to provide ready acknowledge when accessing resources mapped to the Intellec.

The user can also designate a block of memory or I/O as nonexistent. ICE-85B issues error messages when memory or I/O designated as nonexistent is accessed by the user program.

INTEGRATED HARDWARE/SOFTWARE DEVELOPMENT

The user prototype need consist of no more than an 8085 CPU socket and a user bus to begin integration of software and hardware development efforts. Through ICE-85B mapping capabilities, Intellec® System equivalents can be accessed for missing prototype hardware. Hardware designs can be tested using the system software which will drive the final product. Figure 1 shows the ICE-85B system hosted on a Series IV development system and connected to a user prototype system.

The system integration phase, which can be so costly when attempting to mesh completed hardware and software products, becomes a convenient two-way debug tool when begun early in the design cycle.

INTERROGATION AND UTILITY COMMANDS

DISPLAY/ CHANGE	Display/Changes the values of symbols and the contents of 8085 registers, pseudo-registers, status flags, interrupt bits, I/O ports and memory.
EVALUATE	Displays the value of an expression in the binary, octal, decimal or hexadecimal.
SEARCH	Searches user memory between locations in a user program for specified contents.
CALL	Emulates a procedure starting at a specified memory address in user memory.
ICALL	Executes a user-supplied procedure starting at a specified memory address in the Intellec® System memory.
EXECUTE	Saves emulated program registers and emulates a user-supplied subroutine to access peripheral chips in the user's system.

REAL TIME TRACE

ICE-85B captures valuable trace information from the emulating CPU and the External Trace Module while the user is executing programs in real time. The 8085 status, the user memory or port addressed, the data read or written, the serial data lines and data from 18 external signals, is stored for the last 1023 machine states executed (511 machine cycles). This provides ample data for determining how the user system was reacting prior to emulation break. It is available whether the break was user-initiated or the result of an error condition.

For detailed information on the actions of CPU registers, flags, or other system operations, the user may operate in single or multi-step sequences tailored to system debug needs.

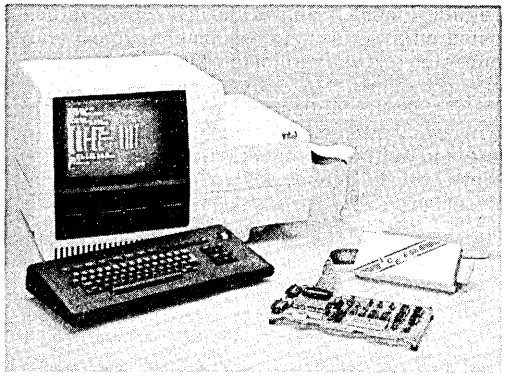


Figure 1. The ICE-85B system hosted by a Series IV development system and connected to a user prototype system.

EMULATION CONTROLS AND COMMANDS

GROUP — Defines into a symbolically named group, a channel or combination of channels from the 8085 Microprocessor and/or the External Trace Module.

GO — Initiates real-time emulation and controls emulation break conditions.

STEP — Initiates emulation in single instruction steps. User may specify the type and amount of information displayed following each step, and define conditions under which stepping should continue.

PRINT — Prints the user-specified portion of the trace memory to the selected list device.

EXTERNAL TRACE MODULE

TTL level signals from 18 points in the user system may be synchronously sampled by the External Trace Module and collected in ICE-85B's trace buffer. The signals can be collected from a single peripheral chip via the supplied 40-pin DIP clip or may be placed by the user on up to 18 separate signal nodes using the supplied 18 individual probe clips. These signals are included in the 42-channel breakpoint comparisons and clock qualifiers. Also, data from these 18 channels may be displayed in meaningful, user-defined groupings.

SYNCHRONOUS OPERATION WITH OTHER DESIGN AIDS

ICE-85B can be synchronized with other Intellec® design aids by means of two external synchronization lines. These lines are used to enable and disable ICE-85B trace data collection and to cause break conditions based on an external signal which may not be included in the ICE-85B breakpoint registers. In addition, ICE-85B can generate signals on these lines which may be used to control other design aids.

BREAK REGISTERS/ TRACE MEMORY

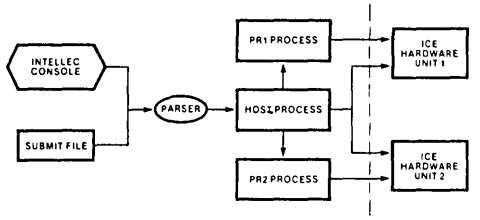
ICE-85B has two breakpoint registers which are used to break emulation, and two trace qualifier registers which are used to control the collection of trace data during emulation. Each register is 42 entries wide, one entry for each channel and each entry can take any one of the three values 0, 1 or "don't care."

The trace buffer, also 42 entries wide, collects data sampled from 24 8085 processor channels and 18 external channels sampled by the External Trace Module. The signals collected from the 8085 include address lines, data lines, status lines and serial input and output lines. The 18 channels extending from the External Trace Module synchronously sample and collect into the trace buffer any user-specified TTL compatible signal from the rest of the prototype system. "Break" and "trace qualification" may therefore occur as a result of a match of any combination of up to 42 channels of CPU and external circuitry signals.

MULTI-ICE™ OPERATION

Multi-ICE software is a debug tool which allows two ICE-85 emulators to begin and stop in sequence. Once started, two ICE emulators emulate simultaneously and independently. Thus, Multi-ICE software permits the debugging of asynchronous or synchronous multi-processor systems.

A conceptual model for the Multi-ICE software can be illustrated with the following block diagram.



Block Diagram of Multi-ICE™ Operation

There are three processes in the Multi-ICE environment: the Host process and the two ICE processes to control the two ICE hardware modules. The processor for these three processes is the microcomputer in the Intellec Microcomputer Development System. Only the Host process is active when Multi-ICE software is invoked. The Parser interfaces with the console, receives commands from the console or from a file, translates them into intermediate code, and loads the code into the Host command code buffer or ICE command code buffers.

The Host process executes commands from its command code buffer using the execution software and hardware of the Host's current environment, either environment 1 or environment 2 (EN1 or EN2), as required. EN1 and EN2 are the operating environments of the two In-Circuit Emulators.

The user can change the execution environment (from EN1 to EN2 or vice versa) with the SWITCH command. Once the environment is selected, ICE operation is the same as with standard ICE software. In addition, the enhanced software capabilities are available to the user.

The two ICE processes (PR1 and PR2) execute commands from their command code buffers in their own environments (PR1 in EN1 and PR2 in EN2). The main functions of the two ICE execution processes are to control the operations of the two ICE hardware sets. The ACTIVATE command controls the execution of the ICE processes. Commands are passed on to each ICE unit to initiate the desired ICE functions.

The two ICE hardware units accept commands from the Host process or ICE processes. Once emulations start, the two ICE hardware sets will operate until a break condition is met or processing is interrupted by commands from the ICE execution processes.

Symbolic Display of Addresses

The user has the option of displaying a 16-bit address in the form of a symbol name or line number plus a hex number offset.

Macro Command

A macro is a set of commands which is given a name. Thus, a group of commands which is executed frequently may be defined as a macro. Each time the user wants to execute that group of commands, he may just invoke the macro by typing a colon followed by the macro name. Up to ten parameters may be passed to the macro.

Macro commands may be defined at the beginning of a debug session and then can be used throughout the whole session. If the user wants to save the macros for later use, he may use the PUT command to save the macro on diskette, or the user may edit the macro file off-line using the Intellec text editor. Later, the user may use the INCLUDE command to bring in the macro definition file that he created.

Example:

```
*DEFINE MACRO INITMEM     ;This macro clears the memory and then loads the programs.
.*SWITCH = EN1            ;Select environment 1 (ICE Module 1)
.*BYTE 0 TO 100=0         ;Initialize memory to 0.
.*LOAD:F1:DRIVER          ;Load user program into memory for ICE Module 1.
.*SWITCH = EN2            ;Select environment 2 (ICE Module 2)
.*LOAD:F1:DR2             ;Load user program into memory for ICE Module 2
.*EM                      ;End of Macro
*                         ;To execute this Macro, user types :INITMEM
```

Compound Command

Compound commands provide conditional execution of commands (IF Command) and execution of commands repeatedly until certain conditions are met (COUNT, REPEAT Commands).

Compound commands and Macro commands may be nested any number of times.

Example:

```
*DEFINE .I = 0             ;Define symbol .I to 0
*COUNT 100H                ;Repeat the following commands 100H times
.*IF .I AND 1 THEN         ;Check if .I is odd
..*BYT .I = .I             ;Fill the memory at location .I to value .I
..*END
.*.I = .I + 1              ;Increment .I by 1
.*END                      ;Command executes upon carriage-return after END
```

INCLUDE File Capability

The INCLUDE command causes input to be taken from the file specified until the end of the file is encountered, at which point, input continues to be

taken from the previous source. Nesting of INCLUDES is permitted. Since the command code file can be complex, the ability to edit offline becomes desirable. The INCLUDE command allows the user to pull in command code files and Macro commands created offline which can then be used for the particular debugging session.

Example:

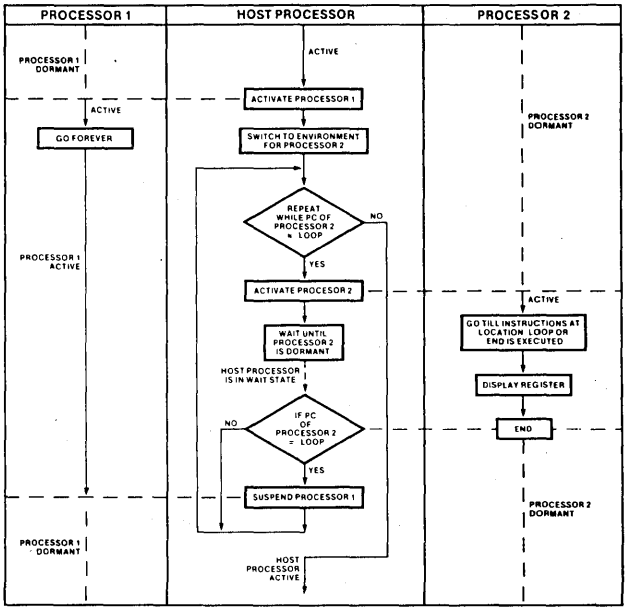
```
*INCLUDE :F1:PROG1      ;Cause input to be taken
                         from file PROG1
*MAP 0 LENGTH 64K       ;Contents of the file
=USER                    PROG1 are listed on
                         screen as they are
                         executed.
*MAP IO 0 TO FF
=USER
*SWITCH = EN2
*LOAD :F2:LED.HEX
*SWITCH = EN1           ;End of the file PROG1
*                        ;After the end of file is
                         reached, control is
                         returned to console.
```

Software Synchronization of Processes

Up to three processes (Host, PR1 and PR2) can be active simultaneously in the system. An ICE process can be activated (ACTIVATE), suspended (SUSPEND), killed (KILL), or continued (CONTINUE). The Host process can wait for other processes to become dormant before it becomes active again. Through these synchronization commands, the user can create a system test file off-line yet be able to synchronize the three processes when the actual system test is executed.

Example:

The capability of the software synchronization commands is demonstrated by the following example. The flowchart shows the synchronization requirements. The program steps show the actual implementation.




```
*ACTIVATE PR1           ;Activate PR1
.*GO FROM 800           ;Start ICE Module 1
.*END                   ;End of Activate block
PR1 EMULATION BEGUN
*SWI=EN2                ;Switch execution Environment to EN2
*REPEAT                 ;Repeat the following block of
                         commands while PC is not equal to .Loop
.*WHILE PC < > .LOOP
.*ACT PR2               ;Activate PR2
..*GO TILL .LOOP OR .END ;Go till instruction at location .Loop
                         or at location .END is executed
..*REGISTER             ;Display the registers
..*END                  ;End of Activate block
.*WAIT PR2              ;Wait until PR2 is dormant
.*IF PC=.LOOP THEN
..*SUSPEND PR1
..*END                  ;End of IF block
*END                    ;End of REPEAT block
*
```

Flowchart of the Example for Demonstrating Multi-ICE™ Synchronization Capability

SPECIFICATIONS

ICE-85B™ Operating Environment

Required Hardware:
- Intellec® Model 800 Series II, Series III, or Series IV Microcomputer Development System
  - (64K bytes RAM for Multi-ICE software)
  - (32K btyes RAM single ICE software)
  - System Console (Model 800 only)
  - ICE-85B Module

Required Software:
- System Monitor
- ISIS v 3.4 or later
- ICE-85B or Multi-ICE Software

Equipment Supplied

18-Channel External Trace Module
Printed Circuit Boards (2)
Interface Cable and Emulation Buffer Module
Operator's Manuals
ICE-85B Software
Multi-ICE Software
- Contains software that supports 85/85 Emulators, 85/49 Emulators and 85/41A Emulators

Emulation Clock

User's system clock or ICE-85B adaptor socket (10.0 MHz Crystal)

Physical Characteristics

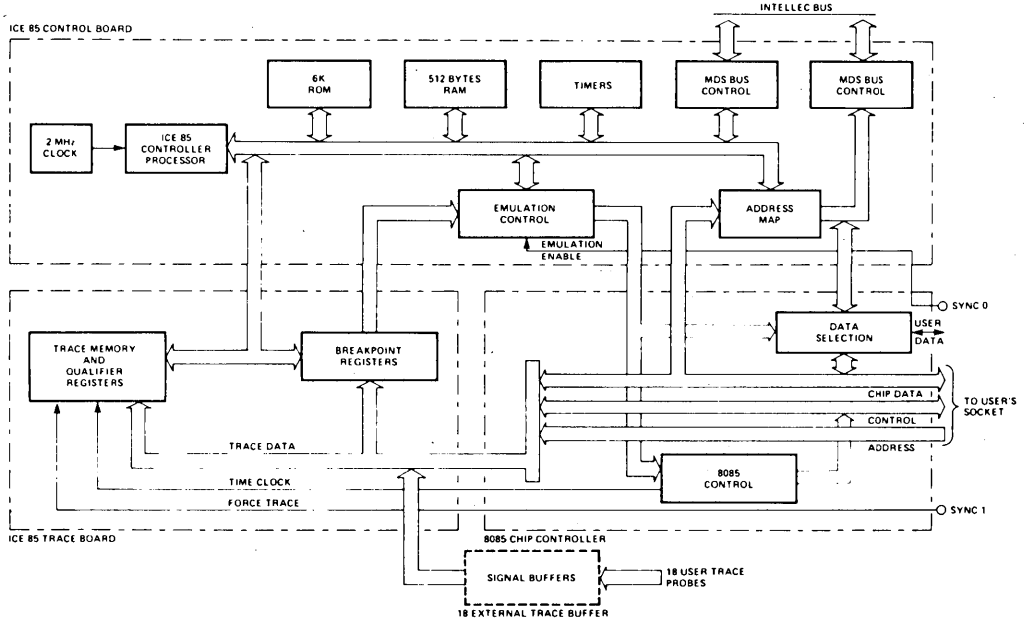
Printed Circuit Boards:
- Width: 12.00 in. (30.48 cm)
- Height: 6.75 in. (1715 cm)
- Depth: 0.50 in. (1.27 cm)
- Packaged Weight: 6.00 lb (2.73 kg)

Electrical Characteristics

DC Power:
- V_{CC} = + 5V ± 5%
- I_{CC} = 12A maximum; 10A typical
- V_{DD} = + 12V ± 5%
- I_{DD} = 80 mA maximum; 60 mA typical
- V_{BB} = − 10V ± 5%
- I_{BB} = 1 mA maximum; 10 μA typical

Environmental Characteristics

Operating Temperature: 0° to 40°C
Operating Humidity: Up to 95% relative humidity without condensation.



ICE-85B™ BLOCK DIAGRAM

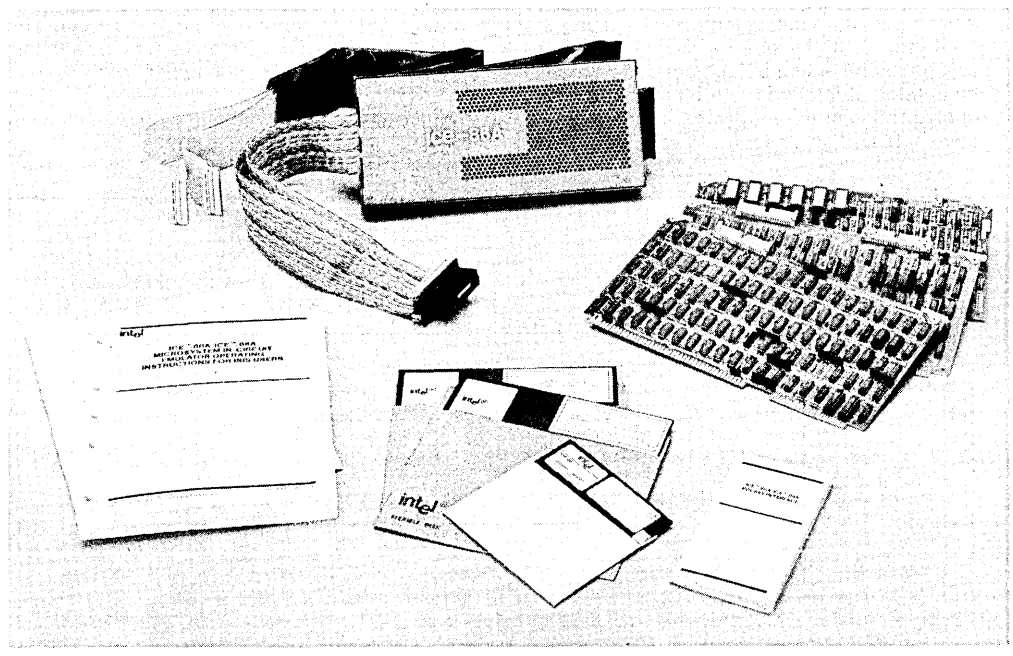
Ordering Information

Part Number	Description
ICE-85B	8085 CPU In-Circuit Emulator, 18-Channel External Trace Module and Multi-ICE software

ICE™-86A
iAPX 86 IN-CIRCUIT EMULATOR

- **Real-time in-circuit emulation of iAPX 86 microsystems**
- **Emulate both minimum and maximum modes of the 8086 CPU**
- **Full symbolic debugging**
- **Breakpoints to halt emulation on a wide variety of conditions**
- **Comprehensive trace of program execution**
- **Disassembly of trace or program memory from object code into assembler mnemonics**
- **Software debugging with or without a user system**
- **Handles full 1 megabyte of iAPX 86 address space**

The Intel ICE™-86A in-circuit emulator provides sophisticated hardware and software debugging capabilities for iAPX 86 microsystems and iAPX 86 single-board computers. These capabilities include in-circuit emulation for the 8086 central processing unit plus extensions to debug systems including the 8087 numeric processor extension. The emulator includes three circuit boards which reside in any Intellec® microcomputer development system (see Figure 1). A cable and buffer box connect the Intellec system to the user system by replacing the user's 8086, thus extending powerful Intellec system debugging functions into the user system (see Figure 2). Using the ICE-86A module, the designer can execute prototype 8086 software in continuous or single-step modes and can substitute blocks of Intellec system memory for user equivalents. Breakpoints allow the user to stop emulation on user-specified conditions of the iAPX 86 system, and the trace capability gives a detailed history of the program execution prior to the break. All user access to the prototype system software may be done symbolically by referring to the source program variables and labels.



JULY, 1984
ORDER NUMBER: 980931-003

INTEGRATED HARDWARE/SOFTWARE DEVELOPMENT

The ICE-86A emulator allows hardware and software development to proceed interactively. This is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-86A module, prototype hardware can be added to the system as it is designed. Software and hardware testing occurs while the product is being developed.

The ICE-86A emulator assists during three stages of development:

1. It can be operated without being connected to the user's system, so the ICE-86A module's debugging capabilities can be used to facilitate program development before any of the user's hardware is available.

2. Integration of software and hardware can begin when any functional element of the user system hardware is connected to the 8086 socket. Because of the ICE-86A emulator mapping capabilities, Intellec memory, ICE module memory, or diskette memory can be substituted for missing prototype memory. Time-critical program modules are debugged before hardware implementation by using the 2K-bytes of high-speed ICE-resident memory. As each section of the user's hardware is completed, it is added to the prototype. Thus each section of the hardware and software is system tested as it becomes available.

3. When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-86A module is then used for real-time emulation of the 8086 to debug the system as a completed unit.

Thus the ICE-86A module provides the user with the ability to debug a prototype of production system at any stage in its development without introducing extraneous hardware or software test tools.

SYMBOLIC DEBUGGING

Symbols and high-level language statement numbers may be substituted for numeric values in any of the ICE-86A emulator commands. This allows the user to make symbolic references to I/O ports, memory addresses, and data in the user program. Thus, the user need not remember the addresses of variables of program subroutines.

Symbols can be used to reference variables, procedures, program labels, and source statements. A variable can be displayed or changed by referring to it by name rather than by its absolute location in memory. Using symbols for statement labels, program labels, and procedure names simplifies both tracing and breakpoint setting. Disassembly of a section of code from either trace or program memory into its assembly mnemonics is readily accomplished.

Furthermore, each symbol may have associated with it one of the data types BYTE, WORD, INTEGER, SINTEGER (for short, 8-bit integer),

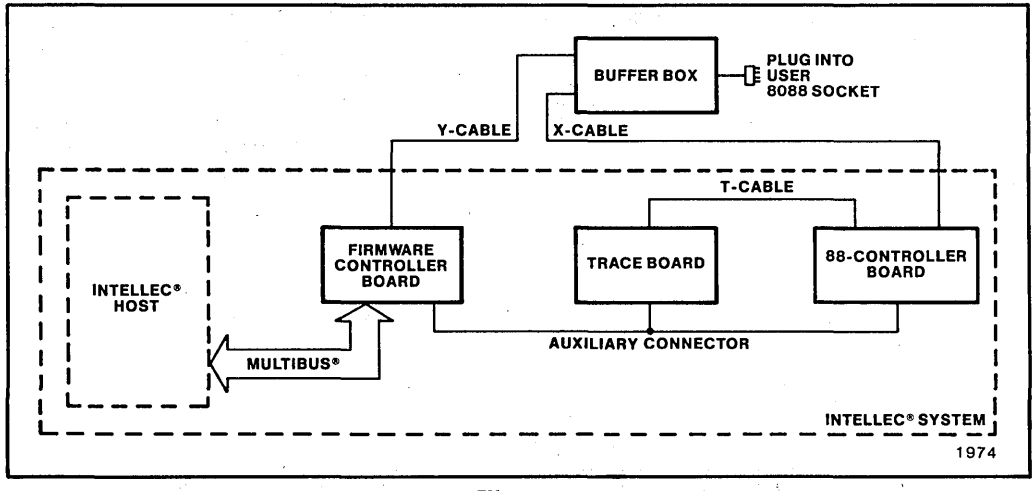


Figure 1. ICE-86A™ Emulator Block Diagram

980931-003

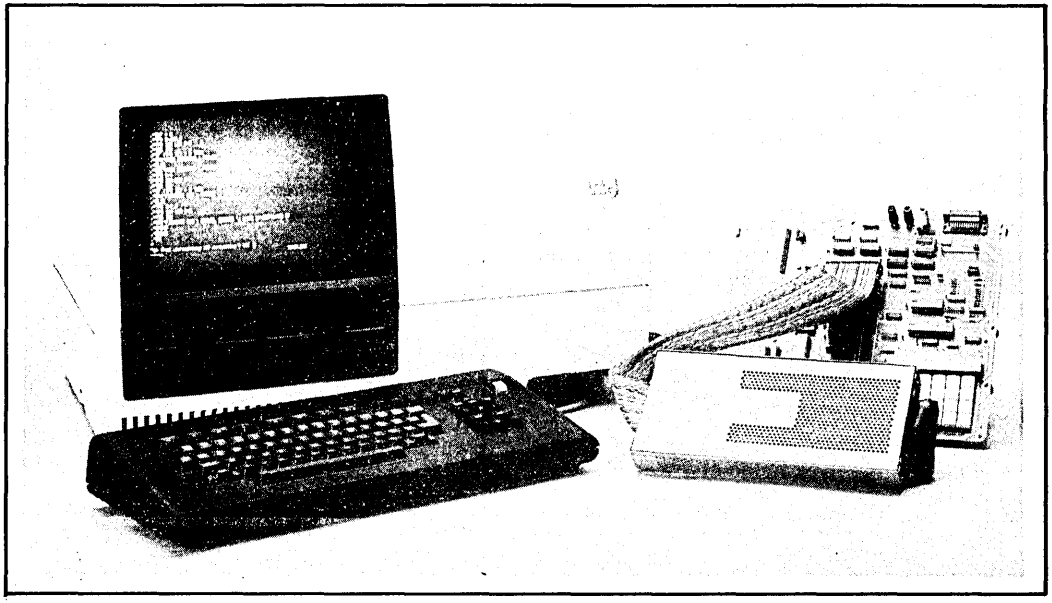


Figure 2. A typical iAPX 86 development configuration. It is based on an Series IV development system, which hosts the ICE-86A™ emulator. The ICE-86A™ module is shown connected to a user prototype system, in this case, an SDK-86.

POINTER, REAL, DREAL, or TREAL. Thus, the user need not remember the type of a source program variable when examining or modifying it. For example, the command "!VAR" displays the value in memory of variable VAR in a format appropriate to its type, while the command "!VAR = !VAR + 1" increments the value of the variable.

The user symbol table generated along with the object file during a PL/M-86, PASCAL-86 or FORTRAN-86 compilation or an ASM-86 assembly is loaded into memory along with the user program which is to be emulated. The user can utilize the available symbol table space more efficiently by using the SELECT option to choose which program modules will have symbols loaded in the symbol table. The user may also add to this symbol table any additional symbolic values for memory addresses, constants, or variables that are found useful during system debugging.

The ICE-86A module provides access through symbolic definition to all of the 8086 registers and flags. the READY, NMI, $\overline{\text{TEST}}$, HOLD, RESET, INTR, MN/$\overline{\text{MX}}$, and $\overline{\text{RQ}}$/$\overline{\text{GT}}$ pins of the 8086 can also be read. Symbolic reference to key ICE-86A emulation information are also provided.

MACROS AND COMPOUND COMMANDS

The ICE-86A module provides a programmable diagnostic facility which allows the user to tailor its operation using macro commands and compound commands.

A macro is a set of ICE-86A commands which is given a single name. Thus, a sequence of commands which is executed frequently may be invoked simply by typing in a single command. Users first define the macro by entering the entire sequence of commands which they want to execute. They then name the macro and store it for future use. They execute the macro by typing its name and passing up to ten parameters to the commands in the macro. Macros may be saved on a disk file for use in subsequent debugging sessions.

Compound commands provide conditional execution of commands (IF), and execution of commands until a condition is met or until they have been executed a specified number of times (COUNT, REPEAT).

Compound commands and macros may be nested up to 8 deep.

MEMORY MAPPING

Memory for the user system can be resident in the user system or "borrowed" from the Intellec System through the ICE-86A emulator's mapping capability. The speed of emulation by the ICE-86A module depends on which mapping options are being used.

The ICE-86A emulator allows the memory which is addressed by the 8086 to be mapped in 8K-byte blocks to the following locations:

1. Physical memory in the user's system, which provides 100 percent real-time emulation at the user-system clock rate (up to 5 MHz) with no wait-states.
2. Either of two 1K-byte blocks of ICE-86A module high-speed memory, which allows nearly full-speed emulation (with two additional wait-states per 8086-controlled bus cycle).
3. Intellec System memory, which provides emulation at approximately 0.02 percent of real-time with a 5 MHz clock.
4. A random-access diskette file, with emulation speed comparable to Intellec System memory, except the emulation must wait when a new page is accessed on the diskette.

The user can also designate a block of memory as non-existent. The ICE-86A module issues an error message when any such guarded memory is addressed by the user program.

As the user prototype progresses to include memory, emulation becomes real time.

OPERATION MODES

The ICE-86A software is a RAM-based program that provides the user with easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-86A commands are configured with a broad range of modifiers which provide the user with maximum flexibility in describing the operation to be performed.

Emulation

Emulation commands to the ICE-86A emulator control the process of setting up, running, and halting an emulation of the user's iAPX 86 system. Breakpoints and tracepoints enable the ICE-86A module to halt emulation and provide a detailed trace of execution in any part of the user's program. A summary of the emulation commands is shown in Table 1.

Table 1. Summary of ICE™-86A Emulation Commands

Command	Description
GO	Initializes emulation and allows the user to specifiy the starting point and breakpoints. Example: GO FROM.START TILL.DELAY EXECUTED where START and DELAY are statement lables.
STEP	Allows the user to single-step through the program

Breakpoints: The ICE-86A module has two breakpoint registers that allow the user to halt emulation when a specified condition is met. The breakpoint registers may be set up for execution or non-execution breaking. An execution breakpoint consists of a single address which causes a break whenever the 8086 executes from its queue an instruction byte which was obtained from the address. A non-execution breakpoint causes an emulation break when a specified condition other than an instruction execution occurs. A non-execution breakpoint condition, using one or both breakpoint registers, may be specified by any one of or a combination of:

1. *A set of address values* — breaking on a set of address values has three valuable features:
   a. The ability to break on a single address.
   b. The ability to set any number of breakpoints within a limited range (1,024 bytes maximum) of memory.
   c. The ability to break in an unlimited range. Execution is halted on any memory access to an address greater than (or less than) any 20-bit breakpoint address.
2. *A particular status of the 8086 bus* — one or more of: memory or I/O read or write, instruction fetch, halt, or interrupt acknowledge.
3. *A set of data values* — features comparable to break on a set of address values, explained in point one.

980931-003

4. *A segment register* — break occurs when the register is used in an effective address calculation.

An emulation break can also be set to occur on an external signal condition. An external breakpoint match output and emulation status lines are provided on the buffer box. These allow synchronization of other test equipment when a break occurs or when emulation is begun. Execution breakpoints set to occur on instructions requiring only 2 or 3 clock cycles to complete will break after completion of the following instruction.

Tracepoints: The ICE-86A module has two tracepoint registers which establish match conditions to conditionally start and stop trace collection. The trace information is gathered at least twice per bus cycle, first when the address signals are valid and second when the data signals are valid. If the 8086 execution queue is otherwise active, additional frames of trace are collected.

Each trace frame contains the 20 address/16 data lines and detailed information on the status of the 8086. The trace memory can store 1,024 frames, or an average of about 300 bus cycles, providing ample data for determining how the 8086 was reacting prior to emulation break. The trace memory contains the last 1,024 frames of trace data collected, even if this spans several separate emulations. The user has the option of displaying each frame of trace data or displaying by instruction in actual ASM-86 Assembler mnemonics. Unless the user chooses to disable trace, the trace information is always available after an emulation.

Interrogation and Utility

Interrogation and utility commands give the user convenient access to detailed information about the user program and the state of the 8086 that is useful in debugging hardware and software. Changes can be made in both memory and the 8086 registers, flags, input pins, and I/O ports. Commands are also provided for various utility operations such as loading and saving program files, defining symbols and macros, displaying trace data, setting up the memory map, and returning control to ISIS-II. A summary of the basic interrogation and utility commands is shown in Table 2.

iAPX 86/20 DEBUGGING

The ICE-86A module has the extended capabilities to debug iAPX 86/20 microsystems which contain both the 8086 microprocessor and the 8087 Numeric Processor Extension (NPX). An iAPX 86/20 system is configured in the 8086's maximum mode and communication between the processors is accomplished through the $\overline{RQ/GT}$ signals. Debugging can be done either using the 8087 chip itself (in which case the 8086 ESCAPE instruction is interpreted as a floating point instruction) or using the 8087 software emulator E8087 (where the 8086 INTERRUPT instruction is interpreted as a floating point instruction). Three new data types are defined to use the NPX:

REAL (4 byte short real)
DREAL (8 byte long real)
TREAL (10 byte temporary real)

While the 8087 NPX is not a programmable part, it does interact closely with the 8086 and can execute instructions in parallel with it. The ICE-86A module provides information about the relative timing of instruction execution in each processor so that the complete system can be debugged. Other debugging capabilities available through the ICE-86A module are: symbolically disassemble NPX call instructions from memory or trace history; display or change the control, status and flag values of the NPX; display the NPX stack either in hexadecimal or disassembled form; and display the last instruction address, last operand, and last operand address. The 8087 can only communicate with user memory.

Multiprocessor Operation

The ICE-86A emulator supports 8089 configurations in both local and remote modes. The ICE-86A emulator may be operating either in minimum or maximum mode. In maximum mode, the 8086 $\overline{RQ/GT}$ lines are employed. This is required for the 8089 local mode configuration to provide local bus arbitration between the two processors.

DESIGN CONSIDERATIONS

- When the ICE-86A system is operating in interrogation mode, responses to HOLD/HOLD ACKNOWLEDGE can require up to 450 microseconds.
- A HOLD sequence error will occur if an additional hold pulse is inserted before the ICE-86A system responds to the previous hold pulse.
- To enter emulation, user READY must be high to avoid a READY$TIMEOUT error.

980931-003

- The ICE-86A system generates an extra bus cycle upon entry into emulation. Users should ignore the extra bus cycle.
- If a user applies a RESET during generation of HOLD, a failure message may result.

Table 2. Selected ICE™-88A Module Interrogation and Utility Commands

Memory/Register Commands
Display or change the contents of:
- Memory
- 8088 registers
- 8088 status flags
- 8088 input pins
- 8088 I/O ports
- ICE-86A pseudo-registers (e.g., emulation timer)

Memory Mapping Commands
Display, declare, set or reset the ICE-86A memory mapping.

Symbol Manipulation Commands
Display any or all symbols, program modules, and program line numbers and their associated values (locations in memory).

Set the domain (choose the particular program module) for the line numbers.

Define new symbols as they are needed in debugging.

Remove any or all symbols, modules, and program statements.

Change the value of any symbol.

Select program modules whose symbols will be used in debugging.

TYPE
Assign or change the type of any symbol in the symbol table.

DASM
Disassemble user program memory into ASM-86 assembler mnemonics.

RQ/GT
Set or display the status of the request/grant facility which enables the ICE-86A module to share the system bus with coprocessors.

BUS
Display which device in the user's iAPX 86 system is currently master of the system bus.

CAUSE
Display the cause of the most recent emulation break.

PRINT
Display the specified portion of the trace memory.

LOAD
Fetch user symbol table and object code from the input file.

EVALUATE
Display the value of an expression in binary, octal, decimal, hexadecimal, and ASCII.

CLOCK
Select the internal (ICE-86A module provided, for stand-alone mode only) or an external (user-provided) system clock.

RWTIMEOUT
Allows the user to time out READ/WRITE command signals based on the time taken by the 8086 to access Intellec memory or diskette memory.

ENABLE/DISABLE RDY
Enable or disable logical AND or ICE-86A emulator ready with the user ready signal for accessing Intellec memory, ICE memory, or diskette memory.

980931-003

DC CHARACTERISTICS OF THE ICE™-86A MODULE USER CABLE

1. **Output low Voltages [V_{OL} (Max)=0.4V]**

	I_{OL} (Min)
AD0-AD15	12 mA (24 mA @ 0.5V)
A16/S3-A19/S7, $\overline{BHE}$/S7, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, M/$\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$	8 mA (16 mA @ 0.5V)
HLDA	7 mA
$\overline{RQ}$/GT	16 mA

2. **Output High Voltages [V_{OH} (Min)=2.4V]**

	I_{OH} (Min)
AD0-AD15	−3 mA
A16/S3-A19/S7, $\overline{BHE}$/S7, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, S0, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, M/IO, DT/R, $\overline{DEN}$, ALE, $\overline{INTA}$, HLDA	−2.6 mA
$\overline{RQ}$/GT	250 mA

3. **Input Low Voltages [V_{IL} (Max)=0.18V]**

	I_{IL} (Max)
AD0-AD15	−0.2 mA
NMI, CLK	−0.4 mA
READY	−0.8 mA
INTR, HOLD, $\overline{TEST}$, RESET	−1.4 mA
MN/$\overline{MX}$ (0.1 μF to GND)	−3.3 mA

4. **Input High Voltages [V_{IH} (Min))=2.0V]**

	I_{IH} (Max)
AD0-AD15	80 μA
NMI, CLK	20 μA
READY	40 μA
INTR, HOLD, $\overline{TEST}$, RESET	−0.4 mA
MN/$\overline{MX}$ (0.1 μF to GND)	−1.1 mA

5. **No current is taken from the user circuit at the V_{CC} pin.**

SPECIFICATIONS

ICE-86A Operating Environment

REQUIRED HARDWARE

Intellec Model 800, Series II, Series III, or Series IV microcomputer development system with the following:

- Three adjacent slots for the ICE-86A module.
- 64K bytes of Intellec memory. If user prototype program memory is desired, additional memory above the basic 64K is required.

System console (Model 800 only)
Disk drive (Model 800 only)
ICE-86A module

REQUIRED SOFTWARE

System Monitor
ISIS, version 4.3 or subsequent versions

Equipment Supplied

Printed circuit boards (3)
Interface cable and emulation buffer module
Operator's manual
ICE-86A software, diskette-based
—8 inch single and double density
—5¼ inch double density

Emulation Clock

User system clock up to 5 MHz or 2 MHz
ICE-86A internal clock in stand-alone mode

Physical Characteristics

PRINTED CIRCUIT BOARDS

Width: 12.00 in (30.48 cm)
Height: 6.75 in (17.15 cm)
Depth: 0.50 in (1.27 cm)
Package Weight: 9.00 (4.10 kg)

Electrical Characteristics

DC POWER

V_{CC} = +5V +5% −1%
I_{CC} = 17A maximum; 11A typical
V_{DD} = +12V ±5%
I_{DD} = 120 mA maximum; 80 mA typical
V_{BB} = −10V ±5% or −12V ±5% (optional)
I_{BB} = 25 mA maximum; 12 mA typical

Environmental Characteristics

OPERATING TEMPERATURE

0° to 40°C

OPERATING HUMIDITY

Up to 95% relative humidity without condensation.

980931-003

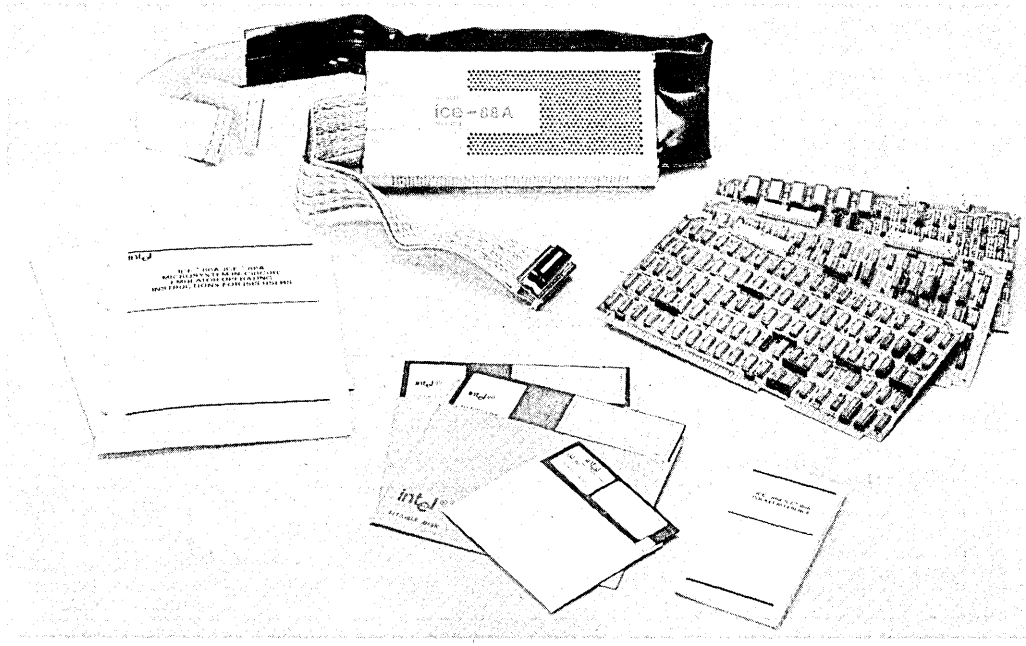
ORDERING INFORMATION

Part Number	Description
ICE-86A	iAPX 86 microsystem in-circuit emulator, cable assembly, and interactive software.

ICE™-88A
iAPX 88 IN-CIRCUIT EMULATOR

- **Real-time in-circuit emulation of iAPX 88 microsystems**
- **Emulate both minimum and maximum modes of 8088 CPU, including RQ/GT**
- **Handles full 1 megabyte of iAPX address space**
- **Breakpoints to halt emulation on a wide variety of conditions**
- **Comprehensive trace of program execution**
- **Full symbolic debugging**
- **Disassembly of trace or program memory from object code into assembler mnemonics**
- **Full 8087 support, including trace disassembly and 8087 data type entry and display options**

The Intel ICE™-88A in-circuit emulator provides sophisticated hardware and software debugging capabilities for iAPX 88 microsystems and iAPX 88 single-board computers. These capabilities include in-circuit emulation for the 8088 central processing unit plus extensions to debug systems including the 8087 numeric processor extension. The emulator includes three circuit boards which reside in any Intellec® microcomputer development system (see Figure 1). A cable and buffer box connect the Intellec system to the user system by replacing the user's 8088, thus extending powerful Intellec system debugging functions into the user system (see Figure 2). Using the ICE-88A module, the designer can execute prototype iAPX 88 software in continuous or single-step modes and can substitute blocks of Intellec system memory for user equivalents. Breakpoints allow the user to stop emulation on user-specified conditions of the iAPX 88 system, and the trace capability gives a detailed history of the program execution prior to the break. All user access to the prototype system software may be done symbolically by referring to the source program variables and labels.



JULY, 1984
ORDER NUMBER: 210357-002

INTEGRATED HARDWARE/ SOFTWARE DEVELOPMENT

The ICE-88A emulator allows hardware and software development to proceed interactively. This is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-88A module, prototype hardware can be added to the system as it is designed. Software and hardware testing occurs while the product is being developed.

The ICE-88A emulator assists in three stages of development:

1. It can be operated without being connected to the user's system, so the ICE-88A module's debugging capabilities can be used to facilitate program development before any of the user's hardware is available.

2. Integration of software and hardware can begin when any functional element of the user system hardware is connected to the 8088 socket. Because of the ICE-88A emulator mapping capabilities, Intellec memory, ICE module memory, or diskette memory can be substituted for missing prototype memory. As each section of the user's hardware is completed, it is added to the prototype. Thus each section of the hardware and software is system tested as it becomes available.

3. When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-88A module is then used for real-time emulation of the 8088 to debug the system as a completed unit.

Thus, the ICE-88A module provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

SYMBOLIC DEBUGGING

Symbols and high-level language statement numbers may be substituted for numeric values in any of the ICE-88A emulator commands. This allows the user to make symbolic references to I/O ports, memory addresses, and data in the user program. Thus, the user need not remember the addresses of variables or program subroutines.

Symbols can be used to reference variables, procedures, program labels, and source statements. A variable can be displayed or changed by referring to it by name rather than by its absolute location in memory. Using symbols for statement labels, program labels, and procedure names simplifies both tracing and breakpoint setting. Disassembly of a section of code from either trace or program memory into its assembly mnemonics is readily accomplished.

Furthermore, each symbol may have associated with it one of the data types BYTE, WORD, INTEGER, SINTEGER (for short, 8-bit integer), POINTER, REAL, DREAL, or TREAL. Thus, the user need not remember the type of a source program variable when examining or modifying it. For example, the command "!VAR" displays the value in memory of variable VAR in a format appropriate to its type, while the command "!VAR = !VAR + 1" increments the value of the variable.

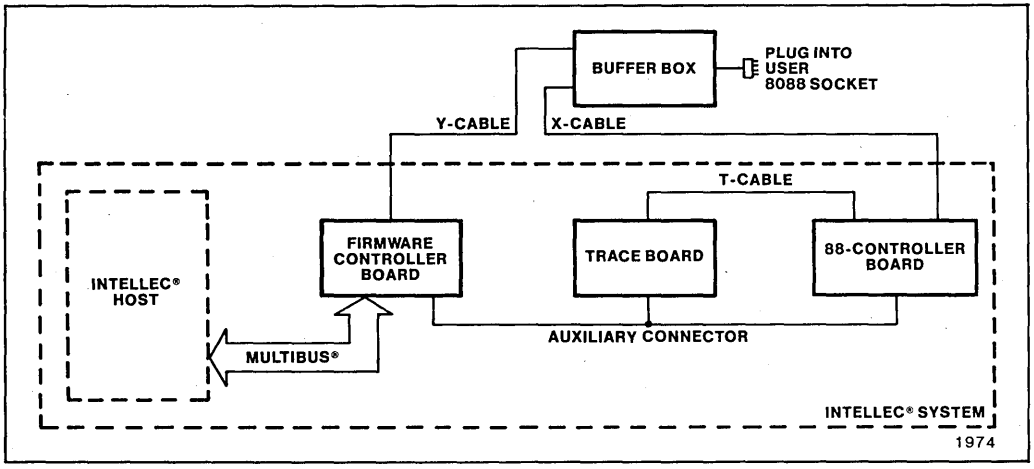


Figure 1. ICE™-88A Emulator Block Diagram

210357-002

The user symbol table generated along with the object file during a PL/M-86/88, Pascal-86/88, or FORTRAN-86/88 compilation or an ASM-86/88 assembly is loaded into memory along with the user program which is to be emulated. The user can utilize the available symbol table space more efficiently by using the SELECT option to choose which program modules will have symbols loaded in the symbol table. The user may also add to this symbol table any additional symbolic values for memory addresses, constants, or variables that are found useful during system debugging.

The ICE-88A module provides access through symbolic definition to all of the 8088 registers and flags. The READY, NMI, $\overline{\text{TEST}}$, HOLD, RESET, INTR, MN/$\overline{\text{MX}}$, and $\overline{\text{RQ}}$/$\overline{\text{GT}}$ pins of the 8088 can also be read. Symbolic references to key ICE-88A emulation information are also provided.

MACROS AND COMPOUND COMMANDS

The ICE-88A module provides a programmable diagnostic facility which allows the user to tailor its operation using macro commands and compound commands.

A macro is a set of ICE-88A commands which is given a single name. Thus, a sequence of commands which is executed frequently may be invoked simply by typing in a single command. Users first define the macro by entering the entire sequence of commands which they want to execute. They then name the macro and store it for future use. They execute the macro by typing its name and passing up to ten parameters to the commands in the macro. Macros may be saved on a disk file for use in subsequent debugging sessions.

Compound commands provide conditional execution of commands (IF), and execution of commands until a condition is met or until they have been executed a specified number of times (COUNT, REPEAT).

Compound commands and macros may be nested up to eight deep.

MEMORY MAPPING

Memory for the user system can be resident in the user system or borrowed from the Intellec system through the ICE-88A emulator's mapping capability. The speed of emulation by the ICE-88A module depends on which mapping options are being used.

The ICE-88A emulator allows the memory which is addressed by the 8088 to be mapped in *1K-byte blocks to the following locations:*

1. Physical memory in the user's system, which provides 100 percent real-time emulation at the user-system clock rate (up to 5 MHz) with no wait states.

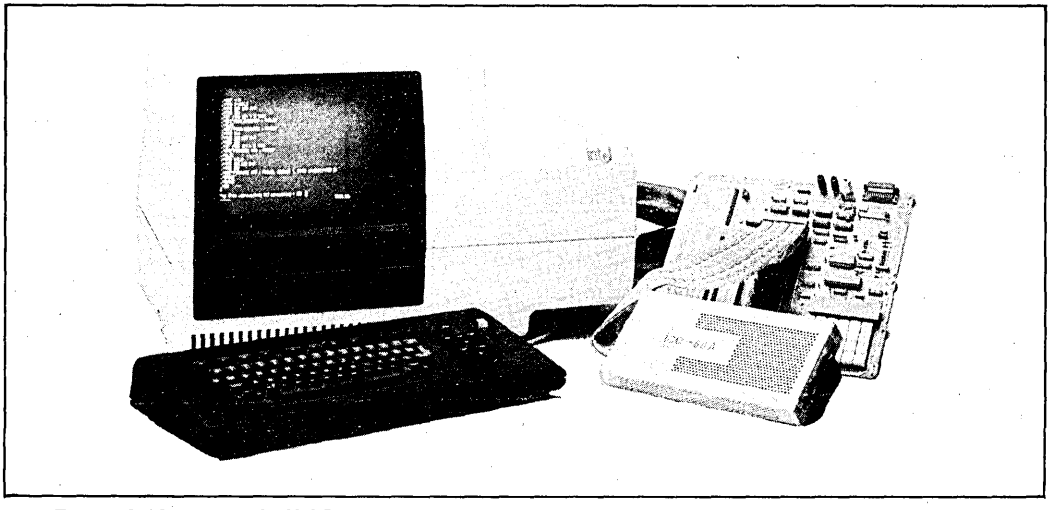


Figure 2. A typical iAPX 88 development configuration. It is based on an Series IV development system, which hosts the ICE-88A™ emulator. The ICE-88A™ module is shown connected to a user prototype system.

2. Either of two 1K-byte blocks of ICE-88A module high-speed memory, which allows nearly full-speed emulation (with two additional wait states per 8088-controlled bus cycle).

3. Intellec system memory, which provides emulation at approximately 0.02 percent of real-time with a 5 MHz clock.

4. A random-access diskette file, with emulation speed comparable to Intellec system memory, except the emulation must wait when a new page is accessed on the diskette.

The user can also designate a block of memory as non-existent. The ICE-88A module issues an error message when any such guarded memory is addressed by the user program.

As the user prototype progresses to include memory, emulation becomes real time.

OPERATION MODES

The ICE-88A software is a RAM-based program that provides the user with easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-88A commands are configured with a broad range of modifiers which provide the user with maximum flexibility in describing the operation to be performed.

Emulation

Emulation commands to the ICE-88A emulator control the process of setting up, running and halting an emulation of the user's iAPX 88 system. Breakpoints and tracepoints enable the ICE-88A module to halt emulation and provide a detailed trace of execution in any part of the user's program. A summary of the emulation commands is shown in Table 1.

Breakpoints: The ICE-88A module has two breakpoint registers that allow the user to halt emulation when a specified condition is met. The breakpoint registers may be set up for execution or non-execution breaking. An execution breakpoint consists of a single address which causes a break whenever the 8088 executes from its queue an instruction byte which was obtained from the address. A non-execution breakpoint causes an emulation break when a specified condition other than an instruction execution occurs. A non-execution breakpoint condition, using one or both breakpoint registers, may be specified by any one of or a combination of the following:

1. *A set of address values* — Break on a set of address values has three valuable features:
   a. The ability to break on a single address.
   b. The ability to set any number of breakpoints within a limited range (1,024 bytes maximum) of memory.
   c. The ability to break in an unlimited range. Execution is halted on any memory access to an address greater than (or less than) any 20-bit breakpoint address.
2. *A particular status of the 8088 bus* — one or more of: memory or I/O read or write, instruction fetch, halt, or interrupt acknowledge.
3. *A set of data values* — features comparable to break on a set of address values, explained in point one.
4. *A segment register* — break occurs when the register is used in an effective address calculation.

Table 1. Summary of ICE™-88A Emulation Commands

Command	Description
GO	Initializes emulation and allows the user to specifiy the starting point and breakpoints. Example: GO FROM .START TILL .DELAY EXECUTED where START and DELAY are statement labels.
STEP	Allows the user to single-step through the program.

Emulation break can also be set to occur on an external signal condition. An external breakpoint match output and emulation status lines are provided on the buffer box. These allow synchronization of other test equipment when a break occurs or when emulation is begun. Execution breakpoints set to occur on instructions requiring only two or three clock cycles to complete will break after completion of the following instruction.

Tracepoints: The ICE-88A module has two tracepoint registers which establish match conditions to conditionally start and stop trace collection. The trace information is gathered at

least twice per bus cycle, first when the address signals are valid and second when the data signals are valid. If the 8088 execution queue is otherwise active, additional frames of trace are collected.

Each trace frame contains the 20 address/16 data lines and detailed information on the status of the 8088. The trace memory can store 1,024 frames, or an average of about 300 bus cycles, providing ample data for determining how the 8088 was reacting prior to emulation break. The trace memory contains the last 1,024 frames of trace data collected, even if this spans several separate emulations. The user has the option of displaying each frame of trace data or displaying by instruction in actual ASM-86 assembler mnemonics. Unless the user chooses to disable trace, the trace information is always available after an emulation.

Interrogation and Utility

Interrogation and utility commands give the user convenient access to detailed information about the user program and the state of the 8088 that is useful in debugging hardware and software. Changes can be made in both memory and the 8088 registers, flags, input pins, and I/O ports. Commands are also provided for various utility operations such as loading and saving program files, defining symbols and macros, displaying trace data, setting up the memory map, and returning control to ISIS-II. A summary of the basic interrogation and utility commands is shown in Table 2.

iAPX 88/20 DEBUGGING

The ICE-88A module has the extended capabilities to debug iAPX 88/20 microsystems which contain both the 8088 microprocessor and the 8087 numeric processor extension (NPX). An iAPX 88/20 system is configured in the 8088's maximum mode and communication between the processors is accomplished through the $\overline{RQ}/\overline{GT}$ signals. Debugging can be done either using the 8087 chip itself (in which case the 8088 ESCAPE instruction is interpreted as a floating-point instruction) or using the 8087 software emulator E8087 (where the 8088 INTERRUPT instruction is interpreted as a floating-point instruction). Three new data types are defined to use the NPX:

REAL (4 byte short real)
DREAL (8 byte long real)
TREAL (10 byte temporary real)

While the 8087 NPX is not a programmable part, it does interact closely with the 8088 and can execute instructions in parallel with it. The ICE-88A module provides information about the relative timing of instruction execution in each processor so that the complete system can be debugged. Other debugging capabilities available through the ICE-88A module are: symbolically disassemble NPX call instructions from memory or trace history; display or change the control, status, and flag values of the NPX; display the NPX stack either in hexadecimal or disassembled form; and display the last instruction address, last operand, and last operand address. The 8087 can only communicate with user memory.

DESIGN CONSIDERATIONS

- When the ICE-88A system is operating in interrogation mode, responses to HOLD/HOLD ACKNOWLEDGE can require up to 450 microseconds.
- A HOLD sequence error will occur if an additional hold pulse is inserted before the ICE-88A system responds to the previous hold pulse.
- To enter emulation, user READY must be high to avoid a READY$TIMEOUT error.
- If a user applies a RESET during generation of HOLD, a failure message may result.
- The ICE-88A system generates an extra bus cycle upon entry into emulation. Users should ignore the extra bus cycle.
- The ICE-88A system also generates extra bus cycles under another condition: While normally accessing the user system (that is, for download and memory interrogation commands), in addition to the bus cycles required for read-after-write, extra bus cycles are generated during interrogation. Users should ignore the extra bus cycles.

DC CHARACTERISTICS OF THE ICE™-88A MODULE USER CABLE

1. Output Low Voltages [V_{OL} (Max)=0.4V]

	I_{OL} (Min)
AD0-AD7	12 mA
A8-A15	(24 mA @ 0.5V)
A16/S3-A10/S6, $\overline{SSO}$, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, $\overline{IO}$/M, DT/R, $\overline{DEN}$, ALE, $\overline{INTA}$	8 mA (16 mA @ 0.5V)
HLDA	7 mA
$\overline{RQ}/\overline{GT}$	16 mA

210357-002

Table 2. Selected ICE™-88A Module Interrogation and Utility Commands

Memory/Register Commands
Display or change the contents of:
- Memory
- 8088 registers
- 8088 status flags
- 8088 input pins
- 8088 I/O ports
- ICE-88A pseudo-registers (e.g. emulation timer)

Memory Mapping Commands
Display, declare, set, or reset the ICE-88A memory mapping.

Symbol Manipulation Commands
Display any or all symbols, program modules, and program line numbers and their associated values (locations in memory).

Set the domain (choose the particular program module) for the line numbers.

Define new symbols as they are needed in debugging.

Remove any or all symbols, modules, and program statements.

Change the value of any symbol.

Select program modules whose symbols will be used in debugging.

TYPE
Assign or change the type of any symbol in the symbol table.

DASM
Disassemble user program memory into ASM-86/88 assembler mnemonics.

RQ/GT
Set or display the status of the request/grant facility which enables the ICE-88A module to share the system bus with co-processors.

BUS
Display which device in the user's iAPX 88 system is currently master of the system bus.

CAUSE
Display the cause of the most recent emulation break.

PRINT
Display the specified portion of the trace memory.

LOAD
Fetch user symbol table and object code from the input file.

EVALUATE
Display the value of an expression in binary, octal, decimal, hexadecimal, and ASCII.

CLOCK
Select the internal (ICE-88A module provided, for stand-alone mode only) or an external (user-provided) system clock.

RWTIMEOUT
Allows the user to time out READ/WRITE command signals based on the time taken by the 8088 to access Intellec memory or diskette memory.

ENABLE/DISABLE RDY
Enable or disable logical AND or ICE-88A emulator Ready with the user Ready signal for accessing Intellec memory, ICE memory or diskette memory.

2. Output High Voltages [V_{OH} (Min)=2.4V]

	I_{OH} (Min)
AD0-AD7	−3 mA
A8-A15	
A16/S3-A19/S6, $\overline{SSO}$, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, $\overline{IO}$/M, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$, HLDA	−2.6 mA
$\overline{RQ/GT}$	250 mA

3. Input Low Voltages [V_{IL} (Max)=0.8V]

	I_{IL} (Max)
AD0-AD7	−0.2 mA
NMI, CLK	−0.4 mA
READY	−0.8 mA
INTR, HOLD, $\overline{TEST}$, RESET	−1.4 mA
MN/$\overline{MX}$ (0.1 μF to GND)	−3.3 mA

210357-002

4. **Input High Voltages [V_{IH} (Min)=2.0V]**

	I_{IH} (Max)
AD0-AD7	80 μA
NMI, CLK	20 μA
READY	40 μA
INTR, HOLD, $\overline{TEST}$, RESET	−0.4 mA
MN/$\overline{MX}$ (0.1 μF to GND)	−1.1 mA

5. **No current is taken from the user circuit at V_{CC} pin.**

SPECIFICATIONS

ICE™-88A Operating Environment

REQUIRED HARDWARE

Intellec Model 800, Series II, Series III, or Series IV microcomputer development system with the following features:

- Three adjacent slots for the ICE-88A module.
- 64K bytes of Intellec memory. If user prototype program memory is desired, additional memory above the basic 64K is required.

System console (Model 800 only)
Disk drive (Model 800 only)
ICE-88A module

REQUIRED SOFTWARE

System monitor
ISIS, version 4.3 or subsequent versions

Equipment Supplied

Printed circuit boards (3)
Interface cable and emulation buffer module
Operator's manual
ICE-88A software, diskette-based
— 8 inch single and double density
— 5¼ inch double density

Emulation Clock

User system clock up to 5 MHz or 2 MHz
ICE-88A internal clock in stand-alone mode

Physical Characteristics

PRINTED CIRCUIT BOARDS

Width: 12.00 in (30.48 cm)
Height: 6.75 in (17.15 cm)
Depth: 0.50 in (1.27 cm)
Package Weight: 9.00 (4.10 kg)

Electrical Characteristics

DC POWER

V_{CC} = +5V ± 5%
I_{CC} = 17A maximum; 11A typical
V_{DD} = +12V ± 5%
I_{DD} = 120 mA maximum; 80 mA typical
V_{BB} = −10V ± 5% or −12V ± 5% (optional)
I_{BB} = 25 mA maximum; 12 mA typical

Environmental Characteristics

Operating Temperature: 0° to 40°C

Operating Humidity: Up to 95% relative humidity without condensation.

ORDERING INFORMATION

Part Number	Description
ICE-88A	iAPX 88 microsystem in-circuit emulator, cable assembly, and interactive software.

210357-002

PROM Programming 6

iUP-200A/iUP-201A UNIVERSAL PROM PROGRAMMERS

MAJOR iUP-200A/iUP-201A FEATURES:

- **Support for all Intel PROM families through multiple-device personality modules, which may also be used with the Intel personal development system (iPDS™).**
- **Serial interface to all Intellec® development systems.**
- **Powerful PROM programming software (iPPS).**
- **iUP system self-tests plus device integrity checks.**
- **Support for new personality modules that provide state of the art fast programming algorithms, the inteligent Identifier™, and a security bit.**

ADDITIONAL iUP-201A FEATURES:

- **Off-line editing, device duplication, and PROM memory locking.**
- **32K-byte iUP RAM.**
- **24-character alphanumeric display.**
- **Full hexadecimal plus 12-function keypad.**

The iUP-200A and iUP-201A universal programmers program and verify data in all the Intel programmable ROMs (PROMs). They can also program the PROM memory portions of Intel's single-chip microcomputer and peripheral devices. When used with any Intellec® development system, the iUP-200A and iUP-201A universal programmers provide on-line programming and verification using the Intel PROM programming software (iPPS). In addition, the iUP-201A universal programmer supports off-line, stand-alone program editing, PROM duplication, and PROM memory locking. The iUP-200A universal programmer is expandable to an iUP-201A model.

March, 1984
Order Number:210319-003

FUNCTIONAL DESCRIPTION

The iUP-200A universal programmer operates in on-line mode. The iUP-201A universal programmer operates in both on-line and off-line mode.

On-line System Hardware

The iUP-200A and iUP-201A universal programmers are free-standing units that, when connected to any Intel development system having at least 64K bytes of host memory, provide on-line PROM programming and verification of Intel programmable devices. In addition, the universal programmer can read the contents of the ROM versions of these devices.

The universal programmer communicates with the host through a standard RS-232C serial data link. A serial converter is needed when using the MDS 800 as a host system. (Serial converters are available from other manufacturers.)

Each universal programmer contains an 8085 CPU, selectable power supply, 4.3K bytes of static RAM, a programmable timer, an interface for personality modules, an interface for the host system, and 12K bytes of programmed EPROM. The iUP-201A also has a keyboard and display. The programmed EPROM contains the firmware needed for all universal programmer editing and control functions.

A personality module adapts the universal programmer to a family of PROM devices; it contains all the hardware and firmware necessary to program either a family of Intel PROMs or a single Intel device. The user inserts the personality module into the universal programmer front panel. The personality module comes ready to use; no additional sockets or adapters are required.

Figure 1 shows the iUP-200A on-line system configuration, and Figure 2 shows the on-line system data flow.

On-line System Software

The Intel PROM programming software (iPPS) is included with both the iUP-200A and iUP-201A models of the universal programmer. Created to run on any Intellec development system, the iPPS software provides user control through an easy-to-use interactive interface. The iPPS software performs the following functions to make PROM programming quick and easy:

- Reads PROMs and ROMs
- Programs PROMs directly or from a file

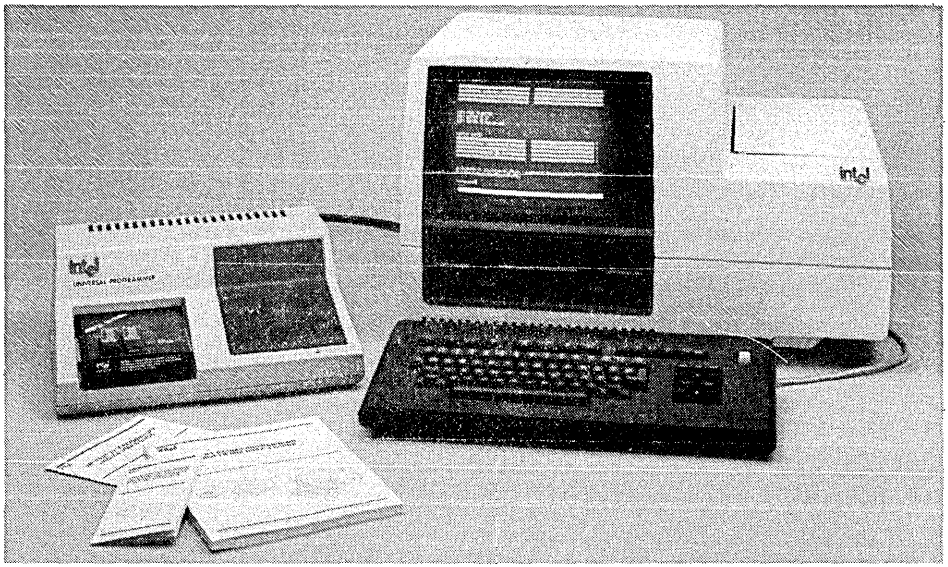


Figure 1 On-Line System Configuration

Order Number: 210319-003

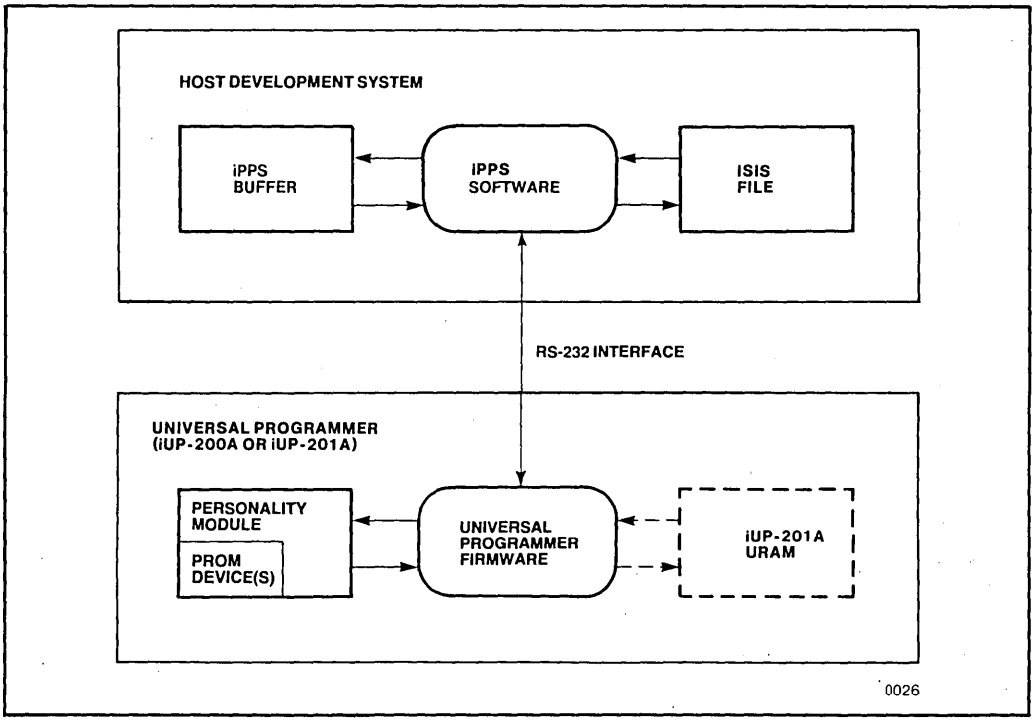


Figure 2 On-Line System Data Flow

- Verifies PROM data with buffer data
- Locks EPROM memory from unauthorized access (on devices which support this feature)
- Prints PROM contents on the network or development system printer
- Performs interactive formatting operations such as interleaving, nibble swapping, bit reversal, and block moves
- Programs multiple PROMs from the source file, prompting the user to insert new PROMs
- Uses a buffer to change PROM contents

All iPPS commands, as well as program address and data information, are entered through the development system ASCII keyboard and displayed on the system CRT. Table 1 summarizes the iPPS commands.

The iPPS software lets the user load programs into a PROM from Intellec system memory or directly from a disk file. Access to the disk lets the user create and manipulate data in a virtual buffer with an address range up to 16M. This large block of data can be formatted into different PROM word sizes for program storage into several different PROM types. In addition, a program stored in the target PROM, the Intellec system memory, or a system disk file can be interleaved with a second program and entered into a specific target PROM or PROMs.

The iPPS software supports data manipulation in the following Intel formats: 8080 hexadecimal ASCII, 8080 absolute object, 8086 hexadecimal ASCII, 8086 absolute object, and 80286 absolute object. Addresses and data can be displayed in binary, octal, decimal, or hexadecimal. The user can easily change default data formats as well as number bases.

The user invokes the iPPS software from the ISIS operating system (Intellec 800, Series II, and Series III, versions V3.4 and later; Series IV, versions V1.0 and later). The software can be run under control of ISIS submit files, thereby freeing the user from repetitious command entry.

Order Number: 210319-003

Table 1 iPPS Command Summary

Command	Description
PROGRAM CONTROL GROUP	CONTROLS EXECUTION OF THE iPPS SOFTWARE.
EXIT	Exits the iPPS software and returns control to the ISIS operating system.
<ESC>	Terminates the current command.
REPEAT	Repeats the previous command.
ALTER	Edits and re-executes the previous command.
UTILITY GROUP	DISPLAYS USER INFORMATION AND STATUS AND SETS DEFAULT VALUES.
DISPLAY	Displays PROM, buffer, or file data on the console.
PRINT	Prints PROM, buffer, or file data on the local printer.
QUEUE	Prints PROM, buffer, or file data on the network spooled printer.
HELP	Displays user assistance information.
MAP	Displays buffer structure and status.
BLANKCHECK	Checks for unprogrammed PROMs.
OVERLAY	Checks whether non-blank PROMs can be programmed.
TYPE	Selects the PROM type.
INITIALIZE	Initializes the default number base and file type.
WORKFILES	Specifies the drive device for temporary work files.
BUFFER GROUP	EDITS, MODIFIES, AND VERIFIES DATA IN THE BUFFER.
SUBSTITUTE	Examines and modifies buffer data.
LOADDATA	Loads a section of the buffer with a constant.
VERIFY	Verifies data in the PROM with buffer data.
FORMATTING GROUP	REARRANGES DATA FROM THE PROM, BUFFER, OR FILE.
FORMAT	Formats and interleaves buffer, PROM, or file data.
COPY GROUP	COPIES DATA FROM ONE DEVICE TO ANOTHER.
COPY (file to PROM)	Programs the PROM with data in a file on disk.
COPY (PROM to file)	Saves PROM data in a file on disk.
COPY (buffer to PROM)	Programs the PROM with data from the buffer.
COPY (PROM to buffer)	Loads the buffer with data in the PROM.
COPY (buffer to file)	Saves the contents of the buffer in a file on disk.
COPY (file to buffer)	Loads the buffer from a file on disk.
COPY (file to URAM)	Loads file data into the iUP RAM (iUP-201A model only).
COPY (URAM to file)	Saves iUP URAM data in a file on disk (iUP-201A model only).
COPY (buffer to URAM)	Loads the buffer into the iUP URAM (iUP-201A model only).
COPY (URAM to buffer)	Loads iUP URAM data into the buffer (iUP-201A model only).
SECURITY GROUP	LOCKS SELECTED DEVICES TO PREVENT UNAUTHORIZED ACCESS.
KEYLOCK	Locks the PROM from unauthorized access.

System Expansion

The iUP-200A universal programmer can be easily upgraded (by the user) to an iUP-201A universal programmer for off-line operation. The upgrade kit (iUP-PAK-A) is available from Intel or your local Intel distributor.

Off-line System

The iUP-201A universal programmer has all the on-line features of the iUP-200A universal programmer plus off-line editing, PROM duplication, program verification, and locking of PROM memory independent of the host system. The iUP-201A universal programmer also accepts Intel hexadecimal programs developed on non-Intel development systems. Just a few keystrokes download the program into the iUP RAM for editing and loading into a PROM.

Off-line commands are entered using the off-line command keys summarized in Table 2.

In addition to the hardware components included as part of the iUP-200A, the iUP-201A contains a 24-character alphanumeric display, full hexadecimal 12-function keypad, and 32K bytes of iUP RAM. Figure 3 illustrates the iUP-201A keyboard and display.

The two logical devices accessible during off-line operation are the PROM device and the iUP RAM. A typical operation is copying the data from a PROM (or ROM) into the iUP RAM, modifying this data in iUP RAM, and programming the modified data back into a PROM device. The address range of the iUP RAM is automatically determined by the universal programmer when PROM type selection is made. Figure 4 shows the off-line system data flow.

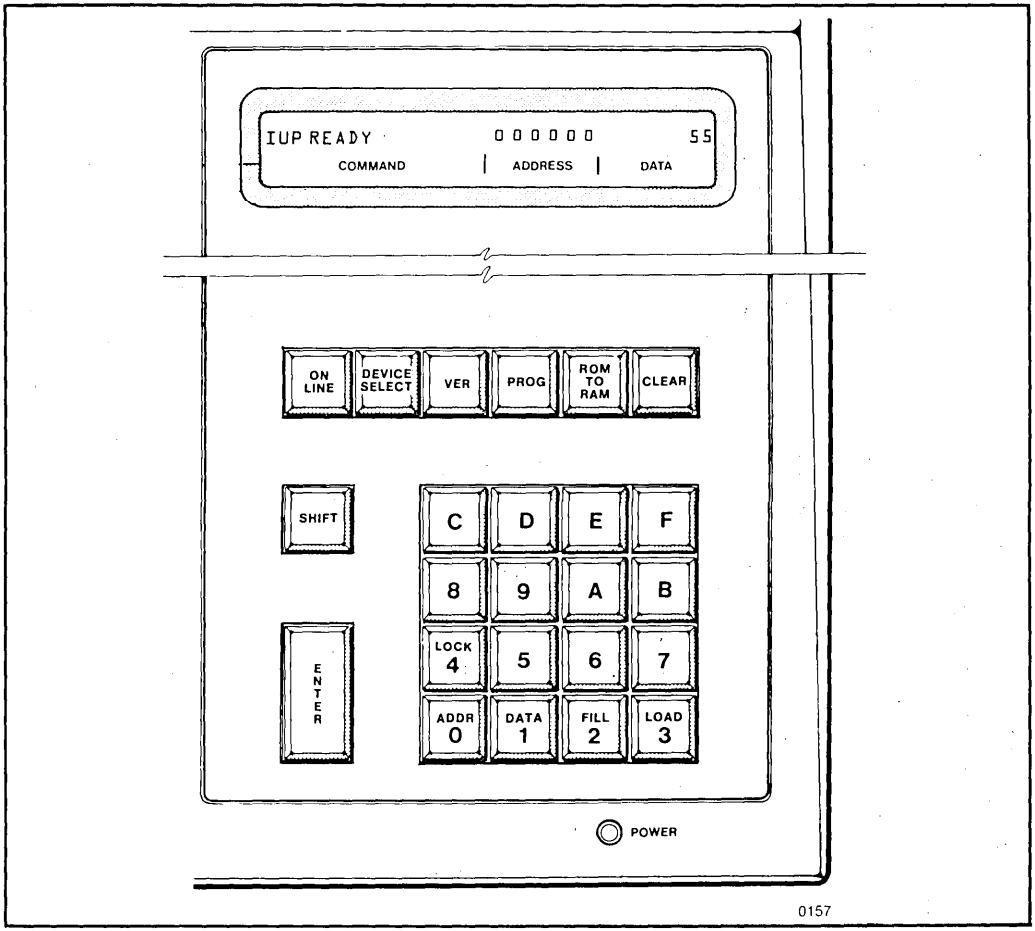


Figure 3 iUP-201A Keyboard and Display

Order Number: 210319-003

Table 2 Off-Line Command Keys Summary

Key	Function
ON LINE	Selects either on-line or off-line operation. Wher on-line, all other function keys are disabled.
DEVICE SELECT	Selects the PROM type when using a personality module able to program multiple PROM devices.
VER	Verifies the contents of the installed PROM device with the contents of the iUP RAM. The universal programmer display indicates the address and the XOR of any mismatches.
PROG	Performs a device blank check and then programs the target PROM with data from the iUP RAM. If the blank check fails, pressing PROG again performs an overlay check to verify that non-blank PROMs can be programmed.
ROM TO RAM	Loads the iUP RAM with the data from the PROM device installed in the personality module.
CLEAR	Terminates the current off-line function, clears a user entry, or restores the display after an error.
ENTER	Transfers information from the universal programmer display (addresses, data, or baud rate) into the iUP RAM.
SHIFT + ADDR 0	Selects an address field for display.
SHIFT + DATA 1	Selects a data field for keypad editing and entry.
SHIFT + FILL 2	Loads a contiguous section of iUP RAM locations with a constant.
SHIFT + LOAD 3	Downloads Intel hexadecimal data from any development system which has an RS-232C port.
SHIFT + LOCK 4	Locks a PROM from unauthorized access.

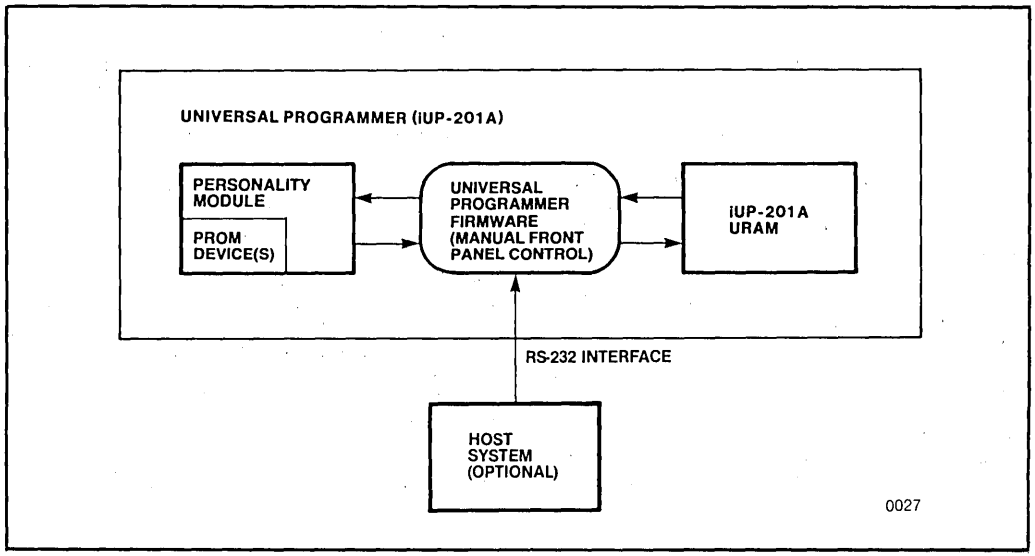


Figure 4 Off-Line System Data Flow

SYSTEM DIAGNOSTICS

Both the iUP-200A and iUP-201A universal programmers include self-contained system diagnostics that verify system operation and aid the user in fault isolation. Diagnostics are performed on the power supply, CPU internal firmware ROM, internal RAM, timer, the iUP-201A keyboard, and the iUP RAM. In addition, tests are made on any personality module installed in the programmer the first time the module is accessed. The personality module tests include the power select circuitry and up to 4K of module firmware. Straight-forward messages are provided on the development system display in on-line mode and on the iUP-201A display in off-line mode.

PERSONALITY MODULES

A personality module is the interface between the iUP-200A/iUP-201A universal programmer (or an iPDS system) and a selected PROM (or ROM). Personality modules contain all the hardware and firmware for reading and programming a family of Intel devices. Each personality module is a single molded unit inserted into the front panel of the universal programmer. No additional adapters or sockets are needed. Table 3 lists the available personality modules.

Each personality module connects to the universal programmer through a 41-pin connector. Module firmware is uploaded into the iUP RAM and executed by the internal 8085A processor.

Table 3 iUP Personality Modules

Personality Module	PROM Type	PROMs and ROMs Supported
iUP-Fast 27/K	EPROM	2764, 2764A, 27128, 27256
iUP-F27/128	E^2/EPROM	2716, 2732, 2732A, 2764, 27128, 2815, 2816
iUP-F87/51A	Microcontroller	8748, 8748H, 8048, 8749H, 8048H, 8049, 8049H, 8050H, 8751, 8751H, 8051
iUP-F87/44A	Peripheral	8741A, 8041A, 8742, 8042, 8744H, 8044AH, 8755A

Order Number: 210319-003

The personality module firmware contains routines necessary to read and program a family of PROMs. In addition, the personality module sends specific information about the selected PROM to the universal programmer to help perform PROM device integrity checks.

LEDs on each personality module indicate operational status. On some personality modules a column of LEDs indicate which PROM device type the user has selected. On some personality modules an LED below the socket indicates which socket is to be used. A red indicator light tells the user when power is being supplied to the selected device. Figure 5 shows the personality modules supported on the universal programmer.

In addition to the testing done by the iUP system self-tests, each personality module contains diagnostic firmware that performs selected PROM tests and indicates status. These tests are performed in both on-line and off-line modes. The PROM installation test verifies that the device is installed in the module correctly and that the ZIF socket is closed. The PROM blank check determines whether a device is blank. The universal programmer automatically determines whether the blank state is all zeros or all ones. The overlay check (performed when a PROM is not blank) determines which bits are programmed, compares those bits against the program to be loaded, and allows programming to continue if they match. As with the system self-tests, straight-forward messages are provided. The user can invoke all of the PROM device integrity checks except the installation test (which occurs automatically any time an operation is selected).

Figure 6 illustrates a typical testing sequence.

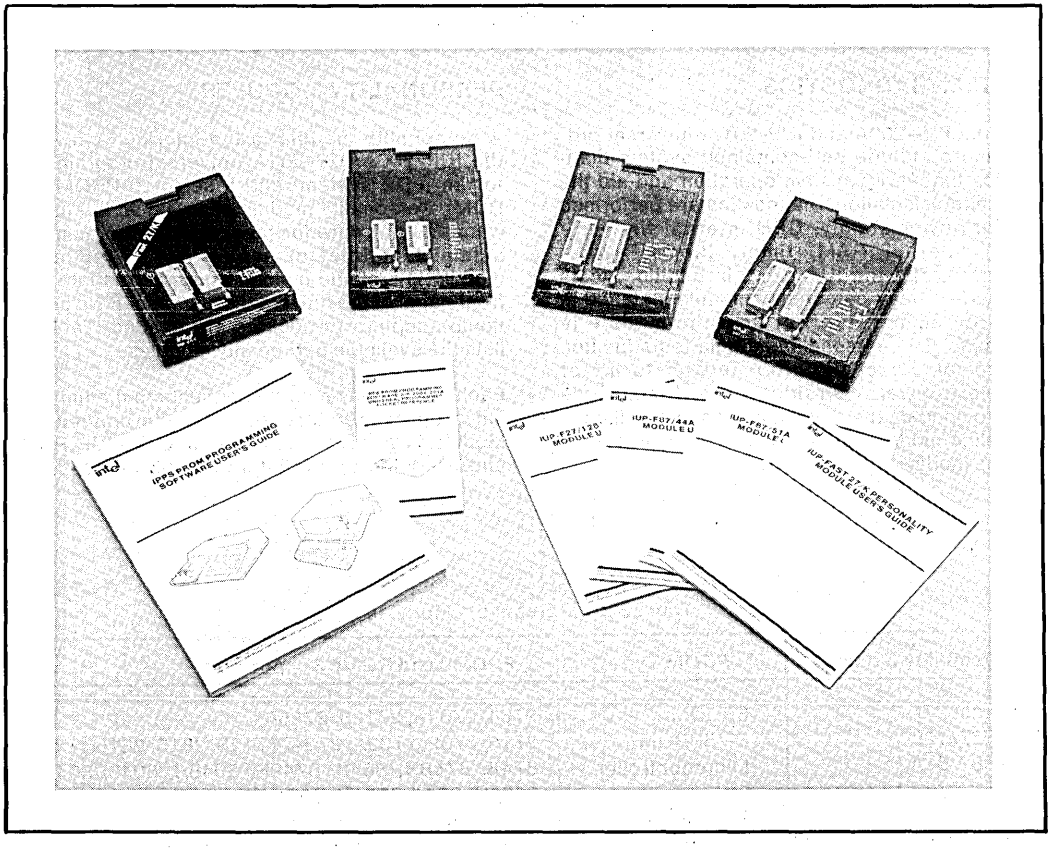


Figure 5 Personality Modules

Order Number: 210319-003

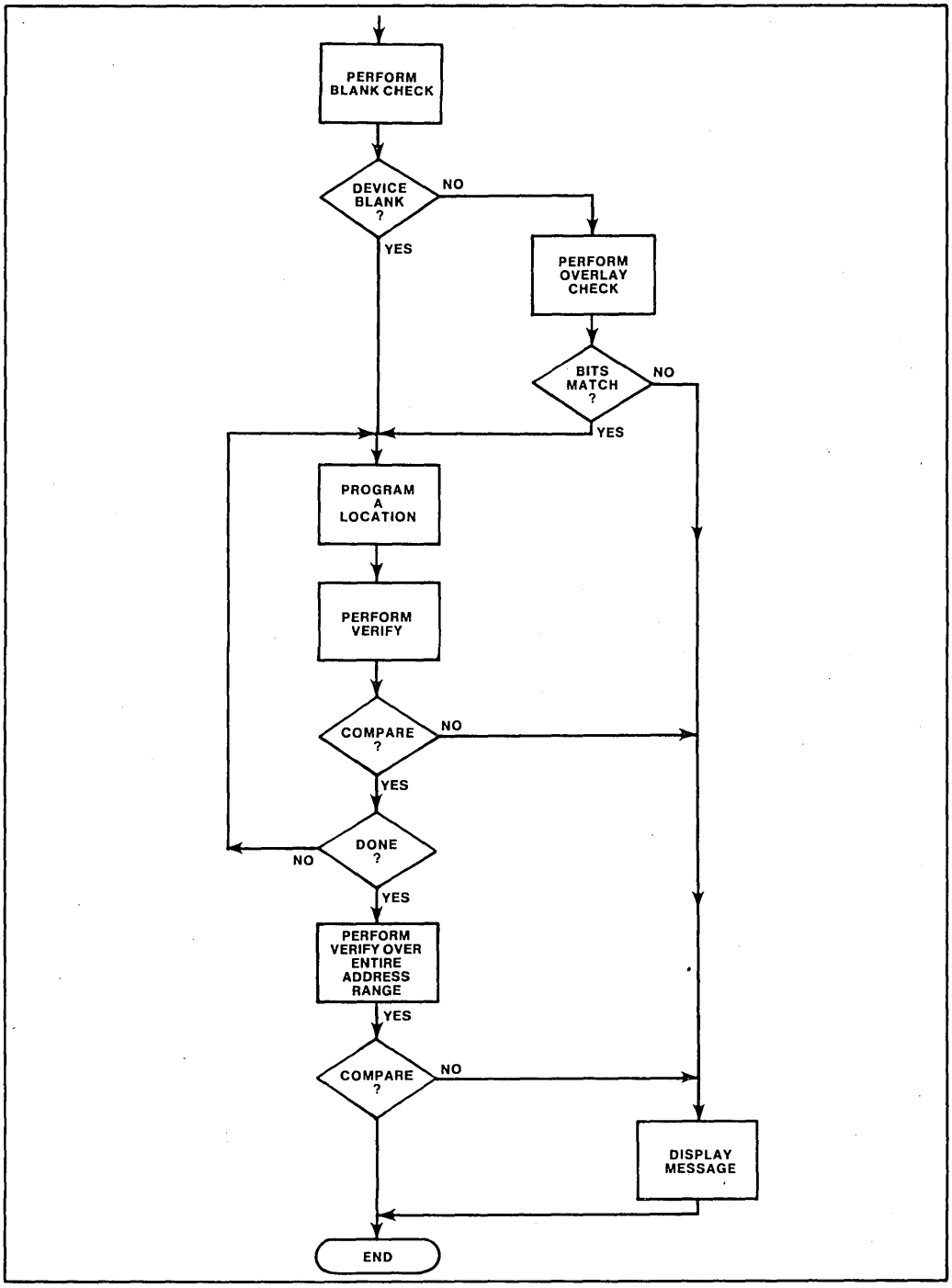


Figure 6 PROM Testing Sequence

Order Number: 210319-003

iUP-200A/iUP-201A SPECIFICATIONS

Control Processor

Intel 8085A microprocessor
6.144 MHz clock rate

Memory

RAM — 4.3 bytes static
ROM — 12K bytes EPROM

Interfaces

Keyboard — 16-character hexadecimal and 12-function keypad (iUP-201A model only)
Display — 24-character alphanumeric (iUP-201A model only)

Software

Monitor — system controller in pre-programmed EPROM
iPPS — Intel PROM programming software on supplied diskette

Physical Characteristics

Depth — 15 inches (38.1 cm)
Width — 15 inches (38.1 cm)
Height — 6 inches (15.2 cm)
Weight — 15 pounds (6.9 kg)

Electrical Characteristics

Selectable 100, 120, 200, or 240 Vac ± 10%; 50-60 Hz
Maximum power consumption — 80 watts

Environmental Characteristics

Reading temperature — 10°C to 40°C
Programming temperature — 25°C ± 5°
Operating humidity — 10% to 85% relative humidity

Reference Material

164852 — *iUP-200A/201A Universal Programmer User's Guide.*
164861 — *iPPS PROM Programming Software User's Guide.*
164853 — *iPPS PROM Programming Software/iUP-200A/201A Universal Programmer Pocket Reference.*

PERSONALITY MODULE SPECIFICATIONS

Memory

EPROM — up to 4K bytes

Physical Characteristics

Width — 5.5 inches (1.4 cm)
Height — 1.6 inches (4.1 cm)
Depth — 7.0 inches (17.8 cm)
Weight — 1 pound (.45 kg)

Electrical Characteristics

Maximum power consumption (module) — 7.5 watts
Maximum power consumption (device) — 2.5 watts
Maximum power consumption (total from iUP) — 10 watts

Environmental Characteristics

Reading temperature — 10°C to 40°C
Programming temperature — 25°C ± 5°
Operating humidity — 10% to 85% relative humidity

Reference Material

Appropriate personality module user's guide:

164376 — *iUP-Fast 27/K Personality Module User's Guide.*
162848 — *IUP-F27/128 Personality Module User's Guide.*
164855 — *iUP-F87/51A Personality Module User's Guide.*
164853 — *iUP-F87/44A Personality Module User's Guide.*

ORDERING INFORMATION

Part number	Description		
iUP-200A	Intel on-line universal programmer	iUP-Fast 27/K*	EPROM personality module
iUP-201A	Intel on-line/off-line universal programmer	iUP-F27/128	EPROM and E^2PROM personality module

Order Number: 210319-003

iUP-F87/51A	Microcontroller personality module
iUP-F87/44A	Peripheral personality module
iUP-200/201 U1 Upgrade Kit	Upgrades an iUP-200/201 universal programmer to an iUP-200A/201A universal programmer
iUP-PAK-A Upgrade Kit	Upgrades an iUP-200A universal programmer to an iUP-201A universal programmer

*The iUP-Fast 27/K personality module can be used only with an iUP-200A/201A universal programmer or an iUP-200/iUP-201 universal programmer upgraded to an A with the iUP-200/201 U1 upgrade kit. If used in an iPDS, this personality module requires version 1.4 or later of the iPPS-iPDS software. All iPDS-140 units shipped after June 1984 will contain this software.

PROM PROGRAMMING PERSONALITY MODULES

MAJOR PERSONALITY MODULE FEATURES:

- **Adapts an iUP-200A/iUP-201A Universal Programmer or Intel Personal Development System (iPDS™) to a family of PROM devices.**
- **Comes ready to use.**
- **Includes the Fast 27/K personality module that programs Intel's latest PROM devices in one tenth the time.**
- **Supports multiple PROM device types.**

Personality modules custom-fit the iUP-200A/iUP-201A Universal Programmer or the iPDS™ system to a family of PROM devices. Each personality module comes ready to use—just plug it into a Universal Programmer or an iPDS system and begin reading or programming parts. The personality modules can be used off-line or controlled from a host or iPDS system using Intel's powerful PROM programming software (iPPS). Selected personality modules support the latest PROM programming features such as the int$_e$ligent Programming™ algorithms (reduce programming time up to a factor of 10), the int$_e$ligent Identifier™ (automatically selects the correct int$_e$ligent Programming algorithm), and the security bit function (protects PROM memory from unauthorized access).

May, 1984
ORDER NUMBER: 280003-001

PERSONALITY MODULE DESCRIPTION

The personality module adapts the universal programmer or the iPDS system to a specific family of PROM devices; it contains all the hardware and firmware necessary to read and program a family of Intel PROMs. The personality module comes ready to use; the user merely inserts the personality module into the universal programmer front panel or the side door of the iPDS chassis. No additional sockets or adapters are required.

As Table 1 shows, each personality module supports a different family of PROM devices.

Each personality module connects to the universal programmer/iPDS system through a 41-pin connector. LEDs on the personality module indicate its operational status. A column of LEDs or a hexadecimal display indicates which PROM device type the user has selected. On some personality modules, an LED below the socket indicates which socket is to be used. A red indicator light tells the user when power is applied to the selected device.

After specifying the PROM device type, the user inserts the PROM to be programmed or read in the socket on the personality module. The personality module checks for correct PROM installation. In addition, each personality module contains diagnostic firmware that performs the following selected PROM tests and indicates status.

- The PROM installation test verifies that the device is installed in the module correctly and that the ZIF socket is closed.
- The PROM blank check determines whether a device is blank. The universal programmer/iPDS system automatically determines whether the blank state is all zeros or all ones.
- The overlay check (performed when a PROM is not blank) determines which bits are programmed, compares those bits with the program to be loaded, and allows programming to continue if they match.

The user can invoke all the PROM device integrity checks except the installation test (which occurs automatically any time an operation is selected).

PROM PROGRAMMERS

The personality modules are used with either the universal programmer or the iPDS system. Both the iUP-200A and iUP-201A models of the universal programmer program PROM devices in on-line mode. The iPPS software which controls on-line programming runs on the host system. The iUP-201A universal programmer adds an additional feature: off-line programming directly from the universal programmer's keyboard. Figure 1 shows an iUP-201A universal programmer with a personality module inserted.

The iPDS system features stand-alone on-line programming controlled by the iPDS-iPPS software which runs on the iPDS system. The iPDS system operates in on-line mode only. Figure 2 shows an iPDS system with a personality module inserted.

Table 2 compares the features of the universal programmer with the features of the iPDS system.

Table 1. Personality Modules

Personality Module	PROM Type	PROMs and ROMs Supported
iUP-Fast 27/K	EPROM	2764, 2764A, 27128, 27256
iUP-F27/128	E²PROM/EPROM	2716, 2732, 2732A, 2764, 27128, 2815, and 2816
iUP-F87/51A	Microcontroller	8748, 8748H, 8048, 8749H, 8048H, 8049, 8049H, 8050H, 8751, 8751H, 8051
iUP-F87/44A	Peripheral	8741A, 8041A, 8742, 8042, 8744H, 8044AH, 8755A

280003-001

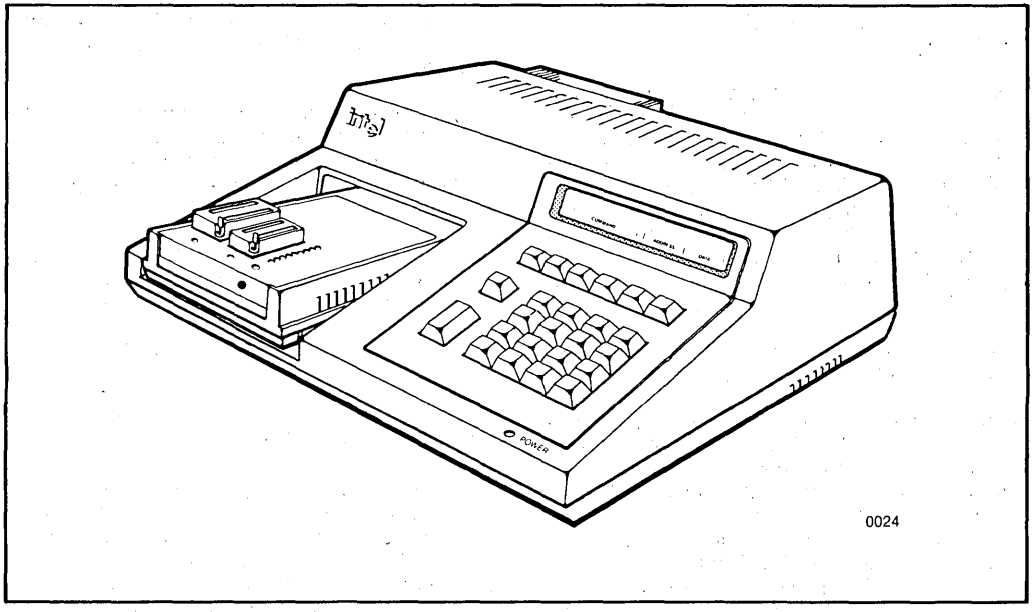


Figure 1. iUP-201A Universal Programmer

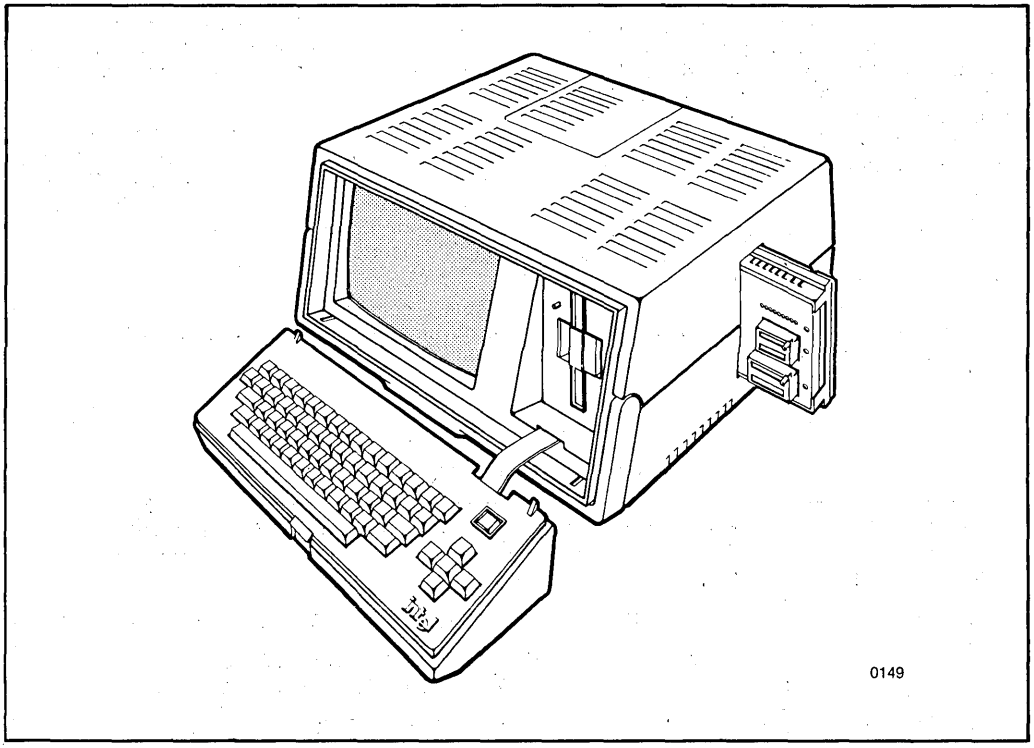


Figure 2. iPDS™ System

280003-001

Table 2. PROM Programmers

Features	iUP-200A Universal Programmer	iUP-201A Universal Programmer	iPDS™ System
Function	PROM programmer	PROM programmer	Development system and PROM programmer
Operating mode	On-line mode	On-line mode and off-line mode	On-line mode
Configuration	Requires host system running iPPS software	Requires host system in on-line mode; stand-alone in off-line mode	Stand-alone plugged into iPDS system
Data display	On CRT of host system terminal	On built-in single-line display in stand-alone mode	On iPDS CRT
Input keyboard	From host system terminal	Built-in keyboard	From iPDS Keyboard

THE iPPS SOFTWARE

The iPPS software, included with both the iUP-200A and iUP-201A models of the universal programmer and with the iPDS system, brings increased flexibility to PROM programming. The iPPS software provides user control through an easy-to-use interactive interface and performs the following functions to make PROM programming quick and easy:

- Reads PROMs and ROMs.
- Programs PROMs directly or from a file.
- Verifies PROM data with buffer data.
- Locks EPROM memory from unauthorized access (on devices which support this feature).
- Prints PROM contents on the network printer (universal programmer only) or the development system printer.
- Performs interactive formatting operations such as interleaving, nibble swapping, bit reversal, and block moves.
- Programs multiple PROMs from the source file, prompting the user to insert new PROMs.
- Uses a buffer to change PROM contents.

With the iPPS software the user can load programs into a PROM from system memory or directly from a disk file. Access to the disk lets the user create and manipulate data in a virtual buffer. This block of data can be formatted into different PROM word sizes for program storage into several different PROM types. In addition, a program stored in the target PROM, the system memory, or a system disk file can be interleaved with a second program and entered into a specific target PROM or PROMs.

The iPPS software supports data manipulation in the following Intel formats: 8080 hexadecimal ASCII, 8080 absolute object, 8086 hexadecimal ASCII, 8086 absolute object, and 80286 absolute object. Addresses and data can be displayed in binary, octal, decimal, or hexadecimal. The user can easily change default data formats as well as number bases.

The user invokes the iPPS software from the ISIS operating system (Intellec 800, Series II, and Series III, ISIS versions V3.4 and later; Series IV, ISIS versions V1.0 and later; iPDS system, ISIS versions 1.4 and later). The software can be run under control of ISIS submit files, thereby freeing the user from repetitious command entry.

Note that the universal programmer and the iPDS system each has its own version of the iPPS software. To distinguish between them, the iPPS software for the iPDS system is called iPPS-iPDS software.

PERSONALITY MODULE FEATURES

The personality modules described in the following sections allow a universal programmer/iPDS system to program a wide range of PROM devices, each with its unique needs and requirements: PROMs, EPROMs, E²PROMs, microcontrollers, and microprocessor peripherals.

280003-001

Note that the user needs one of the following configurations to use the Fast 27/K personality module or to use the security bit function on the iUP-F87/51A and iUP-F87/44A personality modules:

iPDS system

Intel PROM programming software (iPPS-iPDS), version 1.4 or later

iPDS-140 EMV/PROM adapter option

universal programmer

- on-line — Intel PROM programming software (iPPS), version 1.4 or later; model 200A or 201A
- off-line — model 201A

The user can easily update an iUP-200/201 universal programmer to an iUP-200A/201A universal programmer with the iUP-200/201 U1 upgrade kit.

The iUP-Fast 27/K Personality Module

The iUP-Fast 27/K personality module lets the user program, read, and verify the contents of Intel's newest 64K and 256K EPROMs. This personality module supports the int$_e$ligent Programming algorithms and the int$_e$ligent Identifier. The int$_e$ligent Identifier lets the personality module interrogate the PROM device in the program/master socket. It determines whether the type selected matches the type of PROM device installed and then selects the proper int$_e$ligent Programming algorithm. The int$_e$ligent Programming algorithms reduce programming time up to a factor of 10.

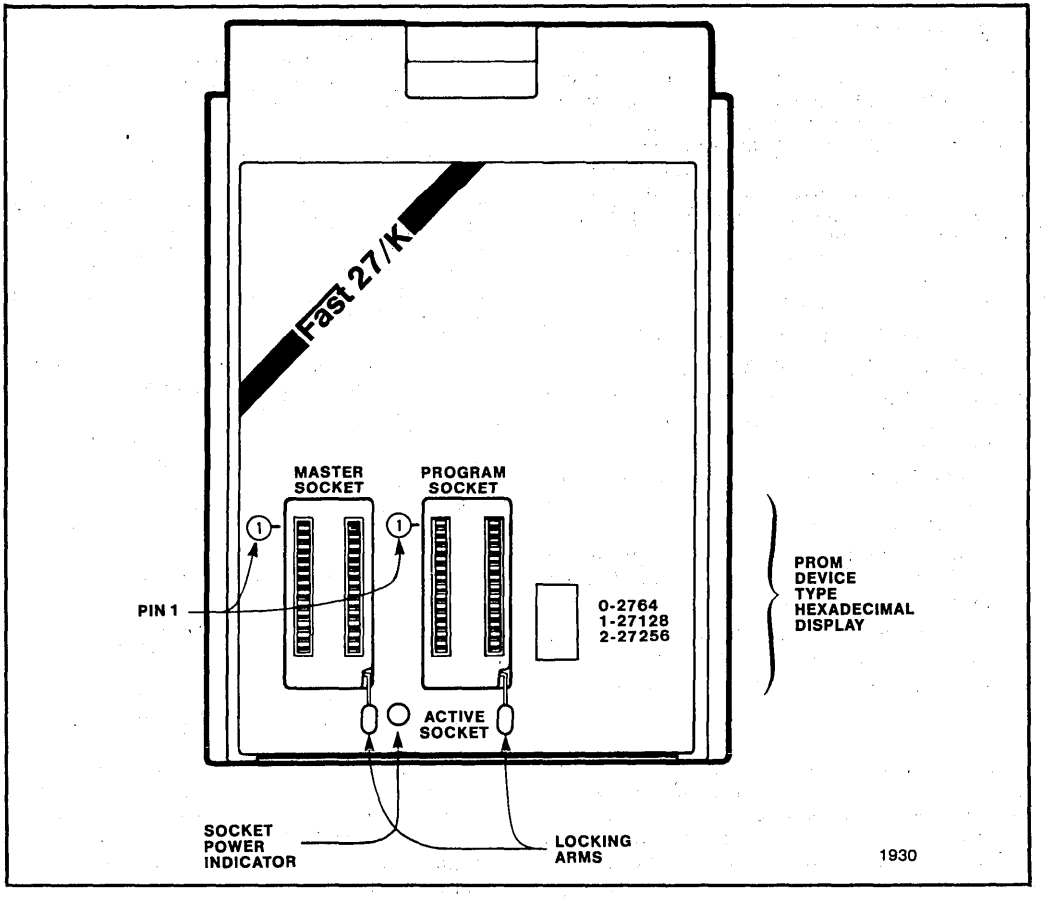


Figure 3. iUP-Fast 27/K Personality Module

280003-001

The iUP-Fast 27K personality module supports the following PROM devices:

2764 2764A 27128 27256

As shown in Figure 3, the iUP-Fast 27/K personality module contains two 28-pin sockets, a hexadecimal display (0 through F), and a red LED that indicates when power is being applied to a socket. The program socket holds the device being programmed. The master socket will be used in future upgrades. The hexadecimal display shows the PROM device type selected.

The iUP-F27/128 Personality Module

The iUP-F27/128 personality module lets the user program, read, and verify the contents of a wide variety of PROM devices, including some of Intel's most popular PROM devices. This personality module supports the following PROM devices:

2716 2732 2732A 2764 27128 2815 2816

As shown in Figure 4, the iUP-F27/128 personality module contains two sockets: one for 24-pin PROM devices and the other for 28-pin PROM devices. The user can use only one socket at a time. An LED below the socket indicates the correct socket to use based on the PROM device type selected, and a row of green LEDs on the right side of the personality module indicate which PROM type is selected. The ACTIVE SOCKET LED indicates when power is being applied to the PROM device and when the universal programmer/iPDS system is accessing the selected socket.

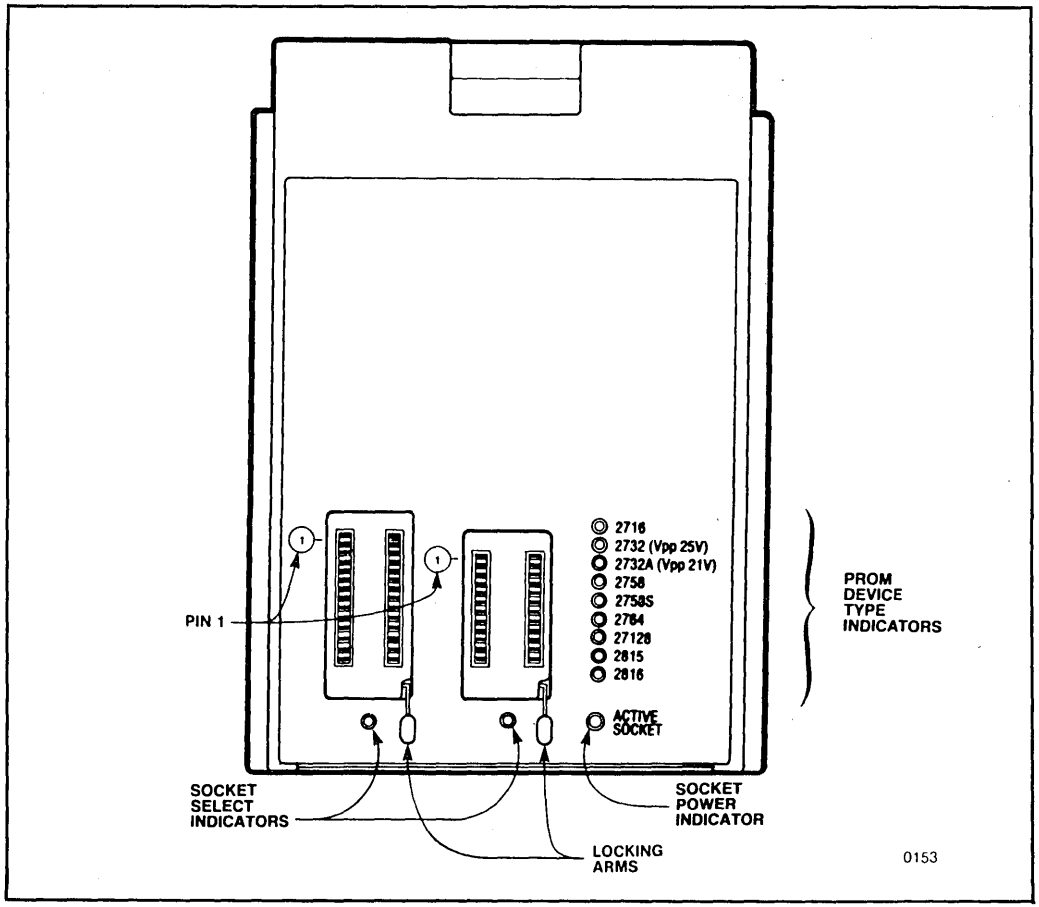


Figure 4. iUP-F27/128 Personality Module

280003-001

The iUP-F87/51A Personality Module

The iUP-F87/51A personality module lets the user program EPROM microcontrollers and read the memory contents of ROM microcontrollers. This personality module supports the security bit function on the 8751H microcontroller. The KEYLOCK command locks the 8751H EPROM memory from unauthorized access by setting the security bit (which cannot be unlocked without erasing the device). As a safety precaution, the KEYLOCK command requires user verification before locking the security bit.

The iUP-F87/51A personality module supports the following PROM devices:

8748	8748H	8048	8048H	8749H
8049	8050H	8751	8751H	8051

As shown in Figure 5, the iUP-F87/51A personality module has two sockets for inserting applicable PROM devices: one for the MCS®-48 family of devices and the other for the MCS-51 family of PROM devices. An LED below the socket indicates the correct socket to use based on the PROM device type selected. One of the green

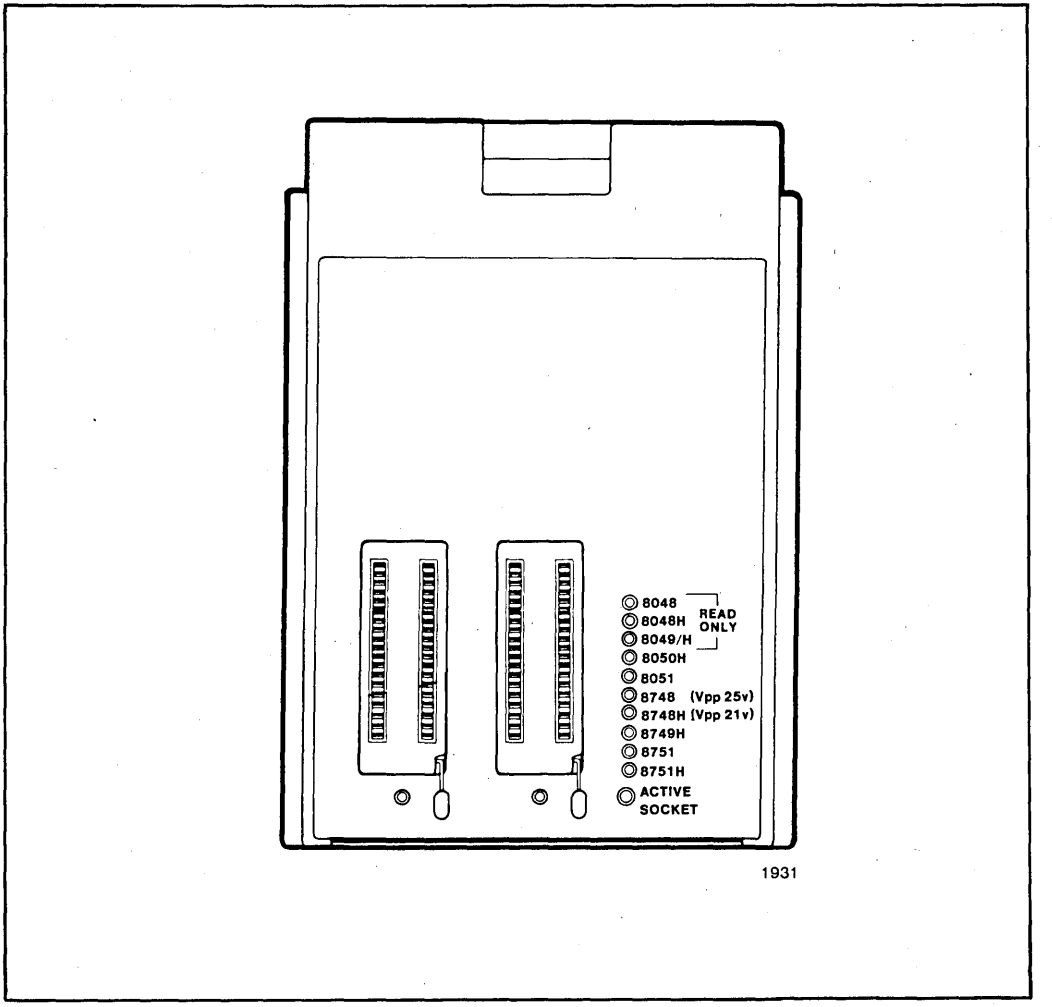


Figure 5. iUP-F87/51A Personality Module

280003-001

LEDs on the right side of the personality module lights to indicate the PROM type selected. The ACTIVE SOCKET LED lights when power is applied to the PROM device and when the universal programmer/iPDS system is accessing the selected socket.

The iUP-F87/44A Personality Module

The iUP-F87/44A personality module lets the user program EPROM versions of the 8044 family of microcontroller/serial interface units and read the memory contents of ROM versions. This personality module supports the security bit function on the 8744H microcontroller. The KEYLOCK command locks the 8744H EPROM memory from unauthorized access by setting the security bit (which cannot be cleared without erasing the device). As a safety precaution, the KEYLOCK command requires user verification before setting the security bit.

The iUP-F87/44A personality module supports the following PROM devices:

8741A	8041A	8742	8042
8744H	8044AH	8755A	

As shown in Figure 6, the iUP-F87/44A personality module has two sockets for inserting applicable PROM devices: one for the 8741A, 8742, and 8755A PROM devices and the other for the 8744H PROM device. An LED below each socket indicates the correct socket to use based on the PROM device type selected. One of the green LEDs on the right side of the personality module lights to indicate the PROM type selected. The

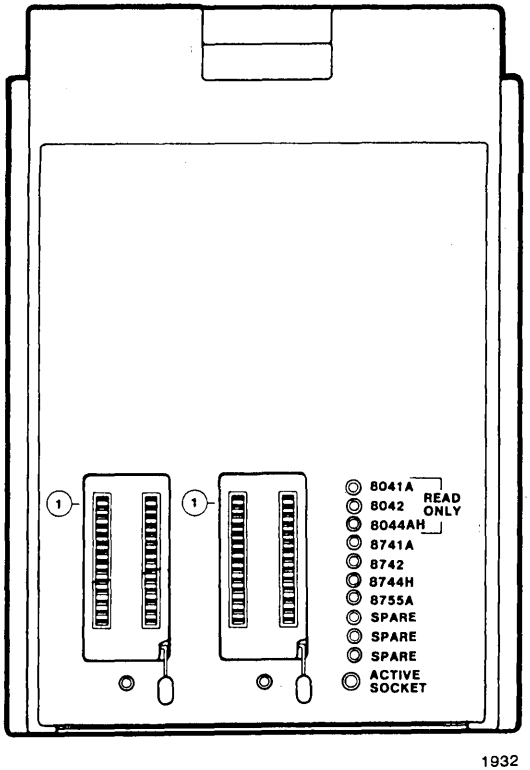


Figure 6. iUP-F87/44A Personality Module

280003-001

ACTIVE SOCKET LED lights when power is applied to the PROM device and when the universal programmer/iPDS system is accessing the selected socket.

PROM PROGRAMMING EXAMPLE

The personality module is the interface that lets the user perform a wide variety of PROM programming, data display, and data editing operations. One of the most popular applications is copying data from a master PROM into a blank PROM. Table 3 outlines and compares the steps for both on-line and off-line copying. Notice the easy-to-use, English-language approach of the iPPS commands, which may be shortened to the first letter for faster entry.

The on-line example assumes that the universal programmer/iPDS system has been powered on and is under control of the ISIS software and that the iPPS software has been initialized. The off-line example assumes that the iUP-201A universal programmer has been powered on and initialized.

PERSONALITY MODULE SPECIFICATIONS

Memory

EPROM — up to 4K bytes

Physical Characteristics

Width — 5.5 inches (1.4 cm)
Height — 1.6 inches (4.1 cm)
Depth — 7.0 inches (17.8 cm)
Weight — 1 pound (.45 kg)

Electrical Characteristics

Maximum power consumption (module) —7.5 watts
Maximum power consumption (device) —2.5 watts
Maximum power consumption (total from PROM programmer) — 10 watts

Environmental Characteristics

Reading temperature 10°C to 40°C
Programming temperature 25°C ± 5°
Operating humidity 10%-85% relative humidity

DOCUMENTATION

Appropriate personality module user's guide:

164376 — *iUP-FAST 27/K Personality Module User's Guide*
162848 — *iUP-F27/128 Personality Module User's Guide*
164855 — *iUP-F87/51A Personality Module User's Guide*
164854 — *iUP-F87/44A Personality Module User's Guide*

Table 3. Typical PROM Programming Sequence

Action	On-line Command	Off-line Function Key
1. Select PROM type.	TYPE	DEVICE SELECT
2. Install the PROM to be copied (the master PROM) in the personality module.		
3. Copy the contents of the master PROM to the buffer.	COPY PROM TO BUFFER	ROM TO RAM
4. Verify that the copy was correct.	VERIFY	VER
5. Remove the master PROM; install a blank PROM		
6. Copy the buffer to the blank PROM.	COPY BUFFER TO PROM	PROG

280003-001

ORDERING INFORMATION

Part number	Description
iUP-Fast 27/K*	EPROM personality module
iUP-F27/128	EPROM and E^2PROM personality module
iUP-F87/51A*	Microcontroller personality module
iUP-F87/44A*	Peripheral personality module

*The iUP-Fast 27/K personality module and the security bit function on the iUP-F87/51A and iUP-F87/44A personality modules can be used with an iUP-200A/201A universal programmer; or an iUP-200/iUP-201 universal programmer upgraded to an A with the iUP-200/201 U1 upgrade kit; or an iPDS system, using version 1.4 or later of the iPPS-iPDS software (iPDS-140 units shipped after June 1984 contain this software).

intel®

APPLICATION NOTE

AP-179

May 1984

PROM Programming With the Intel Personal Development System (iPDS™)

FRED MOSEDALE
DSHO TECHNICAL PUBLICATIONS

Order Number 280015-001

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:

BXP, CREDIT, i, ICE, I^2ICE, ICS, iDBP, iDIS, iLBX, i_m, iMMX, Insite, INTEL, int_el, Intelevision, Intellec, $int_eligent$ Identifier™, int_elBOS, $int_eligent$ Programming™, Intellink, iOSP, iPDS, iRMS, iSBC, iSBX, iSDM, iSXM, Library Manager, MCS, Megachassis, Micromainframe, MULTIBUS, Multichannel™ Plug-A-Bubble, MULTIMODULE, PROMPT, Ripplemode, RMX/80, RUPI, System 2000, and UPI, and the combination of ICE, iCS, iRMX, iSBC, 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.

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

Intel Corporation
Literature Department
3065 Bowers Avenue
Santa Clara, CA 95051

INTRODUCTION

Programmable read-only memory (PROM) devices play a significant role in microprocessor-based products. How can PROM programming devices perform to best serve the needs of those who develop and service such products?

This application note first provides a general answer to this question; then, it proceeds to describe the features and use of PROM programming hardware and software available for the Intel Personal Development System (iPDS™). The description explains how the iPDS system provides the wide range of capabilities needed by those who use PROM programming devices. The description also highlights the iPDS system's ability to program some of Intel's newest EPROMs.

PROMS IN THE LIFE CYCLE OF A PRODUCT

Memory Options: A Review

Before PROM programming needs are discussed, it is important to briefly review memory options available to designers.

Microprocessor-based products need memory to store instructions and data used in controlling their operations. In order to maximize product operating speeds, designers must use memory that can be accessed quickly. Both random access memory (RAM) devices and read-only memory (ROM) devices offer designers quick access, but RAM devices are volatile—their contents are erased when system power is turned off. ROM devices are nonvolatile; thus, designers use ROMs to store programs and data that will not change during the operation of the product.

After a product's program code data has been debugged, you can transfer the code to programmable ROMs (PROMs) or masked ROMs. (The most flexible PROMs are E²PROMs and EPROMs; E²PROMs are electrically erasable and EPROMs can be erased by ultraviolet light.) PROMs are programmed using a relatively simple procedure; by contrast, masked ROMs can only be programmed in a manufacturing environment. Thus, masked ROMs provide less flexibility but are used because they may be more cost-effective in large volumes. However, because the price of PROMs is falling and because inventories with erasable PROMs can be reprogrammed when product programs are changed, erasable PROMs are also attractive for large volumes.

Desirable PROM Programming Features

Once your product's software is debugged, you can load the software into the product's PROMs (so that it becomes the product's firmware). However, usually in the development of a product, the initial programming of PROM devices is not the last operation involving the PROMs. Even if the software is debugged, once it is loaded into the PROMs, you may discover new bugs in the program that you failed to detect before the program was committed to PROMs. So, there may soon be a need to erase the PROMs (if they are EPROMs or E²PROMs) and reprogram them.

During product development and servicing, you will also sometimes need to accomplish the following tasks:

- Check the contents of a PROM.
- Use one PROM to program other PROMs that will be used in other prototype systems.
- Update earlier firmware versions with later versions.

Consider in more detail the (P)ROM-related needs that can arise for you during the product's life cycle, that is, between the time when the product's software has been loaded into PROMs and the time when the product is phased out. There are two basic sets of needs, those having to do with displaying (P)ROM contents and those having to do with programming PROMS.

DISPLAYING AND PRINTING NEEDS

For a variety of reasons, you may need to examine what is stored in a (P)ROM. For example, you may suspect that a (P)ROM's program is in error; or you may have incomplete documentation on what was programmed into a (P)ROM. Thus, you will want to be able to display the contents on a video display terminal and to have a printer print out the contents. You will want to be able to choose the display base (binary, octal, decimal, or hexadecimal) and whether to display the contents as ASCII characters.

PROGRAMMING NEEDS

When you program PROMs, you need a programming device that can program a variety or PROMS and one that offers flexibility and ease of programming. The following list describes PROM programming needs.

- **Need for simple operation** — You want a programming device that satisfies all of the following needs and is simple to operate. You do not want to have to refer to a manual every time you wish to program a PROM.

- **Need to program a wide variety of PROMs** — For greatest flexibility, you want a programming device that can program the various kinds of PROMs that are available. For example, you will want to be able to program microcontrollers with EPROMs, and you will want to be able to program from the small inexpensive 16K and 32K PROMS to the latest 256K PROMS with int$_e$ligent Programming™ algorithms to speed programming. You will also want to be able to program those PROMs that use the new lower programming voltage (12.5 V).

- **Need to be upgradeable** — A PROM programming device should be designed so that it can be upgraded to program PROMs that will be available in the future. Without upgradeability, the device will soon be out-of-date.

- **Need to check PROM contents before programming** — If the PROM you will be programming is blank, it can of course be programmed. Even if it has some bits set at the time of programming, if the same bits must also be set for the program, the PROM can be used. Thus, ideally, a PROM programming device will determine whether the PROM is blank, and if not, determine whether the bits already set are compatible with the program to be loaded into the PROM.

- **Need to recognize file formats** — When a PROM is programmed using an object file generated by a compiler or assembler, the PROM programming device must be able to extract the data that is to be loaded into the PROM from the larger file structure. For greatest flexibility, you will need a PROM programming device that recognizes the file structures generated by compilers and assemblers that you will use when developing program code for the PROMs.

- **Need to support a variety of source program options** — There are three sources you may wish to use for PROM data: an already-programmed PROM, a software development system, or a disk. For greatest programming flexibility, you will want a PROM programming device that makes all three kinds of sources available.

- **Need to support data manipulation and modification** — You may wish to modify a source file for your PROM program. For example, you may discover an error in the source file, or you may realize that for your new processor system, the PROM data must first be 2's complemented. For a variety of reasons, your PROM programming will be much more flexible if the PROM programming device offers a buffer for temporary storage and PROM programming software that can manipulate the source program data in a variety of ways.

- **Need for variety in loading program code into PROMs** — If your product has a 16-bit microprocessor and you are using 8-bit PROMs to store the firmware, you will need to interleave the 16-bit code between two 8-bit PROMs. A PROM programming device with interleaving capability will speed such programming. You may want other kinds of flexibility when programming.

- **Need to transfer long programs that will not fit into one PROM** — Long programs may exceed the storage capacity of the PROMs chosen for your product. You need a programming device that can format the program so that it can be stored in successive PROMs.

- **Need to verify programming** — When programming is finished, you will want to check that the PROM is indeed correctly programmed. A defect in the PROM could corrupt the intended firmware. Checking would involve comparing the source with the programmed PROM.

- **Need to compare buffer with PROM** — If you are interrupted when programming PROMs or if you have not labeled PROMs that you did program, you may forget whether a particular PROM was programmed. In such cases, you will want to compare the particular PROM with the program stored in the buffer of the programming device. Comparison will prevent you from having to reprogram an already-programmed PROM.

- **Need to lock microcontrollers from unauthorized access** — Some advanced microcontrollers can be locked to prevent unauthorized access. To take advantage of this security feature, you need to be able to control the locking of the microcontrollers.

- **Need to automate routine PROM programming tasks** — To speed programming, you will want a programming device that can automate routine programming functions. Automation will not only speed programming, it will also release personnel for other work.

DESIRABLE PROM PROGRAMMING FEATURES: A SUMMARY

In summary, if PROMs (and ROMs) are incorporated in your product, you will have the greatest flexibility if your PROM programming device has the following features. (Of course, for ROMs only reading tasks are needed.)

- Is easy-to-use.
- Can program a wide variety of PROMs.
- Is upgradeable.
- Can display (P)ROM contents in ASCII characters and in a variety of bases.
- Can enable a printer to print out (P)ROM contents in ASCII characters and in a variety of bases.
- Can check for blank PROMs.
- If PROM is not blank, can check PROM contents for compatibility with program.
- Can recognize file formats of your development system object files.
- Supports transfers of program code from development systems, disks, and other PROMs to the new PROM.
- Provides temporary storage and software for manipulating Programming data before loading it into the PROM.
- Supports variety in how data is loaded into PROMs, e.g.:
  - Interleaving 16-bit data into 8-bit PROMs
  - Segmenting long programs so that resulting program segments fit into successive PROMs
- Can verify the accuracy of copying.
- Can compare programming buffer with PROM contents.
- Can lock microcontrollers from unauthorized access.
- Can automate routine PROM programming tasks.

The Intel Personal Development System (iPDS) with the PROM programming option meets these needs. The following sections describe the iPDS PROM programming hardware and software and show how this system can perform all of these tasks for a variety of PROMs.

THE iPDS™ PROM PROGRAMMING SYSTEM

The iPDS system supports integrated hardware and software development; it provides a complete set of software development tools and in-circuit emulators for hardware debugging and hardware-software integration. With its optional PROM programming hardware and software, the iPDS system also supports PROM programming.

Three components comprise the iPDS PROM programming system: the iPDS system (with the ISIS-PDS operating system and the plug-in module adapter board), the PROM programming modules, and the PROM programming software. Each of these components is described briefly in the following sections.

iPDS™ System

To perform PROM programming tasks, the iPDS system must use its ISIS-PDS operating system and the plug-in module adapter board. The PROM programming software (iPPS-PDS) runs under the ISIS-PDS operating system.

The adapter board allows you to use both the PROM programming personality modules and emulation modules. It provides the interface between the modules and the iPDS system.

Figure 1 shows the iPDS system with a PROM programming personality module plugged into its side.

280015-001

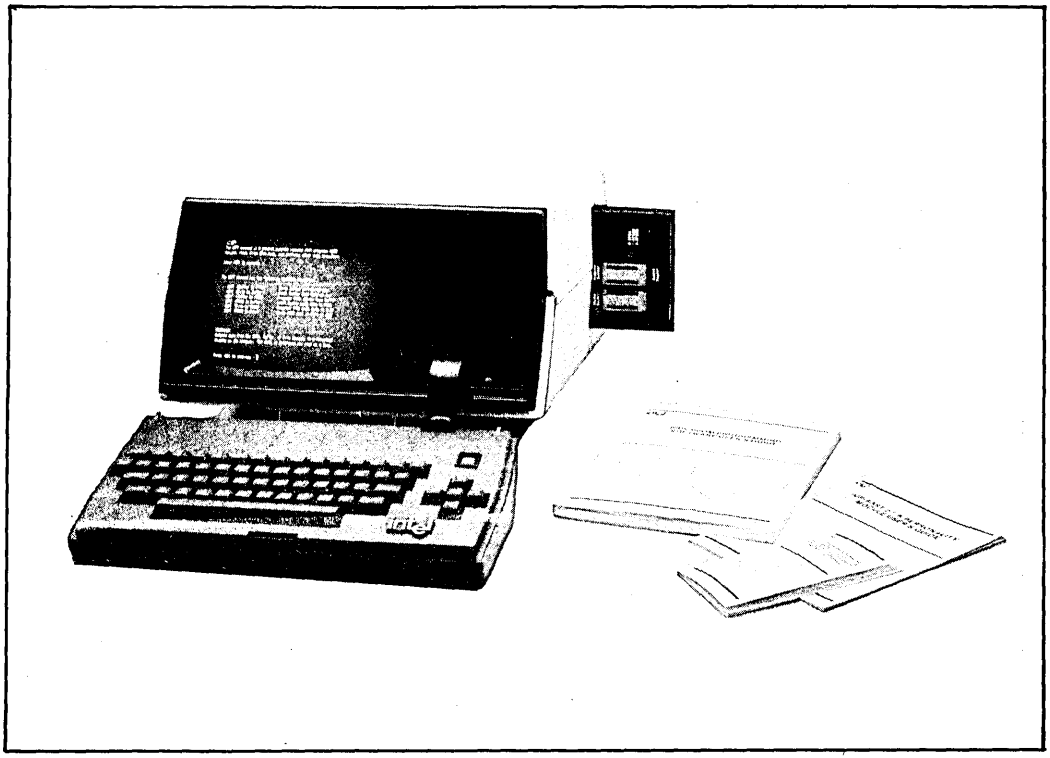


Figure 1 iPDS™ System with PROM Programming Personality Module

PROM Programming Personality Modules

A personality module is the interface between the iPDS system and a selected PROM. Personality modules contain all the hardware and firmware for reading and programming a family of Intel devices. Each personality module is a single molded unit inserted into the side panel of the iPDS unit. No additional adapters or sockets are needed. Table 1 lists the available personality modules, and Figure 2 shows the four modules.

Table 1 PROM Programming Personality Modules

PERSONALITY MODULE	PROM TYPE PROGRAMMED	PROMs AND ROMs SUPPORTED
iUP-Fast 27/K	EPROM	2764, 2764A, 27128, 27256, and provisions for future PROMs
iUP-F27/128	E^2/EPROM	2716, 2732, 2732A, 2764, 27128, 2815, and 2816
iUP-F87/51A	Microcontroller	8748, 8748H, 8048, 8749H, 8048H, 8049, 8049H, 8050H, 8751, 8751H, 8051
iUP-F87/44A	Peripheral	8741A, 8041A, 8742, 8042, 8744H, 8044AH, 8755A

Figure 2 PROM Programming Personality Modules

Each personality module connects to the iPDS system through a 41-pin connector. Module firmware is uploaded into the iPDS system and executed by the iPDS system. The personality module firmware contains routines needed to read and program a family of PROMs. In addition, the personality module sends specific information about the selected PROM to the iPDS system, such as information about the PROM size and its blank state.

LEDs on each personality module indicate its operational status. On some personality modules a column of LEDs or a hexadecimal display indicates which PROM device type the user has selected. On some personality modules with more than one socket, an LED below each socket indicates the socket to be used. In addition, a red indicator light tells the user when power is being supplied to the selected device.

The personality module firmware performs selected PROM tests and indicates status:

- The PROM installation test verifies that the device is installed in the module correctly and that the ZIF socket is closed.
- The PROM blank check determines whether the device is blank. The iPDS system automatically determines whether the blank state for the particular device is defined as all zeros or all ones.
- The overlay check (performed when a PROM is not blank) determines which bits are programmed, compares those bits against the program to be loaded, and allows programming to continue if they match.

Easy-to-read status messages are also provided. The user can invoke all of the PROM device integrity checks except the installation test (which occurs automatically any time an operation is selected). The following sections describe specific features of the three personality modules that program the newer Intel PROMs.

280015-001

iUP-F87/44A AND iUP-F87/51A PERSONALITY MODULES: SPECIAL FEATURE

Each of these personality modules supports the security bit function on one member of the microcontroller family it can program. The iUP-F87/44A module supports the function on the 8744H microcontroller, and the iUP-F87/51A supports the function on the 8751H microcontroller. The KEYLOCK command locks the 8744H (or the 8751H) EPROM memory from unauthorized access by setting the security bit; the microcontroller cannot be unlocked without erasing the EPROM. As a safety precaution, the KEYLOCK command requires user verification before it sets the security bit.

iUP-FAST 27/K PERSONALITY MODULE: SPECIAL FEATURES

The iUP-Fast 27/K personality module supports the int$_e$ligent Identifier™ and the int$_e$ligent Programming algorithms. The int$_e$ligent Identifier is used to check the PROM installed in the personality module socket to determine whether it matches the type selected; then the int$_e$ligent Identifier is used to select the proper int$_e$ligent Programming algorithm. The int$_e$ligent Programming algorithms reduce PROM programming time by as much as a factor of ten. This module has provision for support of future EPROMS and E^{2}PROMs via simple plug-in updates.

The int$_e$ligent Programming™ Algorithm

Using the capabilities of the iPDS PROM programming equipment and employing a new kind of algorithm that recognizes differences among EPROM cells, you can dramatically reduce programming time for the newest high-density EPROMs. As a bonus, the technique helps ensure that EPROMs receive adequate programming — in terms of memory-cell charge — to maintain long-term reliability.

Reducing programming time and costs for EPROMs has become increasingly important because the chips have become a cost-effective, easy-to-use alternative to masked ROM in high-volume applications requiring code flexibility or simplified inventory — a major switch from EPROMs' original small-volume prototyping applications. And, volume usage makes EPROM programming a significant manufacturing consideration.

The conventional programming procedure for most EPROMs uses a nominal 50-msec pulse per EPROM byte, resulting in a total programming time of approximately 1.5 minutes for a 16K-bit chip. With the introduction of the 2764 (64K bits) and devices with even higher densities, however, programming times have increased. A 256K-bit EPROM, for example, requires 24 minutes for programming using the conventional programming method.

Most EPROM cells program in less than 45 msec, however. In fact, empirical data shows that very few cells require longer than 8 msec for programming. Therefore, a procedure that takes into account the characteristics of individual EPROM cells can significantly reduce a device's programming time.

Arbitrarily reducing programming time is risky, however, because a cell's ability to achieve and maintain its programmed state is a function of this time. What is needed, therefore, is a way to verify the level to which individual cells have been programmed. Such a way exists. By determining the charge stored in a cell compared to the minimum charge needed to program the cell to a detectable level, you can check for a program margin that ensures reliable EPROM operation.

Margin checking does not occur in conventional EPROM programming, however. Instead, each EPROM cell receives a 45- to 55-msec write pulse, and manufacturers attempt to ensure program margin by screening out EPROMs having bytes that do not program within 45 msec. This programming procedure is thus an open loop — no actual verification of margin occurs.

By contrast, the int$_e$ligent Programming algorithm guarantees reliability through the closed-loop technique of margin checking. This algorithm uses two different pulse types: initial and over-program. The algorithm first applies a 1-msec initial pulse to an EPROM. After the pulse, it checks the EPROM's output for the desired programmed value. If the output is incorrect, the algorithm repeats the pulse-and-check operation. When the output is correct, the algorithm supplies an over-program pulse; the length of this pulse depends on how many initial pulses were used and varies with the EPROM being programmed. This longer pulse helps ensure that the EPROM cell has an adequate programming margin for reliable operation.

280015-001

Prom Programming Software (iPPS-PDS)

The iPPS-PDS software provides easy-to-use commands that allow you to load programs into a target PROM from another PROM, from iPDS system memory, or directly from a disk file.

The iPPS-PDS software also supports data manipulation in the following Intel formats: 8080 hexadecimal ASCII, 8080 absolute object, 8086 hexadecimal ASCII, 8086 absolute object, and 286 absolute object. Addresses and data can be displayed in binary, octal, decimal, or hexadecimal. You can easily change default data formats as well as number bases.

You invoke the iPPS-PDS software from the ISIS operating system. (The software can be run under control of ISIS submit files, thereby freeing you from repetitious command entry.)

An explanation of the iPPS-PDS software follows. It is divided into three main sections: the iPPS-PDS storage devices, iPPS-PDS commands, and invoking the iPPS-PDS. Also see the Appendix for iPDS PROM programming examples.

iPPS-PDS STORAGE DEVICES

The iPPS-PDS software transfers data between any two of the three storage devices: PROM, buffer, and file. These devices are defined in the following three sections.

PROM Device

The PROM device is plugged into a socket on the personality module installed in the iPDS system. The iPPS-PDS software does not recognize the PROM device until you enter the TYPE command. The TYPE command automatically sets the appropriate buffer size according to the size of the PROM device specified.

Buffer Device

The buffer device is a section of development system memory that the iPPS-PDS software allocates and uses as a working area for temporary storage and for rearranging data. Its boundaries can exist anywhere in a virtual address range from 0 to 16777215 (0 to 2^{24}-1).

When the iPPS-PDS software is initialized, the buffer starting address is set to 0 and the buffer ending address is set to 8K−1, providing an initial buffer size of 8K bytes (the default buffer size when no PROM type is specified). During subsequent iPPS-PDS operations, the size and boundaries can vary. Specific iPPS-PDS commands determine these variations. The most recent command that changed the lower boundary of the buffer determines the buffer starting address. The TYPE command affects both the size and location of the buffer. For example, the TYPE command always resets the buffer start address to 0. The most recent TYPE command controls the size of the buffer.

The iPDS system needs a virtual buffer when PROM size exceeds 8K. If the PROM size exceeds the 8K memory buffer space available on the development system, the iPPS-PDS software creates a virtual buffer area using temporary file space on disk.

Two temporary work files are used to create the virtual buffer. During subsequent virtual buffer operations, the iPPS-PDS software automatically swaps data in and out of development system memory from and to work files.

File Device

The file device is an ISIS file on a disk. It is specified within iPPS-PDS commands.

The data stored in the disk file is in one of the following Intel absolute formats: 8080 hexadecimal, 8080 object, 8086 hexadecimal, 8086 object, or 80286 object. The iPPS-PDS software can read any of these formats as input but writes data to a file in 8080 object, 8086 object, or 80286 object formats only. Basically, these files contain representations of blocks of memory data. Included with the data are addresses for the locations of the data. The data blocks are not necessarily in consecutive address order. The method used to create the file determines the order of the data.

The iPPS-PDS file device has address boundaries that exist in the virtual range from 0 to 16777215 (0 to 2^{24}-1). These boundaries are determined as follows:

- The file's lowest address is the lowest address encountered while reading the file.
- The file's highest address is the highest address encountered while reading the file.

If the iPPS-PDS software creates the file (that is, if the file is a destination device in an iPPS-PDS command), the specific command issued determines these boundaries.

When you specify a particular address range to be read from a file, all sections in the address range that are not present in the file are written in a PROM destination device as the blank state of the currently selected PROM type. If the destination device is the buffer, the nonexistent sections in the file do not overwrite the corresponding sections in the buffer.

During the operation of commands that use the file device as a source, the iPPS-PDS software only reads the actual data from the file and ignores any other information in the file. For example, the file can contain special information used later for debugging. Since the iPPS-PDS software ignores this information, it will not appear in any new files generated. If the data is written back to the original file, the original file is deleted.

iPPS-PDS COMMANDS

Each iPPS-PDS command consists of a keyword that identifies the command, followed by other keywords and associated parameters that are the arguments of the command. You enter all iPPS-PDS commands, as well as program address and data information, through the development system ASCII keyboard; the commands are displayed on the system CRT. Table 2 summarizes the iPPS-PDS commands.

Table 2 iPPS-PDS Command Summary

COMMAND	DESCRIPTION
PROGRAM CONTROL GROUP	**CONTROLS EXECUTION OF THE iPPS-PDS SOFTWARE.**
EXIT	Exits the iPPS software and returns control to the ISIS operating system.
<ESC>	Terminates the current command.
REPEAT	Repeats the previous command.
ALTER	Edits and re-executes the previous command.
UTILITY GROUP	**DISPLAYS USER INFORMATION AND STATUS; SETS DEFAULT VALUES.**
DISPLAY	Displays PROM, buffer, or file data on the console.
PRINT	Prints PROM, buffer, or file data on the local printer.
HELP	Displays user assistance information.
MAP	Displays buffer structure and status.
BLANKCHECK	Checks for unprogrammed PROMs.
OVERLAY	Checks whether non-blank PROMs can be programmed.
TYPE	Selects the PROM type.
INITIALIZE	Initializes default number base and file type.
WORKFILES	Specifies the drive device for temporary work files.
BUFFER GROUP	**EDITS, MODIFIES, AND VERIFIES DATA IN BUFFER.**
SUBSTITUTE	Examines and modifies buffer data.
LOADDATA	Loads a section of buffer with a constant.
VERIFY	Verifies data in the PROM with buffer data.
FORMATTING GROUP	**REARRANGES DATA FROM PROM, BUFFER, OR FILE.**
FORMAT	Formats and interleaves buffer, PROM, or file data.
COPY GROUP	**COPIES DATA FROM ONE DEVICE TO ANOTHER.**
COPY (file to PROM)	Programs PROM with data in a file on disk.
COPY (PROM to file)	Saves PROM data in a file on disk.
COPY (buffer to PROM)	Programs PROM with data from the buffer.
COPY (PROM to buffer)	Loads the buffer with data in the PROM.
COPY (buffer to file)	Saves the contents of buffer in a file on disk.
COPY (file to buffer)	Loads the buffer from a file on disk.
SECURITY GROUP	**LOCKS SELECTED DEVICES; PREVENTS UNAUTHORIZED ACCESS.**
KEYLOCK	Locks the PROM from unauthorized access.

280015-001

Once entered, a command line is verified for correct syntax and executed. If a syntax error is detected, the following error message is displayed:

--SYNTAX ERROR--*specific error.*

If you omit a required keyword, the iPPS-PDS software prompts for the keyword and its associated parameters. If the keyword is entered but its parameters are omitted, either a default value is assumed or an error message is displayed if there is no default. In certain commands, default keywords are also assumed.

You can enter complete iPPS-PDS keywords or any unique abbreviation (only the first character is required). For example, command keywords of C, CO, COP, and COPY are all interpreted as the COPY command.

The iPPS-PDS software accepts numeric entries in any one of four number bases: binary (Y), octal (O or Q), decimal (T), or hexadecimal (H). Numbers can be entered in any of these bases by appending the appropriate letter identifier to specify the base (e.g., 11111111Y, 377Q, 255T, FFH). An explicit number base identifier overrides the default number base, which is initially hexadecimal.

INVOKING iPPS

There are two methods of invoking the iPPS-PDS software: command lines and submit files.

The command line for invoking the iPPS-PDS software (under V1.0 and later versions of the ISIS.PDS operating system) uses the following syntax:

[:F*n*:]IPPS

The symbol ":F*n*:" Specifies the drive on which the iPPS-PDS files are located. When you enter the iPPS-PDS command, the ISIS operating system loads and executes the iPPS-PDS software.

The iPPS-PDS software can also run under the control of a submit file. SUBMIT is an ISIS command that allows you to use a disk text file as input for further ISIS commands or as command inputs to utilities running under the ISIS operating system. Thus, a submit file can contain the ISIS command line to invoke the iPPS-PDS software and then a sequence of commands for the iPPS-PDS software itself.

Summary: The iPDS™ System Meets PROM Programming Needs

Table 3 describes briefly how the iPDS system meets each of the needs identified earlier in this application note.

The iPDS system can be a complete intelligent PROM programmer — and, because the iPDS system is also a development system, it can provide an excellent means to off-load PROM programming from your current development system (just as the iPDS system allows you to off-load other 8-bit development tasks). In addition, with its state-of-the-art PROM programming capability, the iPDS system becomes an attractive solution to your complete development system needs.

Table 3 iPDS™ Features Meet PROM Programming Needs

NEED	iPDS™ FEATURE
Be easy-to-use.	iPPS software and the PROM programming personality modules were designed to provide ease-of-use.
Program a wide variety of PROMs.	Personality modules each permit the programming of a family of PROMs or microcontrollers.
Be upgradeable.	New personality modules will be released as new PROM families appear.
Display (P)ROM contents in ASCII characters or in a variety of bases.	iPPS DISPLAY command displays (P)ROM (or buffer or file) contents in ASCII characters and in binary, octal, decimal, or hexadecimal.
Enable a printer to print out (P)ROM contents in ASCII characters and in a variety of bases.	iPPS PRINT command prints out (P)ROM (or file or buffer) contents in ASCII characters and in binary, octal, decimal, or hexadecimal.
Check for blank PROMs.	iPPS BLANKCHECK command checks for blank PROMS.
If PROM is not blank, check PROM contents for compatibility with program.	iPPS OVERLAY command checks PROM contents for compatibility with program.
Recognize file formats of development system object files.	iPPS command file switch allows you to indicate to the iPDS system which object file format is being used.
Support transfers of program code from development system, disks, and other PROMs to the new PROM.	iPPS COPY commands allow you to copy in either direction between the iPDS disk drive(s), PROMs, and the iPDS buffer storage.
Provide temporary storage and software for manipulating programming data before loading it into the PROM.	iPDS buffer provides temporary storage and the iPPS SUBSTITUTE and LOADDATA commands allow you to manipulate programming data before you load it into a PROM.
Load data into PROMs in a variety of formats, e.g.: — interleaving 16-bit data into two 8-bit PROMs — segmenting long programs so that resulting program segments fit into successive PROMs	iPPS FORMAT command allows you to format data in a variety of ways so that it can be loaded into PROMs in various sequences (including interleaving and segmenting).
Verify the accuracy of copying.	iPPS software automatically checks the accuracy of copying.
Compare programming buffer with PROM contents	iPPS VERIFY command compares buffer data with PROM data.
Control the security feature of advanced microcontrollers for unauthorized access.	iPPS KEYLOCK command locks advanced microcontrollers.
Automate routine PROM programming tasks.	ISIS SUBMIT files permit you to store frequently used command sequences. The files can then be activated with a single command.

280015-001

APPENDIX: PROM PROGRAMMING EXAMPLES

APPENDIX: PROM PROGRAMMING EXAMPLES

Displaying (P)ROM contents and programming PROMs are easy tasks with the iPDS system. The following four examples show typical uses of the iPDS system's PROM programming capabilities:

- Examining the contents of a masked ROM
- Duplicating a PROM
- Interleaving a file between two PROMs
- Locking a microcontroller

EXAMPLES

The examples assume that the iPDS system is under control of the iPPS-PDS software. The boldface characters shown on the iPDS screen displays indicate user entries. The key-in sequence below each screen display gives the actual entries that you must key in to obtain the screen display.

Examining the Contents of a Masked ROM

The DISPLAY command lets you examine the contents of a PROM or a masked ROM.

```
PPS> DISPLAY PROM
000000: C3 40 00 20 20 44 20 2D 20 44 49 53 4B 00 20 20  .@. D - DISK.
000010: 47 20 2D 20 47 45 4E 45 52 41 4C 00 20 20 4B 20  G - GNENERAL.  K
000020: 2D 20 4B 45 59 42 4F 41 52 44 2F 43 52 54 00 FF  - KEYBOARD/CRT..
000030: FF FF FF FF FF FF FF FF C3 36 1C FF FF FF FF FF  ........6......
000040: F3 DB 80 E6 20 CA 03 08 3E 00 D3 D1 DB 80 E6 01  .... ...>.......
000050: C2 66 00 3E 4F D3 D0 3E 58 D3 D0 3E 89 D3 D0 3E  .f.>O..>X..>...>
000060: 99 D3 D0 C3 76 00 3E 4F D3 D0 3E 98 D3 D0 3E 8A  ....v.>O..>...>.
000070: D3 D0 3E 9C D3 D0 21 00 00 11 00 08 AF 47 7B B2  ..>...!......G{.
000080: CA 8A 00 78 86 23 18 C3 7D 00 78 FE 55 C2 8D 00  ...x.#..}.x.U...
000090: 3E 34 D3 E3 3E 1F D3 E0 3E 00 D3 E0 01 30 00 DB  >4..>...>....0..
0000A0: 80 E6 01 C2 A9 00 01 2C 00 3E 72 D3 E3 79 D3 E1  .......,.>r..y..
0000B0: 78 D3 E1 3E B2 D3 E3 3E 00 D3 E2 3E 16 D3 E2 D3  x..>...>...>....
0000C0: 10 3E 22 D3 60 D3 50 DB 80 E6 04 CA C7 00 DB 80  .>".`.P.........
0000D0: E6 04 C2 CE 00 AF D3 F0 D3 F0 D3 F0 D3 F1 3E A1  ..............>.
0000E0: D3 F8 3E 23 D3 60 3E C8 D3 E2 3E 00 D3 E2 D3 50  ..>#.`>...>....P
0000F0: 21 EF 00 2B 7C B5 C2 F3 00 DB 80 E6 04 C2 FD 00  !..+|...........
000100: 3E 00 D3 E2 3E 16 D3 E2 D3 50 DB 80 E6 04 CA 0A  >...>....P......
000110: 10 DB 80 E6 04 C2 11 01 3E 22 D3 60 D3 50 DB 80  ........>".`.P..
000120: E6 04 CA 1E 01 DB 80 E6 04 C2 25 01 21 00 40 11  ..........%.!.@.
ENTER <CR> TO CONTINUE $
ABORTED
PPS>
```

Key-in Sequence

DISPLAY PROM

RETURN

ESC

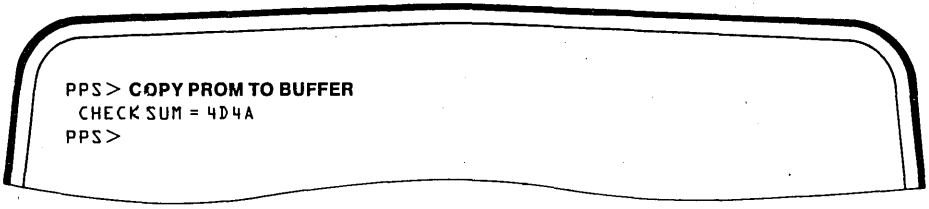
Comments

This example shows the data in the PROM in hexadecimal format, which is the default base in this example. Press the ESC key at any time to end the display. The "S" sign is the echo of the ESC key. You can also display the data in other number bases. Note the ASCII code displayed in the far right column.

Duplicating a PROM

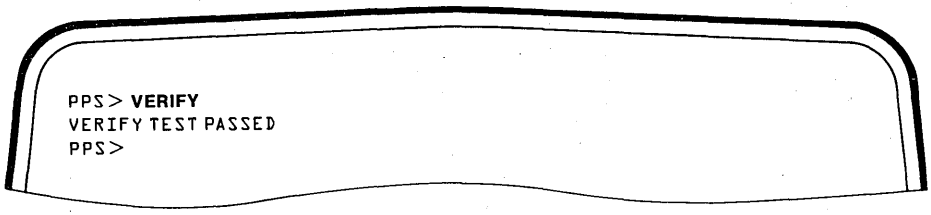
One frequently used application of iPDS PROM programming is copying data from a PROM into a buffer or file, then copying it into another PROM. You can perform this operation using the iPPS-PDS buffer (or an iPDS file for intermediate storage) and the iPPS-PDS COPY commands.

The following example illustrates a direct PROM-to-buffer-to-PROM duplication. If you wish to perform these examples, place the PROM in the PROM socket and reset the iPPS-PDS (using the TYPE command for your type of PROM). A 2716 EPROM that contains sample code is used in this example.



Key-in Sequence	Comments
COPY PROM TO BUFFER RETURN	This command copies every memory location in the PROM to the buffer beginning at destination address 00H in the buffer. The checksum is the 2's complement of the 16-bit sum of all the bytes read.

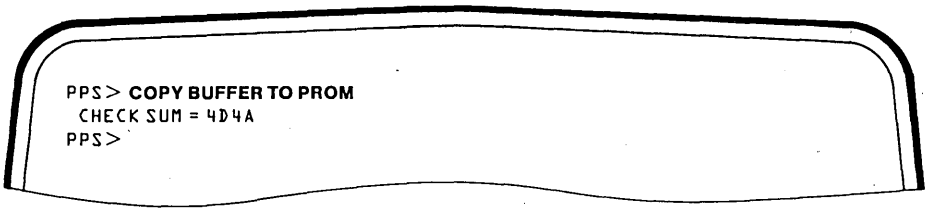
If you want to check the buffer to be sure the data now there matches the original data in the PROM, one command is all that is needed. Enter the VERIFY command, and if the buffer and PROM data match, you will be informed VERIFY TEST PASSED.



Key-in Sequence	Comments
VERIFY RETURN	The data in the buffer matches the data in the PROM.

280015-001

Now that you have verified that the data in the buffer matches the data in the PROM, you are ready to copy the buffer to a blank PROM. Remove the master PROM from the PROM socket and insert the blank PROM. Then use COPY again to copy the contents of the iPPS-PDS buffer to the blank PROM.



Key-in Sequence	Comments
COPY BUFFER TO PROM RETURN	The display of the check-sum and the return of the iPPS prompt indicate that the PROM was successfully programmed.

Note that for copying from the buffer to a PROM, you do not need to use the VERIFY command. The iPPS-PDS software automatically verifies the copying when you copy in this direction.

Interleaving a File Between Two PROMS

It is often desirable to have code or data arranged in 16-bit words and stored on a pair of 8-bit PROMs. This is the case, for example, when working with an 8086 microprocessor that reads from and writes to memory on a 16-bit data bus. The data is interleaved between two PROMs, the odd (or low) bytes stored in one PROM and the even (or high) bytes stored in the other PROM. The FORMAT command handles this interleaving automatically.

In the following example, a file written in Intel 8086 hexadecimal format is interleaved into two PROM devices.

```
PPS>FORMAT DOUBLE.BYT (0,FFFH)
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N-BYTE=4)
LU = 3
INPUT BLOCK SIZE (N BYTES)
N = 2
OUTPUT BLOCK SIZE (N BYTES)
N = 1
INPUT BLOCK STRUCTURE:
NUMBER OF INPUT LOGICAL UNITS = 002

LSB
 --------,
| 00 | 01 |
 --------
NUMBER OF OUTPUT LOGICAL UNITS = 001
OUTPUT SPECIFICATION (<CR> TO EXIT):
*
```

Key-in Sequence

FORMAT DOUBLE.BYT (0,FFFH) RETURN

3 RETURN

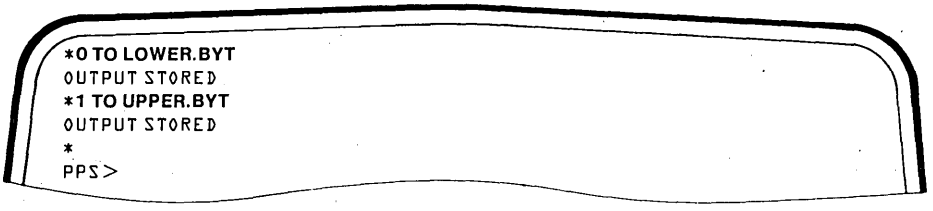
2 RETURN

1 RETURN

Comments

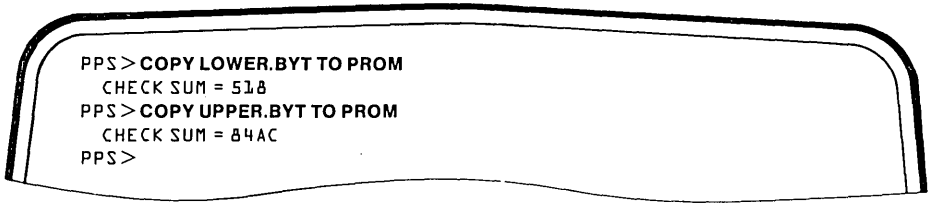
In this example, a file called DOUBLE.BYT is split into two files, with alternate bytes being loaded into alternate files. After establishing the FORMAT command and the file name with the first entry, the iPPS software prompts for the size of the logical unit that is going to be manipulated. Byte is selected as the logical unit. You are then prompted to set up the input block size (in this case two bytes) and the output block size (one byte). A diagram of the input block is displayed with the logical units labeled. The least significant bit in the input block is displayed with the logical units labeled. The least significant bit in the input block is shown on the left. The number of logical units in the output block is also displayed. You are then prompted with an asterisk (*) to enter the output specification.

280015-001



Key-in Sequence	Comments
0 TO LOWER.BYT RETURN **1 TO UPPER.BYT** RETURN RETURN	Once the size of the logical unit, the input block size and the output block sizes have been established, you are prompted for the output specification (how you want the data in the file to be manipulated in terms of logical units). This example specified that the least significant byte in each input block be stored in a file titled LOWER.BYT in the default drive. The iPPS software then sorts through the DOUBLE.BYT file. Next it specifies that the most significant byte be stored in a file titled UPPER.BYT. The iPPS software then sorts through the DOUBLE.BYT file and copies every odd byte to the UPPER.BYT file. OUTPUT STORED is displayed after each output specification is implemented. You then have the option of entering another output specification. Pressing RETURN exits the FORMAT command and returns the iPPS prompt.

You can use the two files created with this FORMAT operation to program two PROMs, which you can then install in parallel to provide 16-bit data words to a 16-bit microprocessor. To copy the files to the PROMs, use the COPY command as follows.

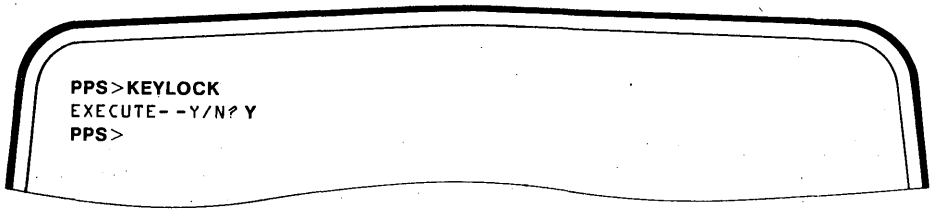


Key-in Sequence	Comments
COPY LOWER.BYT TO PROM RETURN **COPY UPPER.BYT TO PROM** RETURN	You must install a blank PROM in the personality module before entering each COPY command.

280015-001

Locking a Microcontroller

After programming a microcontroller, you can protect it from unauthorized access by locking it with the KEYLOCK command (the KEYLOCK command cannot be used with all EPROMs). The following example locks an 8751H microcontroller, which then cannot be unlocked without erasing it.



Key-in Sequence	Comments
KEYLOCK RETURN	Entering Y locks the EPROM. If you enter N, the command terminates and EPROM remains unlocked.
Y RETURN	

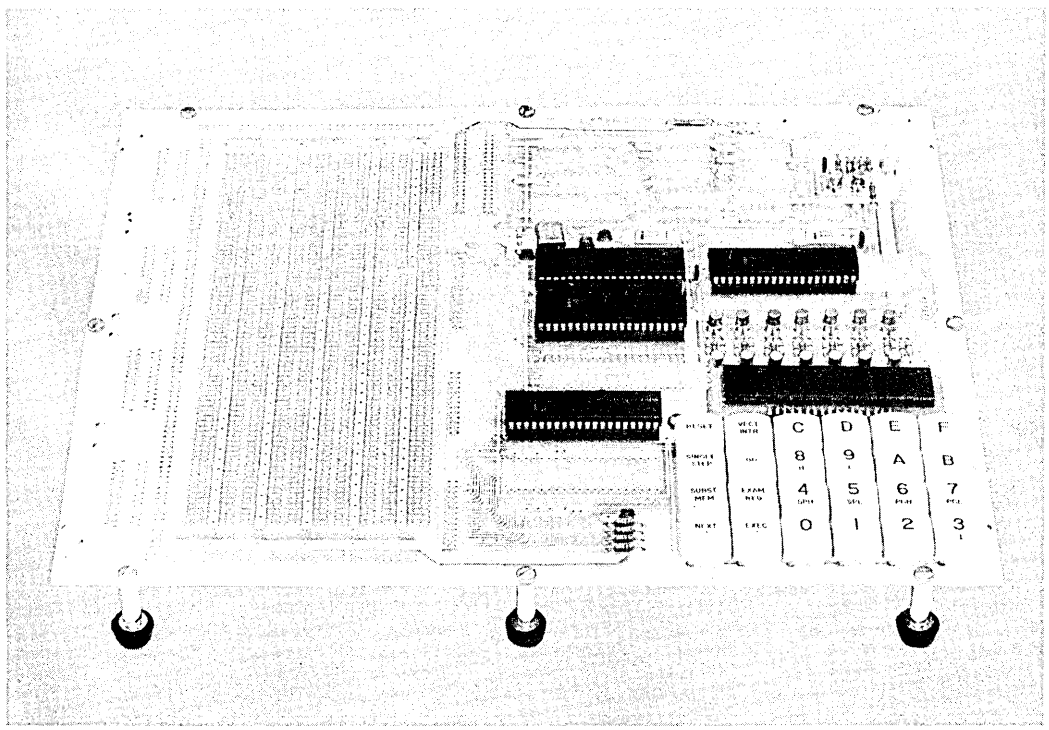
System Design Kits

7

SDK-85
MCS-85™ SYSTEM DESIGN KIT

- **Complete Single Board Microcomputer System Including CPU, Memory, and I/O**
- **Easy to Assemble, Low Cost, Kit Form**
- **Extensive System Monitor Software in ROM**
- **Interactive LED Display and Keyboard**
- **Large Wire-Wrap Area for Custom Interfaces**
- **Popular 8080A Instruction Set**
- **Interfaces Directly with TTY**
- **High Performance 3 MHz 8085A CPU (1.3 μs Instruction Cycle)**
- **Comprehensive Design Library Included**

The SDK-85 MCS-85 System Design Kit is a complete single board microcomputer system in kit form. It contains all components required to complete construction of the kit, including LED display, keyboard, resistors, caps, crystal, and miscellaneous hardware. Included is a preprogrammed ROM containing a system monitor for general software utilities and system diagnostics. The complete kit includes a 6-digit LED display and a 24-key keyboard for a direct insertion, examination, and execution of a user's program. In addition, it can be directly interfaced with a teletype terminal. The SDK-85 is an inexpensive, high performance prototype system that has designed-in flexibility for simple interface to the user's application.



MAY 1983

FUNCTIONAL DESCRIPTION

The SDK-85 is a complete 8085A microcomputer system on a single board, in kit form. It contains all necessary components to build a useful, functional system. Such items as resistors, capacitors, and sockets are included. Assembly time varies from three to five hours, depending on the skill of the user. The SDK-85 functional block diagram is shown in Figure 1.

8085A Processor

The SDK-85 is designed around Intel's 8085A microprocessor. The Intel 8085A is a new generation, complete 8-bit parallel central processing unit (CPU). Its instruction set is 100% software upward compatible with the 8080A microprocessor, and it is designed to improve the present 8080A's performance by higher system speed. Its high level of system integration allows a minimum system of three IC's: 8085A (CPU), 8156 (RAM), and 8355/8755 (ROM/PROM). A block diagram of the 8085A microprocessor is shown in Figure 2.

System Integration — The 8085A incorporates all of the features that the 8224 (clock generator) and 8228 (system controller) provided for the 8080A, thereby offering a high level of system integration.

Addressing — The 8085A uses a multiplexed data bus. The 16-bit address is split between the 8-bit address bus and the 8-bit address/data bus. The on-chip address latches of 8155/8156/8355/8755 memory products allows a direct interface with the 8085A.

System Monitor

A compact but powerful system monitor is supplied with the SDK-85 to provide general software utilities and system diagnostics. It comes in a pre-programmed ROM.

Communications Interface

The SDK-85 communicates with the outside world through either the on-board LED display/keyboard combination, or the user's TTY terminal (jumper selectable

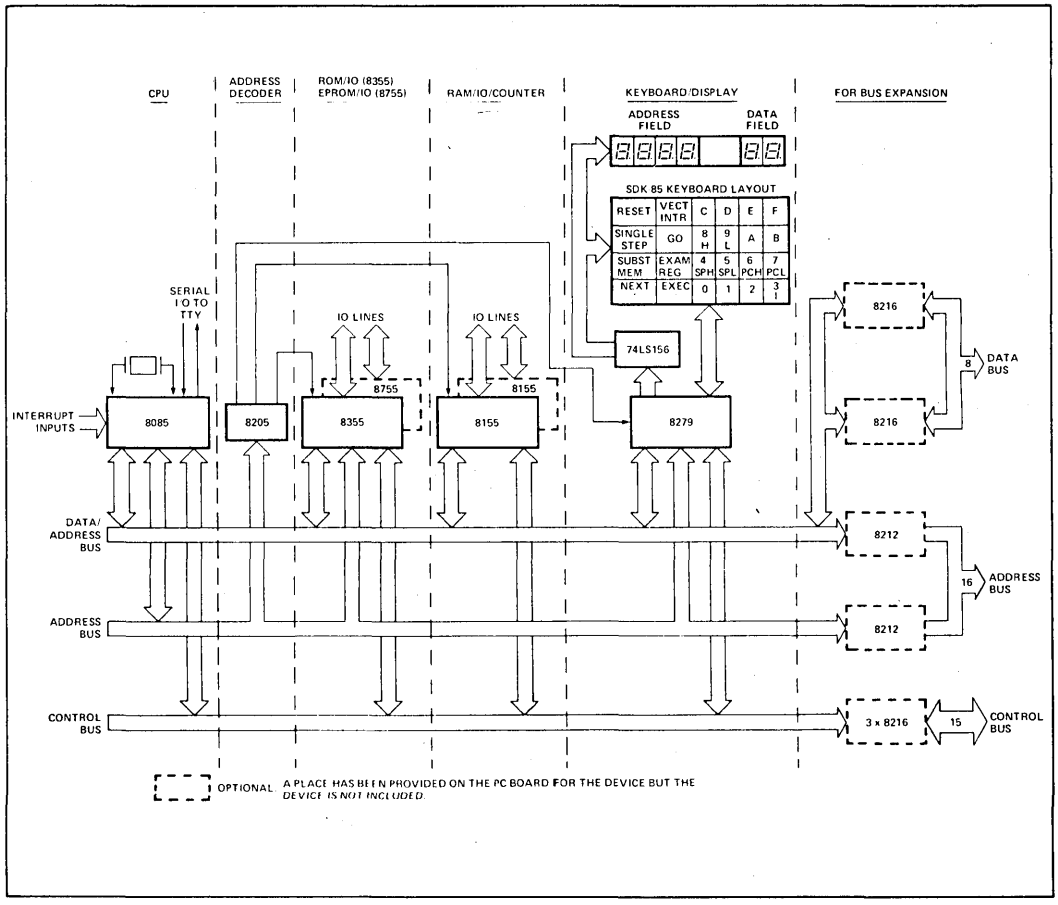
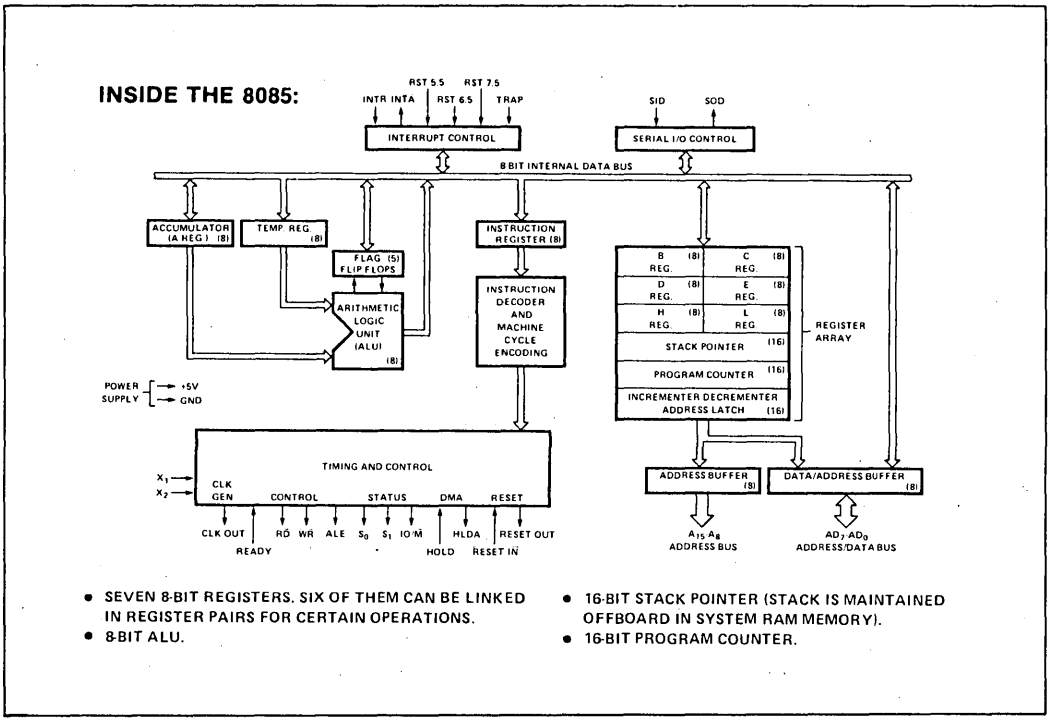


Figure 1. SDK-85 System Design Kit Functional Block Diagram



- SEVEN 8-BIT REGISTERS, SIX OF THEM CAN BE LINKED IN REGISTER PAIRS FOR CERTAIN OPERATIONS.
- 8-BIT ALU.
- 16-BIT STACK POINTER (STACK IS MAINTAINED OFFBOARD IN SYSTEM RAM MEMORY).
- 16-BIT PROGRAM COUNTER.

Figure 2. 8085A Microprocessor Block Diagram

Both memory and I/O can be easily expanded by simply soldering in additional devices in locations provided for this purpose. A large area of the board (45 sq. in.) is laid out as general purpose wire-wrap for the user's custom interfaces.

Assembly

Only a few simple tools are required for assembly; soldering iron, cutters, screwdriver, etc. The SDK-85 user's manual contains step-by-step instructions for easy assembly without mistakes. Once construction is complete, the user connects his kit to a power supply and the SDK-85 is ready to go. The monitor starts immediately upon power-on or reset.

Table 1. Keyboard Monitor Commands

Command	Operation
Reset	Starts monitor.
Go	Allows user to execute user program.
Single step	Allows user to execute user program one instruction at a time—useful for debugging.
Substitute memory	Allows user to examine and modify memory locations.
Examine register	Allows user to examine and modify 8085A's register contents.
Vector interrupt	Serves as user interrupt button.

Commands — Keyboard monitor commands and teletype monitor commands are provided in Table 1 and Table 2 respectively.

Table 2. Teletype Monitor Commands

Command	Operation
Display memory	Displays multiple memory locations.
Substitute memory	Allows user to examine and modify memory locations one at a time.
Insert instructions	Allows user to store multiple bytes in memory.
Move memory	Allows user to move blocks of data in memory.
Examine register	Allows user to examine and modify the 8085A's register contents.
Go	Allows user to execute user programs.

Documentation

In addition to detailed information on using the monitors, the SDK-85 user's manual provides circuit diagrams, a monitor listing, and a description of how the system works. The complete design library for the SDK-85 is shown in Figure 7-11 and listed in the Specifications section under Reference Manuals.

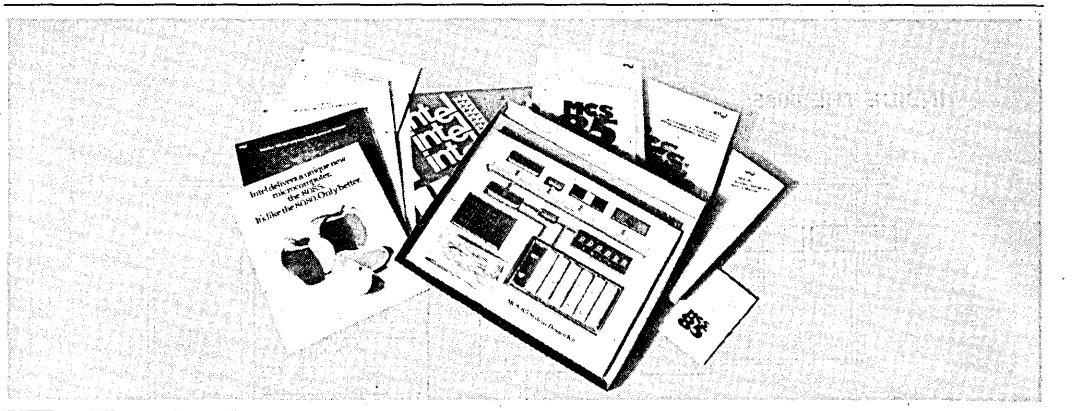


Figure 3. SDK-85 Design Library

8085A INSTRUCTION SET

Table 3 contains a summary of processor instructions used for the 8085A microprocessor.

Table 3. Summary of 8085A Processor Instructions

Mnemonic[1]	Description	Instruction Code[2] D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock[3] Cycles
MOVE, LOAD, AND STORE										
MOVr1r2	Move register to register	0	1	D	D	D	S	S	S	4
MOV M,r	Move register to memory	0	1	1	1	0	S	S	S	7
MOV r,M	Move memory to register	0	1	D	D	D	1	1	0	7
MVI r	Move immediate register	0	0	D	D	D	1	1	0	7
MVI M	Move immediate memory	0	0	1	1	0	1	1	0	10
LXI B	Load immediate register Pair B & C	0	0	0	0	0	0	0	1	10
LXI D	Load immediate register Pair D & E	0	0	0	1	0	0	0	1	10
LXI H	Load immediate register Pair H & L	0	0	1	0	0	0	0	1	10
STAX B	Store A indirect	0	0	0	0	0	0	1	0	7
STAX D	Store A indirect	0	0	0	1	0	0	1	0	7
LDAX B	Load A indirect	0	0	0	0	1	0	1	0	7
LDAX D	Load A indirect	0	0	0	1	1	0	1	0	7
STA	Store A direct	0	0	1	1	0	0	1	0	13
LDA	Load A direct	0	0	1	1	1	0	1	0	13
SHLD	Store H & L direct	0	0	1	0	0	0	1	0	16
LHLD	Load H & L direct	0	0	1	0	1	0	1	0	16
XCHG	Exchange D & E, H & L registers	1	1	1	0	1	0	1	1	4
STACK OPS										
PUSH B	Push register pair B & C on stack	1	1	0	0	0	1	0	1	12
PUSH D	Push register pair D & E on stack	1	1	0	1	0	1	0	1	12
PUSH H	Push register pair H & L on stack	1	1	1	0	0	1	0	1	12
PUSH PSW	Push A and flags on stack	1	1	1	1	0	1	0	1	12
POP B	Pop register pair B & C off stack	1	1	0	0	0	0	0	1	10
POP D	Pop register pair D & E off stack	1	1	0	1	0	0	0	1	10
POP H	Pop register pair H & L off stack	1	1	1	0	0	0	0	1	10
POP PSW	Pop A and flags off stac	1	1	1	1	0	0	0	1	10
XTHL	Exchange top of stack, H & L	1	1	1	0	0	0	1	1	16
SPHL	H & L to stack pointer	1	1	1	1	1	0	0	1	6
LXI SP	Load immediate stack pointer	0	0	1	1	0	0	0	1	10
INX SP	Increment stack pointer	0	0	1	1	0	0	1	1	6
DCX SP	Decrement stack pointer	0	0	1	1	1	0	1	1	6
JUMP										
JMP	Jump unconditional	1	1	0	0	0	0	1	1	10
JC	Jump on carry	1	1	0	1	1	0	1	0	7/10
JNC	Jump on no carry	1	1	0	1	0	0	1	0	7/10
JZ	Jump on zero	1	1	0	0	1	0	1	0	7/10
JNZ	Jump on no zero	1	1	0	0	0	0	1	0	7/10
JP	Jump on positive	1	1	1	1	0	0	1	0	7/10
JM	Jump on minus	1	1	1	1	1	0	1	0	7/10
JPE	Jump on parity even	1	1	1	0	1	0	1	0	7/10
JPO	Jump on parity odd	1	1	1	0	0	0	1	0	7/10
PCHL	H & L to program counter	1	1	1	0	1	0	0	1	6
CALL										
CALL	Call unconditional	1	1	0	0	1	1	0	1	18
CC	Call on carry	1	1	0	1	1	1	0	0	9/18
CNC	Call on no carry	1	1	0	1	0	1	0	0	9/18
CZ	Call on zero	1	1	0	0	1	1	0	0	9/18
CNZ	Call on no zero	1	1	0	0	0	1	0	0	9/18
CP	Call on positive	1	1	1	1	0	1	0	0	9/18
CM	Call on minus	1	1	1	1	1	1	0	0	9/18
CPE	Call on parity even	1	1	1	0	1	1	0	0	9/18
CPO	Call on parity odd	1	1	1	0	0	1	0	0	9/18
RETURN										
RET	Return	1	1	0	0	1	0	0	1	10
RC	Return on carry	1	1	0	1	1	0	0	0	6/12
RNC	Return on no carry	1	1	0	1	0	0	0	0	6/12
RZ	Return on zero	1	1	0	0	1	0	0	0	6/12
RNZ	Return on no zero	1	1	0	0	0	0	0	0	6/12
RP	Return on positive	1	1	1	1	0	0	0	0	6/12
RM	Return on minus	1	1	1	1	1	0	0	0	6/12

continued

Table 3. Summary of 8085A Processor Instructions (Continued)

Mnemonic[1]	Description	Instruction Code[2] D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock[3] Cycles
RPE	Return on parity even	1	1	1	0	1	0	0	0	6/12
RPO	Return on parity odd	1	1	1	0	0	0	0	0	6/12
RESTART										
RST	Restart	1	1	A	A	A	1	1	1	12
INCREMENT AND DECREMENT										
INR r	Increment register	0	0	D	D	D	1	0	0	4
DCR r	Decrement register	0	0	D	D	D	1	0	1	4
INR M	Increment memory	0	0	1	1	0	1	0	0	10
DCR M	Decrement memory	0	0	1	1	0	1	0	1	10
INX B	Increment B & C registers	0	0	0	0	0	0	1	1	6
INX D	Increment D & E registers	0	0	0	1	0	0	1	1	6
INX H	Increment H & L registers	0	0	1	0	0	0	1	1	6
DCX B	Decrement B & C	0	0	0	0	1	0	1	1	6
DCX D	Decrement D & E	0	0	0	1	1	0	1	1	6
DCX H	Decrement H & L	0	0	1	0	1	0	1	1	6
ADD										
ADD r	Add register to A	1	0	0	0	0	S	S	S	4
ADC r	Add register to A with carry	1	0	0	0	1	S	S	S	4
ADD M	Add memory to A	1	0	0	0	0	1	1	0	7
ADC M	Add memory to A with carry	1	0	0	0	1	1	1	0	7
ADI	Add immediate to A	1	1	0	0	0	1	1	0	7
ACI	Add immediate to A with carry	1	1	0	0	1	1	1	0	7
DAD B	Add B & C to H & L	0	0	0	0	1	0	0	1	10
DAD D	Add D & E to H & L	0	0	0	1	1	0	0	1	10
DAD H	Add H & L to H & L	0	0	1	0	1	0	0	1	10
DAD SP	Add stack pointer to H & L	0	0	1	1	1	0	0	1	10
SUBTRACT										
SUB r	Subtract register from A	1	0	0	1	0	S	S	S	4
SBB r	Subtract register from A with borrow	1	0	0	1	1	S	S	S	4
SUB M	Subtract memory from A	1	0	0	1	0	1	1	0	7
SBB M	Subtract memory from A with borrow	1	0	0	1	1	1	1	0	7
SUI	Subtract immediate from A	1	1	0	1	0	1	1	0	7
SBI	Subtract immediate from A with borrow	1	1	0	1	1	1	1	0	7

Mnemonic[1]	Description	Instruction Code[2] D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock[3] Cycles
LOGICAL										
ANA r	And register with A	1	0	1	0	0	S	S	S	4
XRA r	Exclusive Or register with A	1	0	1	0	1	S	S	S	4
ORA r	Or register with A	1	0	1	1	0	S	S	S	4
CMP r	Compare register with A	1	0	1	1	1	S	S	S	4
ANA M	And memory with A	1	0	1	0	0	1	1	0	7
XRA M	Exclusive Or memory with A	1	0	1	0	1	1	1	0	7
ORA M	Or memory with A	1	0	1	1	0	1	1	0	7
CMP M	Compare memory with A	1	0	1	1	1	1	1	0	7
ANI	And immediate with A	1	1	1	0	0	1	1	0	7
XRI	Exclusive Or immediate with A	1	1	1	0	1	1	1	0	7
ORI	Or immediate with A	1	1	1	1	0	1	1	0	7
CPI	Compare immediate with A	1	1	1	1	1	1	1	0	7
ROTATE										
RLC	Rotate A left	0	0	0	0	0	1	1	1	4
RRC	Rotate A right	0	0	0	0	1	1	1	1	4
RAL	Rotate A left through carry	0	0	0	1	0	1	1	1	4
RAR	Rotate A right through carry	0	0	0	1	1	1	1	1	4
SPECIALS										
CMA	Complement A	0	0	1	0	1	1	1	1	4
STC	Set carry	0	0	1	1	0	1	1	1	4
CMC	Complement carry	0	0	1	1	1	1	1	1	4
DAA	Decimal adjust A	0	0	1	0	0	1	1	1	4
INPUT/OUTPUT										
IN	Input	1	1	0	1	1	0	1	1	10
OUT	Output	1	1	0	1	0	0	1	1	10
CONTROL										
EI	Enable interrupts	1	1	1	1	1	0	1	1	4
DI	Disable interrupts	1	1	1	1	0	0	1	1	4
NOP	No-operation	0	0	0	0	0	0	0	0	4
HLT	Halt	0	1	1	1	0	1	1	0	5
NEW 8085 INSTRUCTIONS										
RIM	Read interrupt mask	0	0	1	0	0	0	0	0	4
SIM	Set interrupt mask	0	0	1	1	0	0	0	0	4

Notes

1. All mnemonics copyright © Intel Corporation 1977.
2. DDD or SSS: B = 000, C = 001, D = 010, E = 011, H = 100, L = 101, Memory = 110, A = 111.
3. Two possible cycle times. (6/12) indicates instruction cycles dependent on condition flags.

SPECIFICATIONS

Central Processor

CPU — 8085A

Instruction Cycle — 1.3 μs

Tcy — 330 ns

Memory

ROM — 2K bytes (expandable to 4K bytes) 8355/8755A

RAM — 256 bytes (expandable to 512 bytes) 8155

Addressing

ROM — 0000-07FF (expendable to 0FFF with an additional 8355/8755A)

RAM — 2000-20FF (2800-28FF available with an additional 8155)

Note

The wire-wrap area of the SDK-85 PC board may be used for additional custom memory expansion up to the 64K-byte addressing limit of the 8085A.

Input/Output

Parallel — 38 lines (expandable to 76 lines)

Serial — Through SID/SOD ports of 8085A. Software generated baud rate.

Baud Rate — 110

Interfaces

Bus — All signals TTL compatible
Parallel I/O — All signals TTL compatible
Serial I/O — 20 mA current loop TTY

Note
By populating the buffer area of the board, the user has access to all bus signals that enable him to design custom system expansions into the kit's wire-wrap area.

Interrupts

Three Levels
(RST 7.5) — Keyboard interrupt
(RST 6.5) — TTL input
(INTR) — TTL input

DMA

Hold Request — Jumper selectable. TTL compatible input.

Software

System Monitor — Pre-programmed 8755A or 8355 ROM
Addresses — 0000-07FF
Monitor I/O — Keyboard/display or TTY (serial I/O)

Physical Characteristics

Width — 12.0 in. (30.5 cm)
Height — 10 in. (25.4 cm)
Depth — 0.50 in. (1.27 cm)
Weight — approx. 12 oz

Electrical Characteristics

DC Power Requirement (power supply not included in kit)

Voltage	Current
V_{CC} 5V ± 5%	1.3A
V_{TTY} – 10V ± 10%	0.3A (V_{TTY} required only if teletype is connected)

Environmental Characteristics

Operating Temperature — 0-55°C

Reference Manuals

9800451 — SDK-85 User's Manual (SUPPLIED)

9800366 — MCS-85 User's Manual (SUPPLIED)

9800301 — 8080/8085 Assembly Language Programming Manual (SUPPLIED)

8085/8080 Assembly Language Reference Card (SUPPLIED)

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

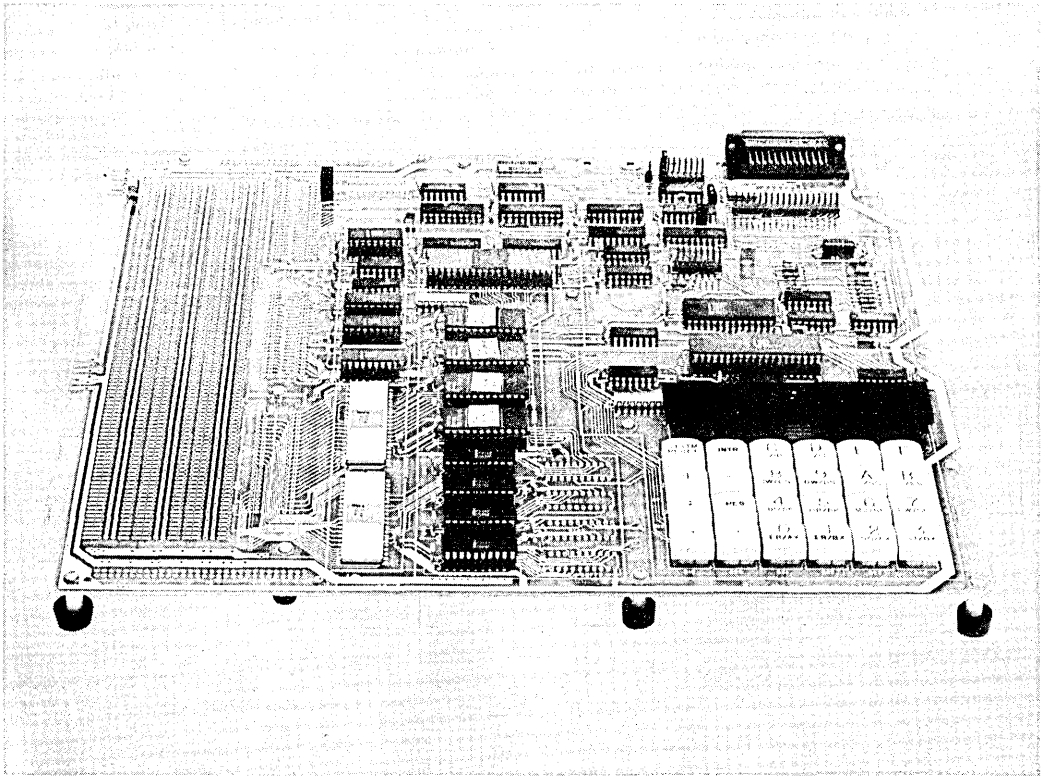
ORDERING INFORMATION

Part Number	Description
SDK-85	MCS-85 system design kit

SDK-86
MCS-86™ SYSTEM DESIGN KIT

- **Complete Single Board Microcomputer System Including CPU, Memory, and I/O**
- **Easy to Assemble Kit Form**
- **High Performance 8086 16-Bit CPU**
- **Interfaces Directly with TTY or CRT**
- **Interactive LED Display and Keyboard**
- **Wire Wrap Area for Custom Interfaces**
- **Extensive System Monitor Software in ROM**
- **Comprehensive Design Library Included**

The SDK-86 MCS-86 System Design Kit is a complete single board 8086 microcomputer system in kit form. It contains all necessary components to complete construction of the kit, including LED display, keyboard, resistors, caps, crystal, and miscellaneous hardware. Included are preprogrammed ROMs containing a system monitor for general software utilities and system diagnostics. The complete kit includes an 8-digit LED display and a mnemonic 24-key keyboard for direct insertion, examination, and execution of a user's program. In addition, it can be directly interfaced with a teletype terminal, CRT terminal, or the serial port of an Intellec system. The SDK-86 is a high performance prototype system with designed-in flexibility for simple interface to the user's application.



MAY 1983
Order Number: 205945-001

FUNCTIONAL DESCRIPTION

The SDK-86 is a complete MCS-86 microcomputer system on a single board, in kit form. It contains all necessary components to build a useful, functional system. Such items as resistors, caps, and sockets are included. Assembly time varies from 4 to 10 hours, depending on the skill of the user. The SDK-86 functional block diagram is shown in Figure 1.

8086 Processor

The SDK-86 is designed around Intel's 8086 microprocessor. The Intel 8086 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor features attributes of both 8-bit and 16-bit microprocessors in that it addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance. Additional features of the 8086 include the following:

- Direct addressing capability to one megabyte of memory
- Assembly language compatibility with 8080/8085
- 14 word x 16-bit register set with symmetrical operations
- 24 operand addressing modes
- Bit, byte, word, and block operations
- 8 and 16-byte signed and unsigned arithmetic in binary or decimal mode, including multiply and divide
- 4 or 5 or 8 MHz clock rate

A block diagram of the 8086 microprocessor is shown in Figure 2.

System Monitor

A compact but powerful system monitor is supplied with the SDK-86 to provide general software utilities and system diagnostics. It comes in preprogrammed read only memories (ROMs).

Communications Interface

The SDK-86 communicates with the outside world through either the on-board light emitting diode (LED) display/keyboard combination or the user's TTY or CRT terminal (jumper selectable), or by means of a special mode in which an Intellec development system transports finished programs to and from the SDK-86. Memory may be easily expanded by simply soldering in additional devices in locations provided for this purpose. A large area of the board (22 square inches) is laid out as general purpose wire-wrap for the user's custom interfaces.

Assembly

Only a few simple tools are required for assembly: soldering iron, cutters, screwdriver, etc. The SDK-86 assembly manual contains step-by-step instructions for easy assembly with a minimum of mistakes. Once construction is complete, the user connects his kit to a power supply and the SDK-86 is ready to go. The monitor starts immediately upon power-on or reset.

Commands — Keyboard mode commands, serial port commands, and Intellec slave mode commands are summarized in Table 1, Table 2, and Table 3, respectively. The SDK-86 keyboard is shown in Figure 3.

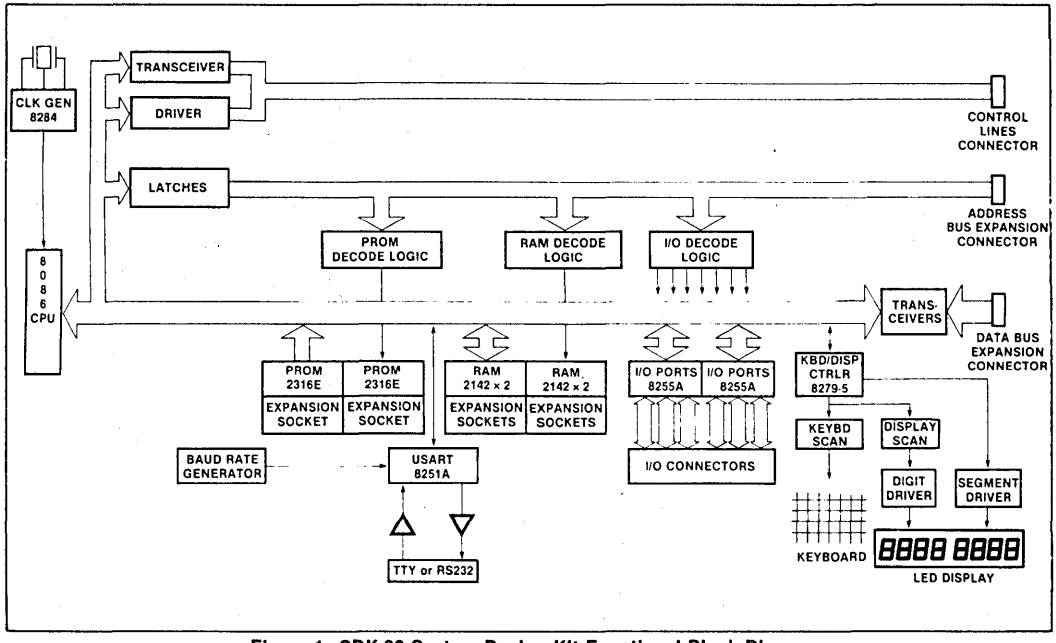


Figure 1. SDK-86 System Design Kit Functional Block Diagram

Documentation

In addition to detailed information on using the monitors, the SDK-86 user's manual provides circuit diagrams, a monitor listing, and a description of how the system works. The complete design library for the SDK-86 is shown in Figure 4 and listed in the specifications section under Reference Manuals.

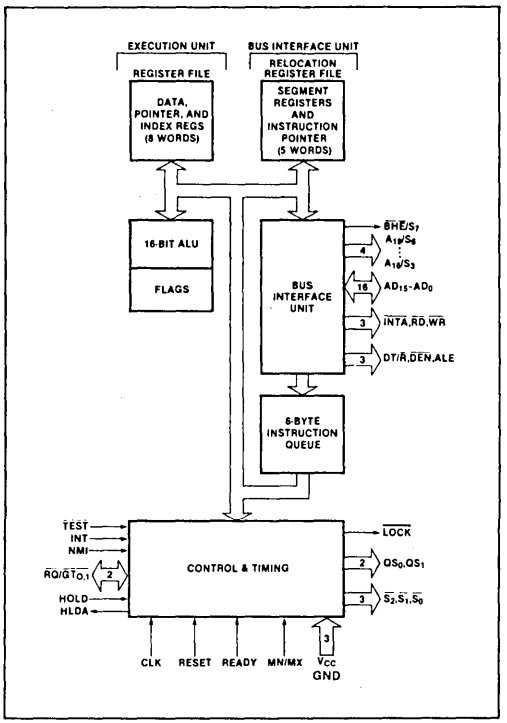


Figure 2. 8086 Microprocessor Block Diagram

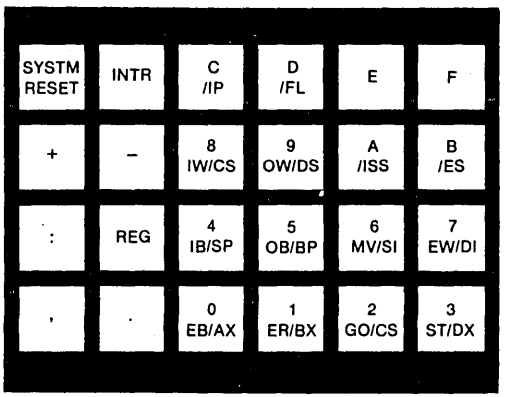


Figure 3. SDK-86 Keyboard

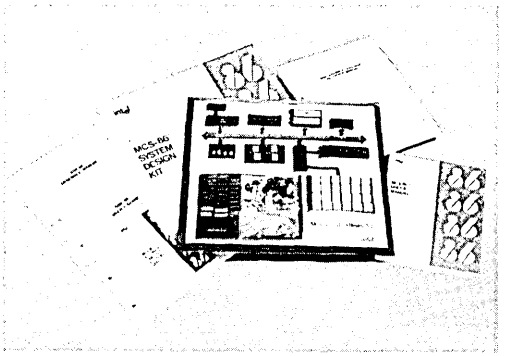


Figure 4. SDK-86 Design Library

Table 1. Keyboard Mode Commands

Command	Operation
Reset	Starts monitor.
Go	Allows user to execute user program, and causes it to halt at predetermined program stop. Useful for debugging.
Single step	Allows user to execute user program one instruction at a time. Useful for debugging.
Substitute memory	Allows user to examine and modify memory locations in byte or word mode.
Examine register	Allows user to examine and modify 8086 register contents.
Block move	Allows user to relocate program and data portions in memory.
Input or output	Allows direct control of SDK-86 I/O facilities in byte or mode.

Table 2. Serial Mode Commands

Command	Operation
Dump memory	Allows user to print or display large blocks of memory information in hex format than amount visible on terminal's CRT display.
Start/continue display	Allows user to display blocks of memory information larger than amount visible on terminal's CRT display.
Punch/read paper tape	Allows user to transmit finished programs into and out of SDK-86 via TTY paper tape punch.

8086 INSTRUCTION SET

Table 4 contains a summary of processor instructions used for the 8086 microprocessor.

Table 4. 8086 Instruction Set Summary

Mnemonic and Description	Instruction Code				
Data Transfer					
MOV = Move:	76543210	76543210	76543210	76543210	
Register/memory to/from register	100010dw	mod reg r/m			
Immediate to register/memory	1100011w	mod000 r/m	data	data if w=1	
Immediate to register	1011w reg	data	data if w=1		
Memory to accumulator	1010000w	addr-low	addr-high		
Accumulator to memory	1010001w	addr-low	addr-high		
Register/memory to segment register	10001110	mod0reg r/m			
Segment register to register/memory	10001100	mod0reg r/m			
PUSH = Push:					
Register/memory	11111111	mod110 r/m			
Register	01010 reg				
Segment register	000reg110				
POP = Pop:					
Register/memory	10001111	mod000 r/m			
Register	01011 reg				
Segment register	000reg111				
XCHG = Exchange:					
Register/memory with register	1000011w	mod reg r/m			
Register with accumulator	10010 reg				
IN = Input					
Fixed port	1110010w	port			
Variable port	1110110w				
OUT = Output					
Fixed port	1110011w	port			
Variable port	1110111w				
XLAT=Translate byte to AL	11010111				
LEA=Load EA to register	10001101	mod reg r/m			
LDS=Load pointer to DS	11000101	mod reg r/m			
LES=Load pointer to ES	11000100	mod reg r/m			
LAHF=Load AH with flags	10011111				
SAHF=Store AH into flags	10011110				
PUSHF=Push flags	10011100				
POPF=Pop flags	10011101				
Arithmetic					
ADD = Add:					
Reg./memory with register to either	000000dw	mod reg r/m			
Immediate to register/memory	100000sw	mod000 r/m	data	data if s:w=01	
Immediate to accumulator	0000010w	data	data if w=1		
ADC = Add with carry:					
Reg./memory with register to either	000100dw	mod reg r/m			
Immediate to register/memory	100000sw	mod010 r/m	data	data if s:w=01	
Immediate to accumulator	0001010w	data	data if w=1		
INC = Increment:					
Register/memory	1111111w	mod000 r/m			
Register	01000 reg				
AAA=ASCII adjust for add	00110111				
BAA=Decimal adjust for add	00100111				
SUB = Subtract:					
Reg./memory and register to either	001010dw	mod reg r/m			
Immediate from register/memory	100000sw	mod101 r/m	data	data if s:w=01	
Immediate from accumulator	0010110w	data	data if w=1		
SBB = Subtract with borrow					
Reg./memory and register to either	000110dw	mod reg r/m			
Immediate from register/memory	100000sw	mod011 r/m	data	data if s:w=01	
Immediate from accumulator	0001110w	data	data if w=1		
DEC = Decrement:					
Register/memory	1111111w	mod001 r/m			
Register	01001 reg				
NEG=Change sign	1111011w	mod011 r/m			

Mnemonic and Description	Instruction Code				
CMP = Compare:	76543210	76543210	76543210	76543210	
Register/memory and register	001110dw	mod reg r/m			
Immediate with register/memory	100000sw	mod111 r/m	data	data if s:w=01	
Immediate with accumulator	0011110w	data	data if w=1		
AAS=ASCII adjust for subtract	00111111				
DAS=Decimal adjust for subtract	00101111				
MUL=Multiply (unsigned)	1111011w	mod100 r/m			
IMUL=Integer multiply (signed)	1111011w	mod101 r/m			
AAM=ASCII adjust for multiply	11010100	00001010			
DIV=Divide (unsigned)	1111011w	mod110 r/m			
IDIV=Integer divide (signed)	1111011w	mod111 r/m			
AAD=ASCII adjust for divide	11010101	00001010			
CBW=Convert byte to word	10011000				
CWD=Convert word to double word	10011001				
Logic					
NOT=Invert	1111011w	mod010 r/m			
SHL/SAL=Shift logical/arithmetic left	110100vw	mod100 r/m			
SHR=Shift logical right	110100vw	mod101 r/m			
SAR=Shift arithmetic right	110100vw	mod111 r/m			
ROL=Rotate left	110100vw	mod000 r/m			
ROR=Rotate right	110100vw	mod001 r/m			
RCL=Rotate through carry flag left	110100vw	mod010 r/m			
RCR=Rotate through carry right	110100vw	mod011 r/m			
AND = And:					
Reg./memory and register to either	001000dw	mod reg r/m			
Immediate to register/memory	1000000w	mod100 r/m	data	data if w=1	
Immediate to accumulator	0010010w	data	data if w=1		
TEST = And function to flags, no result:					
Register/memory and register	1000010w	mod reg r/m			
Immediate data and register/memory	1111011w	mod000 r/m	data	data if w=1	
Immediate data and accumulator	1010100w	data	data if w=1		
OR = Or:					
Reg./memory and register to either	000010dw	mod reg r/m			
Immediate to register/memory	1000000w	mod001 r/m	data	data if w=1	
Immediate to accumulator	0000110w	data	data if w=1		
XOR = Exclusive or:					
Reg./memory and register to either	001100dw	mod reg r/m			
Immediate to register/memory	1000000w	mod110 r/m	data	data if w=1	
Immediate to accumulator	0011010w	data	data if w=1		
String Manipulation					
REP=Repeat	1111001z				
MOVS = Move byte/word	1010010w				
CMPS = Compare byte/word	1010011w				
SCAS = Scan byte/word	1010111w				
LODS = Load byte/wd to AL/AX	1010110w				
STOS = Stor byte/wd frm AL/A	1010101w				
Control Transfer					
CALL = Call:					
Direct within segment	11101000	disp-low	disp-high		
Indirect within segment	11111111	mod010 r/m			
Direct intersegment	10011010	offset-low	offset-high		
		seg-low	seg-high		
Indirect intersegment	11111111	mod011 r/m			

continued

Table 4. 8086 Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code		
JMP - Unconditional Jump:	76543210	76543210	76543210
Direct within segment	11101001	disp-low	disp-high
Direct within segment-short	11101011	disp	
Indirect within segment	11111111	mod 100 r/m	
Direct intersegment	11101010	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	11111111	mod 101 r/m	
RET - Return from CALL:			
Within segment	11000011		
Within seg. adding immed to SP	11000010	data-low	data-high
Intersegment	11001011		
Intersegment adding immediate to SP	11001010	data-low	data-high
JE/JZ - Jump on equal/zero	01110100	disp	
JL/JNGE - Jump on less/not greater or equal	01111100	disp	
JLE/JNG - Jump on less or equal/not greater	01111110	disp	
JB/JNAE - Jump on below/not above or equal	01110010	disp	
JBE/JNA - Jump on below or equal/not above	01110110	disp	
JP/JPE - Jump on parity/parity even	01111010	disp	
JO - Jump on overflow	01110000	disp	
JS - Jump on sign	01111000	disp	
JNE/JNZ - Jump on not equal/not zero	01110101	disp	
JNL/JGE - Jump on not less/greater or equal	01111101	disp	
JNLE/JG - Jump on not less or equal/greater	01111111	disp	
JNB/JAE - Jump on not below/above or equal	01110011	disp	
JNBE/JA - Jump on not below or equal/above	01110111	disp	
JNP/JPO - Jump on not par/par odd	01111011	disp	
JNO - Jump on not overflow	01110001	disp	
JNS - Jump on not sign	01111001	disp	
LOOP Loop CX times	11100010	disp	
LOOPZ/LOOPE Loop while zero/equal	11100001	disp	
LOOPNZ/LOOPNE Loop while not zero/equal	11100000	disp	
JCXZ Jump on CX zero	11100011	disp	
INT - Interrupt			
Type specified	11001101	type	
Type 3	11001100		
INTO Interrupt on overflow	11001110		
IRET Interrupt return	11001111		
Processor Control			
CLC Clear carry	11111000		
CMC Complement carry	11110101		
STC Set carry	11111001		
CLD Clear direction	11111100		
STD Set direction	11111101		
CLI Clear interrupt	11111010		
STI Set interrupt	11111011		
HLT Halt	11110100		
WAIT Wait	10011011		
ESC Escape (to external device)	11011xxx	mod xxx r/m	
LOCK Bus lock prefix	11110000		

Notes

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low

if s:w = 01 then 16 bits of immediate data form the operand.
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand.
if v = 0 then "count" = 1; if v = 0 then "count" in (CL)
x = don't care
if v = 0 then "count" = 1; if v = 1 then "count" in (CL) register.
z is used for string primitives for comparison with ZF FLAG.

SEGMENT OVERRIDE PREFIX

001 reg 110

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

Mnemonics © Intel, 1978

SPECIFICATIONS

Central Processor

CPU — 8086 (5 MHz clock rate)

Note
May be operated at 2.5 MHz or 5 MHz, jumper selectable, for use with 8086.

Memory

ROM — 8K bytes 2316/2716
RAM — 2K bytes (expandable to 4K bytes) 2142

Addressing

ROM — FE000-FFFFF
RAM — 0-7FF (800-FFF available with additional 2142's)

Note
The wire-wrap area of the SDK-86 PC board may be used for additional custom memory expansion.

Input/Output

Parallel — 48 lines (two 8255A's)
Serial — RS232 or current loop (8251A)
Baud Rate — selectable from 110 to 4800 baud

Interfaces

Bus — All signals TTL compatible

Parallel I/O — All signals TTL compatible

Serial I/O — 20 mA current loop TTY or RS232

Note
The user has access to all bus signals which enable him to design custom system expansions into the kit's wire-wrap area.

Interrupts (256 vectored)

Maskable
Non-maskable
TRAP

DMA

Hold Request — Jumper selectable. TTL compatible input.

Software

System Monitor — Preprogrammed 2716 or 2316 ROMs

Addresses — FE000-FFFFF

Monitor I/O — Keyboard/display or TTY or CRT (serial I/O)

Physical Characteristics

Width — 13.5 in. (34.3 cm)

Height — 12 in. (30.5 cm)

Depth — 1.75 in. (4.45 cm)

Weight — approx. 24 oz. (3.3 kg)

Electrical Characteristics

DC Power Requirement
(Power supply not included in kit)

Voltage	Current
V_{CC} 5V ± 5%	3.5A
V_{TTY} – 12V ± 10%	0.3A
	(V_{TTY} required only if teletype is connected)

Environmental Characteristics

Operating Temperature — 0-50°C

Reference Manuals

9800697A — SDK-86 MCS-86 System Design Kit Assembly Manual

9800722 — MCS-86 User's Manual

9800640A — 8086 Assembly Language Programming Manual

8086 Assembly Language Reference Card

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

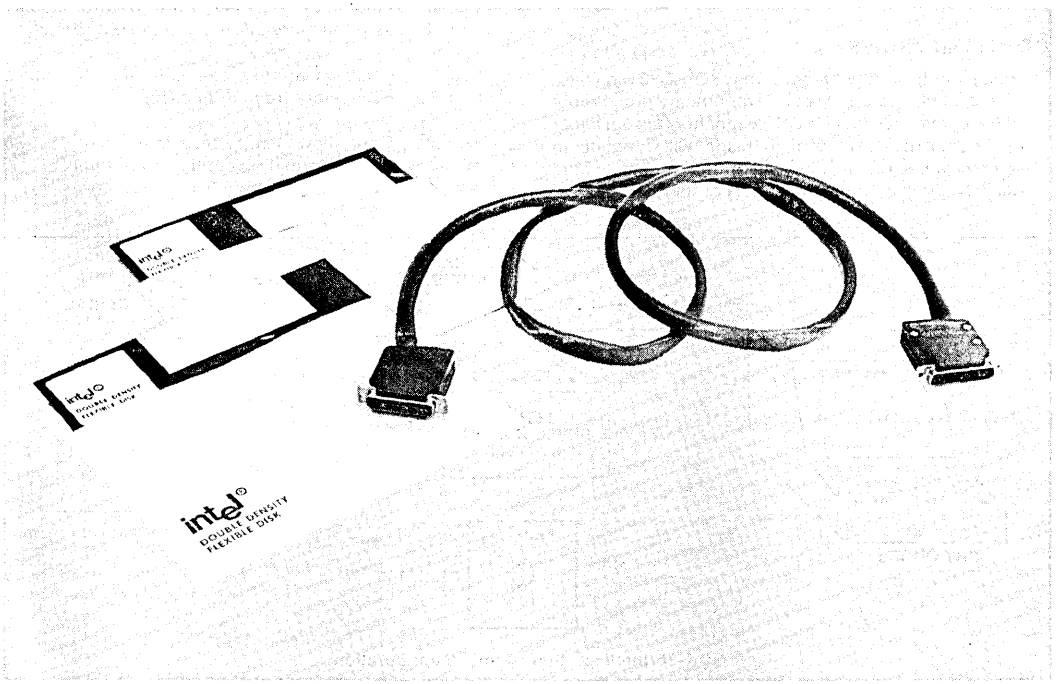
ORDERING INFORMATION

Part Number	Description
SDK-86	MCS-86 system design kit

SDK-C86
MCS-86™ SYSTEM DESIGN KIT
SOFTWARE AND CABLE INTERFACE TO INTELLEC® DEVELOPMENT SYSTEM

- **Provides the Software and Hardware Communications Link Between an Intellec® Development System and the SDK-86**
- **Intellec® System Files can be Accessed and Down-Loaded to the SDK-86 Resident Memory**
- **Data in SDK-86 Memory can be Uploaded and Saved in Intellec® System Files**
- **Enhances and Extends the Power and Usefulness of the SDK-86**
- **Allows the SDK-86 to Become an Execution Vehicle for ISIS-II Developed 8086 Object Code Using the Series II 8086/8088 Software Development Packages**
- **All SDK-86 Serial Port Mode Commands Become Available at Console of the Intellec® System**

The SDK-C86 product provides the software and hardware link for using the SDK-86 monitor in conjunction with an Intellec® Development System while adding features of data transfer between SDK-86 memory and Intellec® System files. The user may enter programs and data into the SDK-86 and then save them on a diskette. Also, programs and data may be created on the Intellec® System using the Series II 8086/8088 Software Development Packages, then loaded into the SDK-86 for testing and checkout. This provides a real time execution environment of the SDK-86 as a peripheral to the Intellec® System.



HARDWARE

There are two serial ports on the Intellec® System back panel, TTY and CRT. Assuming that one of the ports is used for the Intellec® console, the SDK-C86 cable can plug into the unused port. The SDK-86 is jumper selectable to accept either the CRT (RS232) or TTY (20mA current loop) signals.

The edge connector on the SDK-86 has the MULTIBUS™ form factor. No signals are connected to the fingers except the power supply traces. Therefore, the SDK-86 can plug directly into the Intellec® motherboard to obtain power while using the SDK-C86 cable as the communication link.

SOFTWARE

Two programs must be invoked to operate in the SDK-86 slave mode. One program runs on the SDK-86, and another runs in any ISIS-II environment that includes a diskette drive.

The serial I/O monitor is installed on the SDK-86 and operates as though it was talking to a terminal. The software in the Intellec® allows the Intellec®, with a console device, to behave as if it were a terminal to the SDK-86.

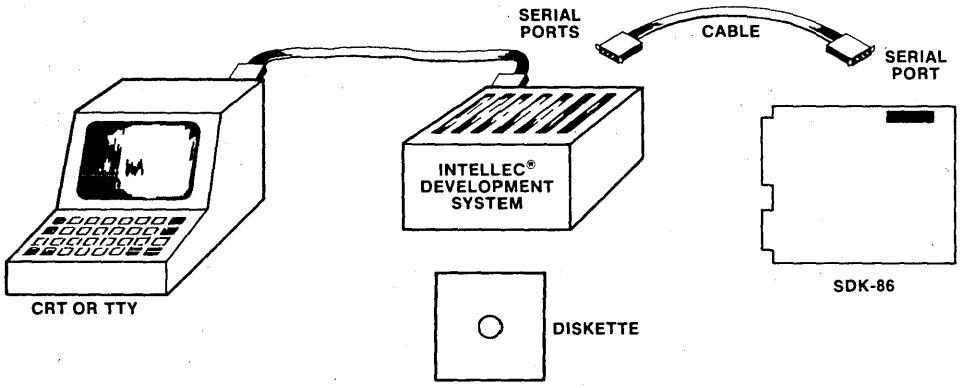
The SDK-C86 software program in the Intellec reads the console input device, then passes the character to the SDK-86 through the serial port. It also receives the characters from the SDK-86 and displays them at the console output device. Besides the basic transfer function, this program also recognizes and performs the Upload and Download functions.

COMMAND MODES

- Transparent: In this mode, the SDK-C86 software passes all characters through without any processing. All the commands of the SDK-86 monitor (except paper tape commands) are available and will function in exactly the same manner as if the terminal were attached directly to the serial port of the SDK-86.
- Upload/Download: In this mode the SDK-C86 software, in the Intellec®, recognizes the mnemonic for Upload or Download from the terminal. It "translates" the Download command to an R (Read hexadecimal tape) command and the Upload command to a W (Write hexadecimal tape). The R and W commands are then passed on to the SDK-86 monitor. Using these paper tape commands allows for a checksummed transfer of data between the Intellec® and the SDK-86 memory.

COMMAND SUMMARY

- Reset — starts the SDK-86 monitor.
- Execute with Breakpoint (G) — Allows you to execute a user program and cause it to halt at a predetermined program step — useful for debugging.
- Single Step (N) — allows you to execute a user program one instruction at a time — useful for debugging.
- Substitute Memory (S, SW) — allows you to examine and modify memory locations in byte or word mode.
- Examine Register (X) — allows you to examine and modify the 8086's register contents.
- Block Move (M) — allows you to relocate program and data portions in memory.
- Input or Output (I, IW, O, OW) — allows direct control of the SDK-86's I/O facilities in byte or word mode.
- Display Memory (D) — allows you to print or display large blocks of memory information in HEX format.
- Load (L) — allows you to load hex format object files into SDK-86 memory from an Intellec.
- Transfer (T) — allows you to save contents of SDK-86 memory in a hex format object file in the Intellec.

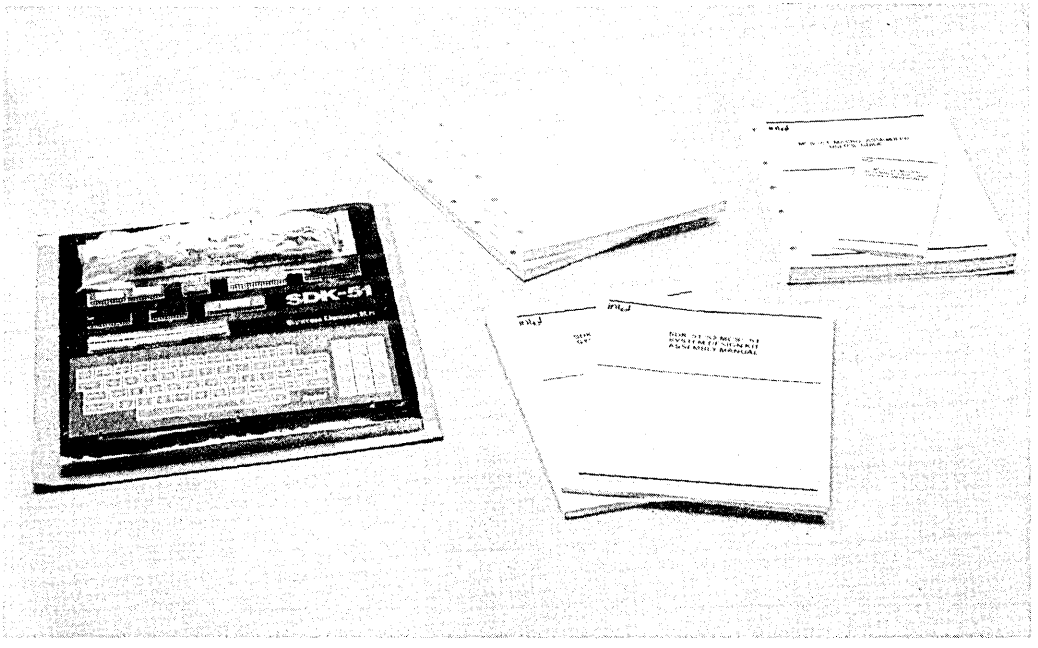


SDK-86/Intellec® Slave Mode Configuration

SDK-51
MCS®-51 SYSTEM DESIGN KIT FOR DESIGNS OF 8051/8052 SINGLE-BOARD MICROCOMPUTERS

- **Complete single-board microcomputer kit:**
  - **—Intel® 8031 CPU**
  - **—Intel 8032 CPU**
  - **—ASCII keyboard and 24-character alpha-numeric display**
  - **—Wire-wrap area for custom circuitry**
  - **—User-configurable RAM**
  - **—Serial and parallel interfaces**
- **Extensive system software in ROM:**
  - **—Single-line assembler and disassembler**
  - **—System debugging commands**
    - **Go**
    - **Step**
    - **Breakpoints**
- **Interface software:**
  - **—Serial port**
  - **—Audio cassette**
  - **—Intellec® system**
- **User's guide, assembly manual, and MCS®-51 design manuals**

The SDK-51 MCS®-51 System Design Kit contains all of the components required to assemble a complete single-board microcomputer based on either Intel's high-performance 8051 or 8052 single-chip microcomputer. SDK-51 uses the external ROM version of the 8051 (8031) and the 8052 (8032). Once you have assembled the kit and supplied +5V power, you can enter programs in MCS-51 assembly language mnemonics, translate them into MCS-51 object code, and run them under control of the system monitor. The kit supports optional memory and interface configurations, including a serial terminal link, audio cassette storage, EPROM program memory, and Intellec® development system upload and download capability.



SEPTEMBER 1984
ORDER NUMBER: 162549-002

FUNCTION DESCRIPTION

The SDK-51 is a kit that includes hardware and software components to assemble a complete MCS-51 family single-board microcomputer. Only common laboratory tools and test equipment are required to assemble the kit. Assembly generally requires 5 to 10 hours, depending on the experience of the user. See Figure 1 for a block diagram of the SDK-51/52 system.

The MCS®-51 Microcomputer Series

MCS-51 is a series of high-performance single-chip microcomputers for use in sophisticated real-time applications such as instrumentation, industrial control, and intelligent computer peripherals. The 8031, 8032, 8051, 8052, and 8751 microcomputers belong to the 8051 family, which is the first family in the MCS-51 series.

In addition to their advanced features for control applications, MCS-51 family devices have a microprocessor bus and arithmetic capability such as hardware multiply and divide instructions, which make the SDK-51 a versatile stand-alone microcomputer board.

The 8031, 8032, 8051, 8052, and 8751 CPUs

The 8031, 8032, 8051, 8052, and 8751 CPUs each combine, on a single chip, a 128×8 data RAM; 32 input/output lines; two 16-bit timer/event counters; a five-source, two-level nested interrupt structure; a serial I/O port; and on-chip oscillator and clock circuits. An 8051 block diagram is shown in Figure 2.

The 8031, the SDK-51's CPU, is a CPU without on-chip program memory. The 8031 can address 64K bytes of external program memory in addition to 64K bytes of external data memory. For systems requiring extra capability, each member of the 8051 family can be expanded using standard memories and the byte-oriented MCS-80 and MCS-85 peripherals. The 8051 is an 8031 with the lower 4K bytes of program memory filled with on-chip mask-programmable ROM while the 8751 has 4K bytes of ultraviolet light-erasable, electrically programmable ROM (EPROM).

The 8031 CPU operates at a 12 MHz clock rate, resulting in 4μs multiply and divide and other instructions of 1μs and 2μs.

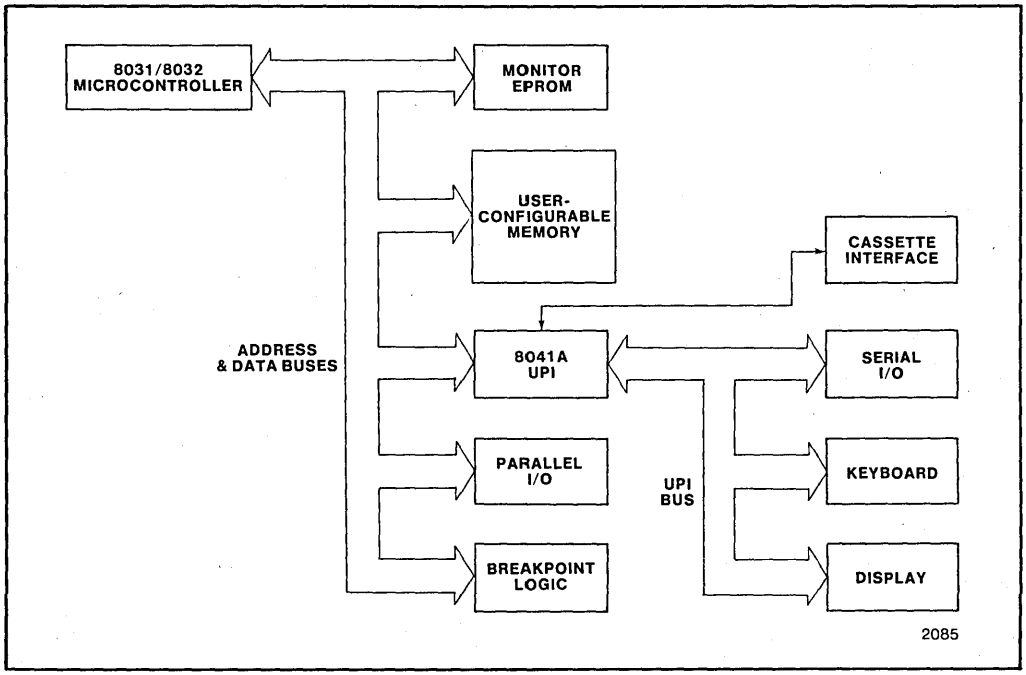


Figure 1. Block Diagram of the SDK-51 System Design Kit

162549-002

The 8032 and 8052 CPUs each combine, on a single chip, a 256×8 data RAM; 32 input/output lines; three 16-bit timer/event counters; a five-source, two-level nested interrupt structure; a serial I/O port; and on-chip oscillator and clock circuits. Refer to the shadowed areas in Figure 2.

For additional information on the 8051 family, see the *Microcontroller Handbook* or the *MCS®-51 Macroassembler User's Guide.*

System Software

A compact but powerful system monitor is contained in 8K bytes of pre-programmed ROM. The monitor includes system utilities such as command interpretation, user program debugging, and interface controls. Table 1 summarizes the SDK-51 monitor commands.

The ROM devices also include a single-line assembler and disassembler. The assembler lets you enter programs in MCS-51 assembly language mnemonics directly from the ASCII keyboard. The disassembler supports debugging by letting you look at MCS-51 instructions in mnemonic form during system interrogation.

Memory

The two 64K external memory spaces are combined into a single memory space which you can configure between program memory and data memory. The kit includes 1K-byte of static RAM. The board has space and printed circuitry for an additional 15K bytes of RAM and 8K bytes of ROM.

Two sets of ROM devices are included in the kit, one for the 8031 microcontroller for 8051 development, and one for the 8032 microcontroller for 8052 development.

User Interface

The kit includes a typewriter-format, ASCII-subset keyboard and a 24-character, alphanumeric LED display. The standard keyboard and display provide full access to all of the SDK-51's capabilities. All of the SDK-51 interfaces are controlled by a pre-programmed Intel 8041 Universal Peripheral Interface.

A 3×4 matrix keyboard can be jumpered to port 1 of the 8031/8032.

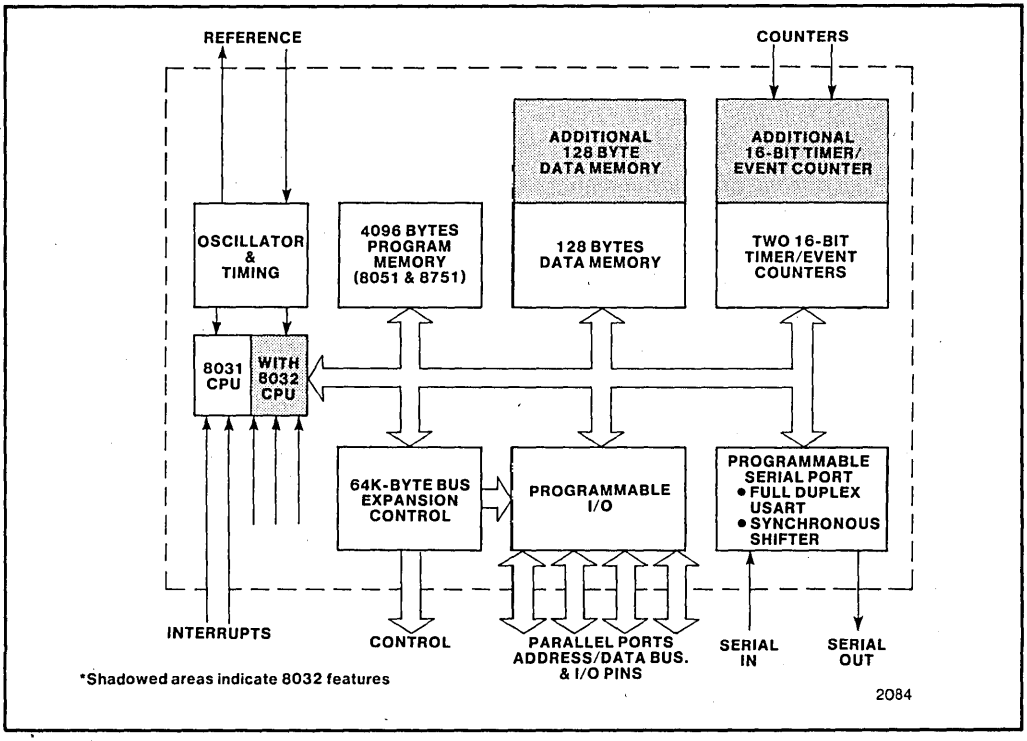


Figure 2. 8031/8032 Block Diagram

162549-002

Optional Interfaces

TERMINAL

An RS-232-compatible CRT or printing terminal or a current-loop-interface terminal may be used as a listing device by connecting it to the board's serial interface connector and supplying +12 and −12 volts to the board.

AUDIO CASSETTE

The kit includes hardware, software, and user's guide instructions to connect and operate an audio cassette tape recorder for low-cost program and data storage.

Table 1. SDK-51 Commands

Command	Operation
Set breakpoint	Define addresses for breaking execution.
Display cause	Ask the system why execution stopped.
Upload, download	Transfer files to and from Intellec® development system.
Save, load	Transfer files to and from optional cassette interface.
Set top of program memory	Define partition between program memory and data memory.
Set baud	Define baud rate value of serial port.
Display memory	Examine and change program memory or data location.
Assemble	Translate an MCS-51 assembly mnemonic into object code.
Disassemble	Translate program memory into MCS-51 assembly language mnemonics.
Go	Start execution between a selected pair of addresses.
Step	Execute a specified number of instructions.

INTELLEC® SYSTEM

An SDK-51 and an Intellec Model 800 or Series II development system with ISIS can upload and download files through the serial interface without adding any software to the Intellec system.

Parallel I/O

The kit includes an Intel 8155 parallel I/O device which expands the 8031/8032 I/O capability providing 32 dedicated parallel lines. Three 40-pin headers between the 8031 and 8155 devices and the wire-wrap area facilitate interconnections with the user's custom circuitry.

Debugging

Hardware breakpoint logic in the SDK-51 checks the address of a program or external data-memory access against values defined by the user and stops execution when it sees a "break" condition. After a breakpoint, you can examine and modify registers, memory locations, and other points in the system. A step command lets you execute instructions in a single-step mode.

Assembly and Test

The SDK-51 assembly manual describes hardware assembly in a step-by-step process that includes checking each hardware subsystem as it is installed. Building the system requires only a few common tools and standard laboratory instruments. Figure 3 shows an assembled SDK-51/52 system.

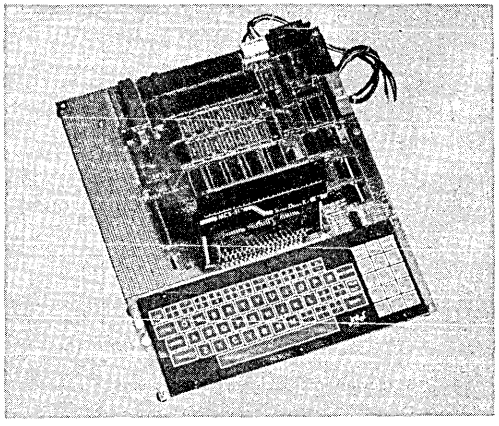


Figure 3. SDK-51 Assembled with Additional RAM and ROM Devices Installed

SPECIFICATIONS

Control Processor

Intel 8031 and 8032 microcomputers
12 MHz clock rate

Memory

RAM — 1K-byte static, expandable in 1K segments to 16K-byte with 2114 RAM devices; user-configurable as program or data memory.

ROM — Printed circuitry for 8K bytes of program memory in 4K segments using 2732A EPROM devices.

Interfaces

Keyboard — 51-key, ASCII subset typewriter format, 12-key (3 × 4) matrix

Display — 24-character, alpha-numeric

Serial — RS-232 with user-selectable baud rate. Printed circuitry for 100 baud 20 mA current loop interface. 8031 serial port.

Parallel — 32 lines, TTL compatible

Cassette — Audio cassette tape storage interface

Software

System monitor preprogrammed in on-board ROM MCS-51 assembler and disassembler preprogrammed in on-board ROM.
Interface control software preprogrammed in 8041's on-chip ROM.

Assembly and Test Equipment Required

Needle-nose pliers
Small Phillips screwdriver
Small diagonal wire cutters
Soldering pencil, ≤30 watts, 1/16" diameter tip
Rosin-core, 60-40 solder, 0.05" diameter
Volt-Ohm-Milliammeter, 1 meg-ohm input impedance
Oscilloscope, 1 volt/division vertical sensitivity, 200 μs/division sweep rate, single trace, internal and external triggering

Physical Characteristics

Length — 13.5 in. (34.29 cm)
Width — 12 in. (30.48 cm)
Height — 4 in. (10.16 cm)
Weight — 3 lb. (1.36 kg)

Electrical Characteristics

DC Power Requirement (supplied by user, cable included with kit)

Voltage	Current
+ 5V ±5%	3A
+12V±5%*	100 mA
−12V±5%*	100 mA

*±12 volts required only for operation with serial interface.

Environmental Characteristics

Operating Temperature — 0 to 40°C
Relative Humidity — 10% to 90%, non-condensing

Reference Manuals

SDK-51/52 MCS®-51 System Design Kit User's Guide
SDK-51/52 MCS®-51 System Design Kit Assembly Manual
SDK-51 MCS®-51 System Design Kit Monitor Listing Manual
SDK-52 MCS®-51 System Design Kit Monitor Listing Manual
MCS®-51 Macro Assembler User's Guide
MCS®-51 Macro Assembly Language Pocket Reference

ORDERING INFORMATION

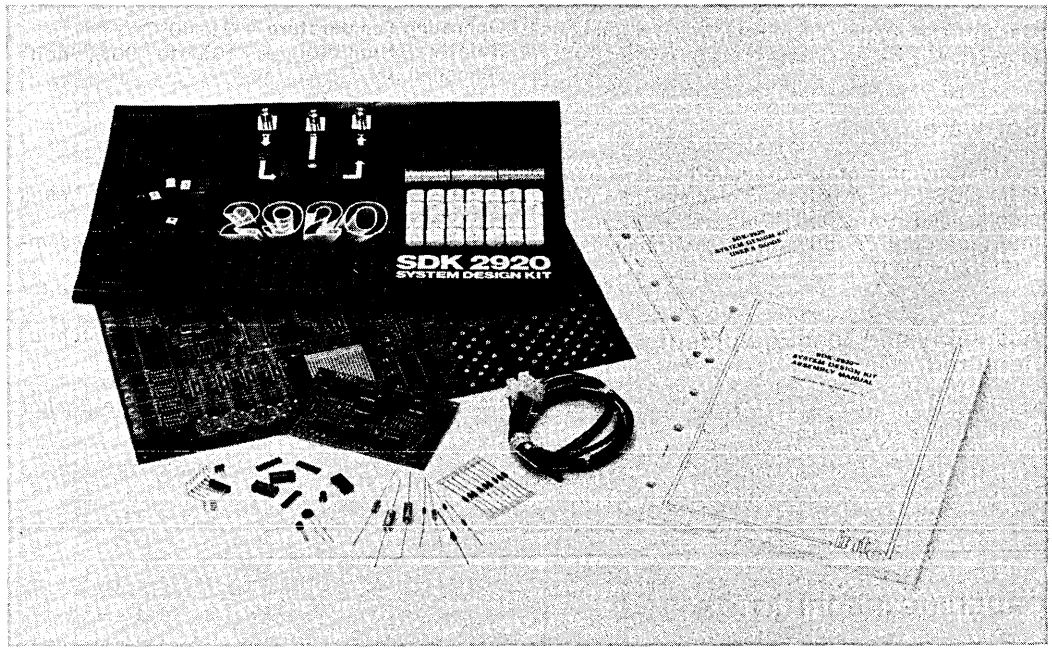
Part Number	Description
MCI-51-SDK	MCS-51 System Design Kit

162549-002

SDK-2920
2920 SYSTEM DESIGN KIT

- **Complete 2920 program development:**
  - **2920 assembly language keyboard**
  - **Single-line assembler/disassembler**
  - **24-character, alphanumeric display**
  - **2920 memory display and modify**
  - **List program memory to line printer with symbol table**
- **Decimal-to-binary conversion program**
- **File handling capabilities:**
  - **Up/down load of object file to Intellec or audio cassette**
  - **Up load source file with symbol table to Intellec**
  - **2920 EPROM programming**
- **Real-time execution of a programmed 2920**
- **Breadboarding area**

The SDK-2920 contains all of the components required to assemble a complete single board microcomputer system for programming and evaluation of the 2920 Analog Signal Processor. The 8085/8041A microcomputer-based program development section allows you to immediately enter programs in 2920 assembly mnemonics, translate them to 2920 object code, and program the on-board 2920 EPROM. The kit supports basic filing options such as up/down loading to/from an Intellec, audio cassette, and line printer. The SDK-2920 also provides the user with a 2920 run mode section allowing real-time execution of a programmed 2920. This section comes complete with BNC connectors and Intel's 2912 PCM line filters required for one input and one output network. The kit supports optional input and output circuitry on the run mode section.



FUNCTIONAL DESCRIPTION

The SDK-2920 is a kit which includes all the necessary hardware and software components to assemble, using common laboratory tools and test equipment, a complete single board 2920 programming and evaluation aid. Assembly generally requires 4 to 8 hours, depending on the experience of the user.

The 2920 Signal Processor

The Intel® 2920 Signal Processor is a programmable, single chip analog and digital signal processor specifically designed to replace analog subsystems in real-time processing applications. Its instruction set plus the high precision (25 bits) digital arithmetic logic unit provides the capability to implement very complex subsystems. Typical functions performed by the 2920 include: lowpass and bandpass filters with up to 20 complex pole and/or zero pairs; threshold detectors; limiters; rectifiers; up to 25-bit multiplication and division; approximations to nonlinear functions such as square law and logarithm; logical operations; input and output multiplexing of signals; logical outputs for decision type processing; and analog outputs for multifrequency oscillators, waveform generators, etc. In addition, several 2920's may be cascaded for very complex processing applications with no loss in throughput rate.

Tables 1 and 2 show the 2920 instruction set and op codes.

Table 1. Shift Op Codes

Operation	Mnemonic	Op Code				Scale Factor
		3	2	1	0	
Shift Right 13 Bits	R13	1	1	0	0	2^{-13}
Shift Right 12 Bits	R12	1	0	1	1	2^{-12}
•	•	•	•	•	•	•
•	•	•	•	•	•	•
•	•	•	•	•	•	•
Shift Right 1 Bit	R01	0	0	0	0	2^{-1}
No Shift	R00	1	1	1	1	1
Shift Left 1 Bit	L01	1	1	1	0	2
Shift Left 2 Bits	L02	1	1	0	1	4

System Software

A compact but powerful system monitor is contained in 6K bytes of preprogrammed ROM. The monitor includes system utilities such as command interpretation, user program debugging, and interface controls.

The monitor ROM devices also include a single-line assembler and disassembler. The assembler lets you enter programs in 2920 assembly language mnemonics. The disassembler supports debugging by letting you look at or change either hexadecimal values or 2920 instructions during program interrogation.

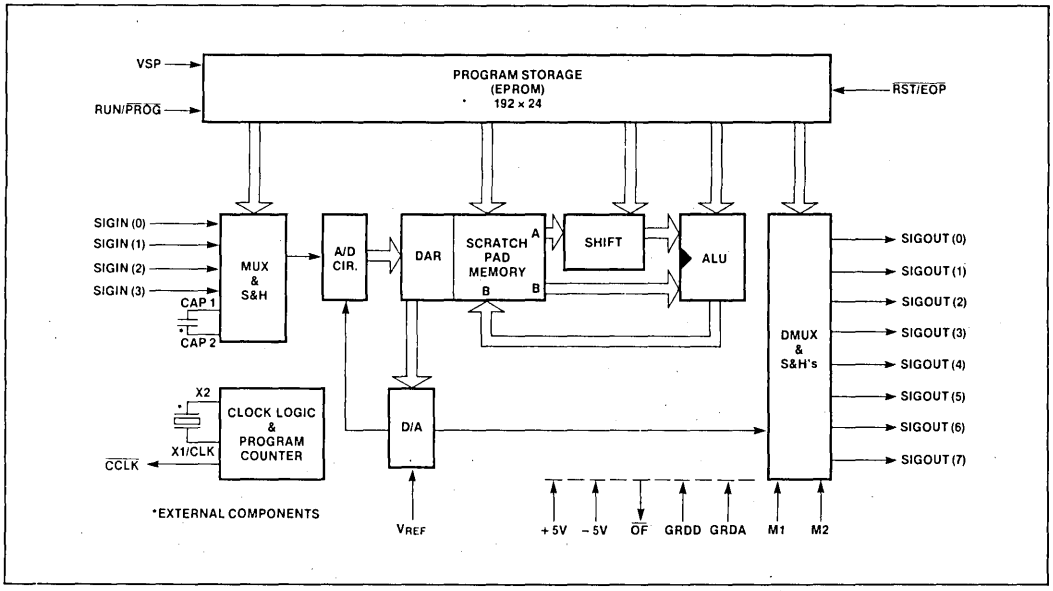
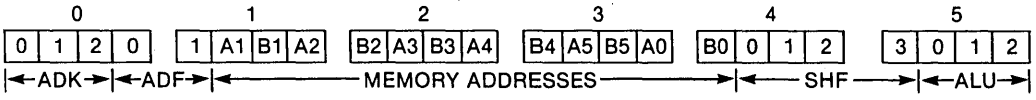


Figure 1. 2920 Function Block Diagram (Run Mode)

Table 2. Instruction Set and Op Codes

Mnemonics	Op Codes[1]			Operations	Notes
Code Condition	ALU	ADF	ADK		
Digital Instructions	**2,1,0**	**1,0**	**2,1,0**		
ADD	110	↑	↑	$(A \times 2^N) + B \rightarrow B$	
SUB	101	|	|	$B - (A \times 2^N) \rightarrow B$	
LDA	111	|	|	$(A \times 2^N) + 0 \rightarrow B$	
XOR	000	|	|	$(A \times 2^N) \oplus B \rightarrow B$	
AND	001	|	|	$(A \times 2^N) \cdot B \rightarrow B$	
ABS	011	|	|	$\lvert(A \times 2^N)\rvert \rightarrow B$	
ABA[11]	100	[3]	[3]	$\lvert(A \times 2^N)\rvert + B \rightarrow B$	
LIM	010	|	|	Sign(A) → ± F.S. → B[4]	
ADD CND( )[2]	110	|	|	$(A \times 2^N) + B \rightarrow B$ IFF DAR(K) = 1 $B \rightarrow B$ IFF DAR(K) = 0	
SUB CND( )[2,8]	101	|	|	$B - (A \times 2^N) \rightarrow B$ & CY → DAR(K) IFF $CY_P = 1$ $B + (A \times 2^N) \rightarrow B$ & CY → DAR(K) IFF $CY_P = 0$	[5]
LDA CND( )[2]	111	|	|	$(A \times 2^N) \rightarrow B$ IFF DAR(K) = 1 $B \rightarrow B$ IFF DAR(K) = 0	
ABA[11] CND( )[9]	100	|	|	$(A \times 2^N) + B \rightarrow B$	
XOR CND( )[9]	000	|	|	$(A \times 2^N) \oplus B \rightarrow B$	
Analog Instructions					
IN(K)	↑	00	0-3	Signal sample from input channel K	
OUT(K)	|	10	0-7	D/A to output channel K	
CVTS	|	00	6	Determine sign bit	
CVT(K)	|	01	0-7	Perform A/D on bit K	
EOP	[7]	00	5	Program counter to zero	[6]
NOP	|	00	4	No operation	
CND(K)	|	11	0-7	Select bit K for conditional instructions	
CNDS	↓	00	7	Select sign bit for conditional instructions	



Note: The input pins for each nibble bit from left to right are D0, D1, D2, D3.

NOTES:

1. Op codes ALU and ADF are in binary notation, ADK is in decimal notation and represents the value "K" when appropriate.
2. CND( ) can be either CND(K) or CNDS testing amplitude bits or the sign bit of the DAR respectively.
3. Determined by analog instructions below.
4. B is set to full scale (F.S.) amplitude with the same sign as the "A" port operand.
5. The previous carry bit (CY_P) is tested to determine the operation. The present carry bit (CY) is loaded into the Kth bit location of the DAR. "Present carry (CY) is generated independent of overflow. It will represent the carry (CY) of a calculated 28-bit result."
6. EOP will also enable overflow correction if it was disabled during a program pass. The EOP must occur in ROM location 188.
7. Determined by digital instructions above.
8. For SUB CNDS Operation $\overline{CY} \rightarrow$ DAR(S).
9. Does not affect DAR. In this case, CND is used with XOR/ABA to enable/disable the ALU overflow saturation algorithm. Use of either instruction causes the ALU output to roll over rather than go to full scale with sign bit preserved. An EOP instruction will also enable the ALU overflow saturation algorithm.
10. Clarification of CY_{OUT} sense for certain operations. For LDA, XOR, AND, ABS; $CY_{OUT} \rightarrow 0$.

Memory

The kit contains 1.25K bytes of RAM for 2920 program development. The RAM is used as 2920 program memory for up to a 192-instruction 2920 program. The RAM space is also used for a symbol table up to 40 user defined symbols.

User Interface

The kit includes a function and hex keyboard and a formatted 24-character, 18-segment display for easy 2920 code entry. The interactive keyboard and display enables the system monitor to step the user through a command entry sequence with

the friendliness of a menu-driven operating system.

Optional Interfaces

An RS-232 or 20 mA current loop compatible CRT or printer may be used as a listing or file storage device by connecting it to the board's serial interface connector and supplying + 12 and − 12 volts to the board. In addition, the kit provides an audio cassette interface, allowing the use of an audio cassette as a mass storage device.

Debugging

Program development is made easy by use of interactive error messages that will inform the user of illegal entries at the time of program development. Syntax errors are detected prior to EPROM programming, giving the user the option to continue or abort the programming.

The run-mode section allows the user to execute a programmed 2920 in real time, with his own input stimulus and output circuit or instrumentation. The kit is supplied with the 2920 (600 ns instructions) and a 6.67 M H_2 crystal.

Assembly and Test

The SDK-2920 assembly manual describes assembly in a step-by-step process that includes checking segments of hardware as they are installed. Building the system requires only a few common tools and standard instruments.

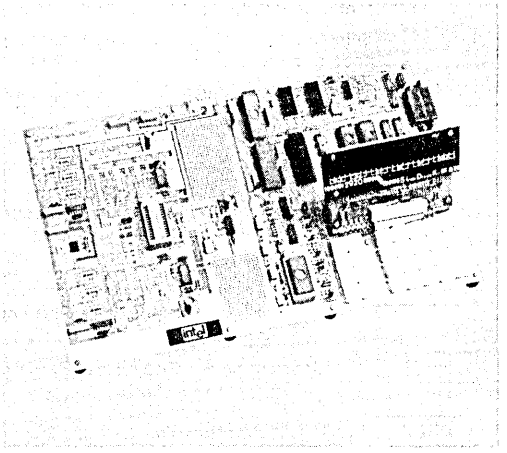


Figure 3. Assembled SDK-2920

CONTROL SECTION

RESET	LIST LOAD
HEX/ASM →	EDIT ←
INSRT NEXT	DEL PREV
SHIFT	CONV CR

2920 DATA KEYS

KP KN	ADD C	ABS D	ABA E	AND F
+/− R	SUB 8	LDA 9	LIM A	XOR B
L .	IN 4	OUT 5	CVTS 6	CVT 7
DAR Y	NOP 0	CNDS 1	CND 2	EOP 3

Figure 2. Keyboard Arrangement

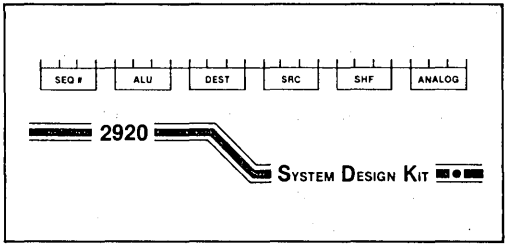


Figure 4. Display Layout

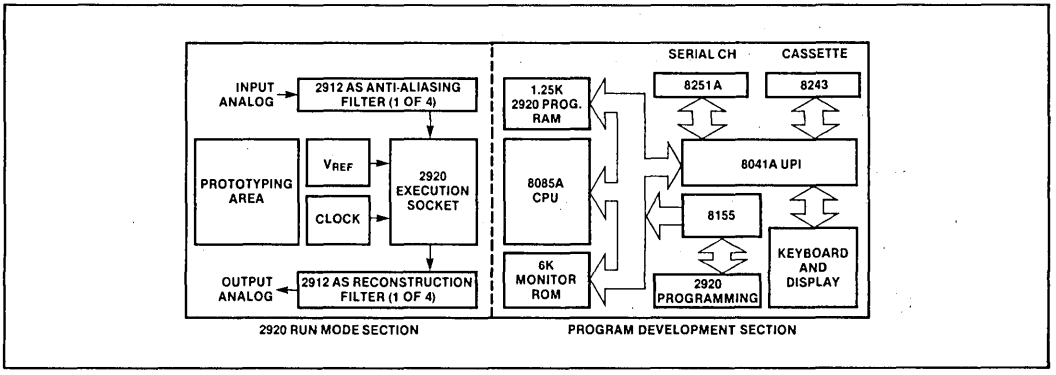


Figure 5. SDK-2920 Functional Block Diagram

Table 3. SDK-2920 Control Commands

RESET	— Sets the monitor to its initialization program and responds to the selection of one of the four modes. The display will prompt the user with EDIT? LOAD? LIST? CONV?.
SHIFT	— Selects the upper case characters or functions.
EDIT	— Selects the edit mode, allowing for 2920 program entry and/or modification. The commands available in the edit mode are shown below in Table 4.
LOAD	— Selects the load mode, providing for up/down loading to/from the RS-232, cassette, or the 20 mA current loop interfaces. It also provides for 2920 EPROM read, program and verify.
LIST	— Selects the list mode, providing for listing the 2920 program source code, symbol table, and 2920 hex code to a line printer via the RS-232 interface.
CONV	— Selects the decimal-to-binary-to-decimal conversion program.

Table 4. Edit Mode Commands

→	Cursor Right	The blinking cursor is moved right one position unless at the end of displayed field.
←	Cursor Left	Blinking cursor is moved left until the sequence number is encountered, then it skips to the left edge of the display.
NEXT	Next Instruction	The next 2920 instruction is displayed unless at end of memory.
PREV	Previous Instruction	The previous 2920 instruction is displayed unless at beginning of memory.
LIST	List Memory	Send disassembled 2920 instructions to serial port.
HEX/ASM	Mode Toggle	Toggle edit mode between symbolic assembly and hexadecimal.
INSRT	Insert Instruction	Expand the program in memory by one location and insert a NOP at current memory display address.
DEL	Delete Instruction	Contract the program in memory by one location and remove the instruction at the current memory display position.

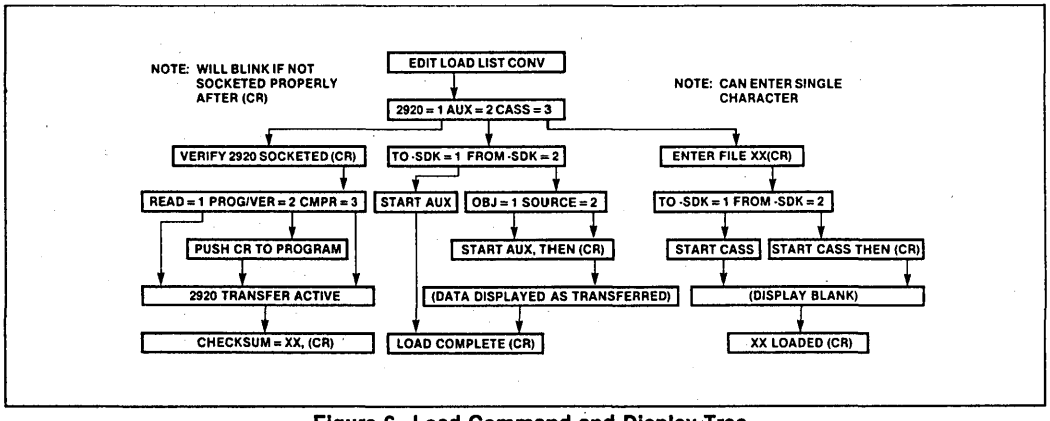


Figure 6. Load Command and Display Tree

SPECIFICATIONS

Control Processor

Intel 8085A microprocessor
6.144 MHz clock rate

Memory

RAM — 1.25K-byte static
ROM — 6K-byte

Interfaces

Keyboard — 28-key with shift, providing 54 functions and characters

Display — 24-character, 18-segment LED

Serial — RS-232 with user-selectable baud rate and 20 mA current loop

Cassette — Hardware and software for audio cassette tape storage interace

Software

System monitor preprogrammed in ROM

2920 assembler and disassembler preprogrammed in ROM

Interface control software preprogrammed in 8041 on-chip ROM

Assembly and Test Equipment Required

- Needle-nose pliers
- Small Phillips screwdriver
- Small flat-blade screwdriver
- Small diagonal wire cutters
- Soldering pencil, <30 watts, 1/16″ diameter tip
- Rosin-core, 60-40 solder, 0.05″ diameter
- Volt-ohm-milliameter, 1 meg ohm input impedance
- Oscilloscope, 1 volt/division vertical sensitivity, 200 μs/division sweep rate, single trace, internal and external triggering

Physical Characteristics

Length — 16 in. (40.64 cm)
Width — 10 in. (25.40 cm)
Height — 4 in. (10.16 cm)
Weight — 3 lb (1.36 kg)

Electrical Characteristics

DC Power Requirements (supplied by user, cables included with the kit)

Program Development Section:

Voltage	Current	Comments
+5V ±5%	1.0A	Required for program development
+12V ±5%	100 mA	Required for 2920 EPROM programming and RS-232 interface
−12V ±5%	100 mA	Required for RS-232 interface

Run Mode Section:

Voltage	Current	Comments
+5V ±5%	300 mA	Required for operation as supplied
	200 mA	Required for each additional 2912/74LS324 pair
−5V ±5%	250 mA	Required for operation as supplied
	200 mA	Required for each additional 2912/74LS324 pair

Environmental Characteristics

Operating Temperature — 0 to 40°C
Relative Humidity — 10% to 90% non-condensing

Reference Manuals

SDK-2920 System Design Kit User's Guide
SDK-2920 System Design Kit Assembly Manual
2920 Analog Signal Processor Design Handbook

ORDERING INFORMATION

Part Number	Description
MCI-20-SDK	2920 System Design Kit

Third Party Software

8

MICROSOFT*, INC. BASIC-80 INTERPRETER SOFTWARE PACKAGE

- **Compatible with other Microsoft BASIC compilers and interpreters**
- **Sophisticated string handling and structured programming features for applications development**
- **Direct transfer of BASIC programs to the 8085, 8086 and 8088**
- **Random and sequential file manipulation where random file record length is user-definable**
- **Read or write memory location capabilities**
- **Meets the requirements for the ANSI subset standard for BASIC, and supports many enhancements**
- **Extensive text editing features built-in**
- **Automatic line number generation and renumbering**
- **Supports assembly language subroutine calls**
- **Trace facilities for easier debugging**

BASIC Release 5.0 from Microsoft is an extensive implementation of BASIC. Microsoft BASIC gives users what they want from a BASIC—ease of use plus the features that are comparable to a minicomputer or large mainframe.

BASIC-80 meets the requirements for the ANSI subset standard for BASIC, as set forth in document BSRX3.60-1978. It supports many unique features rarely found in other BASICs.

FEATURES

—Four variable types: Integer (−32768, +32767), String (up to 255 characters), Single-Precision Floating Point (7 digits), Double-Precision Floating Point (16 digits).

—Trace facilities (TRON/TROFF) for easier debugging.

—Error trapping using the ON ERROR GOTO statement.

—PEEK and POKE statements to read or write any memory location.

—Automatic line number generation and renumbering, including reference line numbers.

—Matrices with up to 255 dimensions.

—Boolean operators OR, AND, NOT, XOR, EQV, IMP.

—Formatted output using the PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.

—Direct access to I/O ports with the INP and OUT functions.

—Extensive program editing facilities via EDIT command and EDIT mode subcommands.

—Assembly language subroutine calls (up to 10 per program) are supported.

—IF/THEN/ELSE and nested IF/THEN/ELSE constructs.

—Supports variable-length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE.

MAY 1983
AFN-02086C

BASIC-80 Commands, Statements, Functions

AUTO	RENUM	NAME
LIST	WIDTH	SAVE
NULL	CONT	EDIT
TROFF	MERGE	NEW
CLEAR	RUN	TRON
LOAD	DELETE	

Program Statements

CALL	RANDOMIZE	RETURN
GOSUB	COMMON	WAIT
END	DEF FN	ON GOSUB
GOTO	ERROR	DIM
STOP	POKE	FOR/NEXT/
WHILE/	RESUME	STEP
WEND	SWAP	IF/THEN/
CHAIN	DEFDBL	ELSE
DEF USR	DEFSTR	ON ERROR
LET	DEFSNG	GOTO
REM	DEFINT	OPTION BASE

Input/Output Statements and Functions

CLOSE	GET	NAME
KILL	POS	PUT
OUT	FIELD	EOF
RESTORE	LSET/RSET	SPC
READ	PRINT	INKEY$
TAB	USING	INPUT
DATA	LOC	OPEN
LINE	MKI$	CVD
INPUT	MKS$	CVI
PRINT	MKD$	CVS
WRITE	LLIST	
LPRINT	LPOS	

Arithmetic Functions

ABS	SIN	LOG
INT	CDBL	FIX
SGN	CSNG	COS
ATN	CINT	RND
EXP	SQR	TAN

String Functions

ASC	STR$	INSTR
LEN	HEX$	RIGHT$
STRING$	OCT$	MID$
CHR$	VAL	SPACE$
LEFT$		

Operators

=	*	XOR
^	<=	NOT
<	+	EQV
>	< >	MOD
-	\	IMP
/	>=	OR
		AND

Special Functions

ERL	ERR	VARPTR
USR	FRE	PEEK

SPECIFICATIONS

Operating Environment

The standard disk version of Microsoft BASIC-80 occupies 24K bytes of memory. Microsoft BASIC-80 Interpreter is compatible with Intel's ISIS operating system or CP/M* operating system.

Required Hardware

Intellec Microcomputer Development System

—iPDS (Personal Development System)
—minimum of 1 diskette drive

Required Software

ISIS Operating System or CP/M Operating System.

Documentation Package

One copy of each manual is supplied with the software package.

Description
BASIC-80 Reference Manual
BASIC Reference Book

ORDERING INFORMATION

Order Code	Description
SD102CPM80F	Microsoft BASIC-80 Interpreter Software Package, CP/M version (Double-Sided, Double Density 5¼" Floppy) iPDS format
SD102ISS80F	Microsoft BASIC-80 Interpreter Software Package, ISIS version (Double-Sided, Double Density 5¼" Floppy) iPDS format

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required.
*Microsoft is a trademark of Microsoft, Inc.
*CP/M is a registered trademark of Digital Research, Inc.
*MP/M-II is a trademark of Digital Research, Inc.

MICROSOFT*, INC. BASIC-80 COMPILER SOFTWARE PACKAGE

- **Produces highly optimized, true machine code**
- **Compiled programs are fast and compact because of extensive optimizations performed during compilation**
- **Supports all the commercial language features of the Microsoft BASIC interpreter (except direct mode commands)**
- **Supports double-precision transcendental functions**
- **Machine code for application programs may be placed on diskette, ROM, or other Media**
- **Provides source program security because only compiled code need be distributed to end-users**
- **Loader format identical to Microsoft's MACRO-80 assembler, COBOL-80 compiler, and FORTRAN-80 compiler: Compiled BASIC programs can be loaded and linked with any of these languages**

Microsoft's BASIC-80 compiler is a powerful tool for programming BASIC applications or microprocessor system software. The single-pass compiler produces extremely efficient, optimized 8080 machine code that is in Microsoft-standard, relocatable binary format. Execution speed is typically 3–10 times faster than Microsoft's BASIC-80 interpreter.

FEATURES

Optimized, Compatible Object Code

The BASIC compiler produces object code that is highly optimized for speed and space, relocatable, and compatible with other Microsoft software products. The loader format is identical to that of the MACRO-80 assembler, COBOL-80 compiler and FORTRAN-80 compiler, so programs written in any one of these four languages can be loaded and linked together. The compiler can also provide a formatted listing of the machine code that is generated.

Compiled programs are fast and compact due to extensive optimizations performed during compilation:

—Expressions are reordered to minimize temporary storage and (wherever possible) to transform floating point division into multiplication.

—Constant multiplications are distributed to allow more complete constant folding.

—Constants are folded wherever possible. The expression reordering finds "hidden" constant operations.

—Peephole optimizations are performed, including strength reduction.

—The code generator is template-driven, allowing optimal sequences to be generated for the most commonly used operations.

—String operations and garbage collection are extremely fast.

Compiled BASIC-80 programs are the ideal end product for BASIC applications' programmers. The machine code for any application program may be placed on a diskette, ROM, or other media. The program not only runs faster than with the interpreter, but the BASIC source program need not be distributed. Thus the original application program is protected from unauthorized alteration.

Language Features

The Microsoft BASIC-80 Compiler supports all the commercial language features of Microsoft BASIC-80, except those commands that are not usable in the compiler environment (i.e., direct mode commands such as LOAD, AUTO, SAVE, EDIT, etc.). That means you get the BASIC language compatible with other Microsoft BASIC packages.

MAY 1983
ORDER NUMBER:210247-002

In addition, the compiler supports double-precision transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR), %INCLUDE, CHAIN and COMMON. The %INCLUDE compiler directive brings another source file into the compilation without retyping the main source file.

BRUN Runtime Module

The BRUN runtime module contains the most common runtime routines needed for most programs. Using the BRUN module provides faster link loading of program modules and allows the user to link much larger programs because the runtime routine library does not reside in memory during linking. The executable files saved on disk are also much smaller since the BRUN module exists separately.

Utility Software Package

The BASIC-80 package includes the Microsoft Utility Software Package. The Utility Software Package includes the MACRO-80 macro assembler, the LINK-80 linking loader and the CREF-80 Cross-Reference Facility. Refer to the description of the Microsoft Utility Software Package for full details.

SPECIFICATIONS

Operating Environment

The BASIC Compiler requires a minimum of 34K bytes of memory (exclusive of the operating system). Microsoft recommends that 48K bytes be available for compiling medium to large programs. The compiler itself occupies about 28K bytes. At runtime, the BRUN module occupies approximately 15.5K bytes. If, as an option, the BRUN module is not used, the runtime library occupies 8K–18K bytes.

Required Hardware

Intellec Microcomputer Development System
—iPDS (Personal Development System)
—minimum of 1 diskette drive

Required Software

CP/M* Operating System

Documentation Package

One copy of each manual is supplied with the software package.

Description
BASIC Compiler User's Manual
BASIC-80 Reference Manual
BASIC Reference Book
Microsoft Utility Software Manual

(Specify by Alpha Character when ordering.)

ORDERING INFORMATION

Order Code	Description
SD124CPM80F	Microsoft BASIC-80 Compiler Software Package, CP/M version (iPDS Format)

SUPPORT:

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required.
*Microsoft is a trademark of Microsoft, Inc.
*CP/M is a registered trademark of Digital Research, Inc.
*MP/M-II is a trademark of Digital Research, Inc.

DIGITAL RESEARCH INC.
CP/M* 2.2 OPERATING SYSTEM

- **High-performance, single-console operating system**
- **Simple, reliable file system matched to microcomputer resources**
- **Table-driven architecture allows field reconfiguration to match a wide variety of disk capacities and needs**
- **Extensive documentation covers all facts of CP/M applications**
- **More than 1,000 commercially available compatible software products**
- **General-purpose subroutines and table-driven data-access algorithms provide a truly universal data management system**
- **Upward compatibility from all previous versions**

CP/M 2.2 is a monitor control program for microcomputer system and application uses on Intel 8080/8085-based microcomputer. CP/M provides a general environment for program execution, construction, storage, and editing, along with the program assembly and check-out facilities.

The CP/M monitor provides rapid access to programs through a comprehensive file management package. The file subsystem supports a named file structure, allowing dynamic allocation of file space as well as sequential and random file access. Using this system, a large number of distinct programs can be stored in both source- and machine-executable form.

CP/M also supports a powerful context editor, Intel-compatible assembler, and debugger subsystems. Nearly all personal software programs can be bought configured to run under CP/M, several of which are available from Intel.

FEATURES

CP/M is logically divided into four distinct modules:

BIOS—Basic I/O System

—Provides primitive operations for access to disk drives and interface to standard peripherals (teletype, CRT, paper tape reader/punch, bubble memory, and user-defined peripherals)

—Allows user modification for tailoring to a particular hardware environment

BDOS—Basic Disk Operating System

—Provides disk management for one to sixteen disk drives containing independent file directories

—Implements disk allocation strategies for fully dynamic file construction and minimization of head movement across the disk

—Uses less than 4K of memory allowing plenty of memory space for applications programs

—Uses less than 4K of memory

—Makes programs transportable from system to system

—Entry points include the following primitive operations which can be programmatically accessed:

SEARCH	Look for a particular disk file by name
OPEN	Open a file for further operations
CLOSE	Close a file after processing
RENAME	Change the name of a particular file
READ	Read a record from a particular file
WRITE	Write a record to a particular file
SELECT	Select a particular disk drive for further operations

SEPTEMBER 1984
ORDER NUMBER:210268-004

CCP—Console Command Processor

—Provides primary user interface by reading and interpreting commands entered through the console

—Loads and transfers control to transient programs, such as assemblers, editors, and debuggers

—Processes built-in standard commands including:

ERA	Erase specified files
DIR	List file names in the directory
REN	Rename the specified file
SAVE	Save memory contents in a file
TYPE	Display the contents of a file on the console

TPA—Transient Program Area

—Holds programs which are loaded from the disk under command of the CCP

—Programs created under CP/M can be checked out by loading and executing these programs in the TPA

—User programs, loaded into the TPA, may use the CCP area for the program's data area

—Transient commands are specified in the same manner as built-in commands

—Additional commands can be easily defined by the user

—Defined transient commands include:

PIP	Peripheral Interchange Program —implements the basic media transfer operations necessary to load, print, punch, copy, and combine disk files; PIP also performs various reformatting and concatenation functions. Formatting options include parity-bit removal, case conversion, Intel hex file validation, subfile extraction, tab expansion, line number generation, and pagination
ED	Text Editor—allows creation and modification of ASCII files using extensive context editing commands: string substitution, string search, insert, delete and block move; ED allows text to be located by context, line number, or relative position with a macro command for making extensive text changes with a single command line
ASM	Fast 8080 Assembler—uses standard Intel mnemonics and pseudo operations with free-format input, and conditional assembly features
DDT	Dynamic Debugging Tool—contains an integral assembler/disassembler module that lets the user patch and display memory in either assembler mnemonic or hexadecimal form and trace program execution with full register and status display; instructions can be executed between breakpoints in real-time, or run fully monitored, one instruction at a time
SUBMIT	Allows a group of CP/M commands to be batched together and submitted to the operating system by a single command
STAT	Lists the number of bytes of storage remaining on the currently logged disks, provides statistical information about particular files, and displays or alters device assignments
LOAD	Converts Intel hex format to absolute binary, ready for direct load and execution in the CP/M environment
SYSGEN	Creates new CP/M system disks for back-up purposes
MOVCPM	Provides regeneration of CP/M systems for various memory configurations and works in conjunction with SYSGEN to provide additional copies of CP/M

BENEFITS

—Easy implementation on any computer configuration which uses an Intel 8080/8085 Central Processing Unit (see the CP/M-86 data sheet for CP/M applications on the iAPX86 CPU)

—iPDS version supports bubble memory option as an additional diskette drive. Also allows diskette duplication with a single drive

—Extensive selection of CP/M-compatible programs allows production and support of a comprehensive software package at low cost

—Field programmability for special-purpose operating system requirements

—Upward compatibility from previous versions of CP/M release 1

—Provides field specification of one to sixteen logical drives, each containing up to eight megabytes

—Files may contain up to 65,536 records of 128 bytes each but may not exceed the size of any single disk

—Each disk is designed for 64 distinct files—more directory entries may be allocated if necessary

—Individual users are physically separated by user numbers, with facilities for file copy operations from one user area to another

—Relative-record random-access functions provide direct access to any of the 65,536 records of an eight-megabyte file

SPECIFICATIONS

Hardware Required

—Model 800 with 720 kit

—DS 235 kit or MDS 225 with 720 kit (integral drive supported except as system boot device)

—iPDS Personal Development System
Optional:
RAM up to 64K

—Additional floppy disk drives

—Single density via 201 controller

—Bubble memory and optional Shugart 460 5¼" disk drive for iPDS

Documentation Package

Title

CP/M 2.2 documentation consisting of 7 manuals:
- An Introduction to CP/M Features and Facilities
- CP/M 2.2 User's Guide
- CP/M Assembler (ASM) User's Guide
- CP/M Dynamic Debugging Tool (DDT) User's Guide
- ED: A Context Editor for the CP/M Disk System User's Manual
- CP/M 2 Interface Guide
- CP/M 2 Alteration Guide

Shipping Media

(Specify by Alpha Character when ordering.)

A—single density (IBM 3740/1 compatible)

B—double density

F—double-sided, double density 5¼" floppy (iPDS format)

Order Code	Product Description
See Price List	CP/M (Control Program for Microcomputers) is a disk-based operating system for the Intel 8080/8085-based systems. CP/M provides a general environment for program development, test, execution and storage. CP/M storage is available via a comprehensive, named-file structure supporting both sequential and random access. CP/M support tools include a Text Editor, a debugger, and an 8080/8085 assembler.

SUPPORT:

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required.
*CP/M is a registered trademark of Digital Research, Inc.
*CP/M-86, MP/M, CP/NET and MP/NET are trademarks of Digital Research, Inc.

WordStar* WORD PROCESSING SOFTWARE

- **Powerful, reliable, and user-friendly word processing software package**
- **Six on-screen menus and ten Help menus provide quick command reference**
- **Printout enhancements provide numerous combined print functions**
- **Simple formatting commands including Hyphen-Help**
- **Streamlines text entry**
- **Horizontal scrolling for wide pages**
- **Wordwrap removes need to worry about right margin**
- **On-screen formatting displays text exactly as it will be printed**
- **All functions easily controlled despite differences in printers and consoles**

WordStar, a popular word processing program written for use under the CP/M† operating system, gives screen editing capabilities in an easy to learn and use format. The program is in use by programmers, and engineers for documentation and program entry, as well as managers and secretaries.

With WordStar, the user can easily make insertions and deletions, move or copy blocks of text, and search for and replace a string of text. WordStar will automatically reformat text upon command as these editing functions are performed.

Documents produced by WordStar can include any combination desired of pagination (page numbers), right and left justification, subscripts, superscripts, underlining, boldface type, overstrikes, crossouts, and even accents for use in foreign languages. Commands for all of these are entered with simple control-character keystrokes which are well documented in the program's six help menus.

All WordStar commands are easily executed using the CTRL key and the standard typewriter keys. Using the CTRL key, the function of standard keys can be changed to perform useful editing commands. The cursor-movement diamond (a group of standard keys on the keyboard) allows fast access to any area of text.

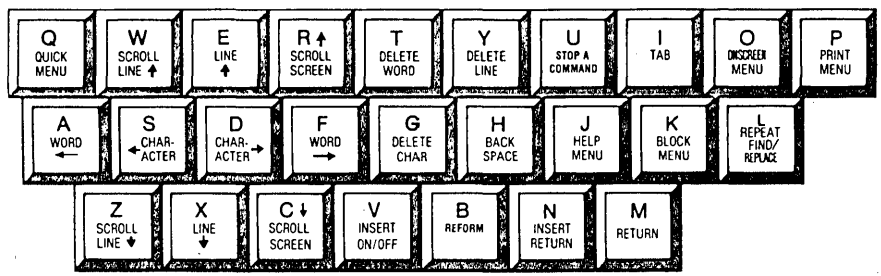


Figure 1. WordStar Keyboard Functions

MAY 1983

*WordStar MailMerge and SpellStar are trademarks of MicroPro International.

†CP/M is a registered trademark of Digital Research Inc.

ORDER NUMBER:210762-002

FEATURES

WordStar is designed to be simple for the novice to use, while remaining sophisticated enough to be appealing to even the most advanced user.

Standard typewriter keys are combined with the "Control" key to provide a wide variety of editing functions (Fig. 1). All cursor control is localized to the ten keys in the "Cursor-Movement Diamond" (Fig. 2), and the on-screen menu details the functions of the other keys, so the user can quickly find functions without memorizing them.

Wordwrap is a feature of WordStar that allows the typist to entirely disregard margins. When typed characters go beyond the right margin, WordStar brings the last full word down to the next line automatically. The only time the Return key needs to be used is between paragraphs. Margins can be automatically right and left justified both during and after entry.

Horizontal Scrolling give the flexibility in creating documents too wide to fit on the video screen. When Wordwrap is disabled and a line is being typed beyond normal screen width, the displayed lines are automatically scrolled offscreen to the left. A single keystroke can be used to move the lines back to their normal position. Editing functions can also use Horizontal Scrolling to examine and modify any part of a wide document.

The On-Screen Formatting feature displays the text on the screen as it will appear when it is printed. This allows the changing of margins, spacing, and other format variables without requiring the use of a number of intermediate printouts.

Hyphen-Help aids in reformatting by positioning the cursor over a word requiring hyphenation at the end of a line, and allowing the user to select a hyphenation point or decide not to hyphenate. Hyphens entered this way are "soft", and will not be printed if the document is reformatted and the hyphen is no longer required. Permanent or "hard" hyphens are inserted while typing and will always be printed.

WordStar's Find and Replace command allows the text to be scanned for a specified character string. Once the string is found it will be replaced quickly with the updated information. Options with this command allow the user to perform functions like finding the "n^{th}" occurrence, performing the operation "n" times, replacing the string without verification by the user each time, searching backward, or to compensate for differences in upper and lower case letters (i.e., at the beginning of a sentence).

Entire blocks of text can be marked at their beginning and ending, then moved to a new area as easily as moving the cursor. Different block control commands allow the duplication and deletion of blocks as well.

Column Move assists in the creation and editing of tables of data. With Column Move, a column can be taken from one table and moved to another table or to another place in the same table. Columns can also be easily duplicated or deleted.

Over 20 Page Formatting commands enable a range of functions from producing automatic page headings to overriding built-in parameters for line height and character width. Margins can be set and number of lines typed per page can be dictated via these very simple commands. These page commands are especially useful in long documents.

Decimal Tab is a feature that assists in aligning figures into columns. When a number is entered into a decimal tab position, it will be automatically aligned so that its decimal point is directly below the decimal point of the number on the line above.

Files can be combined with each other to form derivative documents. One file can be inserted at any point of another, beginning middle or end, with equal simplicity.

Print controls, single letters entered while editing to enhance the printout, permit the user of underscore, boldface, underlining, double-strike, superscript, subscript, overprint, and nearly any combination of the above. This facilitates the generation of mathematical formulas with subscripts and superscripts, and allows the text to include foreign words and phrases with accents above and below certain letters. Alternate character pitch, for italics, and even ribbon color selection can be controlled by WordStar if these options are available on the printer in use.

Can be used with MailMerge* to generate chained printing combining form letters with mailing lists. MailMerge allows names to be drawn from the address and inserted into the text of the form letter.

SpellStar* may also be used with WordStar to check the spelling in a document against both a 20,000-word standard dictionary and a user-generated supplemental dictionary which can be used to store names, buzz words, or abbreviations.

WordStar is easily adapted to nearly any video terminal and document printer, despite the wide variation of options and communication standards used by these devices. At installation time the user is prompted through a series of questions which configure that WordStar installation to fit the hardware at hand.

When an existing file is edited and saved, any previous version of the file is saved under the original filename as a .BAK extension (i.e., after updating a file entitled "LETTER" there would be two files, LETTER and LETTER.BAK on the diskette). When a document is to be updated, its latest extension is automatically used as input. Whenever a new .BAK extension is created, the older .BAK version is destroyed.

WordStar allows documents of almost any size to be entered and edited. A Memory Management feature automatically transfers text to and from mass storage if the document is too large to be held in main memory at one time.

There are six Main Menus (Fig. 3A-3F) and ten Help Menus to guide even the most inexperienced user through a WordStar editing session. The Main or On-Screen Menus are displayed at the top of the screen along with the text being edited. Should the user desire to fill the entire screen with text, the menu displays can be turned on and off as desired.

The ten Help Menus guide the user through the use of all editing functions from Moving Text to Paragraph Reform.

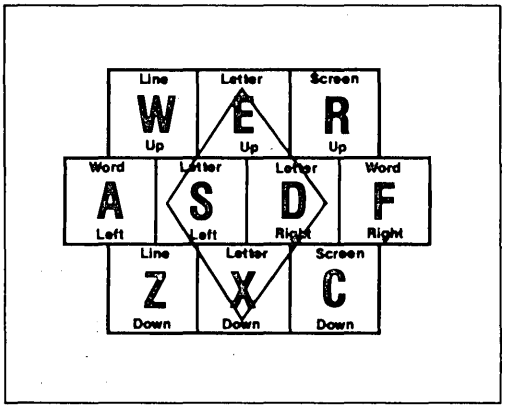


Figure 2. The Cursor Movement Diamond

```
        A:TEST.DOC  PAGE 1 LINE 1  COL 1              INSERT ON
                    < < <      M A I N   M E N U      > > >
*  *  Cursor  Movement  *  *|* Delete *|  * Miscellaneous * | * Other  Menus *
^S char left  ^D char right |^G  char  |^I Tab   ^B Reform  | (from Main only)
^A word left  ^F word right |DEL chr lf|^V Insert On or Off |^J Help  ^K Block
^E line  up   ^X line down  |^T word rt|^L Find/Replce again|^Q Quick ^P Print
*  *   Scrolling    *    *|^Y  line   |RETURN End paragraph|^O Onscreen
^Z line up    ^W line down  |          |^N  Insert a RETURN |
^C screen up  ^R screen down|          |^U  Stop a command  |
L----!----!----!----!----!----!----!----!----!----!----!--------R
```

Figure 3A. Main Menu: guide to the most frequently used commands. This menu —and all other menus—can be called up at any time, or dropped to allow full-screen viewing of the text.

```
^J      A:TEST.DOC  PAGE 1 LINE 1 COL 1              INSERT ON
                    < < <      H E L P   M E N U     > > >
                                          |                    | * Other  Menus *
 H Display and set the help level         | S Status line      | (from Main only)
 B Paragraph reform (CTRL B command)      | R Ruler line       |^J Help  ^K Block
 F Flags in rightmost column of screen    | M Margins and tabs |^Q Quick ^P Print
 D Dot commands, print ctrl(P command)    | P Place markers    |^O Onscreen
                                          | V Moving text      |Space bar returns
                                          |                    |you to Main Menu.
L----!----!----!----!----!----!----!----!----!----!----!--------R
```

Figure 3B. Help Menu: a directory of commands that control help levels and show reference information.

```
^Q      A:TEST.DOC  PAGE 1 LINE 1  COL 1              INSERT ON
                  < < <     Q U I C K   M E N U     > > >
*  *   Cursor Movement  *  *|* Delete *|  * Miscellaneous *  | * Other  Menus *
S left side    D right side |Y line  rt|F Find text in file  | (from Main only)
E top of scrn  X bottom scrn|DEL lin lf|A Find and  Replace  |^J Help  ^K Block
R top of file  C end of file|*  *  *  *|L Find  misspelling  |^Q Quick ^P Print
B top of block K end of block          |Q Repeat command or  |   ^O Onscreen
0-9  marker    Z up    W down          |  key  until  space  |Space bar returns
V last Find or block                   |  bar  or other key  |you to Main Menu.
L----!----!----!----!----!----!----!----!----!----!----!--------R
```

Figure 3C. Quick Menu: expanded cursor movement, deletion, find/replace commands, and place marker commands.

```
^K      A:TEST.DOC  PAGE 1 LINE 1  COL 1              INSERT ON
                  < < <     B L O C K   M E N U     > > >
 * Saving Files * |* Block Operations *| * File Operations *| * Other  Menus *
 S Save and resume|B  Begin   K  End    |R  Read    P  Print  | (from Main only)
 D Save—done      |H  Hide / Display    |O  Copy    E  Rename |^J Help  ^K Block
 X Save and exit  |C  Copy    Y  Delete |J  Delete             |^Q Quick ^P Print
 Q Abandon file   |V  Move    W  Write  |* Disk  Operations *|^O Onscreen
* Place  Markers *|N  Column    off (ON)|L Change logged disk|Space bar returns
0-9 Set/hide # 0-9|                     |F Directory on (OFF)|you to Main Menu.
L----!----!----!----!----!----!----!----!----!----!----!--------R
```

Figure 3D: Block Menu: instructions for using block and place markers, saving and printing a file, and inserting other files.

```
^O      A:TEST.DOC  PAGE 1 LINE 1  COL 1              INSERT ON
                  < < <  O N S C R E E N   M E N U  > > >
* Margins & Tabs * |* Line  Functions *| *  More Toggles  * | * Other  Menus *
L Set left  margin |C Center text       |J Justify   off (ON)| (from Main only)
R Set right margin |S Set line spacing  |V Vari-tabs off (ON)|^J Help  ^K Block
X Release  margins |                    |H Hyph-help off (ON)|^Q Quick ^P Print
I Set   N Clear tab|   *  Toggles  *    |E Soft hyph on (OFF)|^O Onscreen
G Set paragraph tab|W Wrd wrap off (ON)|D Prnt disp off (ON)|Space bar returns
F Ruler from line  |T Rlr line off (ON)|P Pge break off (ON)|you to Main Menu.
L----!----!----!----!----!----!----!----!----!----!----!--------R
```

Figure 3E. Onscreen Menu: functions that perform onscreen document formatting (such as line spacing, tabs, margins, justification, and wordwrap).

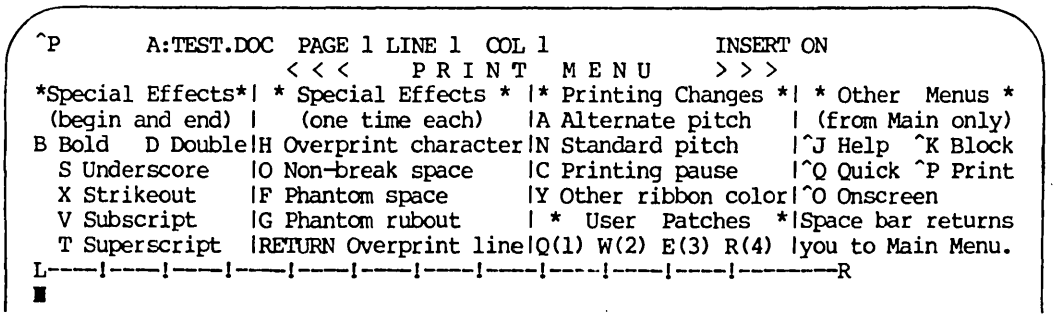


Figure 3F. Print Menu: special print control characters including subscripts, superscripts, boldface, double strike, and strikeout.

BENEFITS

WordStar is an advanced word-processing program that can turn any CP/M based personal computer into a sophisticated yet easy to learn and use text processor. It takes very little time for even the least-trained user to learn to productively generate documentation with WordStar.

The simplifying features of WordStar do not detract from its acceptance by advanced users. Menus and other features are designed to be unobtrusive when they are not needed. WordStar's sophistication means that it will not run out of horsepower as the user progresses, but will always be an appealing and highly productive tool.

With WordStar there is no question about the appearance of the printed output, since the text can be displayed on the screen exactly as it is to be printed.

Time savings when using WordStar will be considerable. Generation of new text is easier than by handwritten/typed means. When WordStar is used for program editing it supplies powerful features unavailable in other editors. With WordStar, both code and documentation can be generated at the same time within the same environment.

Table 1.

EDITING COMMAND INDEX

^	hold CTRL key, type letter
^A	cursor left word
^B	reform paragraph
^C	scroll up screenful
^D	cursor right character
^E	cursor up line
^F	cursor right word
^G	delete character right
^H	cursor left character
^I	tab
^J	help PREFIX
^K	editing PREFIX
^L	find/replace again
^M	(Same as RETURN)
^N	insert hard carriage return
^O	formatting PREFIX
^P	print control PREFIX
^Q	editing PREFIX
^R	scroll down screenful
^S	cursor left character
^T	delete word right
^U	interrupt
^V	insert on/off
^W	scroll down line
^X	cursor down line
^Y	delete line
^Z	scroll up line
^JB	explain reform
^JD	summarize print directives
^JF	explain Flags
^JH	set Help level
^JI	command index
^JM	explain tabs and Margins
^JP	explain Place markers
^JR	explain Ruler line
^JS	explain Status line
^JV	explain moVing text
^K0–^K9	set/hide marker 0–9
^KB	mark/hide Block beginning
^KC	Copy block
^KD	Done edit (save)
^KE	rEname file
^KF	File directory on/off
^KH	Hide/display marked block
^KJ	delete additional file
^KK	mark blocK end
^KL	change Logged disk
^KN	column mode on/off
^KO	cOpy file
^KP	Print
^KQ	abandon edit
^KR	Read additional file
^KS	Save and reedit
^KV	moVe block
^KW	Write block to additional file
^KX	save and eXit
^KY	delete block
^OC	Center cursor line
^OD	print control display on/off
^OE	soft hyphen Entry on/off
^OF	*margins & tabs from line*
^OG	paraGraph tab
^OH	Hyphen-Help on/off
^OI	set tab stop
^OJ	Justification on/off
^OL	set left margin
^ON	clear tab stop
^OP	Page break display on/off
^OR	set Right margin
^OS	set line Spacing
^OT	ruler display on/off
^OW	Wordwrap
^PA–^PZ	enter ^A–^Z
^PM	make next line overprint
^PO	enter non-break space
^Q0–^Q9	cursor to marker 0–9
^QA	find and replace
^QB	cursor Block beginning
^QC	cursor end file
^QD	cursor right end line
^QE	cursor top screen
^QF	Find
^QK	cursor blocK end
^QL	find misspelling
^QP	cursor Previous position
^QQ	repeat next command
^QR	cursor beginning of file
^QS	cursor left Side screen
^QV	cursor source
^QW	continuous downward scroll
^QX	cursor bottom of screen
^QY	delete to end line
^QZ	continuous upward scroll
^Qdel	delete to beginning line
^DEL	delete character left
^ESCAPE	error release
^LINE FEED	(same as ^J)
^RETURN	hard carriage return
^TAB	tab

Table 1. (Continued)

NO-FILE COMMANDS			
D	open a Document file	**O**	cOpy file
E	rEname file	**P**	Print
F	File directory on/off	**R**	Run program
H	set Help level	**S**	run SpellStar (optional)
L	change Logged disk	**X**	eXit to operating system
M	run MailMerge (optional)	**Y**	delete file
N	open a Non-document file		

SPECIFICATIONS

OPERATING ENVIRONMENT

Hardware Required

8080 or 8085 CPU
5¼" or 8" Diskette drive
Printer
64K Bytes of memory
Console with absolute cursor addressing
Note: Intellec Series II and III require iMDX-511

Optional hardware

Additional mass storage

Software Required

CP/M 2.2 operating system

DOCUMENTATION PACKAGE

Wordstar Training Guide
Wordstar Operator's Guide

Wordstar General Information Manual
Wordstar Reference Manual
Wordstar Installation Manual

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

Intel software license is required.

ORDERING INFORMATION

Description

WordStar word processing software package for use under the CP/M operating system

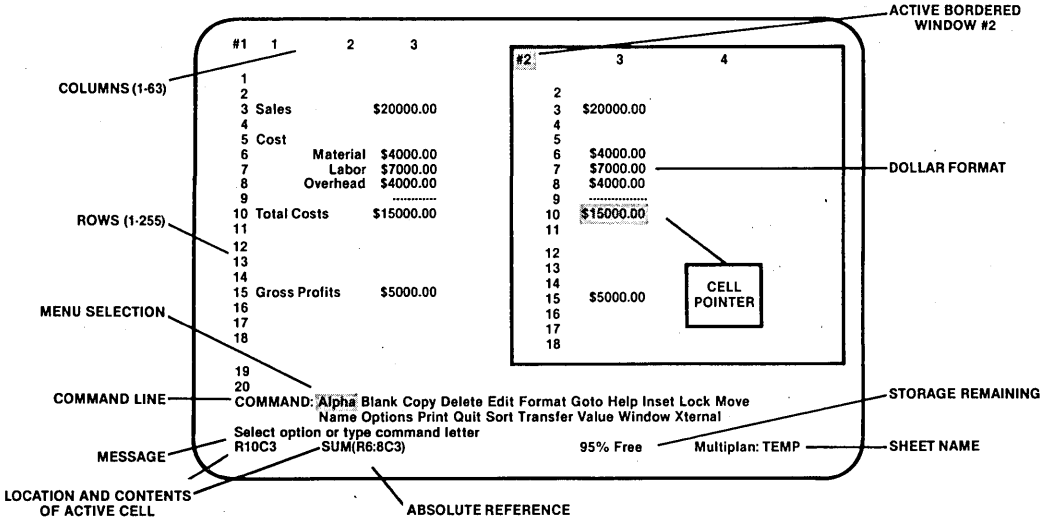
Order Code	Shipping Media
SD111CPM80ASU	A—Single-density 8" diskette
SD111CPM80BSU	B—Double-density 8" diskette
SD111CPM80FSU	F—iPDS Format 5¼" diskette

MICROSOFT*
MULTIPLAN* SPREADSHEET

- **Simplifies the design and use of very large spreadsheets, and multiple interrelated spreadsheets**
- **Automatically updates subtotals, totals, percentages, growth curves, etc**
- **Can perform multiple iterations to solve closed-loop problems**
- **Formulas automatically revised when reordering rows and columns in displays**
- **Can be used in time, monetary, and inventory budgeting**
- **Wide array of sophisticated functions to simplify formulas**
- **Cells and areas can be named for clarity**
- **Can reference and update several interrelated spreadsheets at once**
- **Simple to use, intuitive commands. Single keystroke command entry**
- **"Windows" allow several portions of large sheets to be viewed at once**
- **Contains the features of the most popular spreadsheet programs, as well as its contribution of new features**

Multiplan is a productivity tool designed to help the user to analyze data in spreadsheet format. As an aid to both business and personal needs, Multiplan is an extremely powerful modeling and planning tool.

Multiplan is easy to learn and use, yet its versatility is enhanced by the skill of the user. Multiplan allows the user to operate in as intuitive a way as possible, and its widespread capabilities allow accomplishment of a variety of tasks. Advanced users are unencumbered by simplifying features, and have enough power to satisfy their needs.



Typical Multiplan Screen Display

ORDER NUMBER:210767-002

FEATURES

— Names can be used to express "cells" (worksheet elements), or groups of cells. These names, in turn, can be used as parts of formulas and commands. Named areas can be combined in various ways for ease of use.

— A wide range of functions unique to Multiplan is available in addition to the functions typical to the most popular spreadsheet programs. These functions allow the user to select windows, sort data, draw from other worksheets, and a number of other important operations.

— Expressions can be clarified by the use of names as in "PROFIT = SALES – COSTS" rather than "R12C1 = R1C3 – R5C12".

— Active sheets can draw data automatically from inactive "supporting" sheets through the use of named cells and areas. This unique feature allows the user to streamline the processing of data, and to generate an entire pad full of interrelated spreadsheets.

— Multiplan offers a worksheet size of up to 255 rows by 63 columns, a broad worksheet simulator in which words, numbers, and formulas may be entered into information cells. Added to the access of data in inactive sheets, this large sheet size allows the user to perform very rigorous analyses in a minimum amount of time.

— With Multiplan the user gains the capability to plan against several different situations to allow comparison of one set of circumstances against another. A good example of this would be the generation of several sheets, one based on steady growth versus others based on several potential problems. This way, contingency planning will become less tedious and more effective.

— By altering a single critical number, the impact on other dependent numbers will be automatically updated to help the user observe sensitivities and interdependencies. This helps the user to plan resources efficiently, and schedule more effectively.

— Multiplan overcomes the limitations of paper worksheets by allowing the user to instantly move, insert, or delete entire rows or columns of data. The remaining rows, columns, or free space will expand or contract automatically as necessary, thereby eliminating the costly and tiresome work of typing or hand-printing the worksheet over and over.

— All commands can be invoked by a single keystroke and selections are menu driven. Multiplan even offers proposed responses to commands, to encourage its use by even the most unskilled user. Multiplan's commands, prompts, and messages, as well as the screen and keyboard, communicate with each other and the user directly and naturally to allow the untrained user to accomplish objectives easily.

— A special edit area helps the user to make additions and deletions quickly and easily.

— Up to eight windows are available to allow users to view different parts of a very large worksheet simultaneously. The windows can be aligned, scrolled together, opened, or closed at will.

— An iteration option allows the simulation of closed-loop problems involving mutually interdependant formulas. The number of iterations can be chosen, or iterations can continue until a given constraint is met.

— Formulas can be moved from one worksheet location to another without having to be rewritten by the user.

— Reference to a particular cell need not be in absolute terms, but can be expressed as a location relative to other cells. A formula containing this sort of relative reference may be copied into other cells and will be automatically changed to reflect its new position.

— The sheet display may be redesigned or formatted in various ways without affecting the data stored in Multiplan. Thus, the same data can be presented in different order in different reports with a minimum of effort.

Commands

The following is a brief list of commands available under Multiplan. All of these commands are invoked by the single keystroke of their first letter (i.e. "C" for Copy or "F" for Format) with the exception of eXternal, which is invoked by typing an "X."

Several of the commands offer a number of selections of operational modes, which are displayed when the command is invoked. In order to choose a mode, either press the TAB key until the cursor rests over the selected mode, then hit RETURN, or type the first letter of the selected mode, then hit RETURN.

For more detailed descriptions of the commands, please see the Multiplan User's Manual.

ALPHA
Replaces the contents of the active cell with a character string. If the active cell already contains a string, that string is the proposed response of the command, so that it can be edited.

BLANK
Deletes contents of all specified cells. Names are not affected; if a cell was referred to by a name before use of this command, that name will still apply.

COPY
Presents a choice of three ways of copying the contents of some cells into other cells. To duplicate one cell across several to its right, choose Right. To duplicate one cell across several below it, choose Down. To copy any cell or cells to any others, choose From.

DELETE
Presents a two-way choice to delete cells. To delete a row or rows, choose R. To delete a column or columns, choose C. To blank out the cells, use the Blank command.

EDIT
Makes contents of the active cell available for editing. Place the cell pointer on the cell to be edited and press E. The cell's contents are then placed on the command line for modification. The edit cursor is placed at the end of the current contents rather than highlighting the whole command, as is done for other defaults. If the cell contains a string, it is presented in double quotes. After having edited the cell's contents, press RETURN to put the changed contents back in the cell (or press ABORT to cancel any changes).

FORMAT
Presents a choice of three kinds of format adjustment. To set a specific format for a cell or group of cells, choose Cells. To set the width of a column or columns, choose Width. To set the default format—the format that applies wherever a specific format hasn't been set—choose Default.

GOTO
Presents a choice of ways to move the cell pointer over the sheet. To display a specific row and column, choose Row-col. To display a named area, choose Name.

HELP
Provides helpful information about Multiplan. When help is requested, the spreadsheet is replaced by text from the HELP file and the HELP command menu appears on the screen. Help is available in the areas of Applications, Commands, Editing, Formulas, and the Keyboard. The spreadsheet display is reinstated when the RESUME subcommand is entered.

INSERT
Presents a choice of ways to insert new cells into the sheet. To insert new rows choose Row. To insert new columns choose Column.

LOCK
Provides two ways to lock cells in protection against accidental change. Either individual cells or all cells containing formulas can be moved, deleted, formatted or sorted after having been locked, but their contents cannot be changed.

MOVE
Presents a choice of ways to move cells around the sheet. To move whole rows, choose Row. To move whole columns, choose Column.

Table 1. Multiplan Commands

ALPHA	— Replaces cell contents with a character string.
BLANK	— Clears cell contents.
COPY DOWN	— Used to fill a column with identical values.
COPY FROM	— Duplicates one or a number of cells to another location.
COPY RIGHT	— Used to make a row of identical values.
DELETE COLUMN	— Removes columns from the spreadsheet.
DELETE ROW	— Removes rows from the spreadsheet.
EDIT	— Allows editing of the contents of a single cell.
FORMAT CELLS	— Used to help align cells in a column.
FORMAT WIDTH	— Limits the width of all cells in a given column.
FORMAT DEFAULT CELLS	— Sets formats for all previously unformatted cells.
FORMAT DEFAULT WIDTH	— Sets formats for all previously unformatted columns.
FORMAT OPTIONS COMMAS	— Displays numbers with commas separating every third digit.
FORMAT OPTIONS FORMULAS	— Displays formulas instead of their values.
GOTO ROW-COL	— Moves the cell pointer to the specified row and column.
GOTO NAME	— Moves the cell pointer to the named area.
GOTO WINDOW	— Places the specified cell within the given window.
HELP APPLICATIONS	— Illustrates solutions to a number of common problems.
HELP COMMANDS	— Lists and describes all commands.
HELP EDITING	— Describes Editing functions.
HELP FORMULAS	— Gives Formula construction rules.
HELP KEYBOARD	— Explains special functions of the keyboard.
HELP NEXT	— Gives the next screenful of HELP text.
HELP PREVIOUS	— Gives the previous screenful from HELP call.
HELP RESUME	— Returns to the spreadsheet from HELP call.
HELP START	— Begins the HELP tutorial.
INSERT COLUMN	— Used to add a column to an existing spreadsheet.
INSERT ROW	— Used to add a row to an existing spreadsheet.
LOCK CELLS	— Protects the indicated cell from alteration.
LOCK FORMULAS	— Locks out alteration of all cells containing formulas or text.
MOVE COLUMN	— Changes the order of the columns on the sheet.
MOVE ROW	— Changes the order of the rows on the sheet.
NAME	— Assigns a name to a cell or number of cells.
OPTIONS	— Allows the user to disallow recalculation upon every change of a cell value, to mute the audible alarm, or to enable the Iteration option.
PRINT FILE	— Outputs the spreadsheet to a diskette file.
PRINT MARGINS	— Sets up the margins on the printed output.
PRINT OPTIONS	— Allows optional printing modes to be used.
PRINT PRINTER	— Prints the spreadsheet on the system's printer.
QUIT	— Ends the Multiplan session without saving the active sheet.
SORT	— Sorts a range of rows to put values in a specified column into ascending or descending numerical order.
TRANSFER CLEAR	— Clears the active sheet.
TRANSFER DELETE	— Deletes the specified file.
TRANSFER LOAD	— Loads a sheet from the disk file.
TRANSFER OPTIONS	— Modifies the context of the following transfer operation.
TRANSFER RENAME	— Renames the active sheet.
TRANSFER SAVE	— Saves the active sheet on diskette.
VALUE	— Enters a value or formula into the active cell.
WINDOW BORDER	— Changes the border of the specified window.
WINDOW CLOSE	— Removes a window from the screen.
WINDOW LINK	— Sets or breaks link for synchronized scrolling between windows.
WINDOW SPLIT HORIZONTAL	— Horizontally divides a window into two windows.
WINDOW SPLIT VERTICAL	— Vertically divides one window into two windows.
WINDOW SPLIT TITLES	— Divides one window into two or four which scroll together.
XTERNAL COPY	— Copies data from an inactive sheet to the active sheet.
XTERNAL LIST	— Displays the relationships between the active sheet and the other sheets.
XTERNAL USE	— Sets a substitute name for a supporting sheet.

Commands (Continued)

NAME
Assigns a name to a cell or area of cells. The name defined may then be used wherever a reference to that cell or area is needed in a command or formula.

OPTIONS
The Options command can be used to set and reset various options provided with Multiplan.

The Recalc option controls how often Multiplan performs formula calculations. If the option is on, Multiplan recalculates all formulas whenever a cell is changed. If the option is off, recalculation is done only when the Recalc control key is pressed or during Transfer Save.

The Recalc option has an effect on how quickly Multiplan finishes entering a new value in a cell. The length of time Multiplan takes to recalculate the sheet depends on how many cells are in use, and on the complexity of the formulas in them. When you want to make a number of entries on a busy sheet, turn the Recalc option off to get the quickest response. Turn it on again when you are interested in seeing the effect of each change.

The Mute option silences Multiplan's audible alarm.

The Iterate option gives the user a means of solving problems which involve circular or "closed loop" references. Whereas formulas which count on each other's results (i.e., A = B+C, B = A+C) are disallowed in other spreadsheet programs, Multiplan allows spreadsheets with such references to be reiterated upon in an orderly manner either until a maximum number of iterations has been reached, or until a cell has reached a predetermined value.

PRINT
Presents a choice of four actions related to printing the active sheet. To begin printing, choose Go. To put printable output in a disk file, choose File. To set the margins that will be used on the printed output, choose Margins. To fix the part of the worksheet to be printed, or to insert a control line at the top of the output, choose Options.

QUIT
Ends the Multiplan session without saving the active sheet. Multiplan requests confirmation; if it is given, Multiplan terminates, returning control to the computer operating system. The active sheet is lost unless it has previously been saved.

SORT
Reorders the rows on the spreadsheet so that the data in a specified column appears in ascending or descending numerical order. The column to be sorted may contain numbers, text, or other values, and if such values are mixed, they are presented in ascending order numerically, alphabetically and by error value, after which any blank cells follow.

TRANSFER
Offers a choice of five commands, which affect an entire sheet.

To load a saved sheet, replacing the active sheet, choose Load.

To save the active sheet in a disk file, choose Save.

To give the active sheet a new name, choose Rename.

To clear the active sheet, deleting all its contents, and restoring all its default settings, choose Clear.

To delete the disk copy of the active sheet, choose Delete.

VALUE
This command is used to enter a formula or number into the active cell. VALUE may either be selected from the command menu or by typing a numerical value, a mathematical symbol, or a left parentheses.

WINDOW
Presents a choice of four things that can be done with windows.

To open a new window by splitting the active window horizontally or vertically, or to open a window used strictly for titles, choose Split.

To close a window by removing it from the screen, choose Close.

To synchronize scrolling of windows, choose Link.

To move a window to a particular part of the sheet, choose Home.

To add or remove a decorative border around a window, choose Border.

XTERNAL
Presents a choice of actions relating to the use of data from other sheets in the formulas of the active sheet.

To copy data, or blocks of data from an inactive spreadsheet to the active sheet, choose Copy.

To display the relationships between the active sheet and other sheets, showing which sheets support (provide values for) the active one and which sheets depend on (use values from) the active sheet, choose List.

To assign a substitute name for an inactive sheet, specify Use.

Table 2. Multiplan Functions

ABS	— Calculates the absolute value of an argument.
AND	— True if, and only if, all values are true; otherwise returns false.
ATAN	— Gives the arctangent of an argument.
AVERAGE	— Returns the average of all cells referenced by up to 5 arguments.
COLUMN	— Gives the current column number.
COS	— Calculates an argument's cosine.
COUNT	— Finds the number of cells fitting the referenced criteria.
DOLLAR	— Formats numbers as dollar amounts.
EXP	— This is the inverse natural logarithm of the argument.
FALSE	— Returns the logical false value.
FIXED	— Rounds the first argument to the precision specified by the second.
IF	— Returns value specified after "THEN" if argument is true, or the "ELSE" specified value if false.
INDEX	— Returns the value of the cell in a named area offset by an index value.
INT	— Truncates the argument's fractional part.
ISERROR	— Returns true if, and only if, the argument is an error value.
ISNA	— Returns true if, and only if, the argument is an #N/A value.
LEN	— Gives the number of characters in the argument's string.
LN	— Calculates the natural logarithm of its argument.
LOG10	— Returns the common logarithm of its argument.
LOOKUP	— Used to search for dependent variables in a lookup table.
MAX	— Finds the largest numeric value in an area of cells.
MID	— Produces the middle characters of a string.
MIN	— Finds the smallest numeric value in an area of cells.
MOD	— Gives the remainder of the integer division of the two arguments.
NA	— Returns the #N/A value.
NOT	— Gives the logical inverse of the argument.
NPV	— Calculates the net present value of a constant annuity.
OR	— True if, and only if, any of the arguments are true; otherwise returns a false.
PI	— Returns Pi (3.14159...).
REPT	— Forms a string consisting of a repeated substring.
ROUND	— Rounds the first argument to the precision specified by the second.
ROW	— Gives the current row number.
SIGN	— Performs the Signum function on the argument.
SIN	— Returns the sine of the argument.
SQRT	— Calculates the square root of the argument.
STDEV	— Calculates the standard deviation of the arguments.
SUM	— Adds the sum of all cells in a specified area.
TAN	— Calculates the tangent of the argument.
TRUE	— Returns the logical true value.
VALUE	— Used to extract numbers from strings.

BENEFITS

Unlike other spreadsheet programs, Multiplan allows the user to create and view as many as eight different windows within the screen display area. Complete control is allowed over each window, allowing windows without borders, and the freezing and scrolling of title columns and rows.

Multiplan allows formulas to describe the contents of any cell. Formulas are written in a method similar to standard programming languages, and are evaluated according to priority of functions, a unique feature among spreadsheet programs. Parentheses are allowed to clarify the order of calculation. Formulas can use a string of characters as a variable name, and variables may be either numerical data, or strings of characters which may be manipulated to concatenate words and phrases. These are all unusually powerful and intuitively easy-to-learn features many of which are unique to Multiplan.

Multiplan gives the user an unusual amount of flexibility in rearranging the format or layout of a spreadsheet with its three forms of addressing: absolute, relative, or symbolic (by name). Any of the three can be combined in any order to produce the exact results needed in any case.

One of the features that sets Multiplan apart from other spreadsheet programs is the ability to name all cells. The NAME command allows the naming of single cells, an area of cells of any shape, or even a list of unconnected areas of cells. That name can then be used in functions, or even as a response in a command. NAME also allows the user to review all cell names in their proper position on the screen in order to reduce confusion.

Multiplan commands can be entered by single letters on the command lines, after which the program will fill in the rest of the command. This speeds the user through complex operations without leaving any doubt about their functions. Versatile commands handle not only single data cells, rows, or columns as do other spreadsheet programs, but these commands allow Multiplan to move multiple rows or columns, or insert, delete, or handle any rectangular area. All relative references are automatically adjusted to account for these changes.

Multiplan automatically updates all entries affected by a change in a single cell, without requiring the user to command it to do so. This feature allows the user to fiddle with numbers and test for sensitivities and trouble spots.

Another unique benefit of Multiplan is its ability to employ values from one sheet in the formulas of another. This "sheet linkage" can be used to construct a hierarchy of worksheets, with detailed worksheets feeding their totals to a summary worksheet. When a detail sheet is updated and saved on diskette, the dependent summary sheet will be automatically updated the next time it is loaded.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE:

Multiplan requires a minimum system which contains at least:

- 64K bytes of RAM
- 8080/8085 CPU
- Console with absolute cursor positioning
- One Diskette drive

OPTIONAL HARDWARE:

- Line printer

REQUIRED SOFTWARE:

- CP/M* Operating System

Documentation Package

Multiplan users manual

Shipping Media

(Specify by Alphabetical Character when ordering)

A - Single density IBM 3740/1 compatible 8" diskette
B - Double density IBM 3740/1 compatible 8" diskette
F - iPDS™ compatible 5-1/4" diskette

*CP/M is a registered trademark of Digital Research, Inc.

ORDERING INFORMATION

Order Code	Shipping Media	Product Description
SD109CPM80A	A—Single-density 8″ diskette	Multiplan spreadsheet program for use under CP/M* on 8080/8085 based small computers.
SD109CPM80B	B—Double-density 8″ diskette	
SD109CPM80F	F—iPDS Format 5¼″ diskette	

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

DOMESTIC SALES OFFICES

ALABAMA

Intel Corp.
5015 Bradford Drive
Suite 2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

Intel Corp.
11225 N. 28th Drive
Suite 214D
Phoenix 85029
Tel: (602) 869-4980

Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040

Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

Intel Corp.
4350 Executive Drive
Suite 150
San Diego 92111
(619) 452-5880

Intel Corp.*
2000 East 4th Street
Suite 100
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
1350 Shorebird Way
Mt. View 94043
Tel: (415) 968-8086
TWX: 910-339-9279
910-338-0255

COLORADO

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (303) 594-6622

Intel Corp.*
650 S. Cherry Street
Suite 720
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corp.
26 Mill Plain Road
Danbury 06810
Tel: (203) 748-3130
TWX: 710-456-1199

EMC Corp.
222 Summer Street
Stamford 06901
Tel: (203) 327-2934

FLORIDA

Intel Corp.
242 N. Westmonte Drive
Suite 105
Altamonte Springs 32714
Tel: (305) 869-5588

Intel Corp.
1500 N.W. 62nd Street
Suite 104
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407

FLORIDA (Cont'd)

Intel Corp.
11300 4th Street South
Suite 170
St. Petersburg 33702
Tel: (813) 577-2413

GEORGIA

Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

Intel Corp.*
2550 Gulf Road
Suite 815
Rolling Meadows 60008
Tel: (312) 981-7200
TWX: 910-651-5881

INDIANA

Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
St. Andrews Building
1930 St. Andrews Drive N.E.
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

Intel Corp.
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel: (913) 642-8080

LOUISIANA

Industrial Digital Systems Corp.
Tel: (504) 899-1654

MARYLAND

Intel Corp.*
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Intel Corp.
7833 Walker Drive
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

Intel Corp.*
27 Industrial Avenue
Chelmsford 01824
Tel: (617) 256-1800
TWX: 710-343-6333

MICHIGAN

Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel: (313) 851-8096

MINNESOTA

Intel Corp.
3500 W. 80th Street
Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

Intel Corp.*
Raritan Plaza III
Raritan Center
Edison 08837
Tel: (201) 225-3000
TWX: 710-480-6238

NEW MEXICO

Intel Corp.
8500 Menaul Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

Intel Corp.*
300 Vanderbilt Motor Parkway
Hauppauge 11788
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
Suite 2B Hollowbrook Park
15 Myers Corners Road
Wappinger Falls 12590
Tel: (914) 297-6161
TWX: 510-248-0060

Intel Corp.*
211 White Spruce Boulevard
Rochester 14623
Tel: (716) 424-1050
TWX: 510-253-7391

T-Squared
6443 Ridings Road
Syracuse 13206
Tel: (315) 463-8592
TWX: 710-541-0554

T-Squared
7353 Pittsford-Victor Road
Victor 14564
Tel: (716) 924-9101
TWX: 510-254-8542

NORTH CAROLINA

Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

Intel Corp.*
6500 Poe Avenue
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
Chagrin-Brainard Bldg., No. 300
28001 Chagrin Boulevard
Cleveland 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

Intel Corp.
4157 S. Harvard Avenue
Suite 123
Tulsa 74135
Tel: (918) 749-8688

OREGON

Intel Corp.
10700 S.W. Beaverton
Hillsdale Highway
Suite 22
Beaverton 97005
Tel: (503) 641-8086
TWX: 910-467-8741

PENNSYLVANIA

Intel Corp.*
455 Pennsylvania Avenue
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Boulevard
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PENNSYLVANIA (Cont'd)

Q.E.D. Electronics
139 Terwood Road
Willow Grove 19090
Tel: (215) 657-5600

TEXAS

Intel Corp.*
12300 Ford Road
Suite 380
Dallas 75234
Tel: (214) 241-8087
TWX: 910-860-5617

Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

Industrial Digital Systems Corp.
5925 Sovereign
Suite 101
Houston 77036
Tel: (713)988-9421

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

UTAH

Intel Corp.
5201 Green Street
Suite 290
Salt Lake City 84123
Tel: (801) 263-8051

VIRGINIA

Intel Corp.
1603 Santa Rosa Road
Suite 109
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

Intel Corp.
110 110th Avenue N.E.
Suite 510
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086

WISCONSIN

Intel Corp.
450 N. Sunnyslope Road
Suite 130
Chancellory Park I
Brookfield 53005
Tel: (414) 784-8087

CANADA

ONTARIO

Intel Semiconductor of Canada, Ltd.
Suite 202, Bell Mews
39 Highway 7
Nepean K2H 8R2
Tel: (613) 829-9714
TELEX: 053-4115

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TELEX: 06983574

QUEBEC

Intel Semiconductor of Canada, Ltd.
3950 Cote Vertu Rd.
Suite 210
St. Laurent H4R 1V4
Tel: (514) 334-0560
TELEX: 05-824172

*Field Application Location

UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5-6 Tokodai Toyosato-machi
Tsukuba-gun, Ibaraki-ken 300-26
Japan

FRANCE
Intel
5 Place de la Balance
Silic 223
94528 Rungis Cedex
France

UNITED KINGDOM
Intel
Piper's Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel
Seidstrasse 27
D-8000 Munchen 2
West Germany

Printed in U.S.A./C-787/1284/60K/RRD RM
Development Systems

ISBN 0-917017-16-1