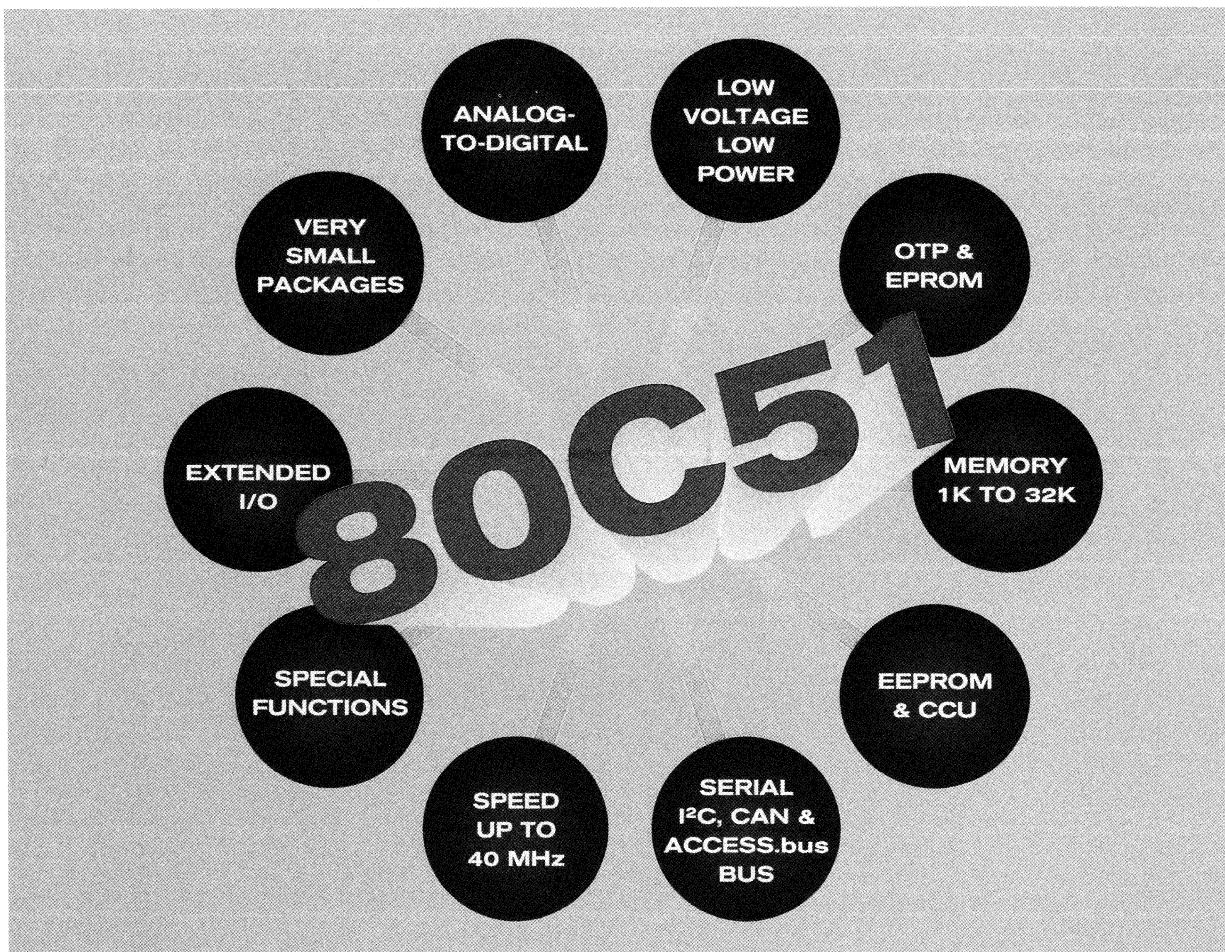


Application Notes and Development Tools for 80C51 Microcontrollers



1995

DATA HANDBOOK

QUALITY ASSURED

Our quality system focuses on the continuing high quality of our components and the best possible service for our customers. We have a three-sided quality strategy: we apply a system of total quality control and assurance; we operate customer-oriented dynamic improvement programmes; and we promote a partnering relationship with our customers and suppliers.

PRODUCT SAFETY

In striving for state-of-the-art perfection, we continuously improve components and processes with respect to environmental demands. Our components offer no hazard to the environment in normal use when operated or stored within the limits specified in the data sheet.

Some components unavoidably contain substances that, if exposed by accident or misuse, are potentially hazardous to health. Users of these components are informed of the danger by warning notices in the data sheets supporting the components. Where necessary the warning notices also indicate safety precautions to be taken and disposal instructions to be followed. Obviously users of these components, in general the set-making industry, assume responsibility towards the consumer with respect to safety matters and environmental demands.

All used or obsolete components should be disposed of according to the regulations applying at the disposal location. Depending on the location, electronic components are considered to be 'chemical', 'special' or sometimes 'industrial' waste. Disposal as domestic waste is usually not permitted.

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

	page
SECTION 1 GENERAL INFORMATION	5
SECTION 2 INTER-INTEGRATED CIRCUIT (I ² C) BUS	37
SECTION 3 I ² C SERIAL BUS APPLICATION NOTES & ARTICLES	65
SECTION 4 ACCESS.bus TECHNICAL OVERVIEW	273
SECTION 5 ACCESS.bus APPLICATION NOTES & ARTICLES	289
SECTION 6 CONTROL AREA NETWORK (CAN) BUS	333
SECTION 7 87C750, 8XC751, 8XC752 APPLICATION NOTES	413
SECTION 8 OTHER 80C51 APPLICATION NOTES & ARTICLES	575
SECTION 9 DEVELOPMENT SUPPORT TOOLS	757
APPENDIX A DATA HANDBOOK SYSTEM	842

Application Notes and Development Tools for 80C51 Microcontrollers

Microcontrollers from Philips Semiconductors

Philips Semiconductors 8 and 16-bit microcontrollers are based on the widely-accepted 8048, 8051 and 68000 architectures. We offer most of the 'industry standard' products in these architectures as well as a large selection of powerful derivative products. These derivatives offer a wide assortment of features, including: additional memory, A/D, PWM, additional timers, DTMF, OSD, OTP, EMC and EMI, plus many others. The variety of product derivatives allows Philips Semiconductors to support a broad range of functions in consumer, telecom, EDP, multi media, automotive and industrial applications.

For details, see:

- 8048 'industry standard' architecture types (PCF84CXXX family) in *"Data Handbook IC14"*.

The PCD33XX family covers telecom terminal family devices based on the 8048 core and instruction set, in *"Data Handbook IC03"*.

- 8051 'industry standard' architecture types in *"Data Handbook IC20"*.
- 68000 compatible 'industry standard' architecture types in *"Data Handbook IC21"*.

The Low Power 80CL51 family of derivatives can be found in *"Data Handbook IC20"*. These devices operate over the wide voltage range of 1.8 to 6.0V and are ideal for portable and battery operations.

Many of Philips Semiconductors ICs offer on-board UART serial ports and I²C-bus. The I²C-bus allows easy connection to over 100 other devices, thereby increasing system capabilities even further. We also offer the Philips/Digital Equipment Corporation ACCESS.bus, a new Standard Desktop bus. And for automotive and industrial applications, we also offer the CAN and the VAN serial bus. The CAN standard, developed by Bosch, and VAN concepts offer high noise immunity and error correction.

Philips Semiconductors 16-bit microcontroller family is based on the 68000 architecture. While these are called 16-bit microcontrollers, the 68000 CPU core architecture is a 32-bit. This offers the user a great deal more processing power when the need arises in a design to move from an 8-bit to a 16-bit microcontroller.

Philips Semiconductors 16-bit microcontrollers are software compatible with existing 68000 code. Future developments include the introduction of SPARC and Trimedia devices.

Philips Semiconductors is developing a family of 16-bit microcontrollers based on the 8051 'XA' (eXtended Architecture). This family of microcontrollers will offer advanced performance for those applications that are computation and memory intensive in an embedded control environment.

Section 1

General Information

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

Contents	7
Product Status	13
80C51 microcontroller family features guide	14
8051 microcontroller cross-reference guide	18
Low power / low voltage microcontroller family	19
80C51 microcontroller development system support	20
8-bit microcontroller demonstration and evaluation boards	22
Microcontroller bulletin boards	23
Philips Fax-On-Demand System	24
CMOS and NMOS 8-bit microcontroller family	25
CMOS 16-bit microcontroller family	33
Ordering information	34

APPLICATION NOTES AND DEVELOPMENT TOOLS FOR 80C51 MICROCONTROLLERS

Preface	3
Section 1 – General Information	
Contents	7
Product Status	13
80C51 microcontroller family features guide	14
8051 microcontroller cross-reference guide	18
Low power / low voltage microcontroller family	19
80C51 microcontroller development system support	20
8-bit microcontroller demonstration and evaluation boards	22
Microcontroller bulletin boards	23
Philips Fax-On-Demand System	24
CMOS and NMOS 8-bit microcontroller family	25
CMOS 16-bit microcontroller family	33
Ordering information	34
Section 2 – Inter-Integrated Circuit (I²C) Bus	
The I ² C-bus and how to use it	39
I ² C peripheral selection guide	58
82B715 I ² C bus extender	60
Section 3 – I²C Serial Bus Application Notes & Articles	
AN422 Using the 8XC751 microcontroller as an I ² C bus master	67
AN425 Interfacing the PCD8584 I ² C-bus controller to 80C51 family microcontrollers	85
AN430 Using the 8XC751/752 in multimaster I ² C applications	104
AN433 I ² C slave routines for the 83C751	140
AN434 Connecting a PC keyboard to the I ² C-bus	146
AN438 I ² C routines for 8XC528	164
AN444 Using the P82B715 I ² C extender on long cables	186
ETV/AN89004 PLM51 I ² C software interface IIC51 (version 0.5)	206
EIE/AN91007 I ² C driver routines for 8XC751/2 microcontrollers	215
Programming the I ² C interface	269
Section 4 – ACCESS.bus Technical Overview	
ACCESS.bus Technical Overview	275
Introduction	275
What is ACCESS.bus?	275
ACCESS.bus Hardware	275
ACCESS.bus Protocols	276
How ACCESS.bus Works	277
Electrical	277
Bus Transactions	277
Synchronization	279
Byte Framing and Acknowledgement	279
Addressing	279
Arbitration	279
Message Format	279
Control/Status Messages	279
Configuration	280
Device Identifiers	281
Device Capabilities Information	281
Application Device Types	282
Keyboard Devices	282
Locator Devices	282
Text Devices	282
Timing Rules	282
Transaction Timing Rules	282
Response Timeouts	282
Software Architecture and Development	282
Device Firmware Development	283
Host Software Architecture	283

Development Support	283
ACCESS.bus Industry Group	283
Philips Semiconductors Support	283
ACCESS.bus development kit	284
ACCESS.bus PC/AT controller board	286
Section 5 – ACCESS.bus Application Notes & Articles	
AN445 ACCESS.bus mouse application code for the 8XC751 microcontroller	See Section 7
Issues in desktop connectivity	291
Finally, a plug-and-play solution	295
Special Report: ACCESS.bus Specs And Products	297
ACCESS.bus: A New Peripheral Bus	299
Embedded control using ACCESS.bus	301
A PC-to-ACCESS.bus interface card	309
Taking a new bus	312
Personal Digital Assistants: What's missing?	318
The portable desktop: New connections for today's mobile user	320
Who's hopping on the ACCESS.bus?	322
ACCESS.bus revisited—ending the peripheral connection nightmare	324
Seriously serial	327
Section 6 – Control Area Network (CAN) Bus	
Control Area Network (CAN) overview	335
82C150 CAN serial linked I/O device (SLIO) with digital and analog port functions	336
82C200 Stand-alone CAN-controller	365
PCA82C250 CAN controller interface	401
Section 7 – 87C750, 8XC751, 8XC752 Application Notes	
AN422 Using the 8XC751 microcontroller as an I ² C bus master	See Section 3
AN423 Software driven serial communication routines for the 83C751 and 83C752 microcontrollers	415
AN426 Controlling air core meters with the 87C751 and SA5775	420
AN427 Timer 1 for the 83/87C748/749 and the 83/87C751/752 (non-I ² C applications) microcontrollers	434
AN428 Using the ADC and PWM of the 83C752/87C752	440
AN429 Airflow measurement using the 83/87C752 and "C"	447
AN430 Using the 8XC751/752 in multimaster I ² C applications	See Section 3
AN433 I ² C slave routines for the 83C751	See Section 3
AN436 "Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller	466
AN439 87C751 fast NiCad charger	476
AN442 (BCM) 87C751 Specification for a bus-controlled monitor	488
AN445 ACCESS.bus mouse application code for the 8XC751 microcontroller	505
AN446 A software duplex UART for the 751/752	535
AN453 Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs	543
AN454 Interfacing the 83C576/87C576 to the ISA bus	562
EIE/AN91007 I ² C driver routines for 8XC751/2 microcontrollers	See Section 3
Section 8 – Other 80C51 Application Notes & Articles	
AN408 80C451 operation of port 6	577
AN417 256k Centronics printer buffer using the 87C451 microcontroller	588
AN418 Counter/timer 2 of the 83C552 microcontroller	601
AN420 Using up to 5 external interrupts on 80C51 family microcontrollers	608
AN424 8051 family warm boot determinations	610
AN440 RAM loader program for 80C51 family applications	612
AN443 IEEE Micro Mouse using the 87C751 microcontroller	621
AN447 Automatic baud rate detection for the 80C51	642
AN448 Determining baud rates for 8051 UARTs and other UART issues	645
ESG89001 Electro magnetic compatibility and printed circuit board (PCB) constraints	648
EIE/AN91001 Workbench EMC evaluation method	667
EIE/AN91006 A/D conversion with P83CL410 PCF1252-x	684
EIE/AN91009 Driver for 8xC851 E2PROM	700
EIE/AN92001 Low RF-emission applications with a P83CE654 microcontroller	715

CONTENTS

EIE/AN93017 Using the analog-to-digital converter of the 8XC552 microcontroller	727
Chips push CAN bus into embedded world	745
Add Text Overlay to Any Video Display	747
Section 9 – Development Support Tools	
Development support tools	759
Ashling CTS51 Microprocessor development systems for Philips microcontrollers	766
BSO/Tasking: The total development solution for the 8051 family	771
CEIBO DB-51 Development Board	778
CEIBO DS-51 In-Circuit Emulator	780
CEIBO DS-300 Peripheral Development Tools	788
CEIBO DS-750 Microcontroller Development Tool	790
CEIBO DS-752 Microcontroller Development Tool	792
CEIBO EB-51 Emulation Board	794
CEIBO MP-51 Programmer	796
MetaLink iceMASTER-PE 8051 Family In-Circuit Emulator	798
MetaLink iceMASTER 8051 Family In-Circuit Emulators	803
NOHAU EMUL51-PC – PC-based in-circuit emulator	816
PDS51 Development System for 8XC51 Microcontroller Derivatives	827
LCP Programmers for 87C51 and Derivatives	831
S87C00KSD 8XC51 and I ² C Bus Evaluation Board	834
OM4130 CAN evaluation board with the 8XC552 and 82C200 CAN controller	835
OM4239 CAN evaluation board with the 8XC592 microcontroller	836
OM4240 Evaluation board for the 8XCE598 microcontroller	837
OM4272 SLIO evaluation board with the 82C150 and 82C250 CAN ICs	838
OM4280 P83C852 Smart Card crypto-controller demonstration kit	839
PEB552 Evaluation board	840
Appendix A – Data Handbook System	842

IC20: 80C51-BASED 8-BIT MICROCONTROLLERS

Preface	3
Section 1 – General Information	
Contents	7
Product Status	13
80C51 microcontroller family features guide	14
8051 microcontroller cross-reference guide	18
Low power / low voltage microcontroller family	19
80C51 microcontroller development system support	20
8-bit microcontroller demonstration and evaluation boards	22
Microcontroller bulletin boards	23
Philips Fax-On-Demand System	24
CMOS and NMOS 8-bit microcontroller family	25
CMOS 16-bit microcontroller family	33
Ordering information	34
Section 2 – 80C51 Technical Description	
80C51 architecture	39
80C51 hardware description	54
80C51 programmer's guide and instruction set	79
80C51 EPROM products	134
Section 3 – 80C51 Family Derivatives	
80C31/80C51/87C51 CMOS single-chip 8-bit microcontroller	141
80C51FA/83C51FA/87C51FA CMOS single-chip 8-bit microcontroller	159
83C51FB/87C51FB CMOS single-chip 8-bit microcontrollers	187
83C51FC/87C51FC CMOS single-chip 8-bit microcontrollers	214
80CL31/80CL51 Low-voltage single-chip 8-bit microcontrollers	242
83L51FA/87L51FA CMOS single-chip 3.0V 8-bit microcontroller	276
83L51FB/87L51FB CMOS single-chip 3.0V 8-bit microcontroller	292
80C32/80C52/87C52 CMOS single-chip 8-bit microcontrollers	308
80C54/87C54 CMOS single-chip 8-bit microcontrollers	329
80C58/87C58 CMOS single-chip 8-bit microcontrollers	349
83C055/87C055 Microcontroller for television and video (MTV)	369
P83CL168; P83CL167 Microcontroller for TV OSD, VST and control functions	388
P83CL268; P83CL267 Low voltage/low power single-chip 8-bit microcontroller with I ² C	485
80CL410/83CL410 CMOS single-chip 8-bit microcontroller	507
80C451/83C451/87C451 CMOS single-chip 8-bit microcontroller	527
80C453/83C453/87C453 CMOS single-chip 8-bit microcontroller	550
83C504/87C504 CMOS single-chip 8-bit microcontroller	568
83C508/87C508 CMOS single-chip 8-bit microcontroller	585
P83C524 8-bit microcontroller	606
87C524 CMOS single-chip 8-bit microcontroller	626
80C528/83C528 CMOS single-chip 8-bit microcontroller	644
87C528 CMOS single-chip 8-bit microcontroller	666
P8xC528 8-bit microcontroller with EMC	693
83C542/87C542 ACCESS.bus™ microcontroller	715
80C550/83C550/87C550 CMOS single-chip 8-bit microcontroller with A/D and watchdog timer	739
8XC552/562 overview	739
8XC552 OVERVIEW	739
83C562 OVERVIEW	739
Differences From the 80C51	739
Program Memory	739
Data Memory	739
Special Function Registers	740
Timer T2	746
Timer T3, The Watchdog Timer	747
Serial I/O	747

Reset Circuitry	780
Interrupts	781
I/O Port Structure	785
Port 1 Operation	785
Port 5 Operation	785
Pulse Width Modulated Outputs	785
Analog-to-Digital Converter	785
Power Reduction Modes	791
Memory Organization	793
80C552/83C552 Single-chip 8-bit microcontroller with 10-bit A/D, capture/compare timer, high-speed outputs, PWM	798
87C552 Single-chip 8-bit microcontroller with 10-bit A/D, capture/compare timer, high-speed outputs, PWM	818
P83CE558/P80CE558/ P89CE558 Single-chip 8-bit microcontroller	839
P83CE559/P80CE559 Single-chip 8-bit microcontroller	908
80C562/83C562 Single-chip 8-bit microcontroller with 8-bit A/D, capture/compare timer, high-speed outputs, PWM	976
80C575/83C575/87C575 CMOS single-chip 8-bit microcontroller	989
83C576/87C576 CMOS single-chip 8-bit microcontroller	1024
80CL580/83CL580 Low-voltage single-chip 8-bit microcontroller	1063
P8XC592 8-bit microcontroller with on-chip CAN	1103
P8XCE598 8-bit microcontroller with on-chip CAN	1212
80C652/83C652 CMOS single-chip 8-bit microcontroller	1320
87C652 CMOS single-chip 8-bit microcontroller	1338
83C654 CMOS single-chip 8-bit microcontroller	1358
87C654 CMOS single-chip 8-bit microcontroller	1376
83CE654 CMOS single-chip 8-bit microcontroller with Electromagnetic Compatibility improvements	1397
83C748/87C748 CMOS single-chip 8-bit microcontroller	1411
83C749/87C749 CMOS single-chip 8-bit microcontroller	1424
83C750/87C750 CMOS single-chip 8-bit microcontrollers	1439
83C751/87C751 CMOS single-chip 8-bit microcontroller	1449
83C752/87C752 CMOS single-chip 8-bit microcontroller with A/D, PWM	1467
83CL781/83CL782 Low-voltage single-chip 8-bit microcontrollers	1485
80C851/83C851 CMOS single-chip 8-bit microcontroller with on-chip EEPROM	1518
Section 4 – High Performance 16-bit 80C51 XA (eXtended Architecture)	
80C51XA Architectural overview	1535
XA-G3 CMOS single-chip 16-bit microcontroller	1545
80C51XA Development tools	1557
Section 5 – Package Outlines	
Plastic Dual In-Line Package	
DIP8: plastic dual in-line package; 8 leads (300 mil)	SOT97-1 1563
24-pin (300 mils wide) plastic dual in-line (N) package	SOT101/0410D 1564
DIP28: plastic dual in-line package; 28 leads (600 mil)	SOT117-1 1565
28-pin (600 mils wide) plastic dual in-line (N) package	0413B 1566
DIP40: plastic dual in-line package; 40 leads (600 mil)	SOT129-1 1567
Plastic Shrink Dual In-Line Package	
SDIP42: plastic shrink dual in-line package; 42 leads (600 mil)	SOT270-1 1568
SDIP64: plastic shrink dual in-line package; 64 leads (750 mil)	SOT274-1 1569
Ceramic Dual In-Line Package	
24-Pin (300 mils wide) Ceramic Dual In-line (F) Package (with Window (FA) Package)	0586B 1570
28-Pin (600 mils wide) Ceramic Dual In-line (F) Package (with Window (FA) Package)	0589B 1571
40-Pin (600 mils wide) Ceramic Dual In-line (F) Package (with Window (FA) Package)	0590B 1572
Plastic Leaded Chip Carrier	
28-Pin (300 mils wide) Plastic Leaded Chip Carrier (A) Package	SOT261/0401F 1573
44-Pin Plastic Leaded Chip Carrier (A) Package	0403G 1574
44-pin plastic leaded chip carrier; pocket version (A) package	SOT187 1575
PLCC68: plastic leaded chip carrier; 68 leads	SOT188-2 1576
68-Pin Plastic Leaded Chip Carrier (A) Package	0398E 1577

Ceramic Leaded Chip Carrier			
	44-pin CerQuad J-Bend (K) Package	1472A	1578
	68-pin CerQuad J-Bend (K) Package	1473A	1579
	68-Pin Chip Carrier, J-Bend (L) Package	1240C	1580
	Ceramic leaded chip carrier (window); 68 leads	NO330	1581
Plastic Quad Flat Package			
QFP44:	plastic quad flat package; 44 leads (lead length 1.3 mm); body 10 x 10 x 1.75 mm	SOT307-2	1582
	44-lead quad flat-pack; plastic	SOT205AG	1583
LQFP44:	plastic low profile quad flat package; 44 leads; body 10 x 10 x 1.4 mm	SOT389-1	1584
QFP64:	plastic quad flat package; 64 leads (lead length 2.35mm); body 14 x 20 x 2.75mm	SOT208-1	1585
QFP64:	plastic quad flat package; 64 leads (lead length 1.95mm); body 14 x 20 x 2.7mm; high stand-off height	SOT319-1	1586
QFP80:	plastic quad flat package; 80 leads (lead length 1.95mm); body 14 x 20 x 2.7 mm; high stand-off height	SOT318-1	1587
Ceramic Quad Flat Package			
CQFP80:	ceramic quad flat package; 80 leads	SOT351-1	1588
Plastic Small Outline Package			
SO8:	plastic small outline package; 8 leads; body width 3.9mm	SOT96-1	1589
SO28:	plastic small outline package; 28 leads; body width 7.5mm	SOT136-1	1590
VSO40:	plastic very small outline package; 40 leads	SOT158-1	1591
VSO56:	plastic very small outline package; 56 leads	SOT190-1	1592
Plastic Shrink Small Outline Package			
SSOP24:	plastic shrink small outline package; 24 leads; body width 5.3mm	SOT340-1	1593
SSOP28:	plastic shrink small outline package; 28 leads; body width 5.3mm	SOT341-1	1594
	42-Pin Plastic SSOP (Shrink Small Outline Package) Dual In-Line (D/K) Package	1680	1595
Appendix A – Data Handbook System			1596
Appendix B – Pin Configurations			1598

**80C51-Based
8-Bit Microcontrollers****DEFINITIONS**

Data Sheet Identification	Product Status	Definition
<i>Objective Specification</i>	Formative or In Design	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
<i>Preliminary Specification</i>	Preproduction Product	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
<i>Product Specification</i>	Full Production	This data sheet contains Final Specifications. Philips Semiconductors reserves the right to make changes at any time without notice, in order to improve design and supply the best possible product.

80C51 microcontroller family features guide

Part Number (ROMless)	Memory			Counter Timers	I/O Port	Serial Interfaces	External Interrupt	Comments/ Special Features	
	ROM	EPRM	RAM						
P	83C750	1K	64	1 (16-bit)	2-3/8	-	2	40 MHz, Lowest cost, SSOP	
P	87C750		1K	1 (16-bit)	2-3/8	-	2	40 MHz, Lowest cost, SSOP	
P	83C748	2K	64	1 (16-bit)	2-3/8	-	2	8XC751 w/o I ² C, SSOP	
P	87C748		2K	1 (16-bit)	2-3/8	-	2	8XC751 w/o I ² C, SSOP	
S	83C751	2K	64	1 (16-bit)	2-3/8	I ² C (bit)	2	24-pin Skinny DIP, SSOP	
S	87C751		2K	1 (16-bit)	2-3/8	I ² C (bit)	2	24-pin Skinny DIP, SSOP	
P	83C749	2K	64	1 (16-bit)	2-5/8	-	2	8XC752 w/o I ² C, SSOP	
P	87C749		2K	1 (16-bit)	2-5/8	-	2	8XC752 w/o I ² C, SSOP	
S	83C752	2K	64	1 (16-bit)	2-5/8	I ² C (bit)	2	5 Channel 8-bit A/D, PWM Output, SSOP	
S	87C752		2K	1 (16-bit)	2-5/8	I ² C (bit)	2	5 Channel 8-bit A/D, PWM Output, SSOP	
MAx	8051AH (8031AH)	4K	128	2	4	UART	2	NMOS	
SC	80C51 (80C31)	4K	128	2	4	UART	2	CMOS (Sunnyvale)	
PCx	80C51 (80C31)	4K	128	2	4	UART	2	CMOS (Hamburg)	
SC	87C51		4K	128	2	4	UART	2	CMOS
P	80CL51 (80CL31)	4K	128	2	4	UART	10	Low Voltage (1.8V to 6V), Low Power	
P	83CL410 (80CL410)	4K	128	2	4	I ² C	10	Low Voltage (1.8V to 6V), Low Power	
SC	83C451 (80C451)	4K	128	2	7	UART	2	Extended I/O, Processor Bus Interface	
SC	87C451		4K	128	2	7	UART	2	Extended I/O, Processor Bus Interface
P	83C550 (80C550)	4K	128	2 + Watchdog	4	UART	2	8 Channel 8-bit A/D	
P	87C550		4K	128	2 + Watchdog	4	UART	2	8 Channel 8-bit A/D
P	83C851 (80C851)	4K	128	2	4	UART	2	256B EEPROM, 80C51 Pin compatible	
P	83C542	4K	256	2	1	I ² C	2	ACCESS.bus, replaces 8042 KB controller	
P	87C542		4K	256	2	1	I ² C	2	See Above
P	83C852	6K	256	2 (16-bit)	2/8	-	1	Smartcard Controller with 2K EEPROM (Data, Code) Cryptographic Calc Unit	
P	83CL580 (80CL580)	6K	256	3 + Watchdog	5	UART, I ² C	9	4 Channel 8-bit A/D, PWM Output, Low Voltage (2.5V to 6V), Low Power	
MAx	8052AH (8032AH)	8K	256	3	4	UART	2	NMOS	
P	80C52 (80C32)	8K	256	3	4	UART	2	80C51 Pin Compatible	
P	87C52		8K	256	3	4	UART	2	(see above)
P	83C652 (80C652)	8K	256	2	4	UART, I ² C	2	80C51 Pin Compatible	
S	87C652		8K	256	2	4	UART, I ² C	2	(see above)
P	83C453 (80C453)	8K	256	2	7	UART	2	Extended I/O, Processor Bus Interface	
P	87C453		8K	256	2	7	UART	2	Extended I/O, Processor Bus Interface
S	83C51FA (80C51FA)	8K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA	
S	87C51FA		8K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	83L51FA	8K	256	3 + PCA	4	UART	2	Low Voltage 83C51FA (3V @ 20MHz)	
S	87L51FA		8K	256	3 + PCA	4	UART	2	Low Voltage OTP 87C51FA (3V @ 20MHz)
P	83C575 (80C575)	8K	256	3 + PCA+ Watchdog	4	UART	2	High Reliability, with Low Voltage Detect, OSC Fail Detect, Analog Comparators, PCA	
P	87C575		8K	256	(see above)	4	UART	2	(see above)
P	83C576 (80C576)	8K	256	3 + PCA+ Watchdog	4	UART	2	Same as 8XC575 plus UPI and 10-bit A/D	
P	87C576		8K	256	(see above)	4	UART	2	(see above)
PC	83C562 (80C562)	8K	256	3 + Watchdog	6	UART	2	8 Channel 8-bit A/D, 2 PWM Outputs, Capture/Compare Timer	
PCx	83C552 (80C552)	8K	256	3 + Watchdog	6	UART, I ² C	2	8 Channel 10-bit A/D, 2 PWM Outputs, Capture/Compare Timer	
S	87C552		8K	256	3 + Watchdog	6	UART, I ² C	2	(see above)

Notes: Part number prefixes are noted in the first column.
All combinations of part type, speed, temperature and package may not be available.

80C51 microcontroller family features guide

Part Number (ROMless)	Program Security?	Clock Freq (MHz)	Temperature Ranges (°C)			Package					
			0 to 70	-40 to +85	-55 to +125	PDIP	CDIP	PLCC	CLCC	PQFP/SSOP	
83C750	S	N	3.5 to 40	X	X		N24	F24	A28		DB24 (0-70F)
87C750	S	Y	3.5 to 40	X	X		N24	F24	A28		DB24 (0-70F)
83C748	S	N	3.5 to16	X	X		N24		A28		DB24 (0-70F)
87C748	S	Y	3.5 to16	X	X		N24	F24	A28		DB24 (0-70F)
83C751	S	N	3.5 to16	X	X		N24		A28		DB24 (0-70F)
87C751	S	Y	3.5 to16	X	X		N24	F24	A28		DB24 (0-70F)
83C749	S	N	3.5 to 16	X	X		N28		A28		DB28 (0-70F)
87C749	S	Y	3.5 to 16	X	X		N28	F28	A28		DB28 (0-70F)
83C752	S	N	3.5 to 16	X	X	X	N28		A28		DB28 (0-70F)
87C752	S	Y	3.5 to 16	X	X	X	N28	F28	A28		DB28 (0-70F)
8051AH (8031AH)	S	N	3.5 to 15	X	X		N40		A44		
SC80C51 (80C31)	S	Y	3.5 to 33	X	X	X	N40		A44		B44 (5)
PCx80C51 (80C31)	H	N	1.2 to 30	X	X	X	P (40)		WP (44)		H (44)
87C51	S	Y	3.5 to33	X	X	X	N40	F40	A44	K44	B44 (5)
80CL51 (80CL31)	Z	N	0 to 16 (1)		X		N40 (2)				B44
83CL410(80CL410)	Z	N	0 to 12 (1)		X		N40 (2)				B44
83C451 (80C451)	S	N	3.5 to 16	X	X	X	N64 (4)		A68		
87C451	S	Y	3.5 to16	X	X	X	N64 (4)		A68		
83C550 (80C550)	S	Y	3.5 to 16	X	X		N40		A44		
87C550	S	Y	3.5 to 16	X	X	-40 to +125	N40	F40	A44	K44	
83C851 (80C851)	H	Y	1.2 to 16	X	X		N40		A44		B44
83C542	S	Y	3.5 to 16	X					A44		
87C542	S	Y	3.5 to 16	X					A44	K44	
83C852	H	Y	1 to 12	X			SO28 or die				
83CL580 (80CL580)	Z	N	0 to 12 (1)		X		(3)				B64
8052AH (8032AH)	S	N	3.5 to 15	X	X		N40		A44		
80C52 (80C32)	S	Y	3.5 to 24	X	X		N40		A44		B44 (5)
87C52	S	Y	3.5 to 24	X	X	X	N40	F40	A44	K44	B44 (5)
83C652 (80C652)	H	Y	1.2 to 24	X	X	-40 to +125	N40		A44		B44
87C652	S	Y	1.2 to 20	X	X	X	N40	F40	A44	K44	
83C453 (80C453)	S	N	3.5 to 16	X	X				A68		
87C453	S	Y	3.5 to16	X	X				A68		
83C51FA (80C51FA)	S	Y	3.5 to 24	X	X		N40		A44		B44
87C51FA	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83L51FA	S	Y	3.5 to 20	X	X		N40		A44		B44
87L51FA	S	Y	3.5 to 20	X	X		N40	F40	A44	K44	B44
83C575 (80C575)	S	Y	4 to 16	X		X	N40		A44		B44
87C575	S	Y	4 to 16	X		X	N40	F40	A44	K44	B44
83C576 (80C576)	S	Y	4 to 16	X		X	N40		A44		B44
87C576	S	Y	4 to 16	X		X	N40	F40	A44	K44	B44
83C562 (80C562)	H	N	1.2 to 16	X	X	-40 to +125			A68		B80
83C552 (80C552)	H	N	1.2 to 30	X	X	-40 to +125			A68		B80
87C552	S	Y	1.2 to 16	X					A68	K68	

Notes: Production Centers are indicated in the second column: H – Hamburg, S – Sunnyvale, Z – Zurich.
 All combinations of part type, speed, temperature and package may not be available.

- 1) Oscillator options start from 32kHz.
- 2) Also available in VSO40 package.
- 3) Also available in VSO56 Package.
- 4) Not recommended for new design.
- 5) Package available up to 16 MHz only.

80C51 microcontroller family features guide

Part Number (ROMless)		Memory			Counter Timers	I/O Port	Serial Interfaces	External Interrupt	Comments/ Special Features
		ROM	EPRM	RAM					
P	83CL267	12K		256	3	2 5/8	I ² C	-	OSD, 8 PWM Outputs, 3 Software A/D Inputs, 8 LED Drivers
P	83CL268	12K		256	3	2 5/8	I ² C, 1M Baud	-	(see above)
P	83C055	16K		256	2 (16-bit)	3 1/2	-	2	On-Screen Display, 9 PWM Outputs, 3 Software A/D Inputs
P	87C055		16K	256	2 (16-bit)	3 1/2	-	2	(see above)
P	80C54	16K		256	3	4	UART	2	Standard; 80C51 compatible
P	87C54		16K	256	3	4	UART	2	Standard; 87C51 compatible
P	83C504 (80C504)	16K		256	2	4	UART	2	'654 with Hardware Divide (no I ² C)
P	87C504		16K	256	2	4	UART	2	(see above)
P	83C654	16K		256	2	4	UART, I ² C	2	80C51 Pin Compatible
S	87C654		16K	256	2	4	UART, I ² C	2	(see above)
P	83CE654	16K		256	2	4	UART, I ² C	2	83C654 with Reduced EMI
P	83CL781	16K		256	3	4	UART, I ² C	10	Low Voltage (1.8V to 6V), Low Power
P	83CL782	16K		256	3	4	UART, I ² C	10	83CL781 Optimized 12MHz @ 3.1V
S	83C51FB	16K		256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	87C51FB		16K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	83L51FB	16K		256	3 + PCA	4	UART	2	Low Voltage 83C51FB (3V @ 20MHz)
S	87L51FB		16K	256	3 + PCA	4	UART	2	Low Voltage OTP 87C51FB (3V @ 20MHz)
P	83CL167	16K		256	3	6 1/8	I ² C	-	OSD, 8 PWM Outputs, 4 Software A/D Inputs, 8 LED Drivers
P	83CL168	16K		256	3	6 1/8	I ² C, 1M Baud	-	(see above)
P	83C524	16K		512	3 + Watchdog	4	UART, I ² C-bit	2	512 RAM
P	87C524		16K	512	3 + Watchdog	4	UART, I ² C-bit	2	512 RAM
P	83C592 (80C592)	16K		512	3 + Watchdog	6	UART, CAN	6	CAN Bus Controller with 8 x 10-bit A/D, 2 PWM outputs, Capture/Compare Timer
P	87C592		16K	512	3 + Watchdog	6	UART, CAN	6	(see above)
P	80C58	32K		256	3	4	UART	2	Standard; 80C51 compatible
P	87C58		32K	256	3	4	UART	2	Standard; 87C51 compatible
S	83C51FC	32K		256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	87C51FC		32K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
P	83C528 (80C528)	32K		512	3 + Watchdog	4	UART, I ² C-bit	2	Large Memory for High Level Languages
P	87C528		32K	512	3 + Watchdog	4	UART, I ² C-bit	2	Large Memory for High Level Languages
P	83CE528 (80CE528)	32K		512	3 + Watchdog	4	UART, I ² C-bit	2	8XC528 with Reduced EMI
P	83CE598 (80CE598)	32K		512	3 + Watchdog	6	UART, CAN	6	CAN Bus Controller, 8 x 10-bit A/D, 2 PWM outputs, WD, T2, Reduced EMI
P	87CE598		32K	512	3 + Watchdog	6	UART, CAN	6	(see above)
P	83CE558 (80CE558)	32K		1024	3 + Watchdog	6	UART, I ² C	2	Low EMI, 8 Channel 10-bit A/D, 2 PWM Outputs, Capture/Compare Timer
P	89CE558		32K	1024	3 + Watchdog	6	UART, I ² C	2	32K FLash EEPROM plus above

Notes: Part number prefixes are noted in the first column.

All combinations of part type, speed, temperature and package may not be available.

80C51 microcontroller family features guide

Part Number (ROMless)		Program Security?	Clock Freq (MHz)	Temperature Ranges (°C)			Package				
				0 to 70	-40 to +85	-55 to +125	PDIP	CDIP	PLCC	CLCC	PQFP/SSOP
83CL267	T	N	4.0 to 12	X			R42				B64
83CL268	T	N	4.0 to 12	X			R42				B64
83C055	S	N	3.5 to 20	X			NB42				
87C055	S	N	3.5 to 20	X			NB42				
80C54	S	Y	3.5 to 24	X	X		N40		A44		B44
87C54	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83C504 (80C504)	S	Y	1.2 to 20	X	X	X	N40		A44		B44
87C504	S	Y	1.2 to 20	X	X	X	N40	F40	A44	K44	B44
83C654 (80C654)	H	Y	1.2 to 24	X	X	-40 to +125	R42, N40		A44		B44
87C654	S	Y	1.2 to 20	X	X	X	N40	F40	A44	K44	B44
83CE654	H	Y	1.2 to 16	X	X						B44
83CL781	Z	N	0 to 12 (1)		X		N40				B44
83CL782	Z	N	0 to 12 (1)			-25 to +55	N40				B44
83C51FB	S	Y	3.5 to 24	X	X		N40		A44		B44
87C51FB	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83L51FB	S	Y	3.5 to 20	X			N40		A44		B44
87L51FB	S	Y	3.5 to 20	X			N40	F40	A44	K44	B44
83CL167	T	N	4.0 to 12	X			R42				B64
83CL168	T	N	4.0 to 12	X			R42				B64
83C524	H	Y	1.2 to 16	X	X		N40		A44		B44
87C524	S	Y	3.5 to 20	X	X		N40	F40	A44	K44	B44
83C592 (80C592)	H	Y	1.2 to 16		X	-40 to +125			A68	K68	
87C592	H	Y	1.2 to 16	X			R42		A68	K68	
80C58	S	Y	3.5 to 16	X	X		N40		A44		B44
87C58	S	Y	3.5 to 16	X	X		N40	F40	A44	K44	B44
83C51FC	S	Y	3.5 to 24	X	X		N40		A44		B44
87C51FC	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83C528 (80C528)	H	Y	1.2 to 16	X	X	-40 to +125	N40		A44		B44
87C528	S	Y	3.5 to 20	X	X		N40	F40	A44	K44	B44
83CE528 (80CE528)	H	Y	1.2 to 16	X	X	-40 to +125			A44		B44
83CE598 (80CE598)	H	Y	1.2 to 16		X	-40 to +125					B80
87CE598	H	Y	3.5 to 16	X	X						B80
83CE558 80CE558	H	Y	1.2 to 16	X	X	-40 to +125					B80
89CE558	H	Y	1.2 to 16	X	X					Q80	B80

Notes: Production Centers are indicated in the second column: H – Hamburg, S – Sunnyvale, Z – Zurich.

All combinations of part type, speed, temperature and package may not be available.

- 1) Oscillator options start from 32kHz.
- 2) Also available in VSO40 package.
- 3) Also available in VSO56 Package.
- 4) Not recommended for new design.
- 5) Package available up to 16 MHz only.

8051 microcontroller cross-reference guide

	INTEL	SIEMENS	OKI	MATRA/HARRIS	PHILIPS SEMICONDUCTORS
CMOS	80C31BH	SAB 80C31	MSM80C31	80C31	PCB80C31BH-2/SC80C31BCC
	80C31BH-1		MSM80C31	80C31-1	PCB80C31BH-3/SC80C31BCG
	80C31BH-2			80C3151	/SC80C31BCB
	80C51BH	SAB 80C51	MSM80C51	80C51	PCB80C51BH-2/SC80C51BCC
	80C51BH-1		MSM80C51	80C51-1	PCB80C51BH-3/SC80C51BCG
	80C51BH-2			80C51	/SC80C51BCB
	87C51				SC87C51CC
	87C51-1				SC87C51CG
	87C51-2				SC87C51CB
	80C32	SAB80C32			P80C32EB
	80C32-1			80C32-25	P80C32GB
	80C52		SAB80C52		80C52-25
80C52-1				P80C52GB	
80C54				80C54	
83C54				83C54	
87C54				87C54	
80C58				80C58	
83C58				83C58	
87C58				87C58	
CMOS	83C51FA				S83C51FA
	87C51FA				S87C51FA
	83C51FB				S83C51FB
	87C51FB				S87C51FB
	83C51FC				S83C51FC
	87C51FC				S87C51FC

NOTES:

1. 80XXAHL = 80XX with low power standby pin; H = HMOS.

Low power / low voltage microcontroller family

80C51 LOW POWER FAMILY

Type	Available	ROM	RAM	I/O	I ² C	UART	Features	Package
80CL51	Yes	4k	128	32	No	Yes	Low Voltage 80C51	40-Pin Dual In-Line 40-Pin Very Small Outline 44-Pin Quad Flat Pack
80CL31	Yes	–	128	32	No	Yes	Low Voltage 80C31	40-Pin Dual In-Line 40-Pin Very Small Outline 44-Pin Quad Flat Pack
83CL410	Yes	4k	128	32	Yes	No	80CL51 with I ² C-bus	40-Pin Dual In-Line 40-Pin Very Small Outline 44-Pin Quad Flat Pack
80CL410	Yes	–	128	32	Yes	No	80CL51 with I ² C-bus	40-Pin Dual In-Line 40-Pin Very Small Outline 44-Pin Quad Flat Pack
83CL580	Yes	6k	256	40	Yes	Yes	ADC, PWM, Watchdog, T2	50-Pin Very Small Outline 64-Pin Quad Flat Pack
80CL580	Yes	–	256	40	Yes	Yes	ADC, PWM, Watchdog, T2	50-Pin Very Small Outline 64-Pin Quad Flat Pack
83CL781	Yes	16k	256	32	Yes	Yes	Low voltage 83C654, T2	40-Pin Dual In-Line 44-Pin Quad Flat Pack
83CL782	Yes	16k	256	32	Yes	Yes	Fast 83CL781: 12MHz/3V	40-Pin Dual In-Line 44-Pin Quad Flat Pack
85CL000	Yes	–	256	32	Yes	Yes	For SW development	Piggyback
85CL580	Yes	–	256	40	Yes	Yes	For SW development	Piggyback
85CL782	Yes	–	256	32	Yes	Yes	For SW development	Piggyback

LOW VOLTAGE DEVICES

Type	Available	ROM	RAM	I/O	I ² C	UART	Features	Package
83L51FA	Yes	8k	256	32	No	Yes	PCA, Enhanced UART 3.0V to 4.5V	40-Pin Dual In-Line 44-Pin PLCC 44-Pin Quad Flat Pack
87L51FA	Yes	8k EPROM/ OTP	256	32	No	Yes	PCA, Enhanced UART 3.0V to 4.5V	40-Pin Dual In-Line 44-Pin PLCC 44-Pin Quad Flat Pack
83L51FB	Yes	16k	256	32	No	Yes	PCA, Enhanced UART 3.0V to 4.5V	40-Pin Dual In-Line 44-Pin PLCC 44-Pin Quad Flat Pack
83L51FB	Yes	16k EPROM/ OTP	256	32	No	Yes	PCA, Enhanced UART 3.0V to 4.5V	40-Pin Dual In-Line 44-Pin PLCC 44-Pin Quad Flat Pack

80C51 microcontroller development system support

DEVELOPMENT SYSTEM CONTACTS

COMPANY	ADDRESS	TELEPHONE
Ashling Microsystems Limited	Plassey Technological Park Limerick, Ireland	(353) 61 334 466
	Eastern Systems Inc. 160 East Main Street Westboro, MA 01581	(508) 366-3220
BSO Tasking	Norfolk Place 333 Elm Street Dedham, MA 02026-4530	(800) 458-8276
Ceibo Ltd.	105 Gleason Rd. Lexington, MA 02173	(617) 863-9927
	Merkazim Building, Industrial Zone P.O. Box 2106 Herzeliya 46120, ISRAEL	972-52-555387
Lauterbach Datentechnik GmbH	Fichtenstrasse 27 85649 Hofolding Germany	49 8104 894 328
	945 Concord Street Framingham, MA 01701	(508) 620-4521
MetaLink Corp.	325 E. Elliot Road, Suite 23 Chandler, AZ 85225	(602) 926-0797
Nohau Corp.	51 E. Campbell Ave. Campbell, CA 95008-2053	(408) 866-1820
Philips Semiconductors	Corporate Centre Building BAE-2 P.O. Box 218 5600 MD Eindhoven The Netherlands	31-40-724223
SIGNUM Systems	171 E. Thousand Oaks Blvd., #202 Thousand Oaks, CA 91360	(805) 371-4608

EPROM PROGRAMMING SUPPORT CONTACTS

Advin Systems 1050-L East Duane Ave. Sunnyvale, CA 94086 (408) 736-2503	Logical Devices, Inc. 1201 Northwest 65th Place Ft. Lauderdale, FL 33309 (305) 974-0967	North Valley Products P.O. Box 32899 San Jose, CA 95152 (408) 929-5345
BP Microsystems 10681 Haddington #190 Houston, TX 77043 (800) 225-2102, (713) 461-9430	Logical Systems P. O. Box 6184 Syracuse, NY 13217-6184 (315) 478-0722	Strebor Data Communications 1008 N. Nob Hill American Fork, UT 84003 (801) 756-3605
Data I/O Corp. 10525 Willows Road N.E. P.O. Box 97046 Redmond, WA 98073-9746 (206) 881-6444	Needham's Electronics 4535 Orange Grove Ave. Sacramento, CA 95841 (916) 924-8037	

80C51 microcontroller development system support

SOFTWARE SUPPORT CONTACTS

COMPANY	ADDRESS	TELEPHONE
Archimedes Software, Inc.	2159 Union St. San Francisco, CA 94123	(415) 567-4010
BSO/Tasking	Tasking Software BV P.O. Box 899 3800 AW Amersfoort The Netherlands BSO Tasking 128 Technology Center P.O. Box 9164 Waltham, MA 02254-9164	31-33-55-85-84 (Telephone) 31-33-55-00-33 (Fax) (617) 894-7800 (Telephone) (617) 894-0551 (Fax) (710) 324-0760 (Telex) (800) 458-8276 (Toll Free)
Franklin Software, Inc.	888 Saratoga Ave. #2 San Jose, CA 95129	(408) 296-8051
Keil Software	Bretonischer Ring 15 85630 Grasbrunn Germany	49-89-46-50-57 (Telephone) 49-89-46-81-62 (Fax)

NOTE:

For more information on Development Support, see Section 9, Vol. 2, IC20.

8-bit microcontroller demonstration and evaluation boards

PRODUCT	DESCRIPTION
OM4151, S87C00K	I ² C demonstration board based on 80C51 derivatives
OM4238, P8051DB	8051 family demonstration board
OM4128	8XC552 evaluation board PEB552
OM4130, PCAN-EVAL	CAN controller evaluation board
OM4239	8XC592 evaluation board PEB592
OM4240	8XCE598 evaluation board PEB598
OM4241	8XCE598 evaluation board PDB598
OM4160, SM68070	68070 and 66470 demonstration and evaluation board Microcore 1
OM4160/2	68070 evaluation board Microcore 2
OM4162	9XCE201 evaluation board Microcore 4
OM4280, P83C852DEM	83C852 demonstration kit
OM4281 ¹	83C852 software evaluation kit
P8051DB	80C51 family development board
OM4717	83CL410 solar powered demonstration board
OM5005, P80CLEVAL	80CL51 evaluation board
DS750	8XC750 microcontroller in-circuit emulation development tool

NOTE:

1. The OM4281 is now available only from Ashling Microsystems Ltd. as type SCPC4281.

Microcontroller bulletin boards

To better serve our customers, Philips maintains two microcontroller bulletin boards. These computer bulletin board systems feature microcontroller newsletters, application and demonstration programs for download, and the ability to send messages to microcontroller application engineers.

The telephone numbers are:

North American Bulletin Board
300/1200/2400 baud 8-N-1
(800) 451-6644 (in the U.S.)
or
(408) 991-2406

European Bulletin Board
MAX 14.400 baud
Standards V32/V42/V42.bis/HST
+31 40 721102

European Application Help Desk
+31 40 722749
9a.m. – 16p.m. CET (Central European Time)

Sunnyvale ROMcode Bulletin Board

We also have a ROM code bulletin board through which you can submit ROM codes. This is a closed bulletin board for security reasons. To get an ID, contact your local sales office. The system can be accessed with a 2400, 1200, or 300 baud modem, and is available 24 hours a day.

The telephone number is:

(408) 991-3459

The following application note files are available on the Philips BBS:

App Note	BBS file name	App Note	BBS file name	Articles:
AN417	PRN256K.ZIP	AN434	I2CPCKB.ZIP	Add text overlay to any video display
AN420	INTRUPTS.ASM	AN435	IIC_OS.ZIP	CCI6.ZIP, MTV.ZIP
AN422	I2CAPP.ZIP	AN438	I2C528.EXE	
AN423	RS751.ASM	AN439	BATTCHRG.C	
AN424	WARMBOOT.ZIP	AN440	BOOTSTRP.ZIP	
AN425	I2C8584.ZIP	AN443	MAZEMOUS.ZIP	
AN427	TIMERI.ZIP	AN445	ABMOUSE.ZIP	
AN428	DEMO752.ASM	AN446	DUPUART.ZIP	
AN429	AN429.ZIP	AN447	AUTOBAUD.ZIP	
AN430	MM751.ZIP	EIE/AN91007	MM751B.ZIP	
AN433	SLV751.ZIP	EIE/AN91009	EEPRM851.ZIP	

Philips Fax-On-Demand System

What's the number?

1-800-282-2000

What is it?

The Fax-On-Demand system is a computer facsimile system that allows customers to receive selected documents by Fax automatically.

Philips Fax-On-Demand system is now set up to fax selected datasheets as customers request them, 24 hours per day, 7 days per week.

How does it work?

Each time the system receives a call, the voice card plays the pre-recorded messages, and waits for the caller's responses. Based on the caller's responses, the system will find and fax the appropriate document to the caller's fax machine.

To receive a document, the user must know the document number. This number can be obtained by asking for an index of available documents the first time that he/she calls the system.

How is it set up?

The Philips Fax-On-Demand system has six indexes (so far):

1. Communication
2. Audio & Video (in progress)
3. Microcontrollers
4. Logic
5. Linear
6. PLD

So far, the system has 600 data sheets. We expect this number to rise to 800 data sheets very soon and keep increasing every quarter. As you know, it will take approximately one minute to fax one page. This wouldn't be that bad if the number of pages is less than ten. But if the document is 37 pages long, be ready for a long transmission!!!

Philips Fax-On-Demand number is 1-800-282-2000, try it!

The following listing of products are those for which documents are available via "Fax-On-Demand" for the communications category.

Coming soon: Fax-On-Demand for our European-produced products.

Who do I contact if I have any questions about Fax-On-Demand?

Hamid Mohammadi
Phone: (408) 991-4895
Technical Support Center



CMOS and NMOS 8-bit microcontroller family

80C51 FAMILY CMOS

TYPE	ROM/ EPROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PHILIPS PROBES	THIRD PARTY PODs
80C31 80C51 87C51	0 4k ROM 4k EPROM	128 128 128	33 33 33	DIL40, LCC44 QFP44	UART, 2 timers	87C51:QFP package up to 16MHz	OM1092 + OM1097 (16MHz) OM4120S	8052PC(M) POD-C51B(N)
83C51FA 87C51FA	8k ROM 8k EPROM	256 256	24 24	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA		PDS51FBSD	8351FX(M) POD-C51FX(N)
83L51FA 87L51FA	8k ROM 8k EPROM	256 256	20 20	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA	3V to 4.5V operation		POD-L51P(N)
87C51FB 83C51FB	16k ROM 16k EPROM	256 256	24 24	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA		PDS51FBSD	8351FX(M) POD-C51FX(N)
87L51FB 83L51FB	16k ROM 16k EPROM	256 256	20 20	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA	3V to 4.5V operation		POD-L51P(N)
87C51FC 83C51FC	32k ROM 32k EPROM	256 256	24 24	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA			8351FX(M) POD-C51FX(N)
80C32 80C52 87C52	0 8k ROM 8k EPROM	256 256 256	24 24 24	DIL40, LCC44 QFP44	UART, 3 timers		OM1079 OM5012	8052PC(M) POD-C32(N)
80C54 87C54	16k ROM 16k EPROM	256 256	24 24	DIL40, LCC44 QFP44	UART, 3 timers		OM1079 OM5012	8052PC(M) POD-C32(N)
80C58 87C58	32k ROM 32k EPROM	256 256	24 24	DIL40, LCC44 QFP44	UART, 3 timers		OM1079 OM5012	8052PC(M) POD-C32(N)
80C451 83C451 87C451	0 4k ROM 4k EPROM	128 128 128	16 16 16	DIP64/LCC68	UART, 2 timers Extended I/O		OM4123	83C451PC(M) POD-C451B(N)
83C504 87C504	16K ROM 16K EPROM	256 256	24 24	DIL40, LCC44 QFP44	24 by 8 divide, 2 timers			
87C524 83C524	16k EPROM 16k ROM	512 512	20 12	DIL40/LCC44 QFP44	UART, 3 timers Watchdog timer Bit I ² C		OM4111 + OM4110 + OM4120S	83528PC(M) POD-C528(N)
83C528 87C528	32k ROM 32k EPROM	512 512	16 16, 20	DIL40/LCC44 (QFP44)	UART, 3 timers Watchdog timer Bit I ² C		OM4111 + OM4110 + OM4120S	83C528PC(M) POD-C528(N)
83CE528	32kROM	512	16	CE ONLY QFP				
80C550 83C550 87C550	0 4k ROM 4k EPROM	128 128 128	16 16 16	LCC44 DIL40	UART, 2 timers 8 8-bit ADC inputs, watchdog timer		OM5055 + OM4110	83550(M) POD-C550(N)
80C552 83C552 87C552	0 8k ROM 8k EPROM	256 256 256	16, 24 16, 24 16	LCC68/QFP80	UART, 2 timers Timer with compare and capture, 2 PWM outputs, 8 10-bit ADC inputs, Byte I ² C		OM1092 + OM1095 + OM4120S OM4128	83C552PC(M) POD-C552B(N)
83CE558 89CE558 80CE558	32K ROM 32K FLASH 0	1K 1K	16 16	QFP80	As 8xC552 with PLL-oscillator Auto scan ADC	89C: Q4-92 83C: Q2/3-93	OM4247	
80C562 83C562	0 8k ROM	256 256	16 16	LCC68/QFP80	UART, 2 timers Timer with compare and capture, 2 PWM outputs, 8 8-bit ADC inputs		OM1092 + OM1095 + OM4120S	83C552PC(M) POD-C552B(N)
80C575 83C575 87C575	0 8k 8k EPROM	256 256 256	16 16 16	DIL40, LCC44 QFP44	3 timers 1 Enh. UART, PCA, 4 analog comparators			POD-C575(N)
83C576 87C576	8k ROM 8k EPROM	256 256	16 16	DIL40, LCC44, SDIL42	10-bit A/D, 3 timers, PCA, Watchdog timer			
80C592 83C592 87C592	0 16k ROM 16k EPROM	512 512 512	16 16 16	LCC68/QFP80	8XC552 + CAN interface		OM4110 + OM4112 + OM4120S	POD-592(N)

M = Metlink
N = Nohau

CMOS and NMOS 8-bit microcontroller family

80C51 FAMILY CMOS (Continued)

TYPE	ROM/ EPROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PHILIPS PROBES	THIRD PARTY PODs
87CE598 87CE598 80CE598	32K ROM 32K EPROM 0	512 512 512	16 16 16	QFP80	8xC552 + CAN interface. No I ² C	87CE: prod: Q2'94		
80C652 83C652 87C652	0 8k ROM 8k EPROM	256 256 256	16, 24 16, 24 16, 20	DIL40/LCC44 QFP44	UART, 2 timers Byte I ² C		OM1092 + OM1096 + OM4120S	83652PC(M) POD-C51B(N)
83C654 87C654	16k ROM 16k EPROM	256 256	16,24 16,20	DIL40/LCC44 QFP44	UART, 2 timers Byte I ² C		OM1092 + OM1096 + OM4120S	83654(M) POD-C51B(N)
83CE654 80CE654	16k ROM 0	256 256	16 16	QFP44	UART, 2 timers Byte I ² C	83C654 with Electromagnetic Compatibility improvements	OM1092 + OM1096 + OM4120S	83654(M) POD-C51B(N)
83C750 87C750	1K ROM 1KEPROM	64 64	40 40	SDIP24 skinny	1 timer		OM1094	83751PC(M)
83C751 83C748 87C751 87c748	2k ROM 2k EPROM	64 64	16 16	DIP24 skinny LCC28 DIP24 skinny	1 timer Bit I ² C (8XC751 only)		OM1094P	83751PC(M) POD-C751(N)
83C752 83C749 87C752 87c752	2k ROM 2k EPROM	64 64	16 16	DIP28, LCC28 DIP 28, LCC28	1 timer, PWM output, 5 8-bit ADC inputs, Bit I ² C (8XC752 only)		OM5072	83752A(M) POD-C752(N)
80C851 83C851	0 4k ROM	128 128	16 16	DIL40/LCC44 QFP44	UART, 2 timers 256 byte		OM1092 + OM4120S	80851PC(M) POD-C51(N)
83C852	6k ROM	256	6		2k byte EEPROM smart card hardware CU			
83C055 87C055	16k ROM 16k EPROM	256 256	12 12	DIP42 Shrunk DIP42 Shrunk	As 8XC053	In dev.	OM5054	

* The following microcontrollers have no external memory access: 8XC751, 8XC752, 8XC053, 87C054, 83C852.

M = Metlink

N = Nohau

CMOS and NMOS 8-bit microcontroller family

80CLXXX FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
85CL000	0	256	12	Piggyback	Piggyback CL410, CL411, CL51, P80C51			
85CL580	0	256	12	Piggyback	Piggyback CL580			
85CL781	0	256	12	Piggyback	Piggyback CL781, CL782, CL52			
80CL51 80CL31	4K 0	128 128	16 16	DIL40 VSO40	2 timers, UART		OM1079	QFP: OM5020
83CL410 80CL410	4k 0	128 128	12 12	DIL40 VSO40	2 timers Byte I ² C		OM1079	QFP: OM5020
83CL580	6k	256	16	QFP64/ VSO56	3timers, UART Watchdog timer Byte I ² C, 1 PWM 4*8 bit ADC		OM1079 + OM5004	OM1079: Probe base OM5004: Probe adap
83CL781 83CL782	16k 16k	256 256	12 @ 4.5V 12 @ 3V	DIL40 QFP44	3timers, UART Byte I ² C		OM1079 + OM5004 + tbd	OM1079: Probe base OM5004: Probe adap
83CL167 83CL267	16K 12K	256 256	12 12	SDIL64 QFP64	3timers 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 4*4 bit ADC Byte I ² C 160 char OSD 126 char fonts 4 char sizes Shadow modes ODS PLL osc. 10MHz Blinking	In Dev	OM4840 OM1079	
83CL168 83CL268	16K 12K	256 256	12 12	SDIL64 QFP64	3timers 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 4*4 bit ADC RC preprocessor Byte I ² C 3 wire serial I/O 160 char OSD 126 char fonts 4 char sizes Shadow modes ODS PLL osc. 10MHz Blinking	In Dev	OM4840 + OM1079	

CMOS and NMOS 8-bit microcontroller family

8051 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	THIRD PARTY EMULATOR
8051 8031	4k 0	128 128	15 15	DIL40/PLCC44 DIL40/PLCC44	UART, 2 timers		OM1092 + OM1097 + OM4120S	8052PC(M) OPD-C51B(N)
8052 8032	8k 0	256 256	15 15	DIL40/PLCC44 DIL40/PLCC44	UART, 3 timers UART, 3 timers		OM4111 + OM4110 + OM4120S	8052PC(M) OPD-C51B(N)

CMOS and NMOS 8-bit microcontroller family

8400 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
84C21A 84C41A 84C81A	2k 4k 8k	64 128 256	10 10 10	DIL28/SO28 DIL28/SO28 DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C		OM1083	OM1025 (LSDS)
84C22A 84C42A 84C12A	2k 4k 1k	64 64 64	10 10 16	DIL20/SO20 DIL20/SO20 DIL20/SO20	13 I/O lines 8-bit timer		OM1083 + Adapter_1	OM1025 (LSDS)
84C00B 84C00T	0 0	256 256	10 10	28 pins VSO-56	20 I/O lines 8-bit timer Byte I ² C	Piggyback ROMless	OM1080 OM1080	
84C121 84C121B	1k 0	64 64	10 10	DIL20/SO20 	13 I/O lines 2 8-bit timers 8 bytes EEPROM	 Piggyback	OM1073	OM1025(LEDSD) OM1027
84C122A 84C122B 84C422A 84C422B 84C822A 84C822B 84C822C	1k 4K 8K	32 32 32	10	A: SO20 B: SO24 C: SO28	Controller for remote control A: 12 I/O B: 16 I/O C: 20 I/O		OM4830	
84C230	2l	64	10	DIL40/VSO40	12 I/O lines 8-bit timer 16*4 LCD drive		OM1072	
84C430 84C430BH	4k 0	128 128	10 10	QFP64	24 I/O lines 8-bit timer Byte I ² C 24*4 LCD drive	 Piggyback for C230 and C430	OM1072	
84C633 84C633B	6k 0	256 256	16 16	VSO56	28 I/O lines 8-bit timer 16-bit up/down counter 16-bit timer with compare and capture 16*4 LCD drive		OM1086	
84C440 84C441 84C443 84C444 84C640 84C641 84C643 84C644 84C840 84C841 84C843 84C844	4k 4k 4k 4k 6k 6k 6k 6k 8k 8k 8k 8k	128 128 128 128 128 128 128 128 192 192 192 192	10 10 10 10 10 10 10 10 10 10 10 10	DIP42 shrunk	RC: 29 I/O lines LC: 28 I/O lines 8-bit timer 1 14-bit PWM 5 6-bit PWM 3-bit ADC OSD 2L-16	I ² C, RC I ² C, LC RC LC I ² C, RC I ² C, LC RC LC I ² C, RC I ² C, LC RC LC	OM1074	For emulation of LC versions, use OM1074 + adapter_3 + 2 adapter_5 Baud for LCDS OM4831

CMOS and NMOS 8-bit microcontroller family

8400 FAMILY CMOS (Continued)

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
84C646 84C846	6k 8k	192 192	10 10	DIP42 shrunk	30 I/O lines DOS clock = PLL 8 bit timer 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 3-4 bit ADC DOS: 64 disp. RAM 62 char. fonts Char. blinking Shadow modes 8 foreground colors/char. 8 background colors/word DOS: clock: 8 .. 20MHz	I ² C, RC I ² C, RC	OM4829 + OM4832	OM4833 for LCD584
84C85 84C85B	8k 0	256 256	10 10	DIL40/VSO40	32 I/O lines 8-bit timer Byte I ² C	Piggyback for C85	OM1070	
84C853 84C853B	8k 0	256 256	16 16	DIL40/VSO40	33 I/O lines 8-bit timer 16-bit up/down counter 16-bit timer with compare and capture	Piggyback for C853	OM1081	
84C270 84C470 84C270B 84C470B	2k 4k 0 0	128 128 128 128	10 10 10 10	DIL40/VSO40 DIL40/VSO40	8 I/O lines 16*8 capture keyboard matrix 8-bit timer 470 also handles mech. keys	Piggyback for C270 Piggyback for C470	OM1077	
84C271	2k	128	10	DIL40	8 I/O lines 16*8 mech. keyboard matrix 8-bit timer		OM1078	

8400 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	EMULATOR TOOLS	REMARKS
8411 8421 8441 8461	1k 2k 4k 6k	64 64 128 128	6 6 6 6	DIL28/SO28 DIL28/SO28 DIL28/SO28 DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C			OM1025 (LCDS) + OM1026
8422 8442	2k 4k	64 128	6 6	DIL20 DIL20	13 I/O lines 8-bit timer Bit I ² C			
8401B	0	128	6	28-pin		Piggyback for 84X1		

CMOS and NMOS 8-bit microcontroller family

3300 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
3315A	1.5k	160	10	DIL28/SO28	20 I/O lines 8-bit timer $V_{DD} > 1.8V$		OM1083	OM1025(LCDS)
3343	3k	224	10	DIL28/SO28	20 I/O lines 8-bit timer $V_{DD} > 1.8V$ Byte I ² C		OM1083	OM1025(LCDS)
3344A	2k	224	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM1071	OM1025(LCDS) + OM1028
3346A	4k	128	10	DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C 256 bytes EEPROM $V_{DD} < 1.8V$		OM1076	
3347	1.5k	64	3.58	DIL20/SO20	12 I/O lines 8-bit timer DTMF generator		OM1071 + Adapter_2	OM1025(LCDS) + OM1028
3348A	8k	256	10	DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C $V_{DD} < 1.8V$		OM1083	OM1025(LCDS)
3349A	4k	224	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM1071	OM1025(LCDS) + OM1028
3350A	8k	128	3.58	VSO64	30 I/O lines 8-bit timer DTMF generator 256 bytes EEPROM			
3351A	2k	64	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator 128 bytes EEPROM		OM5000	
3352A	6k	128	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator 128 byte EEPROM		OM5000	
3353A	6k	128	16	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator Ringer out 128 bytes EEPROM	March '92	OM5000	
3354A	8k	256	16	QFP64	36 I/O lines 8-bit timer DTMF generator Ringer out 256 bytes EEPROM	June '92	OM4829 + OM5003	OM4829: Probe base
8755A	0	128	16	DIL28/SO28	8k OTP 20 I/O lines 8-bit timer DTMF generator Melody output 128 bytes EEPROM	In Development		
3301B						Piggyback for 3315, 3343, 3348	OM1083	
3344B						Piggyback for 3344, 3347, 3349	OM1071	
3346B						Piggyback for 3346	OM1076	

CMOS and NMOS 8-bit microcontroller family

3300 FAMILY CMOS (Continued)

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
3350B						Piggyback for 3350A	OM4829+ OM5003	
3351B						Piggyback for 3351A, 3352A, 3353A	OM5000	
3354B						Piggyback for 3354A	OM4829+ OM5010	

CMOS 16-bit microcontroller family

16-BIT CONTROLLERS (68000 ARCHITECTURE)

TYPE	(EP)ROM	RAM	SPEED (MHz)	FUNCTIONS	REMARKS	PHILIPS TOOLS	THIRD-PARTY TOOLS
68070	—	—	17.5	2 DMA channels, MMU, UART, 16-bit timer, I ² C, 68000 bus interface, 16Mb address range		OM4160 Microcore 1 OM4160/2 Microcore 2 OM4161 (SBE68070) OM4767/2 XRAY68070SBE high level symbolic debugger OM4222 68070DS development system OM4226 XRAY68070DS high level symbolic debugger	TRACE32-ICE68070 (Lauterbach)
93C101	34k	512	15	Derivative with low power modes	Not for new design		
90CE201	16MB external ROM	16MB external RAM	24	UART, fast I ² C, 3 timers (16 bit), Watchdog timer. 68000 software compatible, EMC, QFP64	-25 to +85°C	OM4162 Microcore 4	TRACE32 – (Lauterbach)

16-BIT CONTROLLERS (XA ARCHITECTURE)

TYPE	(EP)ROM	RAM	SPEED (MHz)	FUNCTIONS	REMARKS	DEVELOPMENT TOOLS
XA-G1	8k	512	30	3 timers, watchdog, 2 UARTs	-40 to +125°C	Nohau Ceibo MacCraiger Systems
XA-G2	16k	512	30	3 timers, watchdog, 2 UARTs	-40 to +125°C	Nohau Ceibo MacCraiger Systems
XA-G3	32k	512	30	3 timers, watchdog, 2 UARTs	-40 to +125°C	Nohau Ceibo MacCraiger Systems

Ordering Information

MICROCONTROLLER PRODUCTS

Example: P 8 X C X X X E B P N

0 = ROMLESS
 5 = Bond-Out (emulation)
 3 = ROM
 7 = EPROM/OTP
 9 = FEEPROM (FLASH)

Exceptions:
 P80C32 = ROMless
 P80C52 = ROM

This can be 2 or 3 digits

Speed
 C = 12MHz
 E = 3.5MHz to 16MHz
 F = 1.2MHz to 16MHz
 G = 20MHz
 H = 32kHz to 12MHz
 I = 24MHz
 P = 40MHz

Philips North America Package Code
 A = Plastic Leaded Chip Carrier (PLCC)
 B = Quad Flat Pack (QFP)
 FA = Hermetic Cerdip (window)
 KA = CerQuad (window)
 N = Plastic Dual In-Line

Philips Package Code
 A = Plastic Leaded Chip Carrier (PLCC)
 B = Quad Flat Pack (QFP)
 F = Hermetic Cerdip (window)
 L = Cerquad (window)
 P = Plastic Dual In-Line
 Q = Ceramic Quad Flat Pack (window)

Temperature
 B = 0°C to +70°C
 F = -40°C to +85°C
 H = -40°C to +125°C

Example: PSD3XX - 12 I B

Basic Part Number

- = 5V Standard Operation
 L = 3V Low Voltage Operation

Package Code
 A = Plastic Leaded Chip Carrier (PLCC)
 B = Plastic Quad Flat Pack (QFP)
 KA = Ceramic Leaded Chip Carrier (CLCC)

Window
 No*
 No**
 Yes**

Operating Temperature Range
 Blank = Commercial: 0°C to +70°C
 I = Industrial: -40°C to +85°C

Access Time
 X10ns

* Surface Mount
 ** Socketing Recommended

Example: SC 8 X C X X X B C C N 40

0 = ROMLESS
 3 = ROM
 7 = EPROM/OTP

Exceptions:
 SC80C31 = ROMless
 SC80C51 = ROM

This can be 2 or 3 digits

Pin Count

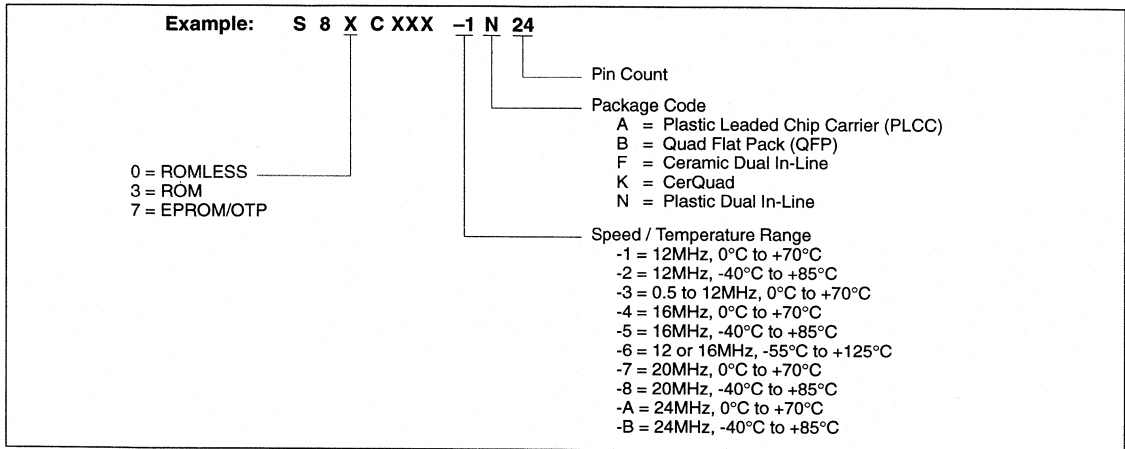
Package Code
 A = Plastic Leaded Chip Carrier (PLCC)
 B = Quad Flat Pack (QFP)
 F = Ceramic Dual In-Line
 FA = Hermetic Cerdip (window)
 KA = CerQuad (window)
 L = Chip Carrier, Leaded
 N = Plastic Dual In-Line

Speed
 B = 0.5 to 12MHz
 C = 12MHz
 G = 16MHz
 L = 20MHz
 P = 24MHz
 Y = 33MHz

Temperature
 C = Commercial 0°C to +70°C
 A = Industrial -40°C to +85°C

Revision (optional)

Ordering Information



Section 2

Inter-Integrated Circuit (I²C) Bus

**Application Notes and
Development Tools for
80C51 Microcontrollers**

CONTENTS

The I ² C-bus and how to use it	39
I ² C peripheral selection guide	58
82B715 I ² C bus extender	60

The I²C-bus and how to use it

The I²C-bus and how to use it (including specification)

1.0 THE I²C-BUS BENEFITS DESIGNERS AND MANUFACTURERS

In consumer electronics, telecommunications and industrial electronics, there are often many similarities between seemingly unrelated designs. For example, nearly every system includes:

- Some intelligent control, usually a single-chip microcontroller
- General-purpose circuits like LCD drivers, remote I/O ports, RAM, EEPROM, or data converters
- Application-oriented circuits such as digital tuning and signal processing circuits for radio and video systems, or DTMF generators for telephones with tone dialling

To exploit these similarities to the benefit of both systems designers and equipment manufacturers, as well as to maximize hardware efficiency and circuit simplicity, Philips developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter IC or I²C-bus. At present, Philips' IC range includes more than 150 CMOS and bipolar I²C-bus compatible types for performing functions in all three of the previously mentioned categories. All I²C-bus compatible devices incorporate an on-chip interface which allows them to communicate directly with each other via the I²C-bus. This design concept solves the many interfacing problems encountered when designing digital control circuits.

Here are some of the features of the I²C-bus:

- Only two bus lines are required; a serial data line (SDA) and a serial clock line (SCL)
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers
- It's a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the standard mode or up to 400 kbit/s in the fast mode
- On-chip filtering rejects spikes on the bus

data line to preserve data integrity

- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF

Figure 1 shows two examples of I²C-bus applications.

1.1 Designer Benefits

I²C-bus compatible ICs allow a system design to rapidly progress directly from a functional block diagram to a prototype. Moreover, since they 'clip' directly onto the I²C-bus without any additional external interfacing, they allow a prototype system to be modified or upgraded simply by 'clipping' or 'unclipping' ICs to or from the bus.

Here are some of the features of I²C-bus compatible ICs which are particularly attractive to designers:

- Functional blocks on the block diagram correspond with the actual ICs; designs proceed rapidly from block diagram to final schematic
- No need to design bus interfaces because the I²C-bus interface is already integrated on-chip
- Integrated addressing and data-transfer protocol allow systems to be completely software-defined
- The same IC types can often be used in many different applications
- Design-time reduces as designers quickly become familiar with the frequently used functional blocks represented by I²C-bus compatible ICs
- ICs can be added to or removed from a system without affecting any other circuits on the bus
- Fault diagnosis and debugging are simple; malfunctions can be immediately traced
- Software development time can be reduced by assembling a library of reusable software modules.

In addition to these advantages, the CMOS ICs in the I²C-bus compatible range offer designers special features which are particularly attractive for portable equipment and battery-backed systems.

They all have:

- Extremely low current consumption
- High noise immunity

- Wide supply voltage range
- Wide operating temperature range.

1.2 Manufacturer benefits

I²C-bus compatible ICs don't only assist designers, they also give a wide range of benefits to equipment manufacturers because:

- The simple 2-wire serial I²C-bus minimizes interconnections so ICs have fewer pins and there are not so many PCB tracks; result – smaller and less expensive PCBs
- The completely integrated I²C-bus protocol eliminates the need for address decoders and other 'glue logic'
- The multi-master capability of the I²C-bus allows rapid testing and alignment of end-user equipment via external connections to an assembly-line computer
- The availability of I²C-bus compatible ICs in SO (small outline), VSO (very small outline) as well as DIL packages reduces space requirements even more.

These are just some of the benefits.

In addition, I²C-bus compatible ICs increase system design flexibility by allowing simple construction of equipment variants and easy upgrading to keep designs up-to-date. In this way, an entire family of equipment can be developed around a basic model. Upgrades for new equipment, or enhanced-feature models (i.e. extended memory, remote control, etc.) can then be produced simply by clipping the appropriate ICs onto the bus. If a larger ROM is needed, it's simply a matter of selecting a microcontroller with a larger ROM from our comprehensive range. As new ICs supersede older ones, it's easy to add new features to equipment or to increase its performance by simply unclipping the outdated IC from the bus and clipping on its successor.

1.3 The ACCESS.bus

Another attractive feature of the I²C-bus for designers and manufacturers is that its simple 2-wire nature and capability of software addressing make it an ideal platform for the ACCESS.bus (Fig.2). This is a lower-cost alternative for an RS-232C interface for connecting peripherals to a host computer via a simple 4-pin connector (see Section 19).

The I²C-bus and how to use it

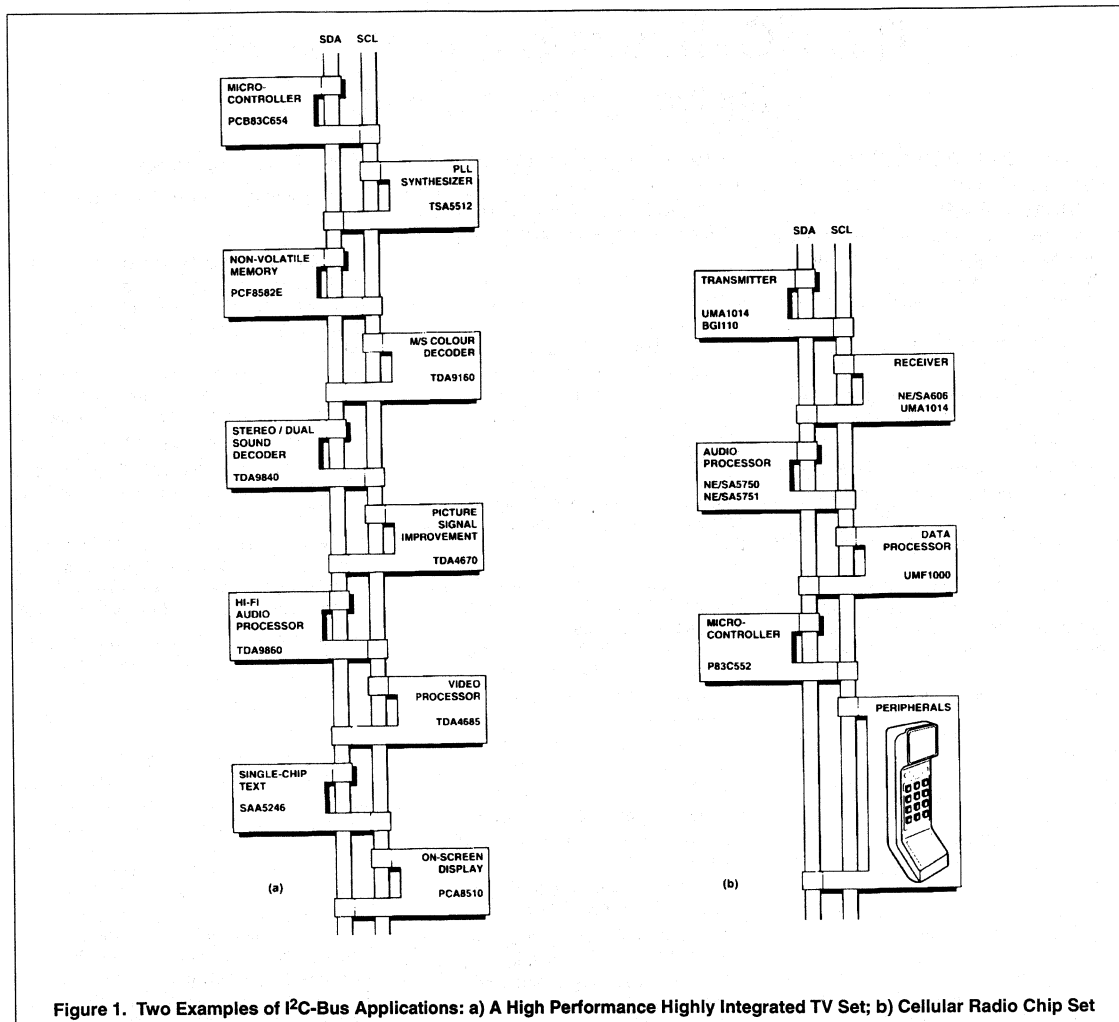


Table 1. Definition of I²C-Bus Terminology

Term	Description
Transmitter	The device which sends the data to the bus
Receiver	The device which receives the data from the bus
Master	The device which initiates a transfer, generates clock signals and terminates a transfer
Slave	The device addressed by a master
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted
Synchronization	Procedure to synchronize the clock signals of two or more devices

The I²C-bus and how to use it

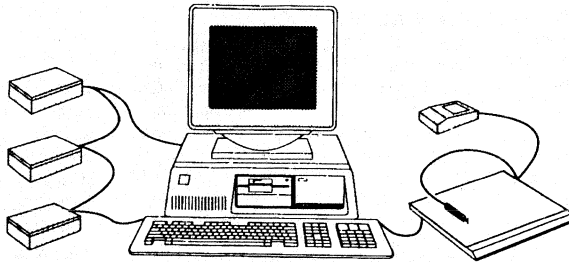


Figure 2. The ACCESS.bus - A Low-Cost Alternative to an RS-232C Interface

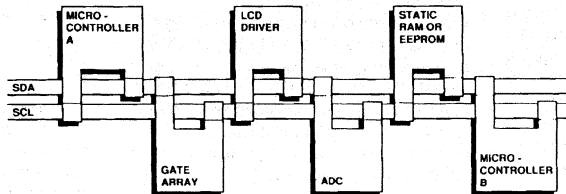


Figure 3. Examples of an I²C-Bus Configuration Using Two Microcontrollers

2.0 INTRODUCTION TO THE I²C-BUS SPECIFICATION

For 8-bit digital control applications, such as those requiring microcontrollers, certain design criteria can be established:

- A complete system usually consists of at least one microcontroller and other peripheral devices such as memories and I/O expanders
- The cost of connecting the various devices within the system must be minimized
- A system that performs a control function doesn't require high-speed data transfer
- Overall efficiency depends on the devices chosen and the nature of the interconnecting bus structure.

In order to produce a system to satisfy these criteria, a serial bus structure is needed. Although serial buses don't have the throughput capability of parallel buses, they do require less wiring and fewer IC connecting pins. However, a bus is not merely an interconnecting wire, it embodies all the formats and procedures for communication within the system.

Devices communicating with each other on a serial bus must have some form of protocol which avoids all possibilities of confusion, data loss and blockage of information. Fast devices must be able to communicate with slow devices. The system must not be dependent on the devices connected to it,

otherwise modifications or improvements would be impossible. A procedure has also to be devised to decide which device will be in control of the bus and when. And, if different devices with different clock speeds are connected to the bus, the bus clock source must be defined. All these criteria are involved in the specification of the I²C-bus.

3.0 THE I²C-BUS CONCEPT

The I²C-bus supports any IC fabrication process (NMOS, CMOS, bipolar). Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus. Each device is recognised by a unique address – whether it's a microcontroller, LCD driver, memory or keyboard interface – and can operate as either a transmitter or receiver, depending on the function of the device. Obviously an LCD driver is only a receiver, whereas a memory can both receive and transmit data. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers (see Table 1). A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.

The I²C-bus is a multi-master bus. This means that more than one device capable of

controlling the bus can be connected to it. As masters are usually micro-controllers, let's consider the case of a data transfer between two microcontrollers connected to the I²C-bus (Fig.3). This highlights the master-slave and receiver-transmitter relationships to be found on the I²C-bus. It should be noted that these relationships are

not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows:

1. Suppose microcontroller A wants to send information to microcontroller B:
 - microcontroller A (master), addresses microcontroller B (slave)
 - microcontroller A (master-transmitter), sends data to microcontroller B (slave-receiver)
 - microcontroller A terminates the transfer.
2. If microcontroller A wants to receive information from microcontroller B:
 - microcontroller A (master) addresses microcontroller B (slave)
 - microcontroller A (master-receiver) receives data from microcontroller B (slave-transmitter)
 - microcontroller A terminates the transfer.

Even in this case, the master (microcontroller A) generates the timing and terminates the transfer.

The possibility of connecting more than one microcontroller to the I²C-bus means that

The I²C-bus and how to use it

more than one master could try to initiate a data transfer at the same time. To avoid the chaos that might ensue from such an event – an arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I²C interfaces to the I²C-bus.

If two or more masters try to put information onto the bus, the first to produce a 'one' when the other produces a 'zero' will lose the arbitration. The clock signals during arbitration are a synchronized combination of the clocks generated by the masters using the wired-AND connection to the SCL line (for more detailed information concerning arbitration see Section 7.0).

Generation of clock signals on the I²C-bus is always the responsibility of master devices; each master generates its own clock signals when transferring data on the bus. Bus clock signals from a master can only be altered when they are stretched by a slow-slave device holding-down the clock line, or by another master when arbitration occurs.

4.0 GENERAL CHARACTERISTICS

Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a pull-up resistor (see Fig.4). When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector in order to perform the wired-AND function. Data on the I²C-bus can be transferred at a rate up to 100 kbit/s in the standard-mode, or up to 400 kbit/s in the fast-mode. The number of interfaces connected to the bus is solely dependent on the bus capacitance limit of 400 pF.

5.0 BIT TRANSFER

Due to the variety of different technology devices (CMOS, NMOS, bipolar) which can be connected to the I²C-bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the associated level of V_{DD} (see Section 15.0 for Electrical Specifications). One clock pulse is generated for each data bit transferred.

5.1 Data Validity

The data on the SDA line must be stable

during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Fig.5).

5.2 START and STOP Conditions

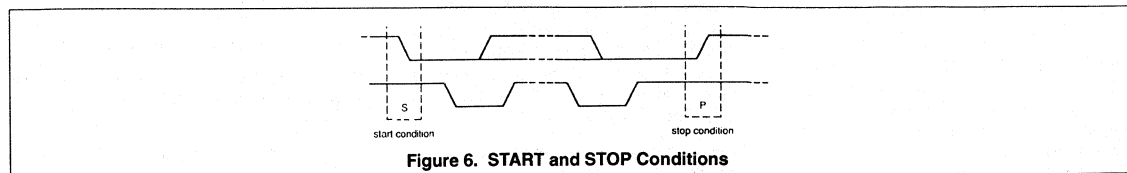
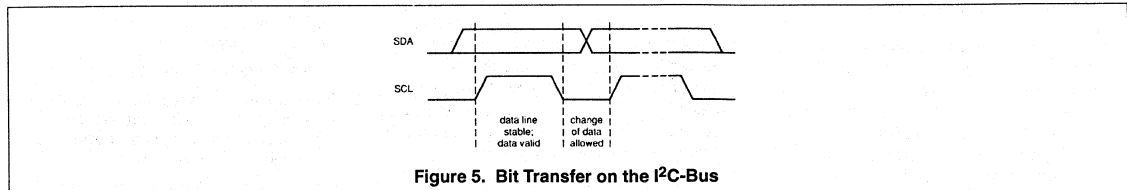
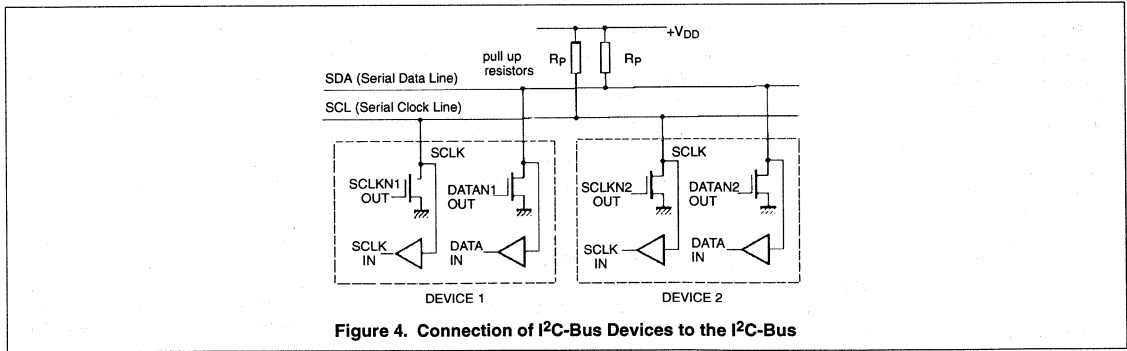
Within the procedure of the I²C-bus, unique situations arise which are defined as START and STOP conditions (see Fig.6).

A HIGH to LOW transition on the SDA line while SCL is HIGH is one such unique case. This situation indicates a START condition.

A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. This bus free situation is specified in Section 15.0.

Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period in order to sense the transition.



The I²C-bus and how to use it

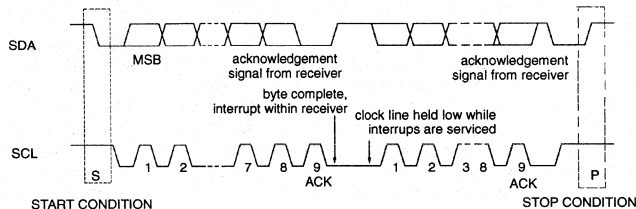


Figure 7. Data Transfer on the I²C-Bus

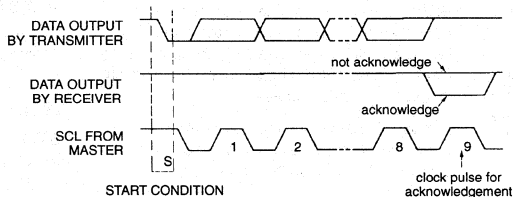


Figure 8. Acknowledge on the I²C-Bus

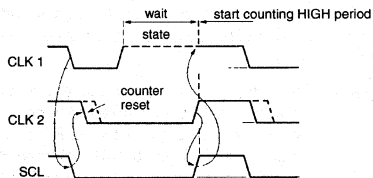


Figure 9. Clock Synchronization During the Arbitration Procedure

6.0 TRANSFERRING DATA

6.1 Byte Format

Every byte put on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first (Fig.7). If a receiver can't receive another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the transmitter into a wait state. Data transfer then continues when the receiver is ready for another byte of data and releases clock line SCL.

In some cases, it's permitted to use a different format from the I²C-bus format (for CBUS compatible devices for example). A message which starts with such an address can be terminated by generation of a STOP

condition, even during the transmission of a byte. In this case, no acknowledge is generated (see Section 9.1.3).

6.2 Acknowledge

Data transfer with acknowledge is obligatory. The acknowledge-related clock pulse is generated by the master. The transmitter releases the SDA line (HIGH) during the acknowledge clock pulse.

The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse (Fig.8). Of course, set-up and hold times (specified in Section 15) must also be taken into account.

Usually, a receiver which has been addressed is obliged to generate an acknowledge after each byte has been received, except when the message starts with a CBUS address (see Section 9.1.3).

When a slave-receiver doesn't acknowledge the slave address (for example, it's unable to receive because it's performing some real-time function), the data line must be left HIGH by the slave. The master can then generate a STOP condition to abort the transfer.

If a slave-receiver does acknowledge the slave address but, some time later in the transfer cannot receive any more data bytes, the master must again abort the transfer. This is indicated by the slave generating the not acknowledge on the first byte to follow. The slave leaves the data line HIGH and the master generates the STOP condition.

If a master-receiver is involved in a transfer, it must signal the end of data to the slave-transmitter by not generating an acknowledge on the last byte that was clocked out of the slave. The slave-transmitter must release the data line to allow the master to generate the STOP condition.

The I²C-bus and how to use it

7.0 ARBITRATION AND CLOCK GENERATION

7.1 Synchronization

All masters generate their own clock on the SCL line to transfer messages on the I²C-bus. Data is only valid during the HIGH period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using the wired-AND connection of I²C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and, once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached (Fig.9). However, the LOW to HIGH transition of this clock may not change the state of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the device with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time.

When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW.

In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period.

7.2 Arbitration

A master may start a transfer only if the bus is free. Two or more masters may generate a START condition within the minimum hold

time ($t_{HD;STA}$) of the START condition which results in a defined START condition to the bus.

Arbitration takes place on the SDA line, while the SCL line is at the HIGH level, in such a way that the master which transmits a HIGH level, while another master is transmitting a LOW level will switch off its DATA output stage because the level on the bus doesn't correspond to its own level.

Arbitration can continue for many bits. Its first stage is comparison of the address bits (addressing information is in Sections 9.0 and 13.0). If the masters are each trying to address the same device, arbitration continues with comparison of the data. Because address and data information on the I²C-bus is used for arbitration, no information is lost during this process.

A master which loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration.

If a master also incorporates a slave function and it loses

arbitration during the addressing stage, it's possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave-receiver mode.

Figure 10 shows the arbitration procedure for two masters. Of course, more may be involved (depending on how many masters are connected to the bus). The moment there is a difference between the internal data level of the master generating DATA 1 and the actual level on the SDA line, its data output is switched off, which means that a HIGH output level is then connected to the bus. This will not affect the data transfer initiated by the winning master.

Since control of the I²C-bus is decided solely on the address and data sent by competing

masters, there is no central master, nor any order of priority on the bus.

Special attention must be paid if, during a serial transfer, the arbitration procedure is still in progress at the moment when a repeated START condition or a STOP condition is transmitted to the I²C-bus. If it's possible for such a situation to occur, the masters involved must send this repeated START condition or STOP condition at the same position in the format frame. In other words, arbitration isn't allowed between:

- A repeated START condition and a data bit
- A STOP condition and a data bit
- A repeated START condition and a STOP condition.

7.3 Use of the Clock Synchronising Mechanism as a Handshake

In addition to being used during the arbitration procedure, the clock synchronization mechanism can be used to enable receivers to cope with fast data transfers, on either a byte level or a bit level.

On the byte level, a device may be able to receive bytes of data at a fast rate, but needs more time to store a received byte or prepare another byte to be transmitted. Slaves can then hold the SCL line LOW after reception and acknowledgement of a byte to force the master into a wait state until the slave is ready for the next byte transfer in a type of handshake procedure.

On the bit level, a device such as a microcontroller without, or with only a limited hardware I²C interface on-chip can slow down the bus clock by extending each clock LOW period. The speed of any master is thereby adapted to the internal operating rate of this device.

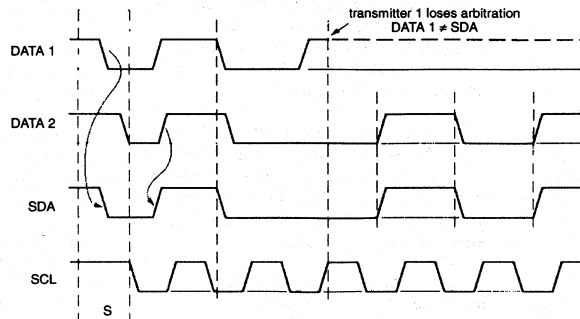


Figure 10. Arbitration Procedure of Two Masters

The I²C-bus and how to use it

8.0 FORMATS WITH 7-BIT ADDRESSES

Data transfers follow the format shown in Fig.11. After the START condition (S), a slave address is sent. This address is 7 bits long followed by an eighth bit which is a data direction bit (R/W) – a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. Various combinations of read/write formats are then possible within such a transfer.

Possible data transfer formats are:

- Master-transmitter transmits to slave-receiver. The transfer direction is not changed (Fig.12)
- Master reads slave immediately after first byte (Fig.13). At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This acknowledge is still generated by the slave. The STOP condition is generated by the master
- Combined format (Fig.14). During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed.

NOTES:

1. Combined formats can be used, for example, to control a serial memory. During the first data byte, the internal memory location has to be written. After the START condition and slave address is repeated, data can be transferred.
2. All decisions on auto-increment or decrement of previously accessed memory locations etc. are taken by the designer of the device.
3. Each byte is followed by an acknowledgement bit as indicated by the A or \bar{A} blocks in the sequence.
4. I²C-bus compatible devices must reset their bus logic on receipt of a START or repeated START condition such that they all anticipate the sending of a slave address.

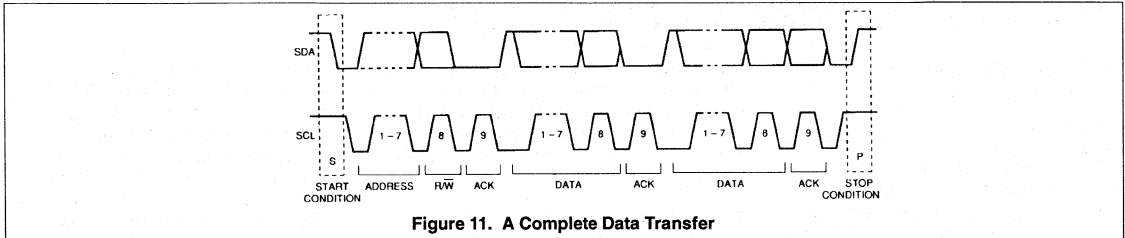


Figure 11. A Complete Data Transfer

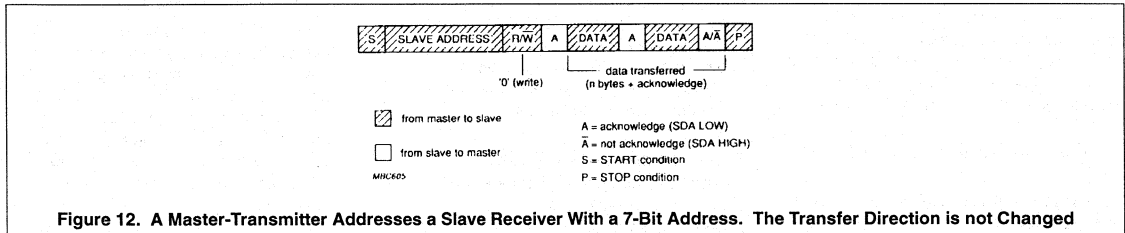


Figure 12. A Master-Transmitter Addresses a Slave Receiver With a 7-Bit Address. The Transfer Direction is not Changed

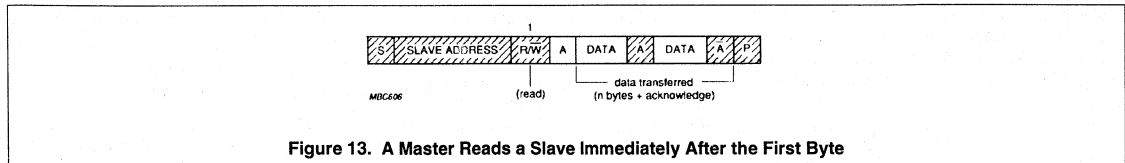


Figure 13. A Master Reads a Slave Immediately After the First Byte

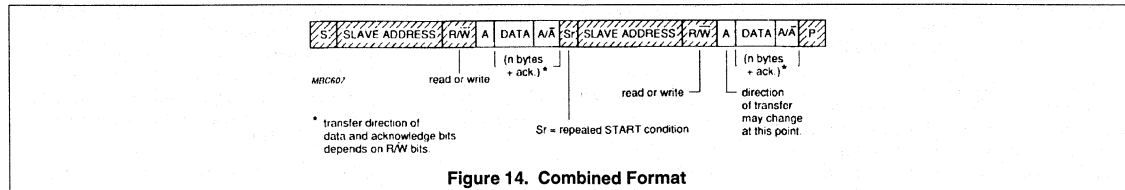


Figure 14. Combined Format

The I²C-bus and how to use it

Table 2. Definition of Bits in the First Byte

Slave address	R/ bit	Description
0000 000	0	General call address
0000 000	1	START byte
0000 001	X	CBUS address
0000 010	X	Address reserved for different bus format
0000 011	X	Reserved for future purposes
0000 1XX	X	Reserved for future purposes
1111 1XX	X	Reserved for future purposes
1111 0XX	X	10-bit slave addressing

NOTES:

1. No device is allowed to acknowledge at the reception of the START byte.
2. The CBUS address has been reserved to enable the inter-mixing of CBUS compatible and I²C-bus compatible devices in the same system. I²C-bus compatible devices are not allowed to respond on reception of this address.
3. The address reserved for a different bus format is included to enable I²C and other protocols to be mixed. Only I²C-bus compatible devices that can work with such formats and protocols are allowed to respond to this address.

9.0 7-BIT ADDRESSING (see Section 13.0 for 10-Bit Addressing)

The addressing procedure for the I²C-bus is such that the first byte after the START condition usually determines which slave will be selected by the master. The exception is the 'general call' address which can address all devices. When this address is used, all devices should, in theory, respond with an acknowledgement. However, devices can be made to ignore this address. The second byte of the general call address then defines the action to be taken. This procedure is explained in more detail in Section 9.1.1.

9.1 Definition of Bits in the First Byte

The first seven bits of the first byte make up the slave address (Fig. 15). The eighth bit is the LSB (least significant bit). It determines the direction of the message. A 'zero' in the least significant position of the first byte means that the master will write information to a selected slave. A 'one' in this position means that the master will read information from the slave.

When an address is sent, each device in a system compares the first seven bits after the START condition with its address. If they match, the device considers itself addressed by the master as a slave-receiver or slave-transmitter, depending on the R/W bit.

A slave address can be made-up of a fixed and a programmable part. Since it's likely that there will be several identical devices in a system, the programmable part of the slave address enables the maximum possible number of such devices to be connected to

the I²C-bus. The number of programmable address bits of a device depends on the number of pins available. For example, if a device has 4 fixed and 3 programmable address bits, a total of 8 identical devices can be connected to the same bus.

The I²C-bus committee coordinates allocation of I²C addresses. Further information can be obtained from the Philips representatives listed on the back cover. Two groups of eight addresses (0000XXX and 1111XXX) are reserved for the purposes shown in Table 2. The bit combination 11110XX of the slave address is reserved for 10-bit addressing (see Section 13).

9.1.1 General Call Address

The general call address is for addressing every device connected to the I²C-bus. However, if a device doesn't need any of the data supplied within the general call structure, it can ignore this address by not issuing an acknowledgement. If a device does require data from a general call address, it will acknowledge this address and behave as a slave-receiver. The second and following bytes will be acknowledged by every slave-receiver capable of handling this data. A slave which cannot process one of these bytes must ignore it by not acknowledging. The meaning of the general call address is always specified in the second byte (Fig. 16).

There are two cases to consider:

- When the least significant bit B is a 'zero'
- When the least significant bit B is a 'one'.

When bit B is a 'zero'; the second byte has the following definition:

- 00000110 (H'06'). Reset and write programmable part of slave address by hardware. On receiving this 2-byte sequence, all devices designed to respond to the general call address will reset and take in the programmable part of their address. Precautions have to be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus
- 00000100 (H'04'). Write programmable part of slave address by hardware. All devices which define the programmable part of their address by hardware (and which respond to the general call address) will latch this programmable part at the reception of this two byte sequence. The device will not reset.
- 00000000 (H'00'). This code is not allowed to be used as the second byte.

Sequences of programming procedure are published in the appropriate device data sheets.

The remaining codes have not been fixed and devices must ignore them.

When bit B is a 'one'; the 2-byte sequence is a 'hardware general call'. This means that the sequence is transmitted by a hardware master device, such as a keyboard scanner, which cannot be programmed to transmit a desired slave address. Since a hardware master doesn't know in advance to which device the message has to be transferred, it can only generate this hardware general call and its own address – identifying itself to the system (Fig. 17).

The seven bits remaining in the second byte contain the address of the hardware master.

The I²C-bus and how to use it

This address is recognised by an intelligent device (e.g. a microcontroller) connected to the bus which will then direct the information from the hardware master. If the hardware master can also act as a slave, the slave address is identical to the master address.

In some systems, an alternative could be that the hardware master transmitter is set in the slave-receiver mode after the system reset. In this way, a system configuring master can tell the hardware master-transmitter (which is

now in slave-receiver mode) to which address data must be sent (Fig.18). After this programming procedure, the hardware master remains in the master-transmitter mode.

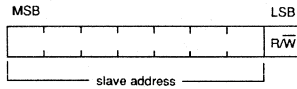


Figure 15. The First Byte After the START Procedure

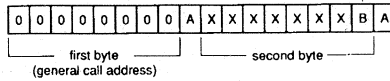


Figure 16. General Call Address Format

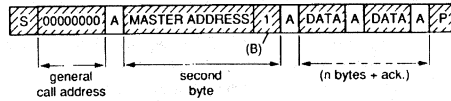
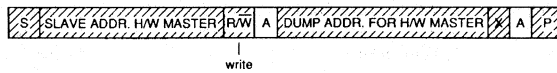
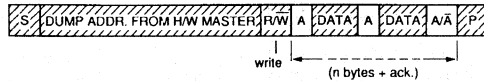


Figure 17. Data Transfer From a Hardware Master-Transmitter



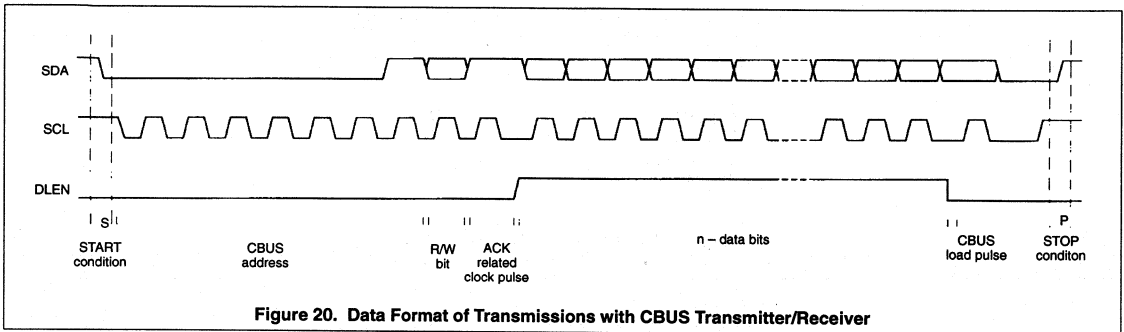
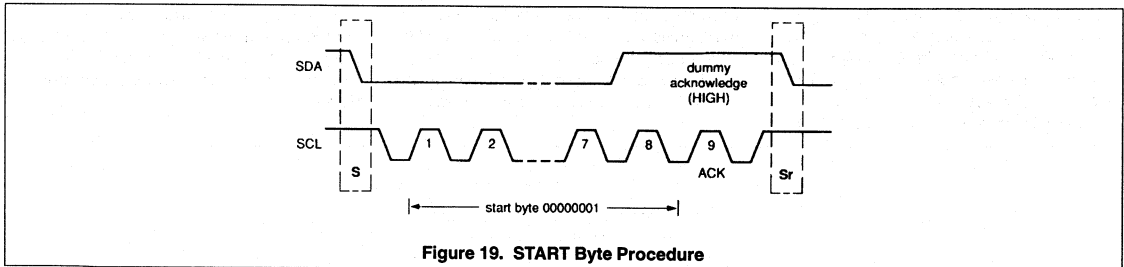
a. Configuring master sends dump address to hardware master



b. Hardware master dumps data to selected slave

Figure 18. Data Transfer by a Hardware-Transmitter Capable of Dumping Data Directly to Slave Devices

The I²C-bus and how to use it



9.1.2 START byte

Microcontrollers can be connected to the I²C-bus in two ways. A microcontroller with an on-chip hardware I²C-bus interface can be programmed to be only interrupted by requests from the bus. When the device doesn't have such an interface, it must constantly monitor the bus via software. Obviously, the more times the microcontroller monitors, or polls the bus, the less time it can spend carrying out its intended function.

There is therefore a speed difference between fast hardware devices and a relatively slow microcontroller which relies on software polling.

In this case, data transfer can be preceded by a start procedure which is much longer than normal (Fig.19). The start procedure consists of:

- A START condition (S)
- A START byte (00000001)
- An acknowledge clock pulse (ACK)
- A repeated START condition (Sr).

After the START condition S has been

transmitted by a master which requires bus access, the START byte (00000001) is transmitted. Another microcontroller can therefore sample the SDA line at a low sampling rate until one of the seven zeros in the START byte is detected. After detection of this LOW level on the SDA line, the microcontroller can switch to a higher sampling rate to find the repeated START condition Sr which is then used for synchronization.

A hardware receiver will reset on receipt of the repeated START condition Sr and will therefore ignore the START byte.

An acknowledge-related clock pulse is generated after the START byte. This is present only to conform with the byte handling format used on the bus. No device is allowed to acknowledge the START byte.

9.1.3 CBUS Compatibility

CBUS receivers can be connected to the I²C-bus. However, a third bus line called DLEN must then be connected and the

acknowledge bit omitted. Normally, I²C transmissions are sequences of 8-bit bytes; CBUS compatible devices have different formats.

In a mixed bus structure, I²C-bus devices must not respond to the CBUS message. For this reason, a special CBUS address (0000001X) to which no I²C-bus compatible device will respond, has been reserved. After transmission of the CBUS address, the DLEN line can be made active and a CBUS-format transmission (Fig.20) sent. After the STOP condition, all devices are again ready to accept data.

Master-transmitters can send CBUS formats after sending the CBUS address. The transmission is ended by a STOP condition, recognised by all devices.

NOTE: If the CBUS configuration is known, and expansion with CBUS compatible devices isn't foreseen, the designer is allowed to adapt the hold time to the specific requirements of the device(s) used.

The I²C-bus and how to use it

10.0 ELECTRICAL CHARACTERISTICS FOR I²C-BUS DEVICES

The electrical specifications for the I/Os of I²C-bus devices and the characteristics of the bus lines connected to them are given in Tables 3 and 4 in Section 15.

I²C-bus devices with fixed input levels of 1.5 V and 3 V can each have their own appropriate supply voltage. Pull-up resistors

must be connected to a 5 V ± 10% supply (Fig.21). I²C-bus devices with input levels related to V_{DD} must have one common supply line to which the pull-up resistor is also connected (Fig.22).

When devices with fixed input levels are mixed with devices with input levels related to V_{DD}, the latter devices must be connected to one common supply line of 5 V ± 10% and must have pull-up resistors connected to their SDA and SCL pins as shown in Fig.23.

Input levels are defined in such a way that:

- The noise margin on the LOW level is 0.1 V_{DD}
- The noise margin on the HIGH level is 0.2 V_{DD}
- As shown in Fig.24, series resistors (R_S) of e.g. 300 Ω can be used for protection against high-voltage spikes on the SDA and SCL lines (due to flash-over of a TV picture tube, for example).

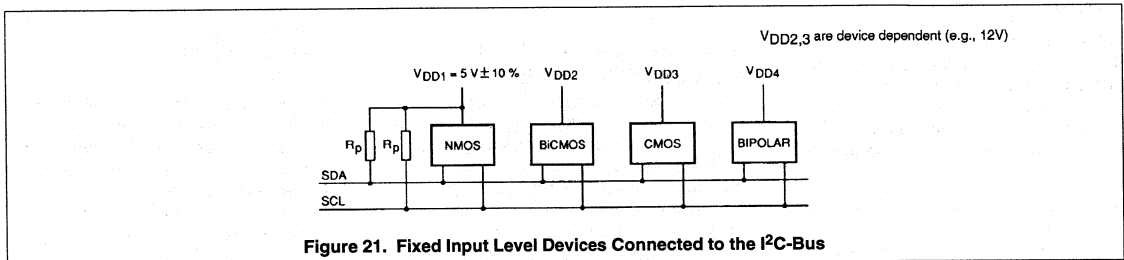


Figure 21. Fixed Input Level Devices Connected to the I²C-Bus

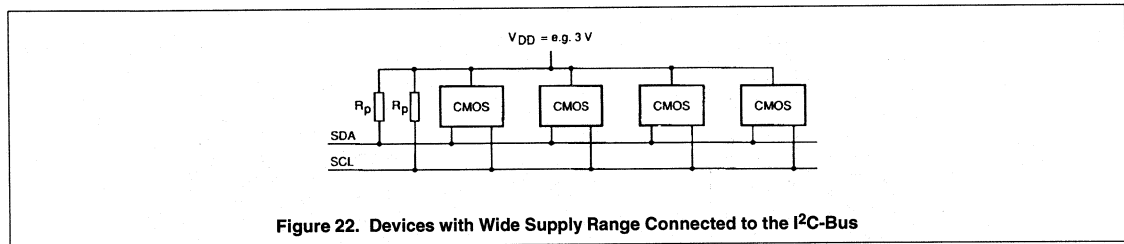


Figure 22. Devices with Wide Supply Range Connected to the I²C-Bus

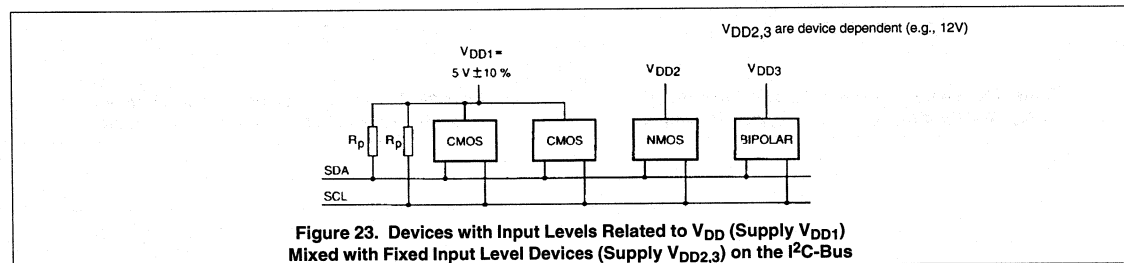


Figure 23. Devices with Input Levels Related to V_{DD} (Supply V_{DD1}) Mixed with Fixed Input Level Devices (Supply V_{DD2,3}) on the I²C-Bus

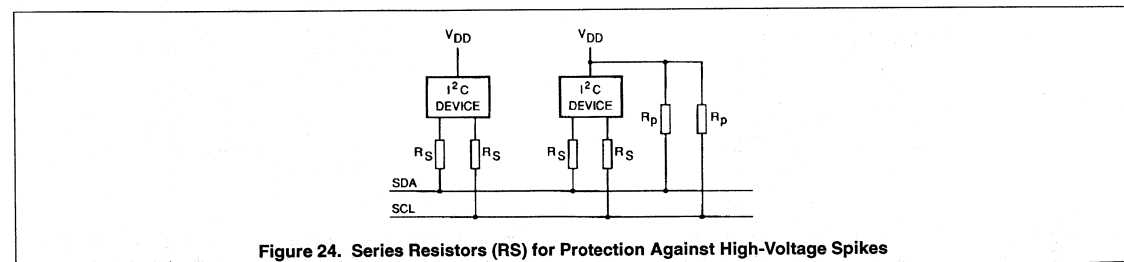


Figure 24. Series Resistors (R_S) for Protection Against High-Voltage Spikes

The I²C-bus and how to use it

10.1 Maximum and minimum values of resistors R_p and R_s

For standard-mode I²C-bus devices, the values of resistors R_p and R_s in Fig.24 depend on the following parameters:

- Supply voltage
- Bus capacitance
- Number of connected devices (input current + leakage current).

The supply voltage limits the minimum value of resistor R_p due

to the specified minimum sink current of 3 mA at V_{OLmax} = 0.4 V for the output stages. V_{DD} as a function of R_p min is shown in Fig.25. The desired noise margin of 0.1V_{DD} for the LOW level, limits the maximum value of R_s. R_s max as a function of R_p is shown in Fig.26.

The bus capacitance is the total capacitance of wire, connections and pins. This capacitance limits the maximum value of R_p due to the specified rise time. Fig.27 shows

R_p max as a function of bus capacitance.

The maximum HIGH level input current of each input/output connection has a specified maximum value of 10 μA. Due to the desired noise margin of 0.2V_{DD} for the HIGH level, this input current limits the maximum value of R_p. This limit depends on V_{DD}. The total HIGH level input current is shown as a function of R_p max in Fig.28.

11.0 EXTENSIONS TO THE I²C-BUS SPECIFICATION

The I²C-bus with a data transfer rate of up to 100 kbit/s and 7-bit addressing has now been in existence for more than ten years with an unchanged specification. The concept is accepted world-wide as a de facto standard and hundreds of different types of I²C-bus compatible ICs are available from Philips and other suppliers. The I²C-bus specification is now extended with the following two features:

- A **fast-mode** which allows a fourfold increase of the bit rate to 0 to 400 kbit/s
- **10-bit addressing** which allows the use of up to 1024 additional addresses.

There are two reasons for these extensions to the I²C-bus specification:

- New applications will need to transfer a larger amount of serial data and will therefore demand a higher bit rate than 100 kbit/s. Improved IC manufacturing technology now allows a fourfold speed increase without increasing the manufacturing cost of the interface circuitry
- Most of the 112 addresses available with the 7-bit addressing scheme have been issued more than once. To prevent problems with the allocation of slave addresses for new devices, it is desirable to have more address combinations. About a tenfold increase of the number of available addresses is obtained with the new 10-bit addressing.

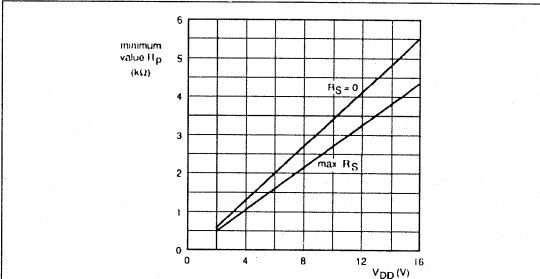


Figure 25. Minimum Value of R_p as a Function of Supply Voltage with the Value of R_s as a Parameter

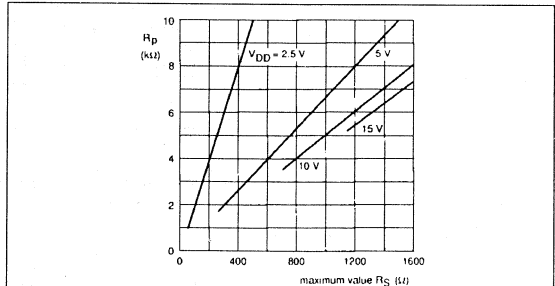


Figure 26. Maximum Value of R_s as a Function of the Value of R_p with Supply Voltage as a Parameter

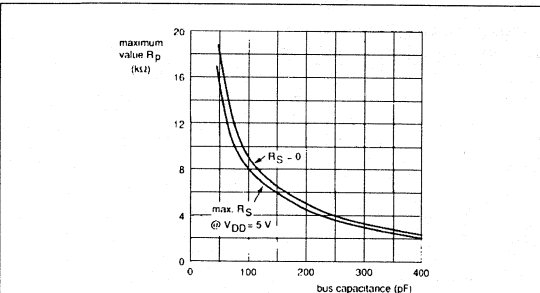


Figure 27. Maximum Value of R_p as a Function of Bus Capacitance for a Standard-Mode I²C-Bus

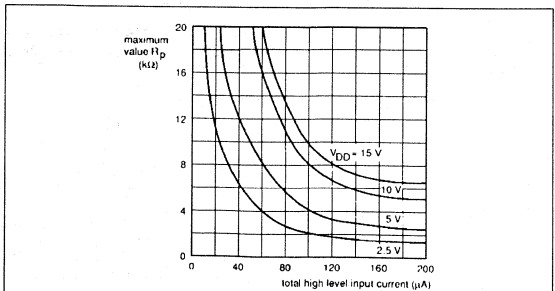


Figure 28. Total HIGH Level Input Current as a Function of the Maximum Value of R_p with Supply Voltage as a Parameter

The I²C-bus and how to use it

All new devices with an I²C-bus interface are provided with the fast-mode. Preferably, they should be able to receive and/or transmit at 400 kbit/s. The minimum requirement is that they can synchronize with a 400 kbit/s transfer; they can then prolong the LOW period of the SCL signal to slow down the transfer. Fast-mode devices must be downward-compatible which means that they must still be able to communicate with 0 to 100 kbit/s devices in a 0 to 100 kbit/s I²C-bus system.

Obviously, devices with a 0 to 100 kbit/s I²C-bus interface cannot be incorporated in a fast-mode I²C-bus system because, since they cannot follow the higher transfer rate, unpredictable states of these devices would occur.

Slave devices with a fast-mode I²C-bus interface can have a 7-bit or a 10-bit slave address. However, a 7-bit address is preferred because it is the cheapest solution in hardware and it results in the shortest message length. Devices with 7-bit and 10-bit addresses can be mixed in the same I²C-bus system regardless of whether it is a 0 to 100 kbit/s standard-mode system or a 0 to 400 kbit/s fast-mode system. Both existing and future masters can generate either 7-bit or 10-bit addresses.

12.0 FAST-MODE

In the fast-mode of the I²C-bus, the protocol, format, logic levels and maximum capacitive load for the SDA and SCL lines quoted in the previous I²C-bus specification are unchanged. Changes to the previous I²C-bus specification are:

- The maximum bit rate is increased to 400 kbit/s
- Timing of the serial data (SDA) and serial clock (SCL) signals has been adapted. There is no need for compatibility with other bus systems such as CBUS because they cannot operate at the increased bit rate
- The inputs of fast-mode devices must incorporate spike suppression and a Schmitt trigger at the SDA and SCL inputs
- The output buffers of fast-mode devices must incorporate slope control of the falling edges of the SDA and SCL signals
- If the power supply to a fast-mode device is switched off, the SDA and SCL I/O pins must be floating so that they don't obstruct the bus lines
- The external pull-up devices connected to the bus lines must be adapted to accommodate the shorter maximum

permissible rise time for the fast-mode I²C-bus. For bus loads up to 200 pF, the pull-up device for each bus line can be a resistor; for bus loads between 200 pF and 400 pF, the pull-up device can be a current source (3mA max.) or a switched resistor circuit as shown in Fig.37.

13.0 10-BIT ADDRESSING

The 10-bit addressing does not change the format in the I²C-bus specification. Using 10 bits for addressing exploits the reserved combination 1111XXX for the first seven bits of the first byte following a START (S) or repeated START (Sr) condition as explained in Section 9.1. The 10-bit addressing does not affect the existing 7-bit addressing. Devices with 7-bit and 10-bit addresses can be connected to the same I²C-bus, and both 7-bit and 10-bit addressing can be used in a standard-mode system (up to 100 kbit/s) or a fast-mode system (up to 400 kbit/s).

Although there are eight possible combinations of the reserved address bits 1111XXX, only the four combinations 11110XX are used for 10-bit addressing. The remaining four combinations 11111XX are reserved for future I²C-bus enhancements.

13.1 Definition of Bits in the First Two Bytes

The 10-bit slave address is formed from the first two bytes following a START condition (S) or a repeated START condition (Sr).

The first seven bits of the first byte are the combination 11110XX of which the last two bits (XX) are the two most-significant bits (MSBs) of the 10-bit address; the eighth bit of the first byte is the R/W bit that determines the direction of the message. A 'zero' in the least significant position of the first byte means that the master will write information to a selected slave. A 'one' in this position means that the master will read information from the slave.

If the R/W bit is 'zero', then the second byte contains the remaining 8 bits (XXXXXXXX) of the 10-bit address. If the R/W bit is 'one', then the next byte contains data transmitted from a slave to a master.

13.2 Formats with 10-bit Addresses

Various combinations of read/write formats are possible within a transfer that includes 10-bit addressing. Possible data transfer formats are:

- **Master-transmitter transmits to slave-receiver with a 10-bit slave address. The transfer direction is not changed (Fig.29).** When a 10-bit

address follows a START condition, each slave compares the first seven bits of the first byte of the slave address (11110XX) with its own address and tests if the eighth bit (R/W direction bit) is 0. It is possible that more than one device will find a match and generate an acknowledge (A1). All slaves that found a match will compare the eight bits of the second byte of the slave address (XXXXXXXX) with their own addresses, but only one slave will find a match and generate an acknowledge (A2). The matching slave will remain addressed by the master until it receives a STOP condition (P) or a repeated START condition (Sr) followed by a different slave address

NOTES:

1. Combined formats can be used, for example, to control a serial memory. During the first data byte, the internal memory location has to be written. After the START condition and slave address is repeated, data can be transferred.
2. All decisions on auto-increment or decrement of previously accessed memory locations etc. are taken by the designer of the device.
3. Each byte is followed by an acknowledgement bit as indicated by the A or \bar{A} blocks in the sequence.
4. I²C-bus compatible devices must reset their bus logic on receipt of a START or repeated START condition such that they all anticipate the sending of a slave address.
 - **Master-receiver reads slave-transmitter with a 10-bit slave address. The transfer direction is changed after the second R/W bit (Fig.30).** Up to and including acknowledge bit A2, the procedure is the same as that described for a master-transmitter addressing a slave-receiver. After the repeated START condition (Sr), a matching slave remembers that it was addressed before. This slave then checks if the first seven bits of the first byte of the slave address following Sr are the same as they were after the START condition (S), and tests if the eighth (R/W) bit is 1. If there is a match, the slave considers that it has been addressed as a transmitter and generates acknowledge A3. The slave-transmitter remains addressed until it receives a STOP condition (P) or until it receives another repeated START condition (Sr) followed by a different slave address. After a repeated START condition (Sr), all the other slave devices will also compare the first seven bits of the first byte of the slave address (11110XX) with their own

The I²C-bus and how to use it

addresses and test the eighth (R/W) bit. However, none of them will be addressed because R/W = 1 (for 10-bit devices), or the 11110XX slave address (for 7-bit devices) does not match)

- **Combined format. A master transmits data to a slave and then reads data from the same slave (Fig.31).** The same master occupies the bus all the

time. The transfer direction is changed after the second R/W bit

- **Combined format. A master transmits data to one slave and then transmits data to another slave (Fig.32).** The same master occupies the bus all the time
- **Combined format. 10-bit and 7-bit addressing combined in one serial**

transfer (Fig.33). After each START condition (S), or each repeated START condition (Sr), a 10-bit or 7-bit slave address can be transmitted. Figure 33 shows how a master-transmits data to a slave with a 7-bit address and then transmits data to a second slave with a 10-bit address. The same master occupies the bus all the time.

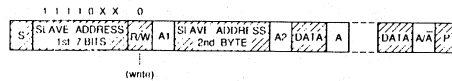


Figure 29. A Master-Transmitter Addresses a Slave-Receiver with a 10-Bit Address

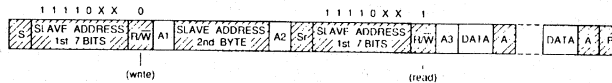


Figure 30. A Master-Receiver Addresses a Slave-Transmitter with a 10-Bit Address

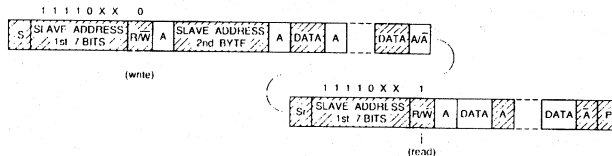


Figure 31. Combined Format. A Master Addresses a Slave with a 10-Bit Address, then Transmits Data to this Slave and Reads Data from this Slave

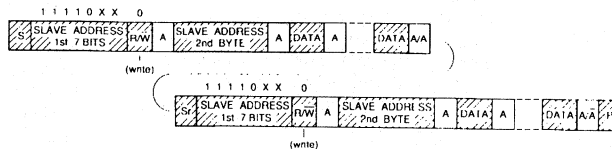


Figure 32. Combined Format. A Master Transmits Data to Two Slaves, Both With 10-Bit Addresses

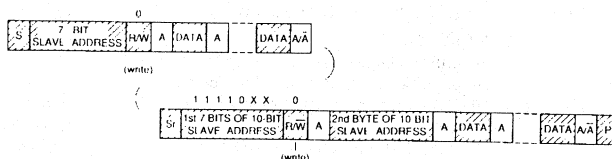


Figure 33. Combined Format. A Master Transmits Data to Two Slaves, One With a 7-Bit Address, and One with a 10-Bit Address.

The I²C-bus and how to use it

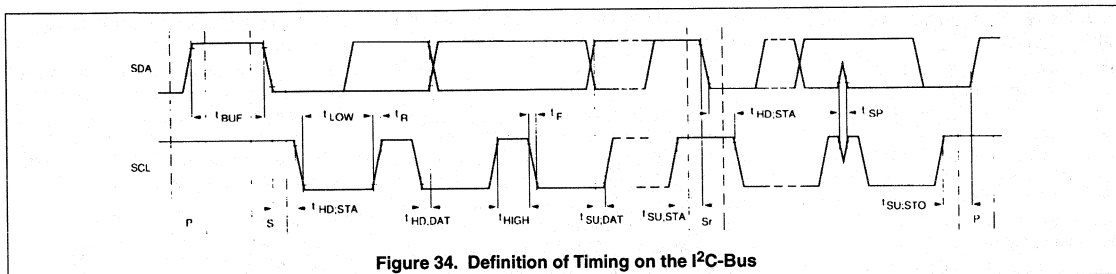


Figure 34. Definition of Timing on the I²C-Bus

Table 3. Characteristics of the SDA and SCL I/O Stages for I²C-Bus Devices

Parameter	Symbol	standard-mode devices		fast-mode devices		Unit
		Min.	Max.	Min.	Max.	
LOW level input voltage: fixed input levels V _{DD} -related input levels	V _{IL}	-0.5 -0.5	1.5 0.3V _{DD}	-0.5 -0.5	1.5 0.3V _{DD}	V
HIGH level input voltage: fixed input levels V _{DD} -related input levels	V _{IH}	3.0 0.7V _{DD}	*1) *1)	3.0 0.7V _{DD}	*1) *1)	V
Hysteresis of Schmitt trigger inputs: fixed input levels V _{DD} -related input levels	V _{hys}	n/a n/a	n/a n/a	0.2 0.05V _{DD}	- -	V
Pulse width of spikes which must be suppressed by the input filter	t _{SP}	n/a	n/a	0	50	ns
LOW level output voltage (open drain or open collector): at 3 mA sink current at 6 mA sink current	V _{OL1} V _{OL2}	0 n/a	0.4 n/a	0 0	0.4 0.6	V
Output fall time from V _{IH min.} to V _{IL max.} with a bus capacitance from 10 pF to 400 pF:	t _{OF}		250 ²⁾ n/a	20 + 0.1C _b ²⁾ 20 + 0.1C _b ²⁾	250 250 ³⁾	ns
with up to 3 mA sink current at V _{OL1}		-	250 ²⁾	20 + 0.1C _b ²⁾	250	
with up to 6 mA sink current at V _{OL2}		n/a	n/a	20 + 0.1C _b ²⁾	250 ³⁾	
Input current each I/O pin with an input voltage between 0.4 V and 0.9V _{DD max.}	I _i	-10	10	10 ³⁾	10 ³⁾	μA
Capacitance for each I/O pin	C _i	-	10	-	10	pF

NOTES:

n/a = not applicable

1. maximum V_{IH} = V_{DD max.} + 0.5 V

2. C_b = capacitance of one bus line in pF. Note that the maximum t_F for the SDA and SCL bus lines quoted in Table 4 (300 ns) is longer than the specified maximum t_{OF} for the output stages (250 ns). This allows series protection resistors (R₀) to be connected between the SDA/SCL pins and the SDA/SCL bus lines as shown in Fig.37 without exceeding the maximum specified t_F.

3. I/O pins of fast-mode devices must not obstruct the SDA and SCL lines if V_{DD} is switched off.

The I²C-bus and how to use it

14.0 GENERAL CALL ADDRESS AND START BYTE

The 10-bit addressing procedure for the I²C-bus is such that the first two bytes after the START condition (S) usually determine which slave will be selected by the master. The exception is the 'general call' address 00000000 (H'00'). Slave devices with 10-bit addressing will react to a 'general call' in the same way as slave devices with 7-bit addressing (see Section 9.1.1).

Hardware masters can transmit their 10-bit address after a 'general call'. In this case, the 'general call' address byte is followed by two successive bytes containing the 10-bit address of the master-transmitter. The format is as shown in Fig.17 where the first DATA byte contains the eight least-significant bits of the master address.

The START byte 00000001 (H'01') can precede the 10-bit addressing in the same way as for 7-bit addressing (see Section 9.1.2).

15.0 ELECTRICAL SPECIFICATIONS

The I/O levels, I/O current, spike suppression, output slope control and pin capacitance for I²C-bus devices are given in Table 3. The I²C-bus timing is given in Table 4. Figure 34 shows the timing definitions for the I²C-bus.

The noise margin for HIGH and LOW levels on the bus lines for fast-mode devices are the same as those specified in Section 10.0 for standard-mode I²C-bus devices.

The minimum HIGH and LOW periods of the SCL clock specified in Table 4 determine the maximum bit transfer rates of 100 kbit/s for standard-mode devices and 400 kbit/s for fast mode devices. Standard-mode and fast-mode I²C-bus devices must be able to follow transfers at their own maximum bit rates, either by being able to transmit or receive at that speed or by applying the clock synchronization procedure described in Section 7 which will force the master into a wait state and stretch the LOW period of the SCL signal. Of course, in the latter case the bit transfer rate is reduced.

16.0 APPLICATION INFORMATION

16.1 Slope-Controlled Output Stages of Fast-Mode I²C-Bus Devices

The electrical specifications for the I/Os of

I²C-bus devices and the characteristics of the bus lines connected to them are given in Tables 3 and 4 in Section 15.

Figures 35 and 36 show examples of output stages with slope control in CMOS and bipolar technology. The slope of the falling edge is defined by a Miller capacitor (C1) and a resistor (R1). The typical values for C1 and R1 are indicated on the diagrams. The wide tolerance for output fall time t_{OF} given in Table 3 means that the design is not critical. The fall time is only slightly influenced by the external bus load (C_b) and external pull-up resistor (R_p). However, the rise time (t_R) specified in Table 4 is mainly determined by the bus load capacitance and the value of the pull-up resistor.

16.2 Switched Pull-Up Circuit for Fast-Mode I²C-Bus Devices

The supply voltage (V_{DD}) and the maximum output LOW level determine the minimum value of pull-up resistor R_p (see Section 10.1). For example, with a supply voltage of $V_{DD} = 5V \pm 10\%$ and $V_{OLmax} = 0.4V$ at 3 mA, $R_{pmin} = (5 - 0.4)/0.003 = 1.7 k\Omega$. As shown in Fig.38, this value of R_p limits the maximum bus capacitance to about 200 pF to meet the maximum t_R requirement of 300 ns. If the bus has a higher capacitance than this, a switched pull-up circuit as shown in Fig.37 can be used.

The switched pull-up circuit in Fig.37 is for a supply voltage of $V_{DD} = 5V \pm 10\%$ and a maximum capacitive load of 400 pF. Since it is controlled by the bus levels, it needs no additional switching control signals. During the rising/falling edges, the bilateral switch in the HCT4066 switches pull-up resistor R_{p2} on/off at bus levels between 0.8 V and 2.0 V. Combined resistors R_{p1} and R_{p2} can pull-up the bus line within the maximum specified rise time (t_R) of 300 ns. The maximum sink current for the driving I²C-bus device will not exceed 6 mA at $V_{OL2} = 0.6V$, or 3 mA at $V_{OL1} = 0.4V$.

Series resistors R_s are optional. They protect the I/O stages of the I²C-bus devices from high-voltage spikes on the bus lines, and minimize crosstalk and undershoot of the bus line signals. The maximum value of R_s is determined by the maximum permitted voltage drop across this resistor when the bus line is switched to the LOW level in order to switch off R_{p2} .

16.3 Wiring Pattern of the Bus Lines

In general, the wiring must be so chosen that crosstalk and interference to/from the bus lines is minimized. The bus lines are most susceptible to crosstalk and interference at the HIGH level because of the relatively high impedance of the pull-up devices.

If the length of the bus lines on a PCB or ribbon cable exceeds 10 cm and includes the V_{DD} and V_{SS} lines, the wiring pattern must be:

SDA _____

V_{DD} _____

V_{SS} _____

SCL _____

If only the V_{SS} line is included, the wiring pattern must be:

SDA _____

V_{SS} _____

SCL _____

These wiring patterns also result in identical capacitive loads for the SDA and SCL lines. The V_{SS} and V_{DD} lines can be omitted if a PCB with a V_{SS} and/or V_{DD} layer is used.

If the bus lines are twisted-pairs, each bus line must be twisted with a V_{SS} return. Alternatively, the SCL line can be twisted with a V_{SS} return, and the SDA line twisted with a V_{DD} return. In the latter case, capacitors must be used to decouple the V_{DD} line to the V_{SS} line at both ends of the twisted pairs.

If the bus lines are shielded (shield connected to V_{SS}), interference will be minimized. However, the shielded cable must have low capacitive coupling between the SDA and SCL lines to minimize crosstalk.

16.4 Maximum and Minimum Values of Resistors R_p and R_s for Fast-Mode I²C-Bus Devices

The maximum and minimum values for resistors R_p and R_s connected to a fast-mode I²C-bus can be determined from Fig.25, 26 and 28 in Section 10.1. Because a fast-mode I²C-bus has faster rise times (t_R) the maximum value of R_p as a function of bus capacitance is less than that shown in Fig.27. The replacement graph for Fig.27 showing the maximum value of R_p as a function of bus capacitance (C_b) for a fast-mode I²C-bus is given in Fig.38.

The I²C-bus and how to use it

Table 4. Characteristics of the SDA and SCL Bus Lines for I²C-Bus Devices

Parameter	Symbol	Standard-mode I ² C-bus		Fast-mode I ² C-bus		Unit
		Min.	Max.	Min.	Max.	
SCL clock frequency	f _{SCL}	0	100	0	400	kHz
Bus free time between a STOP and START condition	t _{BUF}	4.7	-	1.3	-	μs
Hold time (repeated) START condition. After this period, the first clock pulse is generated	t _{HD;STA}	4.0	-	0.6	-	μs
LOW period of the SCL clock	t _{LOW}	4.7	-	1.3	-	μs
HIGH period of the SCL clock	t _{HIGH}	4.0	-	0.6	-	μs
Set-up time for a repeated START condition	t _{SU;STA}	4.7	-	0.6	-	μs
Data hold time: for CBUS compatible masters (see NOTE, Section 9.1.3) for I ² C-bus devices	t _{HD;DAT}	5.0 0 ¹⁾	- -	- 0 ¹⁾	- 0.9 ²⁾	μs μs
Data set-up time	t _{SU;DAT}	250	-	100 ³⁾	-	ns
Rise time of both SDA and SCL signals	t _R	-	1000	20 + 0.1C _b ⁴⁾	300	ns
Fall time of both SDA and SCL signals	t _F	-	300	20 + 0.1C _b ⁴⁾	300	ns
Set-up time for STOP condition	t _{SU;STO}	4.0	-	0.6	-	μs
Capacitive load for each bus line	C _b	-	400	-	400	pF

NOTES:

All values referred to V_{IH min.} and V_{IL max.} levels (see Table 3).

1. A device must internally provide a hold time of at least 300 ns for the SDA signal (referred to the V_{IH min.} of the SCL signal) in order to bridge the undefined region of the falling edge of SCL.
2. The maximum t_{HD;DAT} has only to be met if the device does not stretch the LOW period (t_{LOW}) of the SCL signal.
3. A fast-mode I²C-bus device can be used in a standard-mode I²C-bus system, but the requirement t_{SU;DAT} ≥ 250 ns must then be met. This will automatically be the case if the device does not stretch the LOW period of the SCL signal. If such a device does stretch the LOW period of the SCL signal, it must output the next data bit to the SDA line t_{R max.} + t_{SU;DAT} = 1000 + 250 = 1250 ns (according to the standard-mode I²C-bus specification) before the SCL line is released.
4. C_b = total capacitance of one bus line in pF.

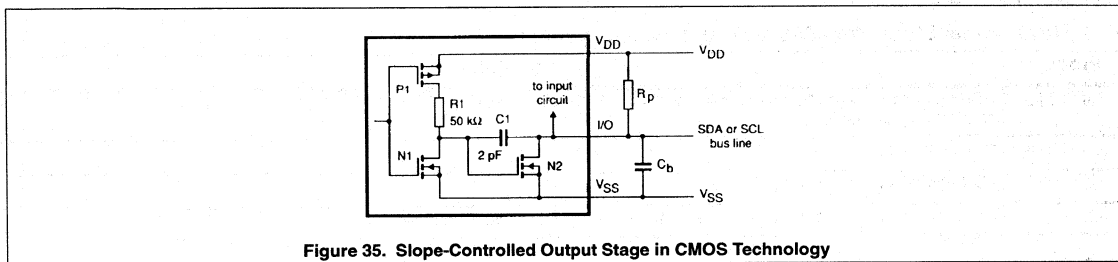


Figure 35. Slope-Controlled Output Stage in CMOS Technology

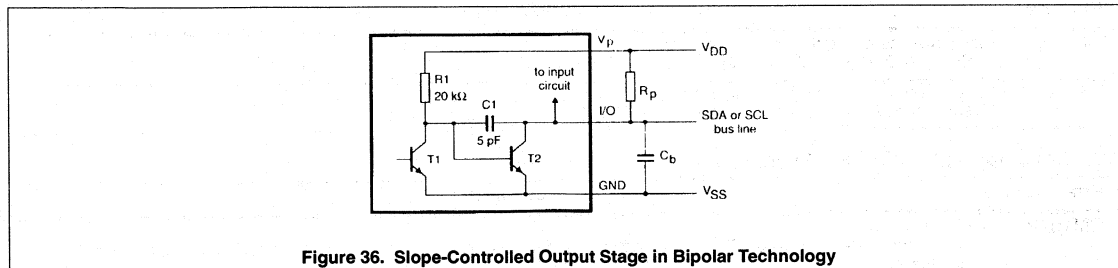
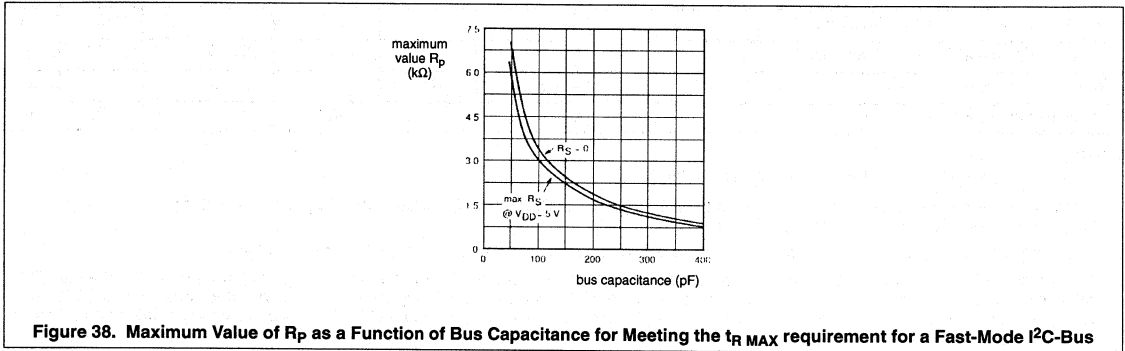
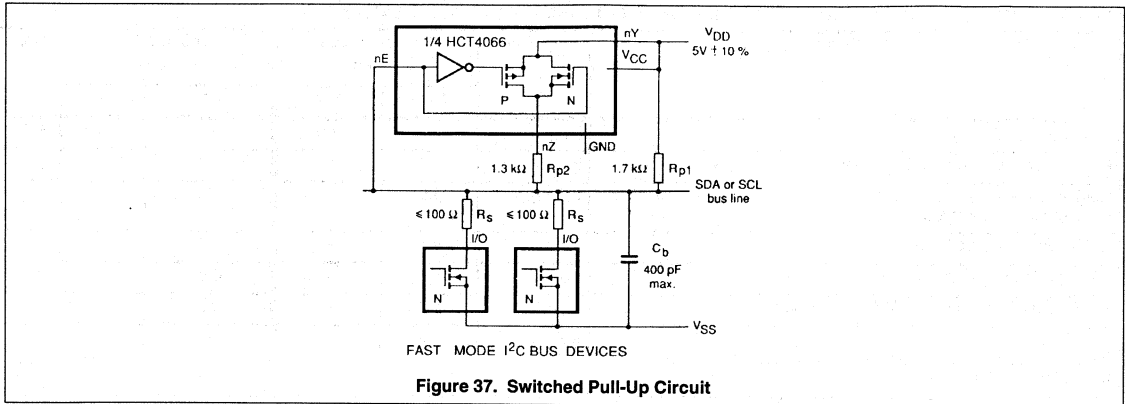


Figure 36. Slope-Controlled Output Stage in Bipolar Technology

The I²C-bus and how to use it



17.0 DEVELOPMENT TOOLS

17.1 Development tools for 8048 and 8051-based systems

Product	Description
OM1016	I ² C-bus demonstration board with microcontroller, LCD, LED, Par. I/O, SRAM, EEPROM, Clock, DTMF generator, AD/DA conversion, infrared link.
OM1018	manual for OM1016
OM1020	LCD and driver demonstration board
OM4151	I ² C-bus evaluation board (similar to OM1016 above but without infrared link).

17.2 Development tools for 68000-based systems

Product	Description
OM4160	Microcore-1 demonstration/evaluation board: SCC68070, 128K EPROM, 512K DRAM, I ² C, RS-232C, VSC SCC66470, resident monitor
OM4160/3	Microcore-3 demonstration/evaluation board: 93C110, 128K EPROM, 64K SRAM, I ² C, RS-232C, 40 I/O, resident monitor

17.3 Development tools for all systems

Product	Description
OM1022	I ² C-bus analyzer. Hardware and software (runs on IBM or compatible PC) to experiment with and analyze the behaviour of the I ² C-bus (includes documentation)

The I²C-bus and how to use it

18.0 SUPPORT LITERATURE

Data handbooks
IC01 1992: Semiconductors for radio and audio systems
IC02 1992: Semiconductors for television and video systems
IC03 1993: Semiconductors for telecom systems
IC14 1992: 8048-based 8-bit microcontrollers
IC20 1994: 8051-based 8-bit microcontrollers
Brochures/leaflets
Microcontrollers and microprocessors for embedded control applications
I ² C-bus compatible ICs and support overview
I ² C-bus control programs for consumer applications

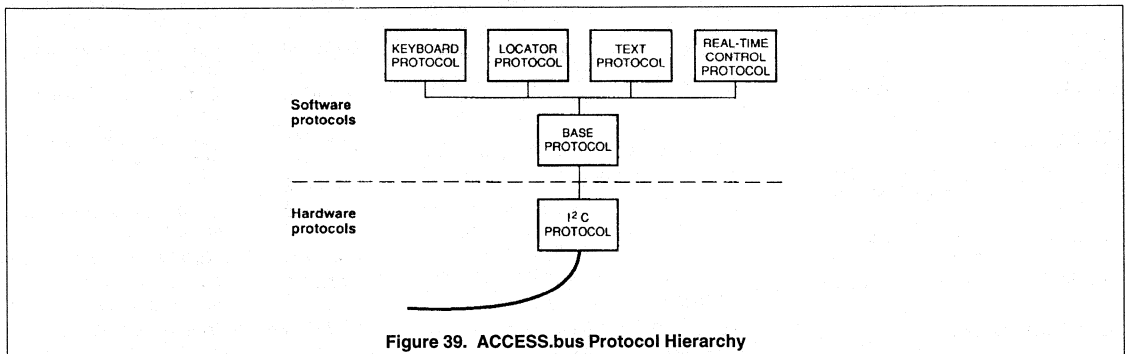


Figure 39. ACCESS.bus Protocol Hierarchy

19.0 APPLICATION OF THE I²C-BUS IN THE ACCESS.bus SYSTEM

The ACCESS.bus (bus for connecting ACCESSory devices to a host system) is an I²C-bus based open-standard serial interconnect system jointly developed and defined by Philips Semiconductors and Digital Equipment Corporation. It is a lower-cost alternative to an RS-232C interface for connecting up to 14 inputs/outputs from peripheral equipment to a desk-top computer or workstation over a distance of up to eight metres. The peripheral equipment can be relatively low speed items such as keyboards, hand-held image scanners, cursor positioners, bar-code readers, digitizing tablets, card readers or modems.

All that's required to implement an ACCESS.bus is an 8051-family

microcontroller with an I²C-bus interface, and a 4-wire cable carrying a serial data (SDA) line, a serial clock (SCL) line, a ground wire and a 12 V supply line (500 mA max.) for powering the peripherals.

Important features of the ACCESS.bus are that the bit rate is only about 20% less than the maximum bit rate of the I²C-bus, and the peripherals don't need separate device drivers. Also, the protocol allows the peripherals to be changed by 'hot-plugging' without re-booting.

As shown in Fig.39, the ACCESS.bus protocol comprises three levels: the I²C-bus protocol, the base protocol, and the application protocol.

The base protocol is common to all ACCESS.bus devices and defines the format of the ACCESS.bus message. Unlike the I²C-bus protocol, it restricts masters to sending and slaves to receiving data. One

item of appended information is a checksum for reliability control. The base protocol also specifies seven types of control and status messages which are used in the system configuration which assigns unique addresses to the peripherals without the need for setting jumpers or switches on the devices.

The application protocol defines the message semantics that are specific to the three categories of peripheral device (keyboards, cursor locators, and text devices which generate character streams e.g. card readers) which are at present envisaged.

Philips Semiconductors offers computer peripheral equipment manufacturers technical support, a wide range of I²C-bus devices and development kits for the ACCESS.bus. Hardware, software and marketing support is also offered by DEC.

I²C peripheral selection guide

GENERAL PURPOSE ICs

LCD Drivers

PCF8566	96-segment LCD driver 1:1 – 1:4 Mux rates
PCF8567	LCD direct mode driver
PCF8568	LCD row driver for dot matrix displays
PCF8569	LCD Column driver for dot matrix displays
PCF8576	160-segment LCD driver 1:1 – 1:4 Mux rates
PCF8577C	64-segment LCD driver 1:1 – 1:2 Mux Rates
PCF8578/79	Row/column LCD dot-matrix driver/display 1:8 – 1:32 Mux rates

LED Drivers

SAA1064	4-digit LED driver
---------	--------------------

I/O Expanders

PCF8574/A	8-bit remote I/O port (I ² C-bus to parallel converter)
PCF8584	8-bit parallel to I ² C converter
SAA1300	5-bit high-current driver

Data Converters

PCF8591	4-channel, 8-bit Mux ADC + one DAC
TDA8442	Quad 6-bit DAC
TDA8444	Octal 6-bit DAC

Memory

PCA8581	128-byte EEPROM
PCF8570/C	256-byte static RAM
PCF8571	128-byte static RAM
PCF8582	256-byte EEPROM
PCF8583	256-byte RAM/clock/calendar
PCF8594	512-byte EEPROM
PCF8598	1K-byte EEPROM

Clocks/Calendars

PCF8573	Clock/calendar
PCF8583	Clock/calendar/ 256-byte RAM

68000-Based CMOS Microcontrollers

68070	68000 CPU/MMU/UART/ DMA/timer
93CXXX	UST/I ² C/34k ROM/ 512 RAM

80C51-Based CMOS Microcontrollers*

83CL267/167	12k ROM, 256 RAM OSD
83CL268/168	12k ROM, 256 RAM OSD
8XCL410	4k ROM/128 RAM, low power
8XC528	32k ROM/512 RAM, T2, WD
8XC542	4k ROM/128 RAM, ACCESS.bus
8XC552	256-byte RAM/8k ROM/ ADC/UART/PWM
8XCL580	6k ROM, 256 RAM, low power
8XC652	256-byte RAM/8k ROM, UART
8XC654	256-byte RAM/16kROM, UART
8XC751	64-byte RAM/2k ROM
8XC752	64-byte RAM/2k ROM, ADC/PWM

8048 Instruction-Set Based CMOS Microcontrollers

PCF84C00	256-byte RAM/bond-out version for prototype development
PCF84C21	64-byte RAM/2k ROM
PCF84C41	128-byte RAM/2k ROM
PCF84C81	256-byte RAM/8k ROM
PCF84C85	256-byte RAM/8k ROM/ Extended I/O
PCF84C430	128-byte RAM/4k ROM/ 96-segment LCD driver

MULTIMEDIA ICs

Desktop Videos

SAA7151B	8-bit digital multistandard TV decoder
SAA7152	Digital comb filter
SAA7157	Clock signal generation circuit for digital video systems; for use with SAA71xx
SAA7165	Video enhancement and D/A processor including digital CT1
SAA7186	Digital video scaler
SAA7191	Digital multistandard TV decoder, square pixel
SAA7191B	SAA7191 variant
SAA7192A	Digital colour space converter with independent LHT
SAA7199B	digital multistandard encoder

SAA9051	Digital multistandard (PAL/NTSC) colour decoder
SAA9056	Digital SECAM colour decoder
SAA9057B	Clock signal generation circuit for digital video systems; for use with SAA90xx
SAA9065	Video enhancement and D/A processor
TDA4680	Video processor
TDA8440	Video/audio switch
Video/Radio/Audio	
SAA4700	VPS dataline processor
SA5751	Audio Processor/Filter Controller
SAA5243	Computer controlled text circuit
SAA5246	Computer controlled text circuit
SAA5248	Integrated teletext decoder and VPS slicer
SAA5252	Closed caption
SAA7158	Line frequency processor and DAC circuit
SAA7194	Digital video decoder/scaler
SAA9042	Digital video teletext (DVTB) processor
SAB3035/36/37	Digital tuning circuits for computer-controlled TV
TDA1551	2 X 22W BTL audio power amp
TDA1551Q	2 X 22W BTL audio power amp with diagnostic
TDA4670	Picture signal improvement circuit
TDA4671	Picture signal improvement circuit
TDA4681	Video processor with automatic cut-off and white level control
TDA4685	Video processor
TDA4686	Video processor (100 Hz)
TDA4687	Video processor
TDA8415	TV/VCR stereo/dual sound processor
TDA8416	TV/VCR stereo/dual sound processor
TDA8417	TV/VCR stereo/dual sound processor
TDA8421	Audio processor with a loudspeaker channel and a headphone channel
TDA8425	Audio processor with a loudspeaker channel only
TDA8426	Hi-fi stereo audio processor
TDA8433	TV deflection processor
TDA8540	4x4 video switch matrix

I²C peripheral selection guide

Video/Radio/Audio (Continued)		Telecom	
TDA9140	Alignment-free multistandard decoder	NE5750/51	Audio processor pair
TEA6320	4 input Tone/volume controller with fader control	NE5752	3 V 5750 variant (samples Q4 92)
TEA6330	Tone/volume controller	NE5753	3 V 5751 variant (samples Q4 92)
TSA6060	A/M Frequency Synthesizer for RDS.	PCD3311/12	Tone generator (DTMF/modem/musical)
TDA8433	Deflection processor	PCD3341	Advanced 10 to 110-number repertory dialer with LCD control
TDA8442	Interface for color decoders	PCD3343	Microcontroller with 224-byte RAM/3k ROM
TDA8443/A	YUV/RGB matrix switch	PCD3348	Microcontroller with 256-byte RAM/8k ROM
TDA8461	PAL/NTSC color decoder and RGB processor	PCD4440	Analog voice scrambler/descrambler
TDA8466	PAL/NTSC color decoder and RGB processor	UMA1000T	Data processor for mobile telephones
TDA9150	Deflection processor	UMA1014T	1GHz frequency synthesizer for mobile telephones
TDA9860	Sound controller w/ 4 inputs	UMF1009	Frequency synthesizer
TEA6100	FM/IF and digital tuning IC for computer-controlled radio		
TEA6300	Sound fader control and preamplifier/source selector for car radio		
TSA5511/12/14	PLL frequency synthesizer for TV		
TSA6057	PLL frequency synthesizer for radio		

* Also available with extended temperature ranges.

FOR FURTHER INFORMATION ON THESE DEVICES, REFER TO *I²C-PERIPHERALS FOR MICROCONTROLLERS DATA HANDBOOK*, AVAILABLE FROM YOUR LOCAL PHILIPS SEMICONDUCTORS SALES OFFICE (SEE INSIDE BACK COVER OF THIS BOOK).

I²C bus extender

82B715

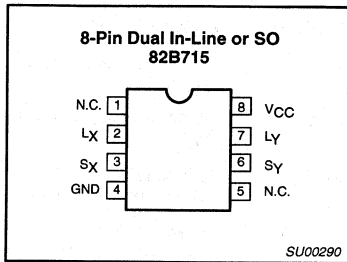
DESCRIPTION

The 82B715 is a bipolar integrated circuit designed for application in I²C bus systems.

While retaining all the operating modes and features of the I²C system it permits extension of the practical separation distance between components on the I²C bus by buffering both the data (SDA) and the clock (SCL) lines.

The I²C bus capacitance limit of 400pF restricts practical communication distances to a few meters. Using one 82B715 at each end of longer cables reduces the cable loading capacitance on the I²C bus by a factor of 10 times and may allow the use of low cost general purpose wiring to extend bus lengths.

PIN CONFIGURATIONS



PINNING

PIN	SYMBOL	FUNCTION
1	N.C.	
2	L _x	Buffered Bus, LDA or LCL
3	S _x	I ² C Bus, SDA or SCL
4	GND	Negative Supply
5	N.C.	
6	S _y	I ² C Bus, SCL or SDA
7	L _y	Buffered Bus, LCL or LDA
8	V _{CC}	Positive Supply

FEATURES

- Dual, bi-directional, unity voltage gain buffer
- I²C bus compatible
- Logic signal levels may include both supply and ground
- X10 impedance transformation
- Wide supply voltage range

QUICK REFERENCE DATA

SYMBOL	PARAMETER	LIMITS			UNIT
		MIN.	TYP.	MAX.	
V _{CC}	Supply voltage	4.5		12	V
I _{CC}	Quiescent current		16		mA
I _{line}	Output sink capability	30			mA
V _{in}	Input voltage range	0		V _{CC}	V
V _{out}	Output voltage range	0		V _{CC}	V
Z _{in} /Z _{out}	Impedance transformation	8	10	13	
T _{amb}	Temperature range	-40		+85	°C

ORDERING INFORMATION

DESCRIPTION	ORDER CODE	DRAWING NUMBER
8-Pin Plastic Dual In-Line (N/P) Package	P82B715P N	SOT97
8-Pin Plastic SOL (Small Outline Large) Dual In-Line (D/T) Package	P82B715T D	SOT96A
82B715 is available in chip form		

I²C bus extender

82B715

FUNCTIONAL DESCRIPTION

The 82B715 bipolar integrated circuit contains two identical buffer circuits which enable I²C and similar bus systems to be extended over long distances without degradation of system performance or requiring the use of special cables.

The buffer has an effective current gain of ten from I²C bus to Buffered bus. Whatever current is flowing out of the I²C bus side, ten times that current will be flowing into the Buffered bus side (see Figure 2).

As a consequence of this amplification the system is able to drive capacitive loads up to ten times the standard limit on the Buffered bus side. This current based buffering

approach preserves the bi-directional, open-collector/open-drain characteristic of the I²C SDA/SCL lines.

To minimize interference and ensure stability, current rise and fall rates are internally controlled.

APPLICATION NOTES

By using two (or more) 82B715 ICs, a sub-system can be built which retains the interface characteristics of an I²C device so that it may be included in, or optionally added to, any I²C or related system.

The sub-system features a low impedance or "Buffered" bus, capable of driving large wiring capacities (see Figure 3).

I²C Systems

As with the standard I²C system, pull-up resistors are required to provide the logic HIGH levels on the Buffered bus. (Standard open-collector configuration of the I²C bus). The size and number of these pull-up resistors depends on the system.

If the buffer is to be permanently connected into the system, the circuit should be configured with only one pull-up resistor on the Buffered bus and none on the I²C bus.

Alternatively a buffer may be connected to an existing I²C system. In this case the Buffered bus pull-up will act in parallel with the I²C bus pull-up.

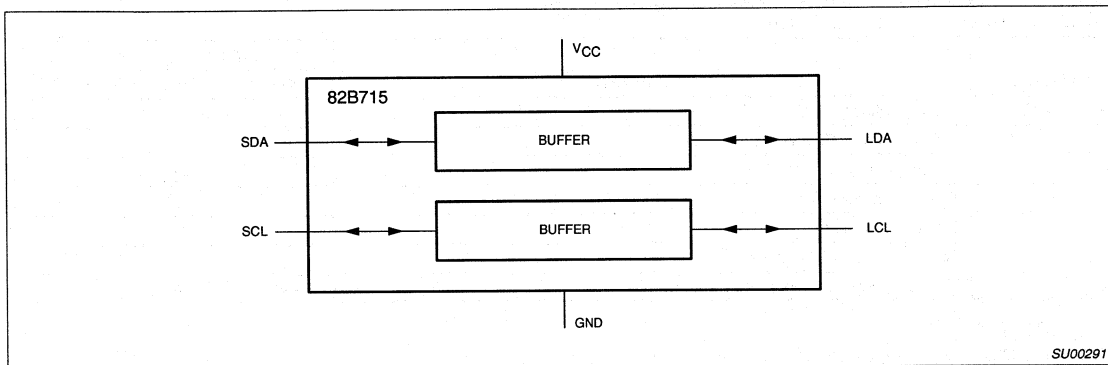


Figure 1. Block Diagram: 82B715

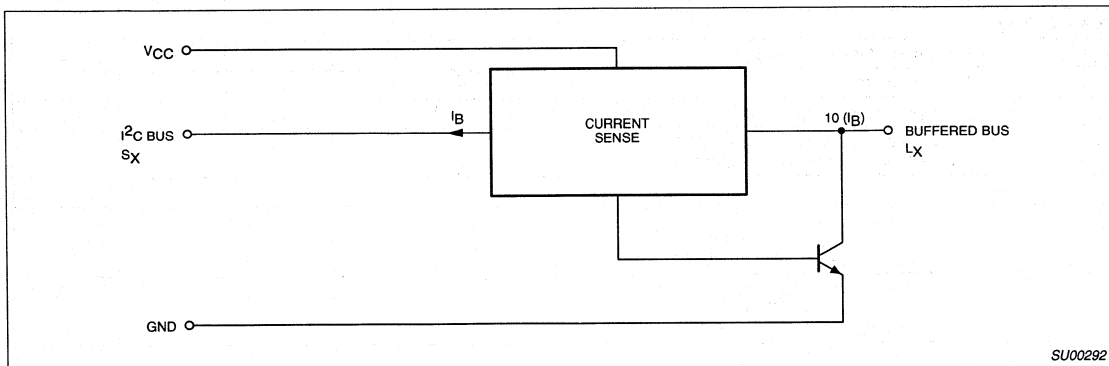
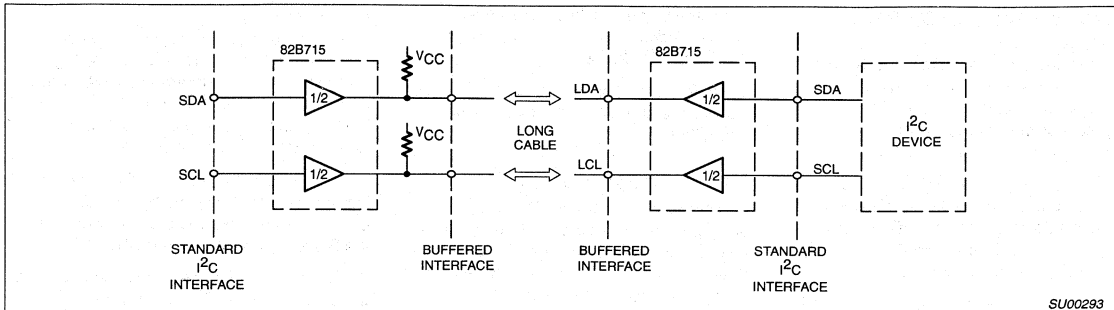


Figure 2. Equivalent Circuit: One Half 82B715

I²C bus extender

82B715



SU00293

Figure 3. Minimum Sub-System with 82B715

RATINGS

Limiting values in accordance with the Absolute Maximum System (IEC 134).
 Voltages with respect to pin GND (DIL-8 pin 4).

SYMBOL	PARAMETER	LIMITS		UNIT
		MIN.	MAX.	
V_{CC} to GND	Supply voltage range V_{CC}	-0.3	+12	V
V_{bus}	Voltage range I ² C Bus, SCL or SDA	0	V_{CC}	V
V_{buff}	Voltage range Buffered Bus	0	V_{CC}	V
I	DC current (any pin)		60	mA
P_{tot}	Power dissipation		300	mW
T_{stg}	Storage temperature range	-55	+125	°C
T_{amb}	Operating ambient temperature range	-40	+85	°C

CHARACTERISTICS

At $T_{amb} = +25^{\circ}\text{C}$ and $V_{CC} = 5$ Volts, unless otherwise specified.

SYMBOL	PARAMETER	LIMITS			UNIT
		MIN.	TYP.	MAX.	
Power Supply					
V_{CC}	Supply voltage (operating)	4.5	—	12	V
I_{CC}	Supply current	—	16	—	mA
I_{CC}	Supply current at $V_{CC} = 12\text{V}$	—	22	—	mA
I_{CC}	Supply current, both I ² C inputs LOW, both buffered outputs sinking 30mA.	—	28	—	mA
Drive Currents					
I_{Sx}, I_{Sy}	Output sink on I ² C bus V_{Sx}, V_{Sy} LOW = 0.4V V_{Lx}, V_{Ly} LOW on Buffered bus = 0.3V	3	—	—	mA
I_{Lx}, I_{Ly}	Output sink on Buffered bus V_{Lx}, V_{Ly} LOW = 0.4V V_{Sx}, V_{Sy} LOW on I ² C bus = 0.3V	30	—	—	mA
Input Currents					
I_{Sx}, I_{Sy}	Input current from I ² C bus when I_{Lx}, I_{Ly} sink on Buffered bus = 30mA	—	—	3	mA
I_{Lx}, I_{Ly}	Input current from Buffered bus when I_{Sx}, I_{Sy} sink on I ² C bus = 3mA	—	—	3	mA
I_{Lx}, I_{Ly}	Leakage current on Buffered bus $V_{Lx}, V_{Ly} = V_{CC}$, and $V_{Sx}, V_{Sy} = V_{CC}$	—	—	200	μA
Impedance Transformation					
Z_{in}/Z_{out}	Input/Output impedance	8	10	13	

I²C bus extender

82B715

Pull-Up Resistance Calculation

In calculating the pull-up resistance values, the gain of the buffer introduces scaling factors which must be applied to the system components. Viewing the system from the Buffered bus, all I²C bus capacitances have effectively 10 times their I²C bus value.

In practical systems the pull-up resistance is determined by the rise time limit for I²C systems. As an approximation this limit will be satisfied if the time constant (product of the net resistance and net capacitance) of the total system is set to 1 microsecond.

The total time constant may either be set by considering each bus node individually (i.e., the I²C nodes, and the Buffered bus node) and choosing pull-up resistors to give time constants of 1 microsecond for each node; or by combining the capacitances into an equivalent capacitive loading on the Buffered bus, and calculating the Buffered bus pull-up resistor required by this equivalent capacitance.

For each separate bus the pull-up resistor may be calculated as follows:

$$R = \frac{1 \mu \text{sec}}{C_{\text{device}} + C_{\text{wiring}}}$$

Where: C_{device} = sum of device capacitances connected to each bus,

and C_{wiring} = total wiring and stray capacitance on each bus.

If these capacitances are not known then a good approximation is to assume that each device presents 10pF of load capacitance and 10pF of wiring capacitance.

The capacitance figures for one or more individual I²C bus nodes should be multiplied by a factor of 10 times, and then added to the Buffered bus capacitance. Calculation of a new Buffered bus pull-up resistor will allow this single pull-up resistor to act for both the included I²C bus nodes and the Buffered bus. Thus it is possible to combine some or all of these separate pull-up resistors into a single resistor on the Buffered bus (the value of which is calculated from the sum of the scaled capacitances on the Buffered bus). If the buffer is to be permanently connected into the system then all the separate pull-up resistors should be combined. But if it is to be connected by adding it onto an existing system, then only those on the additional I²C bus system can be combined onto the Buffered bus if the original system is required to be able to still operate on a stand-alone basis.

A further restriction is that the maximum pull-up current, with the bus LOW, should not exceed the I²C bus specification maximum of

3mA, or 30mA on the Buffered bus. The following formula applies:

$$30\text{mA} > \frac{V_{\text{CC}} - 0.4}{R_{\text{P}}}$$

Where: R_P = scaled parallel combination of all pull-up resistors.

If this condition is met, the fall time specifications will also be met.

Figure 4 shows typical loading calculations for the expanded I²C bus.

Sx, Sy, I²C Bus, SDA or SCL

Because the two buffer circuits in the 82B715 are identical either input pin can be used as the I²C Bus SDA data line, or the SCL clock line.

Lx, Ly, Buffered Bus, LDA or LCL

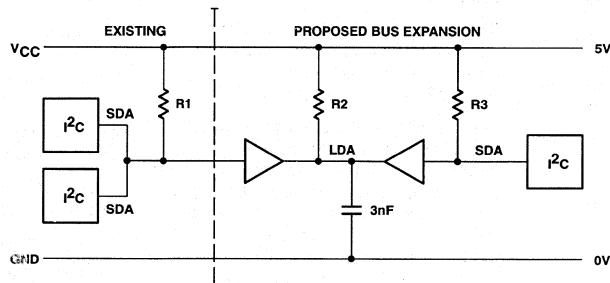
On the buffered low impedance line side, the corresponding output becomes LDA and LCL.

V_{CC}, GND — Positive and Negative Supply Pins

In normal use the power supply voltages at each end of the low impedance line should be comparable. If these differ by a significant amount, noise margin is sacrificed.

I²C bus extender

82B715



EFFECTIVE CAPACITANCE NEAR I²C DEVICES

2 × I ² C Devices	20pF
Strays	20pF
82B715 Buffer	10pF
TOTAL CAP.	50pF

I²C pull-up

$$R1 = \frac{1\mu\text{sec}}{50\text{pF}} = 20\text{K}\Omega$$

EFFECTIVE CAPACITANCE BUFFERED LINE

Wiring Cap.	3000pF
TOTAL CAP.	3000pF

Buffered Bus pull-up

$$R2 = \frac{1\mu\text{sec}}{3000\text{pF}} = 333\Omega$$

EFFECTIVE CAPACITANCE REMOTE I²C DEVICES

1 × I ² C Devices	10pF
Strays	10pF
82B715 Buffer	10pF
TOTAL CAP.	30pF

I²C pull-up

$$R3 = \frac{1\mu\text{sec}}{30\text{pF}} = 33\text{K}\Omega$$

AS AN ADDITION TO AN EXISTING SYSTEM * :

R1 = 20KΩ

$$R2' = \frac{R2 \times 0.1R3}{R2 + 0.1R3} = 300\Omega$$

R3 not required since buffer always connected

FOR A PERMANENT SYSTEM * :

R1 not required since buffer always connected

$$R2' = \frac{0.1R1 \times R2 \times 0.1R3}{0.1R1 + R2 + 0.1R3} = 260\Omega$$

R3 not required since buffer always connected

*** NOTE:**

R1, R2 and R3 are calculated from the capacitive loading and a 1μsec time constant on each bus node. For an addition to an existing system, R2' (the new value for R2) is shown as being calculated from the parallel combination of R2 and the scaled value of R3; while for a permanent system R2, and scaled values of R1 and R3 have been used. Note that this example has used scaled resistor values and combined the node and cable capacitances.

CHECK FOR MAXIMUM PULL-UP CURRENT:

$$\frac{(5 - 0.4)\text{V}}{260\Omega} = 17.6\text{mA} < 30\text{mA}$$

SU00294

Figure 4. Typical Loading Calculation: I²C Bus with 82B715

Section 3

I²C Serial Bus

Application Notes & Articles

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

AN422	Using the 8XC751 microcontroller as an I ² C bus master	67
AN425	Interfacing the PCD8584 I ² C-bus controller to 80C51 family microcontrollers .	85
AN430	Using the 8XC751/752 in multimaster I ² C applications	104
AN433	I ² C slave routines for the 83C751	140
AN434	Connecting a PC keyboard to the I ² C-bus	146
AN438	I ² C routines for 8XC528	164
AN444	Using the P82B715 I ² C extender on long cables	186
ETV/AN89004	PLM51 I ² C software interface IIC51 (version 0.5)	206
EIE/AN91007	I ² C driver routines for 8XC751/2 microcontrollers	215
	Programming the I ² C interface	269

Using the 8XC751 microcontroller as an I²C bus master

AN422

DESCRIPTION

The 83C751/87C751 Microcontroller offers the advantages of the 80C51 architecture in a small package and at a low cost. It combines the benefits of a high-performance microcontroller with on-board hardware supporting the Inter-Integrated Circuit (I²C) bus interface.

The I²C bus, developed and patented by Philips, allows integrated circuits to communicate directly with each other via a simple bidirectional 2-wire bus. The comprehensive family of CMOS and bipolar ICs incorporating the on-chip I²C interface offers many advantages to designers of digital control for industrial, consumer and telecommunications equipment. A typical system configuration is shown in Figure 1.

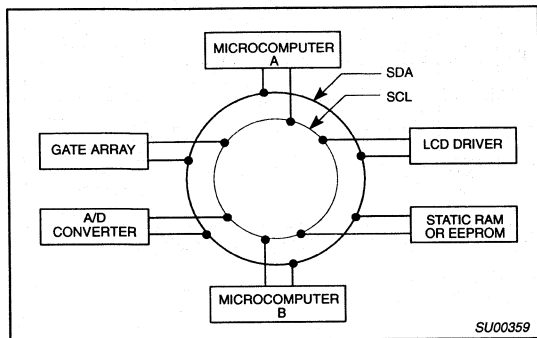


Figure 1. Typical I²C Bus Configuration

Interfacing the devices in an I²C based system is very simple because they connect directly to the two bus lines: a serial data line (SDA) and a serial clock line (SCL). System design can rapidly progress from block diagram to final schematic, as there is no need to design bus interfaces, and functional blocks on a block diagram correspond to actual ICs. A prototype system or a final product version can easily be modified or upgraded by 'clipping' or 'unclipping' ICs to or from the bus. The simplicity of designing with the I²C bus does not reduce its effectiveness; it is a reliable, multimaster bus with integrated addressing and data-transfer protocols (see Figure 2). In addition, the I²C-bus compatible ICs provide cost reduction benefits to equipment manufacturers, some of which are smaller IC packages and a minimization of PCB traces and glue logic.

The availability of microcontrollers like the 83C751, with on-board I²C interface, is a very powerful tool for system designers. The integrated protocols allow systems to be completely software defined. Software development time of different products can be reduced by assembling a library of reusable software modules. In addition, the multimaster capability allows rapid testing and alignment of end-products via external connections to an assembly-line computer.

The mask programmable 83C751 and its EPROM version, the 87C751, can operate as a master or a slave device on the I²C small area network. In addition to the efficient interface to the dedicated function ICs in the I²C family, the on-board interface facilities I/O and RAM expansion, access to EEPROM and processor-to-processor communications.

The multimaster capability of the I²C is very important but many designs do not require it. For many systems, it is sufficient that all communications between devices are initiated by a single, master processor. In this application note, use of the 8XC751 as an I²C bus master is described. Some of the technical features of the bus and the 83C751's special hardware associated with the I²C are discussed. Also included is a software example demonstrating I²C single master communications. Note that the sample routines are quite general, and therefore may be transferred easily to many applications.

The discussion of the I²C bus characteristics in this application note is by no means complete. Additional information for the I²C bus and the S83C751 Microcontroller can be found in the Microcontroller Users' Guide.

THE I²C BUS

The two lines of the I²C-bus are a serial data line (SDA) and a serial clock line (SCL). Both lines are connected to a positive supply via a pull-up resistor, and remain HIGH when the bus is not busy. Each device is recognized by a unique address—whether it is a microcomputer, LCD driver, memory or keyboard interface—and can operate as either a transmitter or receiver, depending on the function of the device. A device generating a message or data is a transmitter, and a device receiving the message or data is a receiver. Obviously, a passive function like an LCD driver could only be a receiver, while a microcontroller or a memory can both transmit and receive data.

Masters and Slaves

When a data transfer takes place on the bus, a device can either be a master or a slave. The device which initiates the transfer, and generates the clock signals for this transfer, is the master. At that time any device addressed is considered a slave. It is important to note that a master could either be a transmitter or a receiver; a master microcontroller may send data to a RAM acting as a transmitter, and then interrogate the RAM for its contents acting as a receiver—in both cases performing as the master initiating the transfer. In the same manner, a slave could be both a receiver and a transmitter.

The I²C is a multimaster bus. It is possible to have, in one system, more than one device capable of initiating transfers and controlling the bus (Figure 2). A microcontroller may act as a master for one transfer, and then be the slave for another transfer, initiated by another processor on the network. The master/slave relationships on the bus are not permanent, and may change on each transfer.

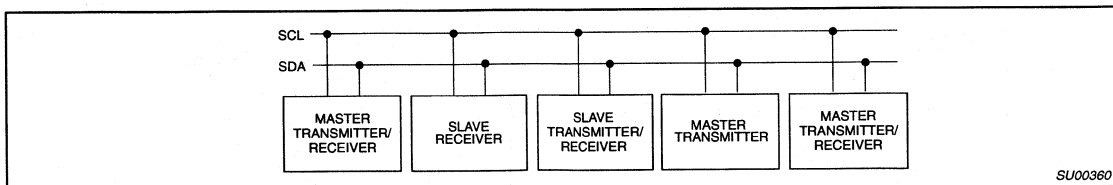


Figure 2. I²C Bus Connection

Using the 8XC751 microcontroller as an I²C bus master

AN422

As more than one master may be connected to the bus, it is possible that two devices will try to initiate a transfer at the same time. Obviously, in order to eliminate bus collisions and communications chaos, an arbitration procedure is necessary. The I²C design has an inherent arbitration and clock synchronization procedure relying on the wired-AND connection of the devices on the bus. In a typical multimaster system, a microcontroller program should allow it to gracefully switch between master and slave modes and preserve data integrity upon loss of arbitration. In this note, a simple case is presented describing the S83C751 operating as a single master on the bus.

Data Transfers

One data bit is transferred during each clock pulse (see Figure 3). The data on the SDA line must remain stable during the HIGH period of the clock pulse in order to be valid. Changes in the data line at this time will be interpreted as control signals. A HIGH-to-LOW transition of the data line (SDA) while the clock signal (SCL) is HIGH indicates a Start condition, and a LOW-to-HIGH transition of the SDA while SCL is HIGH defines a Stop condition (see Figure 4). The bus is considered to be busy after the Start condition and free again at a certain time interval after the Stop condition. The Start and Stop conditions are always generated by the master.

The number of data bytes transferred between the Start and Stop condition from transmitter to receiver is not limited. Each byte, which must be eight bits long, is transferred serially with the most significant bit first, and is followed by an acknowledge bit. (see Figure 5). The clock pulse related to the acknowledge bit is generated by the master. The device that acknowledges has to pull down the SDA line during the acknowledge clock pulse, while the transmitting device releases the SDA line (HIGH) during this pulse (see Figure 6).

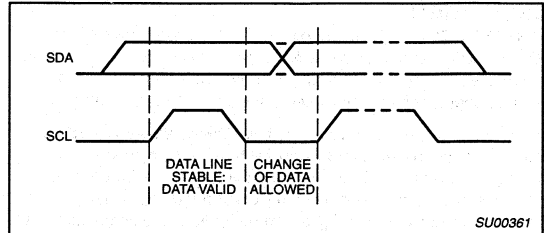


Figure 3. Bit Transfer on the I²C Bus

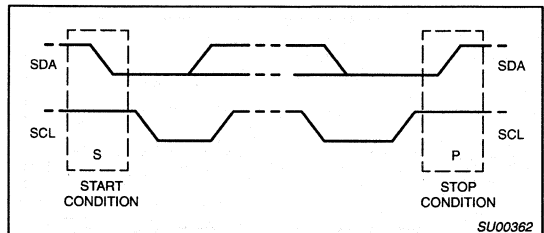


Figure 4. Start and Stop Conditions

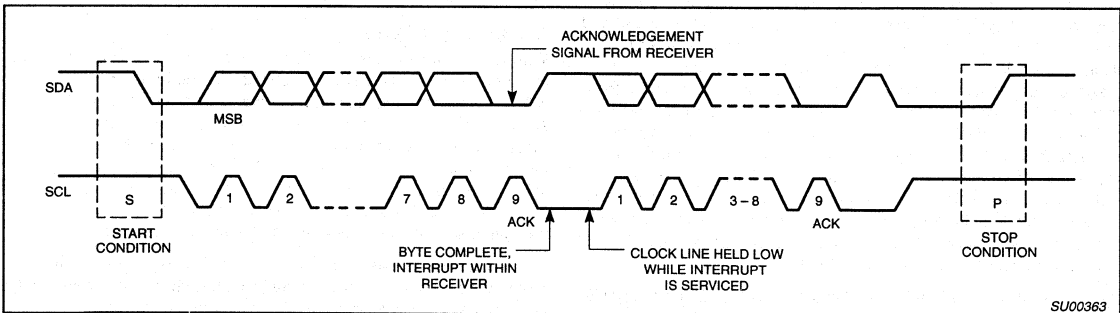


Figure 5. Data Transfer on the I²C Bus

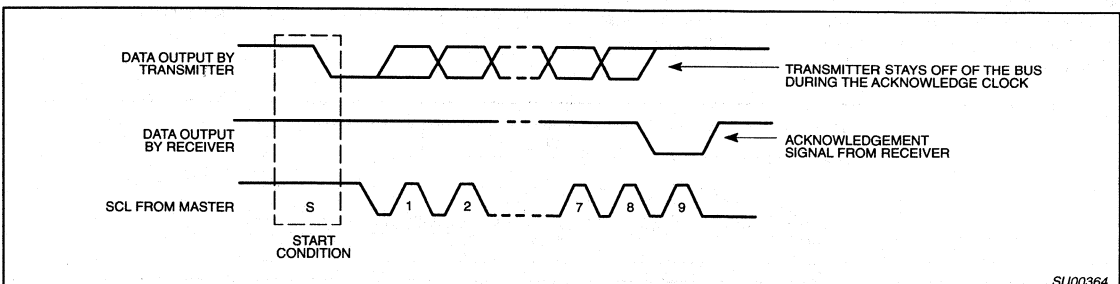


Figure 6. Acknowledge on the I²C Bus

Using the 8XC751 microcontroller as an I²C bus master

AN422

A slave receiver must generate an acknowledge after the reception of each byte, and a master must generate one after the reception of each byte clocked out of the slave transmitter. If a receiving device cannot receive the data byte immediately, it can force the transmitter into a wait state by holding the clock line (SCL) LOW. When designing a system, it is necessary to take into account cases when acknowledge is not received. This happens, for example, when the addressed device is busy in a real time operation. In such a case the master, after an appropriate "time-out", should abort the transfer by generating a Stop condition, allowing other transfers to take place. These "other transfers" could be initiated by other masters in a multimaster system, or by this same master.

There are two exceptions to the "acknowledge after every byte" rule. The first occurs when a master is a receiver: it must signal an end of data to the transmitter by NOT signalling an acknowledge on the last byte that has been clocked out of the slave. The acknowledge related clock, generated by the master should still take place, but the SDA line will not be pulled down. In order to indicate that this is an active and intentional lack of acknowledgement, we shall term this special condition as a "negative acknowledge".

The second exception is that a slave will send a negative acknowledge when it can no longer accept additional data bytes. This occurs after an attempted transfer that cannot be accepted.

The bus design includes special provisions for interfacing to microprocessors which implement all of the I²C communications in software only—it is called "Slow Mode". When all of the devices on the network have built-in I²C hardware support, the Slow Mode is irrelevant.

Addressing and Transfer Formats

Each device on the bus has its own unique address. Before any data is transmitted on the bus, the master transmits on the bus the address of the slave to be accessed for this transaction. A well-behaved slave with a matching address, if it exists on the network, should of course acknowledge the master's addressing. The addressing is done by the first byte transmitted by the master after the Start condition.

An address on the network is seven bits long, appearing as the most significant bits of the address byte. The last bit is a direction (R/W) bit. A zero indicates that the master is transmitting (WRITE) and a one indicates that the master requests data (READ). A complete data

transfer, comprised of an address byte indicating a WRITE and two data bytes is shown in Figure 7.

When an address is sent, each device in the system compares the first seven bits after the Start with its own address. If there is a match, the device will consider itself addressed by the master, and will send an acknowledge. The device could also determine if in this transaction it is assigned the role of a slave receiver or slave transmitter, depending on the R/W bit.

Each node of the I²C network has a unique seven bit address. The address of a microcontroller is of course fully programmable, while peripheral devices usually have fixed and programmable address portions. In addition to the "standard" addressing discussed here, the I²C bus protocol allows for "general call" addressing and interfacing to CBUS devices.

When the master is communicating with one device only, data transfers follow the format of Figure 7, where the R/W bit could indicate either direction. After completing the transfer and issuing a Stop condition, if a master would like to address some other device on the network, it could of course start another transaction, issuing a new Start.

Another way for a master to communicate with several different devices would be by using a "repeated start". After the last byte of the transaction was transferred, including its acknowledge (or negative acknowledge), the master issues another Start, followed by address byte and data—without effecting a Stop. The master may communicate with a number of different devices, combining READS and WRITES. After the last transfer takes place, the master issues a Stop and releases the bus. Possible data formats are demonstrated in Figure 8. Note that the repeated start allows for both change of a slave and a change of direction, without releasing the bus. We shall see later on that the change of direction feature can come in handy even when dealing with a single device.

In a single master system, the repeated start mechanism may be more efficient than terminating each transfer with a Stop and starting again. In a multimaster environment, the determination of which format is more efficient could be more complicated, as when a master is using repeated starts it occupies the bus for a long time and thus preventing other devices from initiating transfers.

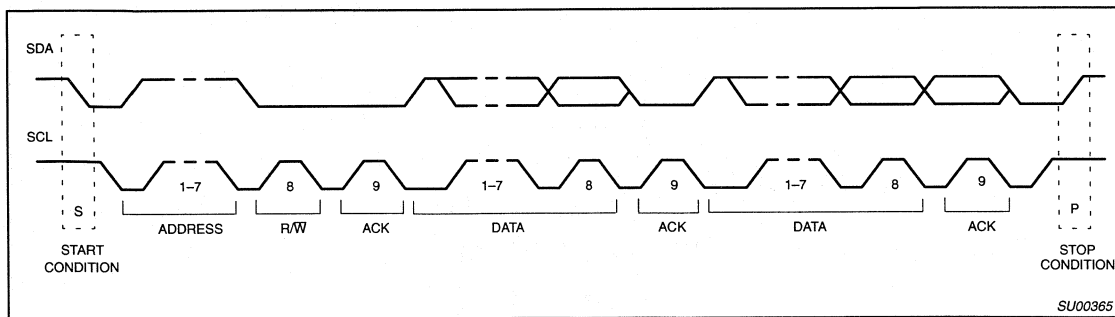


Figure 7. A Complete Data Transfer on the I²C-Bus

SU00365

Using the 8XC751 microcontroller as an I²C bus master

AN422

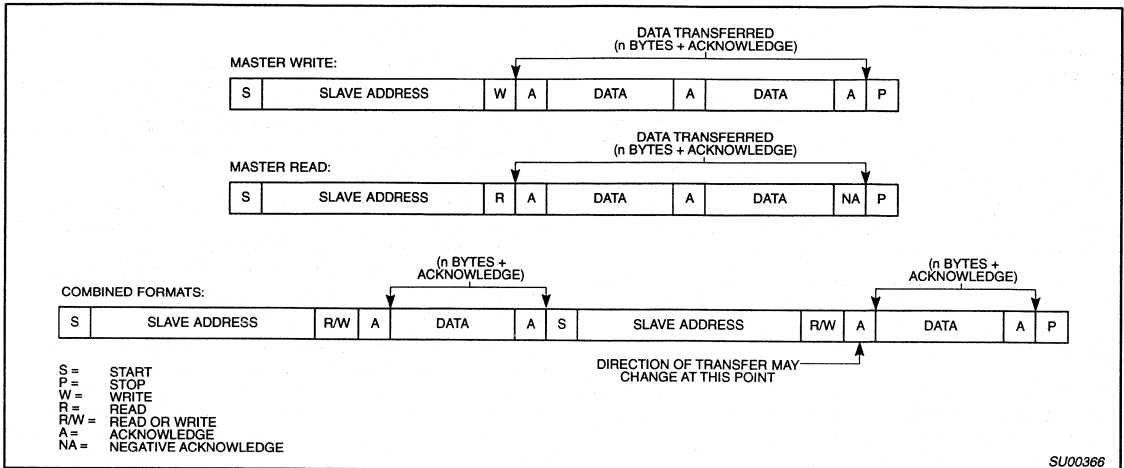


Figure 8. I²C Data Formats

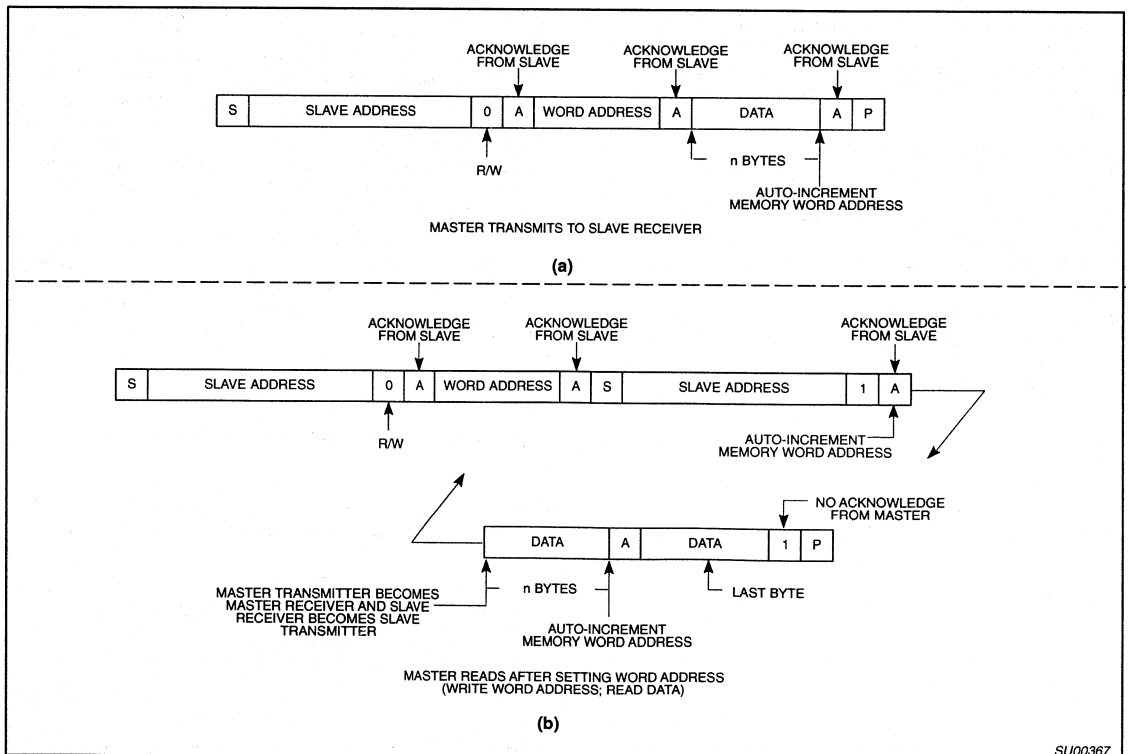


Figure 9. I²C Sub-Address Usage

Using the 8XC751 microcontroller as an I²C bus master

AN422

Use of Sub-Addresses

For some ICs on the I²C bus, the device address alone is not sufficient for effective communications, and a mechanism for addressing the internals of the device is necessary. A typical example when we want to access a specific word inside the device is addressing memories, or a sequence of memory locations starting at a specific internal address.

A typical I²C memory device like the PCF8570 RAM contains a built-in word address register that is incremented automatically after each data byte which is a read or written data byte. When a master communicates with the PCF8570 it must send a sub-address in the byte following the slave address byte. This sub-address is the internal address of the word the master wants to access for a single byte transfer, or the beginning of a sequence of locations for a multi-byte transfer. A sub-address is an 8-bit byte, unlike the device address, it does not contain a direction (R/W) bit, and like any byte transferred on the bus it must be followed by an acknowledge.

A memory write cycle is shown in Figure 9(a). The Start is followed by a slave byte with the direction bit set to WRITE, a sub-address byte, a number of data bytes and a Stop signal. The sub-address is loaded into the word address memory, and the data bytes which follow will be written one after the other starting with the sub-address location, as the register is incremented automatically.

The memory read cycle (see Figure 9(b)) commences in a similar manner, with the master sending a slave address with the direction bit set to WRITE with a following sub-address. Then, in order to reverse the direction of the transfer, the master issues a repeated Start followed again by the memory device address, but this time with the direction bit set to READ. The data bytes starting at the internal sub-address will be clocked out of the device, each followed by a master-generated acknowledge. The last byte of the read cycle will be followed by a negative acknowledge, signalling the end of transfer. The cycle is terminated by a Stop signal.

8XC751 I²C HARDWARE

The on-chip I²C bus hardware support of the 8XC751 allows operation on the bus at full speed, and simplifies the software needed for effective communications on the network. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing errors checks, and takes care of clock stretching and synchronization. The hardware support includes a bus time-out timer, called Timer I. The hardware is synchronized to the software either through polled loops or interrupts.

Two of the port 0 pins are multi-functional. When the I²C is active, the pin associated with P0.0 functions as SCL, and the pin associated with P0.1 functions as SDA. These pins have an open drain output.

Two of the five 8XC751 interrupt sources may be used for I²C support. The I²C interrupt is enabled by the EI2 flag of the interrupt enable register, and its service routine should start at address 023h. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates a new data bit on the bus, or a special condition occurs: Start, Stop or arbitration loss. The interrupt is induced by the ATN flag—see below for the conditions for setting this flag. The Timer I overflow interrupt is enabled by the ET1 flag, and the service routine starts at 01Bh.

The I²C port is controlled through three special function registers: I²C Control (I2CON), I²C Configuration (I2CFG), and I²C Data (I2DAT). The register addresses are shown in Table 1.

Although the following discussion of the hardware and register details is not complete, it should give a better understanding of the programming examples.

Timer I

In I²C applications, Timer I is dedicated to the port timing generation and bus monitoring. In non-I²C applications, it is available for use as a fixed time base.

In its port timing generation function, Timer I is used to generate SCL, the I²C clock. Timer I is clocked once per machine cycle (osc/12), so that the toggle rate of SCL will be some multiple of that rate. Because the 83C751 can be run over a wide range of oscillator frequencies, it is necessary to adjust SCL for the part's oscillator frequency. This allows the I²C bus to be used at its highest transfer rates independent of the oscillator frequency. SCL is adjusted by writing to two bits (CT0 and CT1) in the I2CFG special function register (see Table 2). The inverse of the values in CT0 and CT1 are loaded into the least significant two bit locations of Timer I every time the fourth bit of the timer is toggled. (A value is actually loaded into the least significant three bits, the third bit being 0 unless both CT0 and CT1 are programmed high and in that case the third bit is 1). SCL is then toggled every time the fourth bit of Timer I is toggled. For example: if CT1 = 0 and CT0 = 1 then the least significant three bits of Timer I would be preloaded with 2 (010 binary). Timer I would then count 3, 4, 5, 6, 7, 8 (6 counts or machine cycles). On 8, the fourth bit of Timer I will toggle, SCL will toggle and the 3 least significant bits will again be preloaded with the value 2 (010).

Table 1. I²C Special Function Register Addresses

REGISTER			BIT ADDRESS							
Name	Symbol	Address	MSB						LSB	
I ² C Control	I2CON	98	9F	9E	9D	9C	9B	9A	99	98
I ² C Data	I2DAT	99	–	–	–	–	–	–	–	–
I ² C Configuration	I2CFG	D8	DF	DE	DD	DC	DB	DA	D9	D8

Table 2. CT0, CT1 Timer I Settings

CT1 Values	CT0 Values	Timer I Counts	Oscillator Freq (MHz)
1	0	7	16
0	1	6	15, 14, 13
0	0	5	12, 11
1	1	4	10 or less

Timer I counts = f_{osc} (MHz) \times 0.39 (rounded up to next integer).

Using the 8XC751 microcontroller as an I²C bus master

AN422

For the bus monitoring function, Timer I is used as a “watchdog timer” for bus hang-ups. It creates an interrupt when the SCL line stays in one state for an extended period of time while the bus is active (between a Start condition and a following Stop condition). SCL “stuck low” indicates a faulty master or slave. SCL “stuck high” may mean a faulty device, or that noise induced onto the I²C caused all masters to withdraw from the I²C arbitration.

The time-out interval of Timer I is fixed (cannot be set): it carries out and interrupts (if enabled) when about 1024 machine cycles have elapsed since a change on SCL within a frame. In other words, whenever I²C is active and Timer I is enabled, the falling edge of SCL will reset Timer I. If SCL is not toggled low for 1024 machine cycles, Timer I will overflow and cause an interrupt. (Note: we wrote “about 1024 machine cycles” although for the sake of accuracy—this number is affected by the setting of the CT0 and CT1 bits mentioned above and may vary by up to three machine cycles) The exact number of cycles for a time-out is not critical; what is important is that it indicates SCL is stuck.

In addition to the interrupt, upon Timer I overflow the I²C port hardware is reset. This is useful for multiple master systems in situations where a bus fault might cause the bus to hang-up due to a lack of software response. When this happens, SCL will be released, and I²C operation between other devices can continue.

I2CON Register

The I²C control register (I2CON) can be written to (see Figure 10). When writing to the I2CON register, one should use bit masks as demonstrated in the example program. Trying to clear or set the bits in the register using the bit addressing capabilities of the 8XC751 may lead to undesirable results. The reason is that a command like CLR reads the register, sets the bit and writes it back, and the write-back may affect other bits.

I2CFG Register

The configuration register (I2CFG) is a read/write register (see Figure 11).

I2DAT Register

The I²C data register (I2DAT) is a read/write register, where the MSB represents the data received or data to be sent. The other seven bits are read as 0 (see Figure 12).

Transmit Active State

The transmit active state—Xmit Active—is an internal state in the I²C interface that is affected by the I²C registers as explained above. The I²C interface will only drive the SDA line low when Xmit Active is set. Xmit Active is set by writing the I2DAT register, or by writing I2CON with XSTR = 1 or XSTP = 1. The ARL bit will be set to 1 only when Xmit Active is set—in such a case Xmit Active will be automatically reset upon arbitration loss. Xmit Active is cleared by writing 1 to CXA at I2CON register or by reading the I2DAT register.

I2CON READ	RDAT	ATN	DRDY	ARL	STR	STP	MASTER	—
RDAT	Received DATA bit. The value of SDA latched by the rising edge of SCL. Its contents is identical to RDAT in the I2DAT register. Reading the received data here allows doing so without clearing DRDY and releasing SCL.							
ATN	An “ATteNtion” flag, set when any one of DRDY, ARL, STR or STP is set. This flag allows a single bit testing for terminating “wait loops”, indicating a meaningful event on the bus. This flag also activates the I ² C interrupt request.							
DRDY	Data ReaDY flag. Set by a rising edge of SCL when I ² C is active, except at an idle slave. This flag is cleared by reading or writing the I2DAT register, or by writing a 1 to CDR (at the same address, when I2CON is written).							
ARL	ARbitration Loss flag. Indicates that this device lost arbitration while trying to take control of the bus.							
STR	STaRt flag. Set when a Start condition is detected, except at an idle slave.							
STP	SToP flag. Set when a Stop condition is detected, except at an idle slave.							
MASTER	This flag is set when the controller is a bus master (or a potential master, prior to arbitration).							
I2CON WRITE	CXA	IDLE	CDR	CARL	CSTR	CSTP	XSTR	XSTP
CXA	“Clear Xmit Active”. Writing a 1 to CXA clears the internal transmit-active state.							
IDLE	Setting this bit will cause a slave to enter idle mode and ignore the I ² C bus until the next Start is detected. If the software sets the MASTRQ flag, the device may stop idling by turning into a master.							
CDR	Clear Data Ready. Clears the DRDY flag.							
CARL	Clear Arbitration Lost. Clears the ARL flag.							
CSTR	Clear STaRt. Clears the STR flag.							
CSTP	Clear SToP. Clears the STP flag.							
XSTR	“Xmit repeated STaRt”. Writing a 1 to this bit causes the hardware to issue a Repeated Start signal. A side effect will be setting the internal Xmit Active state. This should be used only when the device is a master.							
XSTP	“Xmit SToP”. Issues a Stop condition. The Xmit active state is set.							

SU00368

Figure 10. I2CON Register

Using the 8XC751 microcontroller as an I²C bus master

AN422

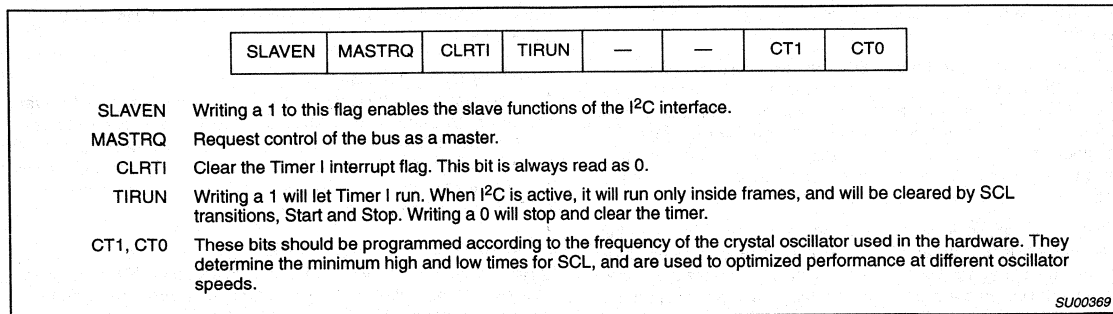


Figure 11. I2CFG Register

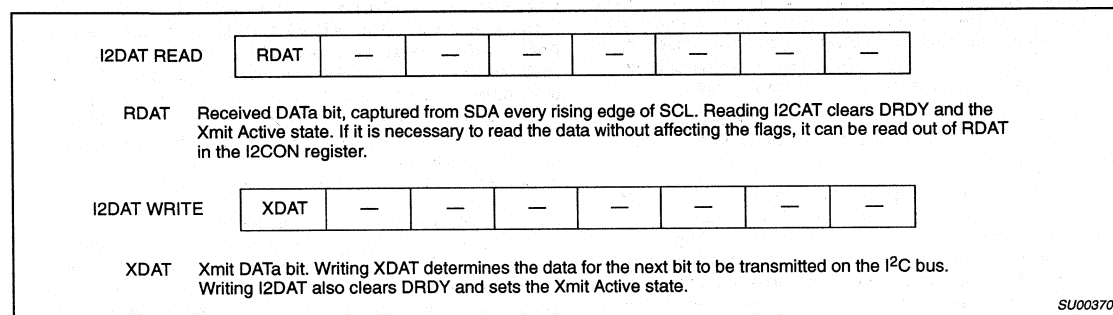


Figure 12. I2DAT Register

PROGRAMMING EXAMPLE

The listing demonstrates communications routines for the 8XC751 as an I²C bus master in a single-master system.

The single-master system is less complicated than a multimaster environment. The programmer does not have to worry about switching between master and slave roles, or the consequences of an arbitration loss.

The I²C interrupt is not used, and therefore disabled. There is no need for frame Start interrupts, as this processor is the only bus master and all data transfers are initiated by it when the appropriate routines are called by the application. No one else generates frame Starts which could be an interrupt source in a multimaster system. Within the frames we monitor bus activity with a wait-loop which polls the ATN flag. As we expect the bus to operate in its full-speed mode, we can assume that only a small amount of time will be wasted in those loops, and the use of interrupts would be less efficient.

The 8XC751 has single-bit I²C hardware interface, where the registers may directly affect the levels on the bus and the software interacting with the register takes part in the protocol implementation. The hardware and the low-level routines dealing with the registers are tightly coupled. Therefore, one should take extra care if trying to modify these lower level routines.

The beginning of the program, at address 0, contains the reset vector, where the microcontroller begins executing code after a hardware reset. In this case, the code simply jumps to the main part of the program, which begins at the label 'Reset' near the end of the listing.

The main program is a simple demonstration of the I²C routines which comprise the balance of the listing. It first enables the Timer I interrupt, and sets up some sample data to be transmitted. Beginning at the label MainLoop, the program then proceeds to transmit one byte of data to a slave device at address 48 hexadecimal, using the routine titled 'SendData'. In our demonstration hardware, this address corresponds to an 8-bit I/O port that drives eight monitor LEDs. The program then reads back one byte of data from the same port using the routine 'RcvData'. The SendData and RcvData routines can send or receive multiple bytes of data, the number of which is determined by the variable 'ByteCnt'.

Upon return from both SendData and RcvData, the program checks the system flag named 'Retry' to see if the transfer was completed correctly. If not, it loops back and attempts the same transfer again.

Next, the program sends four bytes of data to a 256-byte EEPROM device, an 8-pin part called the PCF8582. The routine 'SendSub' is used for this purpose. The EEPROM was located at address A0 hexadecimal on our board. This device uses the sub-addressing feature to select a starting location to address in the EEPROM array. When data is written to the EEPROM, the address is automatically incremented so that the data bytes are stored in consecutive locations.

Finally the program reads back four bytes of data from the EEPROM using the routine 'RcvSub'. Calls to SendSub and RcvSub should also be followed by a test of the Retry flag to insure that all went according to plan.

Using the 8XC751 microcontroller as an I²C bus master

AN422

This entire process is repeated indefinitely by jumping back to MainLoop.

Back at the beginning of the program, the next location after the reset vector is the Timer I interrupt service routine. The microcontroller will go to address 1B hexadecimal if Timer I overflows. This routine stops the timer, clears the timer interrupt, clears the pending interrupt so that other interrupts will be enabled, restores the stack pointer, and jumps to the 'Recover' routine to try to correct whatever stopped the I²C bus and allowed Timer I to overflow.

Next in the listing come the main I²C service routines. These are the routines SendData, RcvData, SendSub, and RcvSub that were called from the main program. Both of the send routines use the data area labeled 'XmtDat' as the transmit data buffer. In this sample program, four bytes were reserved for this area, but it could be larger or smaller depending on the application. The two receive routines use another four byte buffer labeled 'RcvDat' to store received data. All of these routines use the variables 'SlvAdr' and 'ByteCnt' to determine the slave address and the number of bytes to be sent or received, respectively. The SendSub and RcvSub routines use the variable 'SubAdr' as the sub-address to send to the slave device.

Following the main I²C service routines in the listing are the subroutines that are called by the main routines to deal intimately with the I²C hardware.

The 'SendAddr' subroutine requests mastership of the I²C bus and calls the routine 'XmitAddr' to complete sending the slave address. The bulk of the XmitAddr routine is shared with the 'XmitByte' subroutine which sends data bytes on the I²C bus. XmitByte is also used to send I²C sub-addresses. Both subroutines check for an acknowledge from the slave device after every byte is sent on the I²C bus.

The next subroutine 'RDack' calls the 'RcvByte' routine to read in a byte of data. It then sends an acknowledge to the slave device. RDack is used to receive all data except for the last byte of a receive data frame, where the acknowledge is omitted by the bus master. The RcvByte subroutine is called directly for the last byte of a frame.

The 'SendStop' subroutine causes a stop condition on the I²C, thus ending a frame. The 'RepStart' subroutine sends a repeated start condition on the I²C bus, to allow the master to start a new frame without first having to send an intervening stop.

The lower level subroutines deal directly with the hardware. The tight coupling between hardware and software is best demonstrated by the

following explanations, relating to two cases in which the code is not self evident.

Sending the Address

When sending the address byte in the Send Addr subroutine, the first bit is written to I2DAT prior to the loop where the other seven bits are sent (SendAd2). The reason is that we need to clear the Start condition in order to release the SCL line, and this is done explicitly by the subsequent command. When SCL is released, the correct bit (MSB of address) must already be in I2DAT.

Capturing the Received Data

Typically, a program receiving data waits in a loop for ATN, and when detected, checks DRDY. If DRDY = 1 then there was a rising SCL, and the new data can be read from RDAT in I2CON or I2DAT. Reading or writing I2DAT clears DRDY, thus releasing SCL.

When reading the last bit in a byte, it should be read from I2CON, and not I2DAT (see the end of the RcvByte routine). This way the Data Ready (DRDY) flag is not cleared, and the low period on SCL is stretched. The reason for doing so is that upon reception of the last bit of a received byte the master must react with an acknowledge. In order to ensure that we "wait" with the acknowledge clock (release of SCL) until the acknowledge level is issued on SDA, the last bit is read out of I2CON and not I2DAT. SCL is stretched low until the acknowledge level is written into I2DAT by the software.

Bus Faults and Other Exceptions

Bus exceptions are detected either by Timer I time-out, or "illegal" logic states tested for and detected by the software. Upon Timer I time-out, a bus recovery is attempted by the Recover routine. The final section of the listing is this 'Recover' routine. Its job is to try to restore control of the I²C bus to the main program. First, the subroutine 'FixBus' is called. It checks to see if only the SDA line is 'stuck', and if so, tries to correct it by sending some extra clocks on the SCL line, and forcing a stop condition on the bus. If this does not work, another subroutine 'BusReset' is called. This generally happens when a severe bus error occurs, such as a shorted clock line. The philosophy used in this code is that the only chance of recovering from a severe error is to cause a reset of the I²C hardware by deliberately forcing Timer I to time out. This method allows recovery from a temporary short or other serious condition on the I²C bus.

Using the 8XC751 microcontroller as an I²C bus master

AN422

I2CAPP

83C751 Single Master I2C Routines

09/07/89

```

1
2 ;*****
3 ;
4 ; Sample I2C Single Master Routines for the 83C751
5
6 ;*****
7
8 $TITLE(83C751 Single Master I2C Routines)
9 $DATE(09/07/89)
10 $MOD751
11 $DEBUG
12
13
14 ; Value definitions.
15
0002 16 CTVAL      EQU    02h      ;CT1, CT0 bit values for I2C.
17
18
19 ; Masks for I2CFG bits.
20
0010 21 BTIR      EQU    10h      ;Mask for TIRUN bit.
0040 22 BMRQ      EQU    40h      ;Mask for MASTRQ bit.
23
24
25 ; Masks for I2CON bits.
26
0080 27 BCXA      EQU    80h      ;Mask for CXA bit.
0040 28 BIDLE     EQU    40h      ;Mask for IDLE bit.
0020 29 BCDR      EQU    20h      ;Mask for CDR bit.
0010 30 BCARL     EQU    10h      ;Mask for CARL bit.
0008 31 BCSTR     EQU    08h      ;Mask for CSTR bit.
0004 32 BCSTP     EQU    04h      ;Mask for CSTP bit.
0002 33 BXSTR     EQU    02h      ;Mask for XSTR bit.
0001 34 BXSTP     EQU    01h      ;Mask for XSTP bit.
35
36
37 ; RAM locations used by I2C routines.
38
0021 39 BitCnt    DATA   21h      ;I2C bit counter.
0022 40 ByteCnt   DATA   22h
0023 41 SlvAdr    DATA   23h      ;Address of active slave.
0024 42 SubAdr    DATA   24h
43
0025 44 RcvDat    DATA   25h      ;I2C receive data buffer (4 bytes).
45 ; addresses 25h through 28h.
46
0029 47 XmtDat    DATA   29h      ;I2C transmit data buffer (4 bytes).
48 ; addresses 29h through 2Ch.
49
002D 50 StackSave DATA   2Dh      ;Saves stack addr for bus recovery.
51
0020 52 Flags     DATA   20h      ;I2C software status flags.
0000 53 NoAck     BIT     Flags.0 ;Indicates missing acknowledge.
0001 54 Fault     BIT     Flags.1 ;Indicates a bus fault of some kind.
0002 55 Retry     BIT     Flags.2 ;Indicates that last I2C transmission
56 ; failed and should be repeated.
57
0080 58 SCL       BIT     P0.0      ;Port bit for I2C serial clock line.
0081 59 SDA       BIT     P0.1      ;Port bit for I2C serial data line.
60

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

61 ;*****
62 ;                               Begin Code
63 ;*****
64
65 ; Reset and interrupt vectors.
66
0000 21E1 67             AJMP   Reset           ;Reset vector at address 0.
68
69
70 ; A timer I timeout usually indicates a 'hung' bus.
71
001B    72             ORG     1Bh           ;Timer I (I2C timeout)
                                ; interrupt.
001B D2DD 73   TimerI:  SETB   CLRTI          ;Clear timer I interrupt.
001D C2DC 74             CLR     TIRUN
001F 1126 75             ACALL  ClrInt       ;Clear interrupt pending.
0021 852D81 76             MOV     SP,StackSave ;Restore stack for return
                                ; to main.
0024 218A 77             AJMP   Recover       ;Attempt bus recovery.
0026 32    78   ClrInt:  RETI
79
80
81 ;*****
82 ;                               Main Transmit and Receive Routines
83 ;*****
84
85 ; Send data byte(s) to slave.
86 ; Enter with slave address in SlvAdr, data in XmtDat buffer,
87 ; # of data bytes to send in ByteCnt.
88
0027 C200 89   SendData: CLR     NoAck           ;Clear error flags.
0029 C201 90             CLR     Fault
002B C202 91             CLR     Retry
002D 85812D 92             MOV     StackSave,SP ;Save stack address
                                ; for bus fault.
0030 E523 93             MOV     A,SlvAdr       ;Get slave address.
0032 310C 94             ACALL  SendAddr      ;Get bus and send slave addr.
0034 200012 95             JB      NoAck,SDEX      ;Check for missing
                                ; acknowledge.
0037 200112 96             JB      Fault,SDatErr    ;Check for bus fault.
003A 7829 97             MOV     R0,#XmtDat    ;Set start of transmit
                                ; buffer.
98
003C E6    99   SDLoop:  MOV     A,@R0           ;Get data for slave.
003D 08    100             INC     R0
003E 3125 101             ACALL  XmitByte      ;Send data to slave.
0040 200006 102             JB      NoAck,SDEX      ;Check for missing
                                ; acknowledge.
0043 200106 103             JB      Fault,SDatErr    ;Check for bus fault.
0046 D522F3 104             DJNZ   ByteCnt,SDLoop
105
0049 3166 106   SDEX:    ACALL  SendStop      ;Send an I2C stop.
004B 22    107             RET
108
109
110 ; Handle a transmit bus fault.
111
004C 218A 112   SDatErr:  AJMP   Recover       ;Attempt bus recovery.
113
114
115 ; Receive data byte(s) from slave.
116 ; Enter with slave address in SlvAdr,
117 ; # of data bytes requested in ByteCnt.
118 ; Data returned in RcvDat buffer.

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

118
004E C200      119   RcvData:  CLR    NoAck          ;Clear error flags.
0050 C201      120           CLR    Fault
0052 C202      121           CLR    Retry
0054 85812D    122           MOV    StackSave,SP      ;Save stack address
                                ; for bus fault.
0057 E523      123           MOV    A,SlvAdr         ;Get slave address.
0059 D2E0      124           SETB  ACC.0             ;Set bus read bit.
005B 310C      125           ACALL SendAddr         ;Send slave address.
005D 200023    126           JB    NoAck,RDEX       ;Check for missing
                                ; acknowledge.
0060 200123    127           JB    Fault,RDatErr    ;Check for bus fault.
                                128
0063 7825      129           MOV    R0,#RcvDat      ;Set start of receive
                                ; buffer.
0065 D52202    130           DJNZ  ByteCnt,RDLoop   ;Check for count = 1
                                ; byte only.
0068 800A      131           SJMP  RDLast
                                132
006A 3143      133   RDLoop:  ACALL  RDAck          ;Get data and send
                                ; an acknowledge.
006C 200117    134           JB    Fault,RDatErr    ;Check for bus fault.
006F F6        135           MOV    @R0,A           ;Save data.
0070 08        136           INC   R0
0071 D522F6    137           DJNZ  ByteCnt,RDLoop   ;Repeat until last
                                ; byte.
                                138
0074 314F      139   RDLast:  ACALL  RcvByte       ;Get last data byte
                                ; from slave.
0076 20010D    140           JB    Fault,RDatErr    ;Check for bus
                                ; fault.
0079 F6        141           MOV    @R0,A           ;Save data.
                                142
007A 759980    143           MOV    I2DAT,#80h      ;Send negative
                                ; acknowledge.
007D 309EFD    144           JNB   ATN,$            ;Wait for NAK sent.
0080 309D03    145           JNB   DRDY,RDatErr    ;Check for bus
                                ; fault.
                                146
0083 3166      147   RDEX:    ACALL  SendStop      ;Send an I2C bus
                                ; stop.
0085 22        148           RET
                                149
                                150
                                151   ; Handle a receive bus fault.
                                152
0086 218A      153   RDatErr: AJMP   Recover          ;Attempt bus recovery.
                                154
                                155
                                156   ; Send data byte(s) to slave with subaddress.
                                157   ; Enter with slave address in ACC, subaddress in
                                ; SubAdr, # of bytes to send in ByteCnt,
                                ; data in XmtDat buffer.
                                158
                                159
0088 C200      160   SendSub: CLR    NoAck          ;Clear error flags.
008A C201      161           CLR    Fault
008C C202      162           CLR    Retry
008E 85812D    163           MOV    StackSave,SP      ;Save stack address
                                ; for bus fault.
0091 E523      164           MOV    A,SlvAdr         ;Get slave address.
0093 310C      165           ACALL SendAddr         ;Get bus and send
                                ; slave address.

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

0095 20001C      166          JB      NoAck,SSEX      ;Check for missing
                                           ; acknowledge.
0098 20011C      167          JB      Fault,SSubErr    ; Check for bus
                                           ; fault.
                                           168
009B E524        169          MOV     A,SubAdr      ;Get slave subaddress.
009D 3125        170          ACALL  XmitByte      ;Send subaddress.
009F 200012      171          JB      NoAck,SSEX    ;Check for missing
                                           ; acknowledge.
00A2 200112      172          JB      Fault,SSubErr ;Check for bus fault.
00A5 7829        173          MOV     R0,#XmtDat   ;Set start of
                                           ; transmit buffer.
                                           174
00A7 E6          175          SSLoop: MOV    A,@R0      ;Get data for slave.
00A8 08          176          INC     R0
00A9 3125        177          ACALL  XmitByte      ;Send data to slave.
00AB 200006      178          JB      NoAck,SSEX    ;Check for missing
                                           ; acknowledge.
00AE 200106      179          JB      Fault,SSubErr ;Check for bus fault.
00B1 D522F3      180          DJNZ   ByteCnt,SSLoop
                                           181
00B4 3166        182          SSEX:   ACALL  SendStop ;Send an I2C stop.
00B6 22          183          RET
                                           184
                                           185
                                           186          ; Handle a transmit bus fault.
                                           187
00B7 218A        188          SSubErr: AJMP   Recover ;Attempt bus recovery.
                                           189
                                           190
                                           191          ; Receive data byte(s) from slave with subaddress.
                                           192          ; Enter with slave address in SlvAdr, subaddress in SubAdr,
                                           ; # of data bytes requested in ByteCnt.
                                           193          ; Data returned in RcvDat buffer.
                                           194
00B9 C200        195          RcvSub: CLR    NoAck      ;Clear error flags.
00BB C201        196          CLR    Fault
00BD C202        197          CLR    Retry
00BF 85812D      198          MOV     StackSave,SP ;Save stack address
                                           ; for bus fault.
                                           199
00C2 E523        199          MOV     A,SlvAdr     ;Get slave address.
00C4 310C        200          ACALL  SendAddr     ;Send slave address.
00C6 20003E      201          JB      NoAck,RSEX   ;Check for missing
                                           ; acknowledge.
00C9 20013E      202          JB      Fault,RSubErr ;Check for bus fault.
                                           203
00CC E524        204          MOV     A,SubAdr     ;Get slave subaddress.
00CE 3125        205          ACALL  XmitByte     ;Send subaddress.
00D0 200034      206          JB      NoAck,RSEX   ;Check for missing
                                           ; acknowledge.
00D3 200134      207          JB      Fault,RSubErr ;Check for bus fault.
                                           208
00D6 317A        209          ACALL  RepStart     ;Send repeated start.
00D8 20012F      210          JB      Fault,RSubErr ;Check for bus fault.
00DB E523        211          MOV     A,SlvAdr     ;Get slave address.
00DD D2E0        212          SETB   ACC.0        ;Set bus read bit.
00DF 3115        213          ACALL  SendAd2      ;Send slave address.
00E1 200023      214          JB      NoAck,RSEX   ;Check for missing
                                           ; acknowledge.
00E4 200123      215          JB      Fault,RSubErr ;Check for bus fault.
                                           216
00E7 7825        217          MOV     R0,#RcvDat   ;Set start of
                                           ; receive buffer.
00E9 D52202      218          DJNZ   ByteCnt,RSLoop ;Check for count = 1
                                           ; byte only.

```


Using the 8XC751 microcontroller as an I²C bus master

AN422

```

00EC 800A          219          SJMP   RSLast
                                220
00EE 3143          221   RSLoop:  ACALL  RDAck           ;Get data and send
                                ; an acknowledge.
00F0 200117        222          JB     Fault,RSubErr ;Check for bus fault.
00F3 F6           223          MOV    @R0,A          ;Save data.
00F4 08           224          INC    R0
00F5 D522F6        225          DJNZ   ByteCnt,RSLoop ;Repeat until last byte.
                                226
00F8 314F          227   RSLast:  ACALL  RcvByte           ;Get last data byte
                                ; from slave.
00FA 20010D        228          JB     Fault,RSubErr ;Check for bus fault.
00FD F6           229          MOV    @R0,A          ;Save data.
                                230
00FE 759980        231          MOV    I2DAT,#80h    ;Send negative
                                ; acknowledge.
0101 309EFD        232          JNB   ATN,$          ;Wait for NAK sent.
0104 309D03        233          JNB   DRDY,RSubErr  ;Check for bus fault.
                                234
0107 3166          235   RSEX:    ACALL  SendStop        ;Send an I2C bus stop.
0109 22           236          RET
                                237
                                238
                                239   ; Handle a receive bus fault.
                                240
010A 218A          241   RSubErr: AJMP   Recover           ;Attempt bus recovery.
                                242
                                243
                                244   ;*****
                                245   ; Subroutines
                                246   ;*****
                                247
                                248   ; Send address byte.
                                249   ; Enter with address in ACC.
                                250
010C 75D852        251   SendAddr: MOV    I2CFG,#EMRQ+BTIR+CTVAL ;Request I2C bus.
010F 309EFD        252          JNB   ATN,$          ;Wait for bus
                                ; granted.
0112 309908        253          JNB   Master,SAErr   ;Should have
                                ; become the bus
                                ; master.
0115 F599          254   SendAd2:  MOV    I2DAT,A          ;Send first bit,
                                ; clears DRDY.
0117 75981C        255          MOV    I2CON,#BCARL+BCSTR+BCSTP ;Clear start,
                                ; releases SCL.
011A 3120          256          ACALL XmitAddr          ;Finish sending
                                ; address.

011C 22           257          RET
                                258
011D D201          259   SAErr:   SETB   Fault           ;Return bus fault
                                ; status.
011F 22           260          RET
                                261
                                262
                                263   ; Byte transmit routine.
                                264   ; Enter with data in ACC.
                                265   ; XmitByte : transmits 8 bits.
                                266   ; XmitAddr : transmits 7 bits (for address only).
                                267
0120 752108        268   XmitAddr: MOV    BitCnt,#8          ;Set 7 bits of
                                ; address count.
0123 8005          269          SJMP   XmBit2
                                270

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

0125 752108      271  XmitByte:  MOV    BitCnt,#8          ;Set 8 bits of data
                                     ; count.
0128 F599        272  XmBit:     MOV    I2DAT,A          ;Send this bit.
012A 23          273  XmBit2:   RL     A              ;Get next bit.
012B 309EFD      274          JNB    ATN,$            ;Wait for bit sent.
012E 309D0F      275          JNB    DRDY,XMErr        ;Should be data ready.
0131 D521F4      276          DJNZ   BitCnt,XmBit      ;Repeat until all bits sent.
0134 7598A0      277          MOV    I2CON,#BCDR+BCXA      ;Switch to
                                     ; receive mode.
0137 309EFD      278          JNB    ATN,$            ;Wait for acknowledge
                                     ; bit.
013A 309F02      279          JNB    RDAT,XMBX        ;Was there an ack?
013D D200        280          SETB   NoAck              ;Return no acknowledge
                                     ; status.
013F 22          281  XMBX:     RET
                                     282
0140 D201        283  XMErr:    SETB   Fault          ;Return bus fault
                                     ; status.
0142 22          284          RET
                                     285
                                     286
                                     287  ; Byte receive routines.
0143 314F        288          ; RDack : receives a byte of data, then sends
                                     ; an acknowledge.
0145 759900      289          ; RcvByte : receives a byte of data.
0148 309EFD      290          ; Data returned in ACC.
014B 309D15      291
014E 22          292  RDack:    ACALL  RcvByte        ;Receive a data byte.
0152 E4          293          MOV    I2DAT,#0          ;Send receive
                                     ; acknowledge.
0153 4599        294          JNB    ATN,$            ;Wait for acknowledge
0155 23          295          JNB    DRDY,RdErr        ; sent.
0156 309EFD      296          JNB    DRDY,RdErr        ;Check for bus fault.
0159 309D07      297          RET
015C D521F4      298  RcvByte:  MOV    BitCnt,#8          ;Set bit count.
015F A29F        299          CLR    A                ;Init received byte
                                     ; to 0.
0161 33          300  RBit:     ORL    A,I2DAT        ;Get bit, clear ATN.
0162 22          301          RL     A              ;Shift data.
0163 D201        302          JNB    ATN,$            ;Wait for next bit.
0165 22          303          JNB    DRDY,RdErr        ;Should be data ready.
0166 C2DE        304          DJNZ   BitCnt,RBit      ;Repeat until 7 bits
                                     ; are in.
0168 759821      305          MOV    C,RDAT          ;Get last bit, don't
016B 309EFD      306          RLC    A              ; clear ATN.
016E 759820      307          RET                    ;Form full data byte.
0171 309EFD      308
0174 759894      309  RdErr:    SETB   Fault          ;Return bus fault status.
0177 C2DC        310          RET
0179 22          311
                                     312
                                     313  ; I2C stop routine.
0179 22          314
0179 22          315  SendStop: CLR    MASTRQ        ;Release bus
                                     ; mastership.
0168 759821      316          MOV    I2CON,#BCDR+BXSTP ;Generate a bus stop.
016B 309EFD      317          JNB    ATN,$            ;Wait for atn.
016E 759820      318          MOV    I2CON,#BCDR        ;Clear data ready.
0171 309EFD      319          JNB    ATN,$            ;Wait for stop sent.
0174 759894      320          MOV    I2CON,#BCARL+BCSTP+BCXA ;Clear I2C bus.
0177 C2DC        321          CLR    TIRUN          ;Stop timer I.
0179 22          322          RET
0179 22          323
0179 22          324

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

324
325 ; I2C repeated start routine.
326 ; Enter with address in ACC.
327
017A 759822 328 RepStart: MOV I2CON,#BCDR+BXSTR ;Send repeated start.
017D 309EFD 329 JNB ATN,$ ;Wait for ATN.
0180 759820 330 MOV I2CON,#BCDR ;Clear data ready.
0183 309EFD 331 JNB ATN,$ ;Wait for repeated
; start sent.

0189 22 333 RET
334
335
336 ; Bus fault recovery routine.
337
018A 31A4 338 Recover: ACALL FixBus ;See if bus is dead or
; can be 'fixed'.
018C 400D 339 JC BusReset ;If not 'fixed', try
; extreme measures.
018E D202 340 SETB Retry ;If bus OK, return to
; main routine.

0190 C201 341 CLR Fault
0192 C200 342 CLR NoAck
0194 D2DD 343 SETB CLR TI
0196 D2DC 344 SETB TIRUN ;Enable timer I.
0198 D2AB 345 SETB ETI ;Turn on timer I
; interrupts.

019A 22 346 RET
347
348 ;This routine tries a more extreme method of bus recovery.
349 ; This is used if SCL or SDA are stuck and cannot
; otherwise be freed.
350 ; (will return to the Recover routine when Timer I times out)
351
019B C2DE 352 BusReset: CLR MASTRQ ;Release bus.
019D 7598BC 353 MOV I2CON,#0BCh ;Clear all I2C flags.
01A0 D2DC 354 SETB TIRUN
01A2 80FE 355 SJMP $ ;Wait for timer I
; timeout (this will re-
; set the I2C hardware).

356
357
358
359 ; This routine attempts to regain control of the I2C
; bus after a bus fault.

360 ; Returns carry clear if successful, carry set if failed.
361
01A4 C2DE 362 FixBus: CLR MastRQ ;Turn off I2C functions.
01A6 D3 363 SETB C
01A7 D280 364 SETB SCL ;Insure I/O port is not
; locking I2C.

01A9 D281 365 SETB SDA
01AB 308029 366 JNB SCL,FixBusEx ;If SCL is low, bus
; cannot be 'fixed'.

01AE 208113 367 JB SDA,RStop ;If SCL & SDA are high,
; force a stop.

01B1 752109 368 MOV BitCnt,#9 ;Set max # of tries to
; clear bus.

01B4 C280 369 ChekLoop: CLR SCL ;Force an I2C clock.
01B6 31D8 370 ACALL SDelay
01B8 208109 371 JB SDA,RStop ;Did it work?
01BB D280 372 SETB SCL
01BD 31D8 373 ACALL SDelay
01BF D521F2 374 DJNZ BitCnt,ChekLoop ;Repeat clocks until
; either SDA clears or
; we run out of tries.

375

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

01C2 8013      376          SJMP   FixBusEx      ;Failed to fix bus by
                                           ; this method.
                                           377
01C4 C281      378  RStop:   CLR     SDA          ;Try forcing a stop
                                           ; since SCL & SDA
01C6 31D8      379          ACALL  SDelay        ; are both high.
01C8 D280      380          SETB   SCL
01CA 31D8      381          ACALL  SDelay
01CC D281      382          SETB   SDA
01CE 31D8      383          ACALL  SDelay
01D0 308004    384          JNB    SCL,FixBusEx    ;Are SCL & SDA still
                                           ; high? If so, assume bus
01D3 308101    385          JNB    SDA,FixBusEx    ; is now OK, and return
01D6 C3        386          CLR     C              ; with carry cleared.
01D7 22        387  FixBusEx: RET
                                           388
                                           389
                                           390 ; Short delay routine (10 machine cycles).
                                           391
01D8 00        392  SDelay:  NOP
01D9 00        393          NOP
01DA 00        394          NOP
01DB 00        395          NOP
01DC 00        396          NOP
01DD 00        397          NOP
01DE 00        398          NOP
01DF 00        399          NOP
01E0 22        400          RET
                                           401
                                           402 ;*****
                                           403 ;
                                           404 ;           Main Program
                                           405 ;*****
                                           406
01E1 758107    407  Reset:   MOV     SP,#07h      ;Set stack location.
01E4 D2AB      408          SETB   ETI          ;Enable timer I interrupts.
01E6 D2AF      409          SETB   EA          ;Enable global interrupts.
01E8 75290B    410          MOV     XmtDat,#11   ;Set up transmit data.
01EB 752A16    411          MOV     XmtDat+1,#22  ;Set up transmit data.
01EE 752B2C    412          MOV     XmtDat+2,#44  ;Set up transmit data.
01F1 752C58    413          MOV     XmtDat+3,#88  ;Set up transmit data.
01F4 752500    414          MOV     RcvDat,#0     ;Clear receive data.
01F7 752600    415          MOV     RcvDat+1,#0    ;Clear receive data.
01FA 752700    416          MOV     RcvDat+2,#0    ;Clear receive data.
01FD 752800    417          MOV     RcvDat+3,#0    ;Clear receive data.
                                           418
0200 752348    419  MainLoop: MOV     SlvAdr,#48h  ;Set slave address
                                           ; (8-bit I/O port).
0203 752201    420          MOV     ByteCnt,#1    ;Set up byte count.
0206 1127      421          ACALL  SendData     ;Send data to slave.
0208 2002F5    422          JB     Retry,MainLoop
                                           423
020B 752201    424  ML2:    MOV     ByteCnt,#1    ;Set up byte count.
020E 114E      425          ACALL  RcvData     ;Read data from slave.
0210 2002F8    426          JB     Retry,ML2
                                           427
0213 7523A0    428  SL1:    MOV     SlvAdr,#0A0h   ;Set slave address
                                           ; (RAM chip).
0216 752400    429          MOV     SubAdr,#0h     ;Set slave subaddress.
0219 752204    430          MOV     ByteCnt,#4    ;Set up byte count.
021C 1188      431          ACALL  SendSub
021E 2002F2    432          JB     Retry,SL1
                                           433

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

0221 752204      434  SL2:      MOV      ByteCnt,#4      ;Set up byte count.
0224 11B9        435          ACALL   RcvSub
0226 2002F8      436          JB      Retry,SL2
                437
0229 0529        438          INC     XmtDat
022B 052A        439          INC     XmtDat+1
022D 052B        440          INC     XmtDat+2
022F 052C        441          INC     XmtDat+3
0231 80CD        442          SJMP   MainLoop      ;Do it all again.
                443
                444          ENDASSEMBLY COMPLETE, 0 ERRORS FOUND
    
```

I2CAPP		83C751 Single Master I2C Routines	
ACC.		D ADDR	00E0H PREDEFINED
ATN.		B ADDR	009EH PREDEFINED
BCARL.		NUMB	0010H
BCDR.		NUMB	0020H
BCSTP.		NUMB	0004H
BCSTR.		NUMB	0008H
BCXA.		NUMB	0080H
BIDLE.		NUMB	0040H NOT USED
BITCNT.		D ADDR	0021H
BMRQ.		NUMB	0040H
BTIR.		NUMB	0010H
BUSRESET.		C ADDR	019BH
BXSTP.		NUMB	0001H
BXSTR.		NUMB	0002H
BYTECNT.		D ADDR	0022H
CHEKLOOP.		C ADDR	01B4H
CLRINT.		C ADDR	0026H
CLRTI.		B ADDR	00DDH PREDEFINED
CTVAL.		NUMB	0002H
DRDY.		B ADDR	009DH PREDEFINED
EA.		B ADDR	00AFH PREDEFINED
ETI.		B ADDR	00ABH PREDEFINED
FAULT.		B ADDR	0001H
FIXBUS.		C ADDR	01A4H
FIXBUSEX.		C ADDR	01D7H
FLAGS.		D ADDR	0020H
I2CFG.		D ADDR	00D8H PREDEFINED
I2CON.		D ADDR	0098H PREDEFINED
I2DAT.		D ADDR	0099H PREDEFINED
MAINLOOP.		C ADDR	0200H
MASTER.		B ADDR	0099H PREDEFINED
MASTRQ.		B ADDR	00DEH PREDEFINED
ML2.		C ADDR	020BH
NOACK.		B ADDR	0000H
P0.		D ADDR	0080H PREDEFINED
RBIT.		C ADDR	0153H
RCVBYTE.		C ADDR	014FH
RCVDAT.		D ADDR	0025H
RCVDATA.		C ADDR	004EH
RCVSUB.		C ADDR	00B9H
RDACK.		C ADDR	0143H
RDAT.		B ADDR	009FH PREDEFINED
RDATERR.		C ADDR	0086H
RDERR.		C ADDR	0163H
RDEX.		C ADDR	0083H
RDLAST.		C ADDR	0074H
RDLOOP.		C ADDR	006AH
RECOVER.		C ADDR	018AH
REPSTART.		C ADDR	017AH
RESET.		C ADDR	01E1H
RETRY.		B ADDR	0002H
RSEX.		C ADDR	0107H

Using the 8XC751 microcontroller as an I²C bus master

AN422

RSLAST	C ADDR	00F8H	
RSLOOP	C ADDR	00EEH	
RSTOP	C ADDR	01C4H	
RSUBERR	C ADDR	010AH	
SAERR	C ADDR	011DH	
SCL	B ADDR	0080H	
SDA	B ADDR	0081H	
SDATERR	C ADDR	004CH	
SDELAY	C ADDR	01D8H	
SDEX	C ADDR	0049H	
SDLOOP	C ADDR	003CH	
SENDAD2	C ADDR	0115H	
SENDADDR	C ADDR	010CH	
SENDDATA	C ADDR	0027H	
SENDSTOP	C ADDR	0166H	
SENDSUB	C ADDR	0088H	
SL1	C ADDR	0213H	
SL2	C ADDR	0221H	
SLVADR	D ADDR	0023H	
SP	D ADDR	0081H	PREDEFINED
SSEX	C ADDR	00B4H	
SSLOOP	C ADDR	00A7H	
SSUBERR	C ADDR	00B7H	
STACKSAVE	D ADDR	002DH	
SUBADR	D ADDR	0024H	
TIMERI	C ADDR	001BH	NOT USED
TIRUN	B ADDR	00DCH	PREDEFINED
XMBIT	C ADDR	0128H	
XMBIT2	C ADDR	012AH	
XMBX	C ADDR	013FH	
XMERR	C ADDR	0140H	
XMITADDR	C ADDR	0120H	
XMITBYTE	C ADDR	0125H	
XMTDAT	D ADDR	0029H	

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

DESCRIPTION

This application note shows how to use the PCD8584 I²C-bus controller with 80C51 family microcontrollers. One typical way of connecting the PCD8584 to an 80C31 is shown. Some basic software routines are described showing how to transmit and receive bytes in a single master system. An example is given of how to use these routines in an application that makes use of the I²C circuits on an I²C demonstration board.

The PCD8584 is used to interface between parallel microprocessor or microcontroller buses and the serial I²C bus. For a description of the I²C bus protocol refer to the I²C bus specification which is printed in the microcontroller user guide.

The PCD8584 controls the transmission and reception of data on the I²C bus, arbitration, clock speeds and transmission and reception of data on the parallel bus. The parallel bus is compatible with 80C51, 68000, 8085 and Z80 buses. Communication with the I²C-bus can be done on an interrupt or polled basis. This application note focuses on interfacing with 8051 microcontrollers in single master systems.

PCD8584

In Figure 1, a block diagram is shown of the PCD8584. Basically it consists of an I²C-interface similar to the one used in 84Cxx family microcontrollers, and a control block for interfacing to the microcontroller.

The control block can automatically determine whether the control signals are from 80xx or 68xxx type of microcontrollers.

This is determined after the first write action from the microcontroller to the PCD-8584. The control block also contains a programmable divider which allows the selection of different PCD8584 and I²C clocks.

The I²C interface contains several registers which can be written and read by the microcontroller.

S1 is the control/status register. This register is accessed while the A0 input is 1. The meaning of the bits depends on whether the register is written to or read from. When used

as a single master system the following bits are important:

PIN: Interrupt bit. This bit is made active when a byte is sent/received to/from the I²C-bus. When ENI is made active, PIN also controls the external INT line to interrupt the microcontroller.

ES0-ES2: These bits are used as pointer for addressing S0, S0', S2 and S3. Setting ES0 also enables the Serial I/O.

ENI: Enable Interrupt bit. Setting this bit enables the generation of interrupts on the INT line.

STA, STO: These bits allow the generation of START or STOP conditions.

ACK: With this bit set and the PCD8584 is in master/receiver mode, no acknowledge is generated by the PCD8584. The slave/transmitter now knows that no more data must be sent to the I²C-bus.

BER: This bit may be read to check if bus errors have occurred.

BB: This bit may be read to check whether the bus is free for I²C-bus transmission.

S2 is the clock register. It is addressed when A0 = 0 and ES0-ES2 = 010 in the previous write cycle to S1. With the bits S24-S20 it is possible to select 5 input clock frequencies and 4 I²C clock frequencies.

S3 is the interrupt vector register. It is addressed when A0 = 0 and ES0-ES2 = 001 in the previous write cycle to S1. This register is not used when an 80C51 family microcontroller is used. An 80C51 microcontroller has fixed interrupt vector addresses.

S0' is the own address register. It is addressed when A0 = 0 and ES0-ES2 = 000. This register contains the slave address of the PCD8584. In the single master system described here, this register has no functional use. However, by writing a value to S0', the PCD8584 determines whether an 80Cxx or 68xxx type microcontroller is the controlling microcontroller by looking at the CS and WR lines. So independent of whether the PCD8584 is used as master or slave, the

microcontroller should always first write a value to S0' after reset.

S0 is the I²C data register. It is addressed when A0 = 0 and ES0-ES2 = 1x0. Transmission of a byte on the I²C bus is done by writing this byte to S0. When the transmission is finished, the PIN bit in S1 is reset and if ENI is set, an interrupt will be generated. Reception of a byte is signaled by resetting PIN and by generating an interrupt if ENI is set. The received byte can be read from S0.

The SDA and SCL lines have no protection diodes to V_{DD}. This is important for multimaster systems. A system with a PCD8584 can now be switched off without causing the I²C-bus to hang-up. Other masters still can use the bus.

For more information of the PCD8584 refer to the data sheet.

PCD8584/8031 Hardware Interface

Figure 2 shows a minimum system with an 8051 family controller and a PCD8584. In this example, an 80C31 is used. However any 80C51 family controller with external addressing capability can be used.

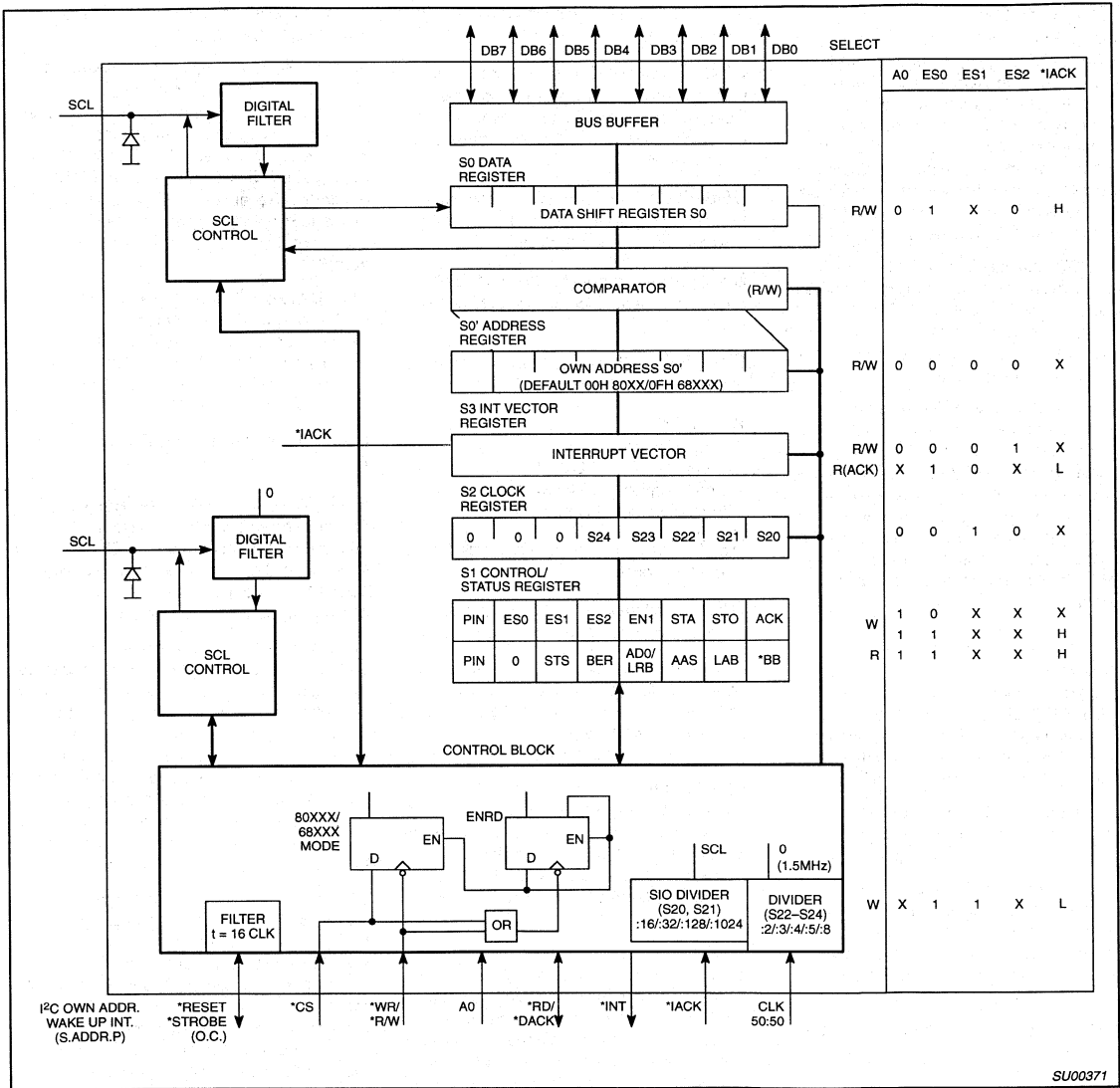
The software resides in EPROM U3. For addressing this device, latch U2 is necessary to demultiplex the lower address bits from the data bits. The PCD8584 is mapped in the external data memory area. It is selected when A1 = 0. Because in this example no external RAM or other mapped peripherals are used, no extra address decoding components are necessary. A0 is used by the PCD8584 for proper register selection in the PCD8584.

U5A is an inverter with Schmitt trigger input and is used to buffer the oscillator signal of the microcontroller. Without buffering, the rise and fall time specifications of the CLK signal are not met. It is also important that the CLK signal has a duty cycle of 50%. If this is not possible with certain resonators or microcontrollers, then an extra flip-flop may be necessary to obtain the correct duty cycle.

U5C and U5D are used to generate the proper reset signals for the microcontroller and the PCD8584.

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

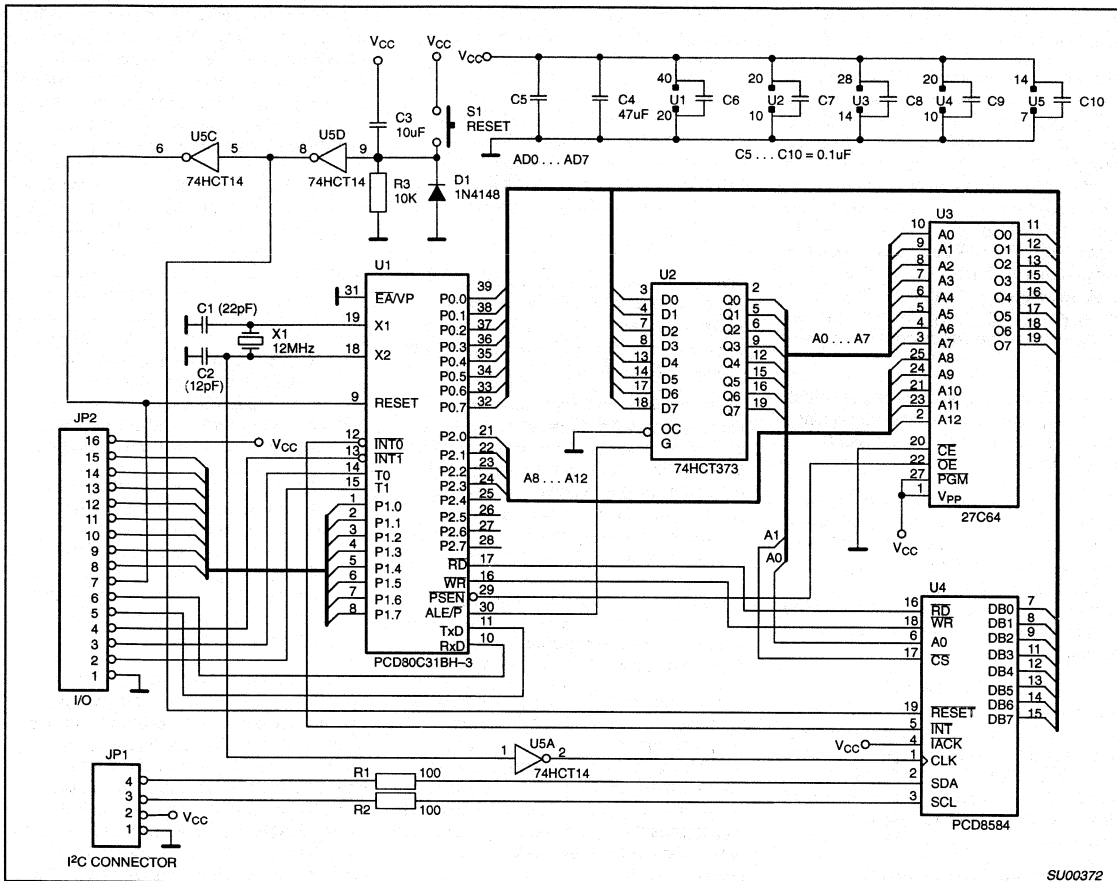


SU00371

Figure 1. PCD8584 Block Diagram

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425



SU00372

Figure 2. PCD8584 to 80C31 Interface

Basic PCD8584/8031 Driver Routines

In the listing section (page 89), some basic routines are shown. The routines are divided in two modules. The module ROUTINE contains the driver routines and initialization of the PCD8584. The module INTERR contains the interrupt handler. These modules may be linked to a module with the user program that uses the routines in INTERR and ROUTINE. In this application note, this module will be called USER. A description of ROUTINE and INTERR follows.

Module ROUTINE

Routine Sendbyte (Lines 17-20)—

This routine sends the contents of the accumulator to the PCD8584. The address is such that A0 = 0. Which register is accessed

depends on the contents of ES0-ES2 of the control register. The address of the PCD8584 is in variable 'PCD8584'. This must have been previously defined in the user program. The DPTR is used as a pointer for addressing the peripheral. If the address is less than 255, then R0 or R1 may be used as the address pointer.

Routine Sendcontr (Lines 25, 26)—

This routine is similar to Sendbyte, except that now A0 = 1. This means that the contents of the accumulator are sent to the control register S1 in the PCD8584.

Routine Readbyte (Lines 30-33)—

This routine reads a register in the PCD8584 with A0 = 0. Which register depends on ES0-ES2 of the control register. The result of the read operation is returned in the accumulator.

Routine Readcontr (Lines 37-39)—

This routine is similar to Readbyte, except that now A0 = 1. This means that the accumulator will contain the value of status register S1 of the PCD8584.

Routine Start Lines (44-56)—

This routine generates a START-condition and the slave address with a R/W bit. In line 44, the variable IIC_CNT is reset. This variable is used as a byte counter to keep track of the number of bytes that are received or transmitted. IIC_CNT is defined in module INTERR.

Lines 45-46 increment the variable NR_BYTES if the PCD8584 must receive data. NR_BYTES is a variable that indicates how many bytes have to be received or transmitted. It must be given the correct value in the USER module. Receiving or transmitting is distinguished by the value of

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

the DIR bit. This must also be given the correct value in the USER module.

Then the status register of PCD8584 must be read to check if the I²C bus is free. First the status register must be addressed by giving ES0-ES2 of the control register the correct value (lines 47-48). Then the Bus Busy bit is tested until the bus is free (lines 49-50). If this is the case, the slave address is sent to data register S0 and the I2C_END bit is cleared (lines 51-53). The slave address is set by the user program in variable USER. The LSB of the slave address is the R/W bit. I2C_END can be tested by the user program whether an I2C reception/transmission is in progress or not.

Next the START condition will be generated and interrupt generation enabled by setting the appropriate bits in control register S1. (lines 54-55).

Now the routine will return back to the user program and other tasks may be performed. When the START condition, slave address and R/W bit are sent, and the ACK is received, the PCD8584 will generate an interrupt. The interrupt routine will determine if more bytes have to be received or transmitted.

Routine Stop (Lines 59-62) —

Calling this routine, a STOP condition will be sent to the I²C bus. This is done by sending the correct value to control register S1 (lines 59-61). After this the I2C_END bit is set, to indicate to the user program that a complete I²C sequence has been received or transmitted.

Routine I2C_Init (Lines 65-76)—

This routine initializes the PCD8584. This must be done directly after reset. Lines 67-70 write data to 'own address' register S0'. First the correct address of S0' is set in control register S1 (lines 67-68), then the correct value is written to it (lines 69-70). The value for S0' is in variable SLAVE_ADR and set by the user program. As noted previously, register S0' must always be the first register to be accessed after reset, because the PCD8584 now determines whether an 80Cxxx or 68xxx microcontroller is connected. Lines 72-76 set the clock register S2. The variable I2C_CLOCK is also set by the user program.

Module INTERR

This module contains the I²C interrupt routine. This routine is called every time a byte is received or transmitted on the I²C bus. In lines 12-15 RAM space for variables is reserved.

BASE is the start address in the internal

80C51 RAM where the data is stored that is received, or where the data is stored that has to be transmitted.

NR_BYTES, IIC_CNT and SLAVE were explained earlier. I2C_END and DIR are flags that are used in the program. I2C_END indicates whether an I²C transmission or reception is in progress. DIR indicates whether the PCD8584 has to receive or transmit bytes. The interrupt routine makes use of register bank 1.

The transmission part of the routine starts at line 42. In lines 42-43, a check is made whether IIC_CNT = NR_BYTES. If true, all bytes are sent and a STOP condition may be generated (lines 44-45).

Next the pointer for the internal RAM is restored (line 46) and the byte to be transmitted is fetched from the internal RAM (line 47). Then this byte is sent to the PCD8584 and the variables are updated (lines 47-49). The interrupt routine is left and the user program may proceed. The receive part starts from line 55. First a check is made if the next byte to be received is the last byte (lines 56-59). If true the ACK must be disabled when the last byte is received. This is accomplished by resetting the ACK bit in the control register S1 (lines 60-61).

Next the received byte may be read (line 62) from data register S0. The byte will be temporary stored in R4 (line 63). Then a check is made if this interrupt was the first after a START condition. If so, the byte read has no meaning and the interrupt routine will be left (lines 68-70). However by reading the data register S0 the next read cycle is started.

If valid data is received, it will be stored in the internal RAM addressed by the value of BASE (lines 71-73). Finally a check is made if all bytes are received. If true, a STOP condition will be sent (lines 75-78).

EXAMPLES

In the listing section (starting on page 8), some examples are shown that make use of the routines described before. The examples are transmission of a sequence, reception of I²C data and an example that combines both.

The first example sends bytes to the PCD8577 LCD driver on the OM1016 demonstration board. Lines 7 to 10 define the interface with the other modules and should be included in every user program. Lines 14 to 16 define the segments in the user module. It is completely up to the user how to organize this.

Lines 24 and 28 are the reset and interrupt vectors. The actual user program starts at

line 33. Here three variables are defined that are used in the I²C driver routines. Note that PCD8584 must be an even address, otherwise the wrong internal registers will be accessed! Lines 37-42 initialize the interrupt logic of the microcontroller. Next the PCD8584 will be initialized (line 45).

The PCD8584 is now ready to transmit data. A table is made in the routine at line 61. For the PCD8577, the data is a control byte and the segment data. Note that the table does not contain the slave address of the LCD driver. In lines 51-54, variables are made ready to start the transmission. This consists of defining the direction of the transmission (DIR), the address where the data table starts (BASE), the number of bytes to transmit (NR_BYTES, without slave address!) and the slave address (SLAVE) of the I²C peripheral that has to be accessed.

In line 55 the transmission is started. Once the I²C transmission is started, the user program can do other tasks because the transmission works on interrupts. In this example a loop is performed (line 58). The user can check the end of the transmission during the other tasks, by testing the I2C_END bit regularly.

The second example program receives 2 bytes from the PCF8574P I/O expander on the OM1016 demonstration board. Until line 45 the program is identical to the transmit routine because it consists of initialization and variable definition. From line 48, the variables are set for I²C reception. The received bytes are stored in RAM area from label TABLE. During reception, the user program can do other tasks. By testing the I2C_END bit the user can determine when to start processing the data in the TABLE.

The third example program displays time from the PCF8583P clock/calendar/RAM on the LCD display driven by the PCF8577. The LED display (driven by SAA1064) shows the value of the analog inputs of the A/D converter PCF8591. The four analog inputs are scanned consecutively.

In this example, both transmit and receive sequences are implemented as shown in the previous examples. The main clock part is from lines 62-128. This contains the calls to the I²C routines. From lines 135-160, routines are shown that prepare the data to be transmitted. Lines 171 to 232 are the main program for the AD converter and LED display. Lines 239 to 340 contain routines used by the main program. This demo program can also be used with the I²C peripherals on the OM1016 demonstration board.

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Routines for PCD8584

```

LOC  OBJ          LINE  SOURCE
                                1  $TITLE (Routines for PCD8584)
                                2  $PAGELENGTH(40)
                                3  ;Program written for PCD8584 as master
                                4  ;
                                5          PUBLIC READBYTE,READCONTR,SENDBYTE
                                6          PUBLIC SENDCONTR,START,STOP
                                7          PUBLIC I2C_INIT
                                8          EXTRN  BIT(I2C_END,DIR)
                                9          EXTRN  DATA(SLAVE,IIC_CNT,NR_BYTES)
                               10          EXTRN  NUMBER(SLAVE_ADR,I2C_CLOCK,PCD8584)
                               11  ;
                               12  ;Define code segment
                               13  ROUTINE  SEGMENT CODE
----- 13          RSEG    ROUTINE
                               14  ;
                               15  ;SENDBYTE sends a byte to PCD8584 with A0=0
                               16  ;Byte to be send must be in accu
0000:   R          17  SENDBYTE:
0000: 900000 R          18          MOV DPTR,#PCD8584 ;Register address
0003: F0          19  SEND:  MOVX @DPTR,A    ;Send byte
0004: 22          20          RET
                               21  ;
                               22  ;SENDCONTR sends a byte to PCD8584 with A0=1
                               23  ;Byte to be send must be in accu
0005:   R          24  SENDCONTR:
0005: 900001 R          25          MOV DPTR,#PCD8584+01H ;Register address
0008: 80F9          26          JMP SEND
                               27  ;
                               28  ;READBYTE reads a byte from PCD8584 with A0=0
                               29  ;Received byte is stored in accu
000A:   R          30  READBYTE:
000A: 900000 R          31          MOV DPTR,#PCD8584 ;Register address
000D: E0          32  REC:  MOVX A,@DPTR    ;Receive byte
000E: 22          33          RET
                               34  ;
                               35  ;READCONTR reads a byte from PCD8584 with A0=1
                               36  ;Received byte is stored in accu
000F:   R          37  READCONTR:
000F: 900001 R          38          MOV DPTR,#PCD8584+01H ;Register address
0012: 80F9          39          JMP REC
                               40  ;
                               41  ;START tests if the I2C bus is ready. If ready a
                               42  ;START-condition will be sent, interrupt generation
                               43  ;and acknowledge will be enabled.
0014: 750000 R          44  START: MOV IIC_CNT,#00 ;Clear I2C byte counter
0017: 200002 R          45          JB DIR,PROCEED ;If DIR is 'receive' then
001A: 0500 R          46          INC NR_BYTES ;increment NR_BYTES
001C: 7440          47  PROCEED:MOV A,#40H    ; Read STATUS register of
                               48          ; 8584
001E: 120005 R          48          CALL SENDCONTR
0021: 12000F R          49  TESTBB: CALL READCONTR
0024: 30E0FA          50          JNB ACC.0,TESTBB; Test BB/ bit
0027: E500 R          51          MOV A,SLAVE
0029: C200 R          52          CLR I2C_END ;Reset I2C ready bit
002B: 120000 R          53          CALL SENDBYTE ;Send slave address
002E: 744D          54          MOV A,#01001101B;Generate START, set ENI,
                               55          ;set ACK
0030: 120005 R          55          CALL SENDCONTR
0033: 22          56          RET
                               57  ;
                               58  ;STOP will generate a STOP condition and set the
                               59  ;I2C_END bit
0034: 74C3          59  STOP:  MOV A,#11000011B

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```
0036: 120005 R 60 CALL SENDCONTR ;Send STOP condition
0039: D200 R 61 SETB I2C_END ;Set I2C_END bit
003B: 22 62 RET
63 ;
64 ;I2C_init does the initialization of the PCD8584
003C: 65 I2C_INIT:
66 ;Write own slave address
003C: E4 67 CLR A
003D: 120005 R 68 CALL SENDCONTR ;Write to control register
0040: 7400 R 69 MOV A,#SLAVE_ADR
0042: 120000 R 70 CALL SENDBYTE ;Write to own slave
;register
71 ;Write clock register
0045: 7420 72 MOV A,#20H
0047: 120005 R 73 CALL SENDCONTR ;Write to control register
004A: 7400 R 74 MOV A,#I2C_CLOCK
004C: 120000 R 75 CALL SENDBYTE ;Write to clock register
004F: 22 76 RET
77 ;
0050: 78 END
```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

ASM51  TSW  ASSEMBLER  I2C INTERRUPT ROUTINE

LOC  OBJ          LINE  SOURCE

      1  $TITLE (I2C INTERRUPT ROUTINE)
      2  $PAGELENGTH(40)
      3  ;
      4          PUBLIC INT0_SRV
      5          PUBLIC DIR,I2C_END
      6          PUBLIC BASE,NR_BYTES,IIC_CNT,SLAVE
      7          EXTRN  CODE(SENDBYTE,SENDCONTR,STOP)
      8          EXTRN  CODE(READBYTE,READCONTR)
      9  ;
     10  ;Define variables in RAM
     11  IIC_VAR SEGMENT DATA
-----
     12  R  BASE:  DS 1          ;Pointer to I2C table (till
                                ;256)
0001:          13  NR_BYTES: DS 1      ;Number of bytes to rcv/trm
0002:          14  IIC_CNT:DS 1      ;I2C byte counter
0003:          15  SLAVE:  DS 1      ;Slave address after START
                                ;
                                16  ;
                                17  ;Define variable segment
                                18  BIT_VAR SEGMENT DATA BITADDRESSABLE
-----
                                19  RSEG BIT_VAR
0000:          R  20  STATUS: DS 1      ;Byte with flags
0000          R  21  I2C_END BIT STATUS.0 ;Defines if a I2C
                                ;transmission is finished
                                ;'1' is finished
                                ;'0' is not ready
0000          R  24  DIR      BIT STATUS.3 ;Defines direction of I2C
                                ;transmission
                                ;'1':Transmit '0':Receive
                                25  ;
                                26  ;
                                27  ;Define code segment for routine
                                28  IIC_INT SEGMENT CODE PAGE
-----
                                29  RSEG IIC_INT
                                30  ;
                                31  ;Program uses registers in R1
                                32  USING 1
                                33  ;
0000:          R  34  INT0_SRV:
0000: C0E0          35          PUSH ACC      ;Save acc. en psw on stack
0002: C0D0          36          PUSH PSW
0004: 75D008        37          MOV PSW,#08H      ;Select register bank 1
0007: 300016        R  38          JNB DIR,RECEIVE ;Test direction bit
                                39          ;8584 is MST/TRM
                                40
                                41  ;Program part to transmit bytes to IIC bus
000A: E502          R  42          MOV A,IIC_CNT      ;Compare IIC_CNT and
                                ;NR_BYTES
000C: B50105        R  43          CJNE A,NR_BYTES,PROCEED
000F: 120000        R  44          CALL STOP      ;All bytes transmitted
0012: 8032          45          JMP EXIT
0014: A800          R  46  PROCEED:MOV R0,BASE      ;RAM pointer
0016: E6           47          MOV A,@R0      ;Source is internal RAM
0017: 0500          R  48          INC BASE      ;Update pointer of table
0019: 120000        R  49          CALL SENDBYTE      ;Send byte to IIC bus
001C: 0502          R  50          INC IIC_CNT      ;Update byte counter
001E: 8026          51          JMP EXIT
                                52  ;
                                53  ;
                                54  ;Program to receive byte from IIC bus
0020:          55  RECEIVE:
0020: E502          R  56          MOV A,IIC_CNT      ;Test if last byte is to be
                                ;received
0022: 04           57          INC A

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

0023: 04          58          INC A
0024: B50105     R          59          CJNE A, NR_BYTES, PROC_RD
0027: 7448          60          MOV A, #01001000B; Last byte to be received.
                                ;Disable ACK
0029: 120000     R          61          CALL SENDCONTR ;Write control word to
                                ;PCD8584
002C: 120000     R          62          PROC_RD: CALL READBYTE ;Read I2C byte
002F: FC          63          MOV R4, A      ;Save accu
                                ;If RECEIVE is entered after the transmission of
64          ;START+address then the result of READBYTE is not
65          ;relevant. READBYTE is used to start the generation
66          ;of the clock pulses for the next byte to read.
                                ;This situation occurs when IIC_CNT is 0
67          CLR A      ;Test IIC_CNT
0030: E4          68          CJNE A, IIC_CNT, SAVE
0031: B50202     R          69          JMP END_TEST  ;START is send. No relevant
0034: 8006          70          ;data in data reg. of 8584
0036: A800     R          71          SAVE:  MOV R0, BASE
0038: EC          72          MOV A, R4      ;Destination is internal RAM
0039: F6          73          MOV @R0, A
003A: 0500     R          74          INC BASE
003C: 0502     R          75          END_TEST: INC IIC_CNT ;Test if all bytes are
                                ;received
003E: E501     R          76          MOV A, NR_BYTES
0040: B50203     R          77          CJNE A, IIC_CNT, EXIT
0043: 120000     R          78          CALL STOP     ;All bytes received
                                ;
79          ;
0046: D0D0     R          80          EXIT:  POP PSW   ;Restore PSW and accu
0048: D0E0     R          81          POP ACC
004A: 32          82          RETI
83          ;
004B:          84          END

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Send a string of bytes to the PCF8577 on OM1016

```

LOC   OBJ          LINE SOURCE
      1 $TITLE (Send a string of bytes to the PCF8577 on
      2           OM1016)
      3 $PAGELENGTH(40)
      4 ;
      5 ;This program is an example to transmit bytes via
      6 ;PCD8584
      7 ;to the I2C-bus
      8 ;
      9 PUBLIC SLAVE_ADR, I2C_CLOCK, PCD8584
     10 EXTRN CODE(I2C_INIT, INT0_SRV, START)
     11 EXTRN BIT(I2C_END, DIR)
     12 EXTRN DATA(BASE, NR_BYTES, IIC_CNT, SLAVE)
     13 ;
     14 ;Define used segments
     15 USER SEGMENT CODE ;Segment for user program
     16 RAMTAB SEGMENT DATA ;Segment for table in
     17 ;internal RAM
     18 RAMVAR SEGMENT DATA ;Segment for RAM variables
     19 ;in RAM
     20 ;
     21 ;
     22 RSEG RAMVAR
     23 STACK: DS 20 ;Reserve stack area
     24 ;
     25 ;
     26 CSEG AT 00H
     27 JMP MAIN ;Reset vector
     28 ;
     29 ;
     30 CSEG AT 03H
     31 JMP INT0_SRV ;I2C interrupt vector
     32 ;(INT0/)
     33 ;
     34 RSEG USER
     35 ;Define I2C clock, own slave address and PCD8584
     36 ;hardware address
     37 SLAVE_ADR EQU 55H ;Own slave address is 55H
     38 I2C_CLOCK EQU 00011100B ;12.00MHz/90kHz
     39 PCD8584 EQU 0000H ;PCD8584 address with A0=0
     40 ;0000: 7581FF R 37 MAIN: MOV SP, #STACK-1 ;Initialize stack pointer
     41 ;Initialize 8031 interrupt registers for I2C
     42 ;interrupt
     43 SETB EX0 ;Enable interrupt INT0/
     44 SETB EA ;Set global enable
     45 SETB PX0 ;Priority level '1'
     46 SETB IT0 ;INT0/ on falling edge
     47 ;
     48 ;Initialize PCD8584
     49 CALL I2C_INIT
     50 ;
     51 ;Make a table in RAM with data to be transmitted.
     52 CALL MAKE_TAB
     53 ;
     54 ;Set variables to control PCD8584
     55 SETB DIR ;DIR='transmission'
     56 MOV BASE, #TABLE ;Start address of I2C-data
     57 MOV NR_BYTES, #05H ;5 bytes must be
     58 ;transferred
     59 MOV SLAVE, #01110100B ;Slave address PCF8577
     60 ; + WR/
     61 CALL START ;Start I2C transmission

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

56 ;
57 ;
001F: 80FE      58 LOOP:   JMP LOOP           ;Endless loop when program
                                   ;is finished
59 ;
60 ;
0021:          61 MAKE_TAB:
0021: 7800      R   62         MOV R0,#TABLE      ;Make data ready for I2C
                                   ;transmission
0023: 7600          63         MOV @R0,#00          ;Controlword PCF8577
0025: 08          64         INC R0
0026: 76FC          65         MOV @R0,#0FCH      ;'0'
0028: 08          66         INC R0
0029: 7660          67         MOV @R0,#60H      ;'1'
002B: 08          68         INC R0
002C: 76DA          69         MOV @R0,#0DAH      ;'2'
002E: 08          70         INC R0
002F: 76F2          71         MOV @R0,#0F2H      ;'3'
0031: 22          72         RET
73 ;
74 ;
----          75         RSEG RAMTAB
0000:          R   76 TABLE: DS 10      ;Reserve space in internal
                                   ;data RAM
77         ;for I2C data to transmit
78 ;
79 ;
000A:          80         END

```


Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

ASM51  TSW  ASSEMBLER  Receive 2 bytes from the PCF8574P on OM1016

LOC   OBJ          LINE  SOURCE
      1  $TITLE (Receive 2 bytes from the PCF8574P on OM1016)
      2  $PAGELENGTH(40)
      3  ;
      4  ;This program is an example to receive bytes via
      5  ;PCD8584
      6  ;from the I2C-bus
      7  ;
      8      PUBLIC  SLAVE_ADR,I2C_CLOCK,PCD8584
      9      EXTRN   CODE(I2C_INIT,INT0_SRV,START)
     10      EXTRN   BIT(I2C_END,DIR)
     11      EXTRN   DATA(BASE,NR_BYTES,IIC_CNT,SLAVE)
     12  ;
     13  ;Define used segments
     14  USER   SEGMENT CODE      ;Segment for user program
     15  RAMTAB SEGMENT DATA     ;Segment for table in
     16  RAMVAR SEGMENT DATA     ;Segment for RAM variables
     17  ;
     18  ;
----
     19      RSEG   RAMVAR
0000:      R    20  STACK:  DS 20      ;Reserve stack area
     21  ;
     22  ;
----
     23      CSEG AT 00H
0000: 020000  R    24      JMP MAIN      ;Reset vector
     25  ;
     26  ;
----
     27      CSEG AT 03H
0003: 020000  R    28      JMP INT0_SRV    ;I2C interrupt vector
     29  ;
     30  ;
----
     31      RSEG USER
     32  ;Define I2C clock, own slave address and PCD8584
     33  ;hardware address
0055      33  SLAVE_ADR EQU 55H      ;Own slave address is 55H
001C      34  I2C_CLOCK EQU 00011100B ;12.00MHz/90kHz
0000      35  PCD8584 EQU 0000H      ;PCD8584 address with A0=0
     36  ;0000: 7581FF  R    37  MAIN:  MOV SP,#STACK-1 ;Initialize stack pointer
     38  ;Initialize 8031 interrupt registers for I2C
     39  ;interrupt
0003: D2A8      39      SETB EX0      ;Enable interrupt INT0/
0005: D2AF      40      SETB EA      ;Set global enable
0007: D2B8      41      SETB PX0      ;Priority level '1'
0009: D288      42      SETB IT0      ;INT0/ on falling edge
     43  ;
     44  ;Initialize PCD8584
000B: 120000  R    45      CALL I2C_INIT
     46  ;
     47  ;Set variables to control PCD8584
000E: C200      R    48      CLR  DIR      ;DIR='receive'
0010: 750000  R    49      MOV  BASE,#TABLE ;Start address of I2C-data
0013: 750002  R    50      MOV  NR_BYTES,#02H ;2 bytes must be received
0016: 75004F  R    51      MOV  SLAVE,#01001111B ;Slave address PCF8574
     52  ; + RD
0019: 120000  R    52      CALL START    ;Start I2C transmission
     53  ;
     54  ;
001C: 80FE      55  LOOP:  JMP LOOP    ;Endless loop when program
     55  ;is finished
     56  ;

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```
-----          57 ;
0000:          R  58      RSEG RAMTAB
          59 TABLE: DS 10      ;Reserve space in internal
          60                          ;data RAM
          61                          ;for received I2C data
          62 ;
000A:          63      END
```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

```

LOC   OBJ           LINE  SOURCE
-----
      1  $TITLE (Demo program for PCD8584 I2C-routines)
      2  $PAGELENGTH(40)
      3  ;Program displays on the LCD display the time (with
      4  ;PCF8583). Dots on LCD display blink every second.
      5  ;On the LED display the values of the successive
      6  ;analog input channels are shown.
      7  ;Program reads analog channels of PCF8591P.
      8  ;Channel number and channel value are displayed
      9  ;successively.
     10  ;Values are displayed on LCD and LED display on I2C
     11  ;demo board.
     12  ;
     13  PUBLIC      SLAVE_ADR,I2C_CLOCK,PCD8584
     14  EXTRN      CODE(I2C_INIT,INT0_SRV,START)
     15  EXTRN      BIT(I2C_END,DIR)
     16  EXTRN      DATA(BASE,NR_BYTES,IIC_CNT,SLAVE)
     17  ;
     18  ;Define used segments
     19  USER        SEGMENT  CODE      ;Segment for user program
     20  RAMTAB     SEGMENT  DATA     ;Segment for table in
     21  ;                                     ;internal RAM
     22  RAMVAR     SEGMENT  DATA     ;Segment for variables
     23  ;
     24  RSEG RAMVAR
-----
0000:      R      24  STACK:  DS 20          ;Stack area (20 bytes)
0014:      25  PREVIOUS: DS 1          ;Store for previous seconds
0015:      26  CHANNEL:DS 1          ;Channel number to be
                                ;sampled
0016:      27  AN_VAL:  DS 1          ;Analog value sampled
                                ;channel
0017:      28  CONVAL:  DS 3          ;Converted BCD value sampled
                                ;channel
-----
0000: 020000  R      29          CSEG AT 00H
      30  LUMP MAIN          ;Reset vector
-----
0003: 020000  R      31  ;
      32          CSEG AT 03H      ;INT0/
      33  LUMP INT0_SRV      ;Vector I2C-interrupt
      34  ;
      35  ;
-----
      36  RSEG USER37      ;Define I2C clock, own slave address and address for
                                ;main processor
0055      38  SLAVE_ADR EQU 55H          ;Own slaveaddress is 55h
001C      39  I2C_CLOCK EQU 00011100B ;12.00MHz/90kHz
0000      40  PCD8584 EQU 0000H          ;Address of PCD8584. This
                                ;must be an EVEN number!!
00A3      41  ;Define addresses of I2C peripherals
      42  PCF8583R EQU 10100011B ;Address PCF8583 with Read
                                ;active
00A2      43  PCF8583W EQU 10100010B ;Address PCF8583 with Write
                                ;active
009F      44  PCF8591R EQU 10011111B ;Address PCF8591 with Read
                                ;active
009E      45  PCF8591W EQU 10011110B ;Address PCF8591 with Write
                                ;active
0074      46  PCF8577W EQU 01110100B ;Address PCF8577 with Write
                                ;active
0076      47  SAA1064W EQU 01110110B ;Address SAA1064 with Write
                                ;active
      48  ;
0000: 7581FF  R      49  MAIN:  MOV SP,#STACK-1 ;Define stack pointer

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

```

LOC   OBJ           LINE   SOURCE
                                50   ;Initialize 80C31 interrupt registers for I2C
                                ;interrupt (INT0/)
0003: D2A8          51       SETB EX0           ;Enable interrupt INT0/
0005: D2AF          52       SETB EA           ;Set global enable
0007: D2B8          53       SETB PX0         ;Priority level is '1'
0009: D288          54       SETB IT0         ;INT0/ on falling edge
                                55   ;Initialize PCD8584
000B: 120000        R   56       CALL I2C_INIT
                                57   ;
000E: 751500        R   58       MOV CHANNEL,#00 ;Set AD-channel
                                59   ;
                                60   ;Time must be read from PCD8583.
                                61   ;First write word address and control register of
                                ;PCD8583.
0011: D200          R   62       SETB DIR           ;DIR='transmission'
0013: 750000        R   63       MOV BASE,#TABLE ;Start address I2C data
0016: 750002        R   64       MOV NR_BYTES,#02H ;Send 2 bytes
0019: 7500A2        R   65       MOV SLAVE,#PCF8583W
001C: E4            66       CLR A
001D: F500          R   67       MOV TABLE,A     ;Data to be sent (word
                                ;address).
001F: F501          R   68       MOV TABLE+1,A   ; " " (control
                                ;byte)
0021: 120000        R   69       CALL START       ;Start transmission.
0024: 3000FD        R   70   FIN_1: JNB I2C_END,FIN_1 ;Wait till transmission
                                ;finished
                                71   ;Send word address before reading time
0027: D200          R   72   REPEAT: SETB DIR ;'transmission
0029: 750000        R   73       MOV BASE,#TABLE ;I2C data
002C: 7500A2        R   74       MOV SLAVE,#PCF8583W
002F: 7401          75       MOV A,#01
0031: F500          R   76       MOV NR_BYTES,A   ;Send 1 byte
0033: F500          R   77       MOV TABLE,A     ;Data to be sent is '1'
0035: 120000        R   78       CALL START       ;Start I2C transmission
0038: 3000FD        R   79   FIN_2: JNB I2C_END,FIN_2 ;Wait till transmission
                                ;finished
                                80   ;
                                81   ;Time can now be read from PCD8583. Data read is
                                82   ;hundredths of sec's, sec's, min's and hr's
003B: C200          R   83       CLR DIR           ;DIR='receive'
003D: 750000        R   84       MOV BASE,#TABLE ;I2C table
0040: 750004        R   85       MOV NR_BYTES,#04; 4 bytes to receive
0043: 7500A3        R   86       MOV SLAVE,#PCF8583R
0046: 120000        R   87       CALL START       ;Start I2C reception
0049: 3000FD        R   88   FIN_3: JNB I2C_END,FIN_3 ;Wait till finished
                                89   ;
                                90   ;Transfer data to R2...R5
004C: 7800          R   91       MOV R0,#TABLE   ;Set pointers
004E: 7902          92       MOV R1,#02H     ;Pointer R2
0050: E6            93   TRANSFER:MOV A,@R0
0051: F7            94       MOV @R1,A
0052: 08            95       INC R0
0053: 09            96       INC R1
0054: D500F9        R   97       DJNZ NR_BYTES,TRANSFER
0057: ED            98       MOV A,R5        ;Mask of hour counter
0058: 543F          99       ANL A,#3FH
005A: FD           100      MOV R5,A
                                101   ;
                                102   ;Data must now be displayed on LCD display.
                                103   ;First minutes and hours (in R4 and R5) must be
                                104   ;converted from BCD to LCD segment data.The segment
                                ;data
                                105   ;will be transferred to TABLE. R0 is pointer to table

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

LOC OBJ LINE SOURCE
005B: 7800 R 106 MOV R0,#TABLE
005D: 7600 107 MOV @R0,#00H ;Control word for PCF8577
005F: 08 108 INC R0 0060: 120080 R 109 CALL CONV
110 ;
111 ;Switch on dp between hours and minutes
0063: 430301 R 112 ORL TABLE+3,#01H
113 ;If lsb of seconds is '0' then switch on dp.
0066: EB 114 MOV A,R3 ;Get seconds
0067: 13 115 RRC A ;lsb in carry
0068: 4003 116 JC PROCEED
006A: 430101 R 117 ORL TABLE+1,#01H;switch on dp
118 ;
119 ;Now the time (hours,minutes) can be displayed on
;the LCD
006D: 120 PROCEED:
006D: D200 R 121 SETB DIR ;Direction 'transmit'
006F: 750000 R 122 MOV BASE,#TABLE
0072: 750005 R 123 MOV NR_BYTES,#05H
0075: 750074 R 124 MOV SLAVE,#PCF8577W
0078: 120000 R 125 CALL START ;Start transmission
126 ;
007B: 3000FD R 127 FIN_4: JNB I2C_END,FIN_4
007E: 8026 128 JMP ADCON ;Proceed with AD-conversion
;part
129 ;
130 ;*****
131 ;Routines used by clock part of demo
132 ;
133 ;CONV converts hour and minute data to LCD data and
;stores
134 ;it in TABLE.
0080: 90009C R 135 CONV: MOV DPTR,#LCD_TAB ;Base for LCD segment
;table
0083: ED 136 MOV A,R5 ;Hours to accu
0084: C4 137 SWAP A ;Swap nibbles
0085: 120096 R 138 CALL LCD_DATA ;Convert 10's hours to LCD
;data in table
0088: ED 139 MOV A,R5 ;Get hours
0089: 120096 R 140 CALL LCD_DATA
008C: EC 141 MOV A,R4 ;Get minutes
008D: C4 142 SWAP A
008E: 120096 R 143 CALL LCD_DATA ;Convert 10's minutes
0091: EC 144 MOV A,R4
0092: 120096 R 145 CALL LCD_DATA ;Convert minutes
0095: 22 146 RET
147 ;
148 ;LCD_DATA gets data from segment table and stores it
;in TABLE
0096: 540F 149 LCD_DATA:ANL A,#0FH ;Mask off LS-nibble
0098: 93 150 MOVC A,@A+DPTR ;Get LCD segment data
0099: F6 151 MOV @R0,A ;Save data in table
009A: 08 152 INC R0
009B: 22 153 RET
154 ;
155 ;LCD_TAB is conversion table for LCD
009C: 156 LCD_TAB:
009C: FC60DA 157 DB 0FCH,60H,0DAH; '0','1','2'
009F: F266B6 158 DB 0F2H,66H,0B6H; '3','4','5'
00A2: 3EE0FE 159 DB 3EH,0E0H,0FEH; '6','7','8'
00A5: E6 160 DB 0E6H ; '9'
161 ;

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

LOC OBJ LINE SOURCE

```

162 ;*****
163 ;
164 ;
165 ;These part of the program reads an analog
;input-channel.
166 ;Displaying is done on the LED-display
167 ;On odd-seconds the channel number will be
;displayed.
168 ;On even-seconds the analog value of this channel is
;displayed
169 ;Then the next channel is displayed.
170 ;
00A6: EB 171 ADCON: MOV A,R3 ;Get seconds
00A7: 13 172 RRC A ;lsb to carry
00A8: 503C 173 JNC NEW_MEAS ;Even seconds; do a
;measurement on the current
;channel

174 ;
175 ;Display and/or update channel
00AA: 33 176 RLC A ;Restore accu
00AB: B51402 R 177 CJNE A,PREVIOUS,NEW_CH ;If new seconds,
;update channel number

00AE: 800A 178 JMP DISP_CH
00B0: 0515 R 179 NEW_CH: INC CHANNEL
00B2: E515 R 180 MOV A,CHANNEL ;If channel=4 then
;channel:=0

00B4: B40403 181 CJNE A,#04,DISP_CH
00B7: 751500 R 182 MOV CHANNEL,#00
00BA: 8B14 R 183 DISP_CH:MOV PREVIOUS,R3 ;Update previous seconds
00BC: E515 R 184 MOV A,CHANNEL ;Get segment value of
;channel

00BE: 900193 R 185 MOV DPTR,#LED_TAB
00C1: 93 186 MOVC A,@A+DPTR
187 ;
00C2: 7800 R 188 MOV R0,#TABLE ;Fill table with I2C data
00C4: 7600 189 MOV @R0,#00 ;SAA1064 instruction byte
00C6: 08 190 INC R0
00C7: 7677 191 MOV @R0,#77H ;SAA1064 control byte
00C9: 08 192 INC R0
00CA: F6 193 MOV @R0,A ;Channel number
00CB: E4 194 CLR A
00CC: 08 195 INC R0
00CD: F6 196 MOV @R0,A ;Second digit
00CE: 08 197 INC R0
00CF: F6 198 MOV @R0,A ;Third digit
00D0: 08 199 INC R0
00D1: F6 200 MOV @R0,A ;Fourth byte
201 ;
00D2: D200 R 202 SETB DIR ;I2C transmission of channel
;number

00D4: 750000 R 203 MOV BASE,#TABLE
00D7: 750006 R 204 MOV NR_BYTES,#06H
00DA: 750076 R 205 MOV SLAVE,#SAA1064W
00DD: 120000 R 206 CALL START
207 ;
00E0: 3000FD R 208 FIN_5: JNB I2C_END,FIN_5
00E3: 020027 R 209 JMP REPEAT ; Repeat clock and AD cycle
; again

210 ;
211 ;

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

```

LOC  OBJ          LINE  SOURCE
                                212 ;Measure and display the value of an AD-channel
00E6: 120108  R    213 NEW_MEAS: CALL AD_VAL  ;Do measurement
                                214 ;Wait till values are available
00E9: 3000FD  R    215 FIN_6:  JNB I2C_END,FIN_6
                                216 ;Relevant byte in TABLE+1. Transfer to AN_VAL
00EC: 7801    R    217         MOV R0,#TABLE+1
00EE: 8616    R    218         MOV AN_VAL,@R0
00F0: E516    R    219         MOV A,AN_VAL  ;Channel value in accu for
                                ;conversion
                                220 ;AN_VAL is converted to BCD value of the measured
                                ;voltage.
                                221 ;Input value for CONVERT in accu
                                222 ;Address for MSByte in R1
00F2: 7917    R    223         MOV R1,#CONVAL
00F4: 120154  R    224         CALL CONVERT
                                225 ;Convert 3 bytes of CONVAL to LED-segments
00F7: 900193  R    226         MOV DPTR,#LED_TAB ;Base of segment table
00FA: 7817    R    227         MOV R0,#CONVAL
00FC: 12018A  R    228         CALL SEG_LOOP
                                229 ;Display value of channel to LED display
00FF: 12012C  R    230         CALL LED_DISP
0102: 3000FD  R    231 FIN_8:  JNB I2C_END,FIN_8 ;Wait till I2C
                                ;transmission is ended
0105: 020027  R    232         JMP REPEAT  ;Repeat clock and AD cycle
                                233 ;
                                234 ;*****
                                235 ;Routines used for AD converter.
                                236 ;
                                237 ;AIN reads an analog values from channel denoted by
                                ;CHANNEL.
                                238 ;Send controlbyte:
0108: D200    R    239 AD_VAL: SETB DIR  ;I2C transmission
010A: 7800    R    240         MOV R0,#TABLE  ;Define control word
010C: A615    R    241         MOV @R0,CHANNEL
010E: 750000  R    242         MOV BASE,#TABLE ;Set base at table
0111: 750001  R    243         MOV NR_BYTES,#01H ;Number of bytes to be
                                ;send
0114: 75009E  R    244         MOV SLAVE,#PCF8591W ;Slave address PCF8591
0117: 120000  R    245         CALL START  ;Start transmission of
                                ;controlword
011A: 3000FD  R    246 FIN_7:  JNB I2C_END,FIN_7 ;Wait until transmission is
                                ;finished
                                247 ;Read 2 data bytes from AD-converter
                                248 ;First data byte is from previous conversion and not
                                249 ;relevant
011D: C200    R    250         CLR DIR  ;I2C reception
011F: 750000  R    251         MOV BASE,#TABLE ;Bytes must be stored in
                                ;TABLE
0122: 750002  R    252         MOV NR_BYTES,#02H; Receive 3 bytes
0125: 75009F  R    253         MOV SLAVE,#PCF8591R ;Slave address PCF8591
0128: 120000  R    254         CALL START
012B: 22      255         RET
                                256 ;
                                257 ;LED_DISP displays the data of 3 bytes from address
                                ;CONVAL
LED_DISP:
012C:          258
012C: 431780  R    259         ORL CONVAL,#80H ;Set decimal point
012F: 7800    R    260         MOV R0,#TABLE
0131: 7917    R    261         MOV R1,#CONVAL
0133: 7600    R    262         MOV @R0,#00  ;SAA1064 instruction byte
0135: 08      263         INC R0

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

```

LOC   OBJ           LINE   SOURCE
0136: 7677           264       MOV @R0,#01110111B ;SAA1064 control byte
0138: 08             265       INC R0
0139: 7600           266       MOV @R0,#00        ;First LED digit
013B: 08             267       INC R0
013C: 120185         R 268       CALL GETBY         ;Second digit
013F: 120185         R 269       CALL GETBY         ;Third digit
0142: 120185         R 270       CALL GETBY         ;Fourth digit
0145: D200           R 271       SETB DIR           ;I2C transmission
0147: 750000         R 272       MOV BASE,#TABLE
014A: 750006         R 273       MOV NR_BYTES,#06
014D: 750076         R 274       MOV SLAVE,#01110110B
0150: 120000         R 275       CALL START         ;Start I2C transmission
0153: 22              276       RET
                277       ;
                278       ;CONVERT calculates the voltage of the analog value.
                279       ;Analog value must be in accu
                280       ;BCD result (3 bytes) is stored from address stored
                ;in R1
                281       ;Calculation: AN_VAL*(5/256)
0154: 75F005         282       CONVERT:MOV B,#05
0157: A4             283       MUL AB
                284       ;b2..b0 of reg. B : 2E+2..2E0
                285       ;b7..b0 of accu  : 2E-1..2E-8
0158: A7F0           286       MOV @R1,B          ;Store MSB (10E0-units)
015A: 09             287       INC R1
                015B: 7700           288       MOV @R1,#00        ;Calculate 10E-1 unit
                ;(10E-1 is 19h)
015D: B41C02         289       TEN_CH: CJNE A,#19H+03H,V1 ;Check if accu <= 0.11
0160: 8002           290       JMP TENS           ;accu=0.11; update tens
0162: 4006           291       V1: JC NX_CON     ;accu<0.11; update hundreds
0164: C3             292       TENS: CLR C       ;Calculate new value
0165: 9419           293       SUBB A,#19H
0167: 07             294       INC @R1           ;Update BCD byte
0168: 80F3           295       JMP TEN_CH
                296       ;Correction may be necessary. With 8 bits '0.1' is
                ;in fact 0.0976.
                297       ;A digit of '0A' may appear. Correct this by
                ;decrementing the digit.
                298       ;The intermediate result result must be corrected
                ;with 10*(0.1-0.0976)
                299       ;This is 06h
016A: B70A03         300       NX_CON: CJNE @R1,#0AH,PROC_CON ; If digit is '0A'
                ;then correct
016D: 17             301       DEC @R1
016E: 2419           302       ADD A,#19H
0170: 09             303       PROC_CON:INC R1
0171: 7700           304       MOV @R1,#00        ;Calculate 10E-2 units
0173: B40302         305       HUND: CJNE A,#03H,V2 ;Check if accu <= 10E-2
0176: 8002           306       JMP HUNS           ;accu=10E-2; update hundreds
0178: 4006           307       V2: JC FINISH     ;accu<10E-2; conversion
                ;finished
                ;Calculate new value
017A: C3             308       HUNS: CLR C
017B: 9403           309       SUBB A,#03H
017D: 07             310       INC @R1           ;Update BCD byte
017E: 80F3           311       JMP HUND
0180: B70A01         312       FINISH: CJNE @R1,#0AH,FIN ;Check if result is '0A'.
                ;Then correct.
0183: 17             313       DEC @R1
0184: 22             314       FIN: RET
                315       ;
                316       ;CALLBY transfers byte from @R1 to @R0
0185: E7             317       GETBY: MOV A,@R1
0186: F6             318       MOV @R0,A

```


Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

ASM51  TSW  ASSEMBLER  Demo program for PCD8584 I2C-routines

LOC   OBJ          LINE  SOURCE
0187: 08          319      INC R0
0188: 09          320      INC R1
0189: 22          321      RET
          322      ;
          323      ;SEG_LOOP converts 3 values to segment values.
          324      ;R0 contains address of source and destination
          325      ;DPTR contains base of table
018A: 7903        326      SEG_LOOP: MOV R1,#03      ;Loop counter
018C: E6          327      INLOOP: MOV A,@R0      ;Get value to be displayed
018D: 93          328      MOV C A,@A+DPTR      ;Get segment value from
          329      ;table
          330      MOV @R0,A      ;Store segment data
          331      INC R0
          332      DJNZ R1,INLOOP
018E: F6          333      RET
          334      ;
          335      ;LED_TAB is conversion table for BCD to LED segments
0193:          336      LED_TAB:
0193: 7D483E        337      DB 7DH,48H,3EH      ; '0','1','2'
0196: 6E4B67        338      DB 6EH,4BH,67H      ; '3','4','5'
0199: 734C7F        339      DB 73H,4CH,7FH      ; '6','7','8'
019C: 4F           340      DB 4FH              ; '9'
          341      ;
          342      ;*****
          343      ;
          344      RSEG RAMTAB
0000:          R 345      TABLE: DS 10
          346      ;
000A:          347      END

```

Using the 8XC751/752 in multimaster I²C applications

AN430

INTRODUCTION

The Philips Semiconductors 83C751/87C751 offers the advantages of the 80C51 architecture in a small package and at a low cost. It combines the benefits of a high performance microcontroller with on-board hardware supporting the Inter Integrated Circuit (I²C) bus interface.

The Inter IC (I²C) bus developed by Philips allows integrated circuits to communicate directly with each other via a simple bidirectional 2-wire bus. The comprehensive family of CMOS and bipolar ICs incorporating the on-chip I²C interface offers many advantages to designers of digital control for industrial, consumer and telecommunications equipment.

Interfacing the devices in an I²C based system is very simple as they connect directly to the two bus lines: a serial data line (SDA) and a serial clock line (SCL). System design can rapidly progress from block diagram to final schematics, as there is no need to design bus interfaces. In addition, functional blocks on the block diagram correspond to actual ICs. A prototype system or a final product version can be easily modified or upgraded by 'clipping' or 'unclipping' ICs to or from the bus. The simplicity of designing with the I²C bus does not reduce its effectiveness: it is a reliable, multimaster bus with integrated addressing and data-transfer protocols. The I²C-bus compatible ICs give cost reduction benefits through smaller IC packages and a minimization of PCB traces and glue logic.

The availability of microcontrollers, like the 83C751, with on-board I²C interface is a very powerful tool for system designers. The integrated protocols allow systems to be completely software defined. Software development time of different products can be reduced by assembling a library of re-usable software modules. In addition, the multimaster capability allows rapid testing and alignment of end-products via external connections to an assembly-line computer.

The mask programmable 83C751 and its EPROM version, 87C751, can operate as a master or a slave device on the I²C small area network. In addition to the efficient interface to the dedicated

function ICs in the I²C family the on-board interface facilitates I/O and RAM expansion, access to EEPROM, and processor-to-processor communications.

The 83C752 and its EPROM version, 87C752, are essentially the 83C751/87C751 with the addition of a five channel multiplexed 8-bit A/D converter and an 8-bit PWM output. As the I²C bus interface is identical, the programming example and the discussion relates to both processors. The multimaster capability of the I²C bus allows easy integration and expansion of relatively complex systems, in which different devices can independently initiate data transfers. Integration of a multimaster system is easy as a Master on the bus does not have to coordinate its data transfer with other potential Master devices—arbitration and synchronization are taken care of by the hardware and bus protocols. Expanding a system with a new device is trivial—it is "clipped" onto the two serial bus lines, and the new device may act as a Master without any modification to the other devices (see Figure 1). Microcontrollers like the S8XC751/752 on the I²C bus are extremely powerful, as they can be programmed to be both Masters and Slaves in the same system. This way the microcontroller may initiate communication on the bus, and when requested, will respond to a data transfer request by another device.

In this Application Note we shall discuss the most important technical features of the I²C bus and describe the special I²C hardware interface of the 8XC751/752. We shall demonstrate with an example how the microcontroller can be programmed for a multimaster environment. The communications routines of the example are quite general, and can be ported to many applications—so we shall discuss in detail the software interface to these routines.

The description of the 8XC751 I²C interface hardware and part of the general discussion of the I²C bus is similar to Application Note AN422 which dealt with the microcontroller in a single-master environment. Most of the added discussions relate to the multimaster aspects of the bus. Additional information for the I²C bus and the 83C751/752 Microcontroller can be found in the Philips Semiconductors Microcontroller Data Handbook (IC20).

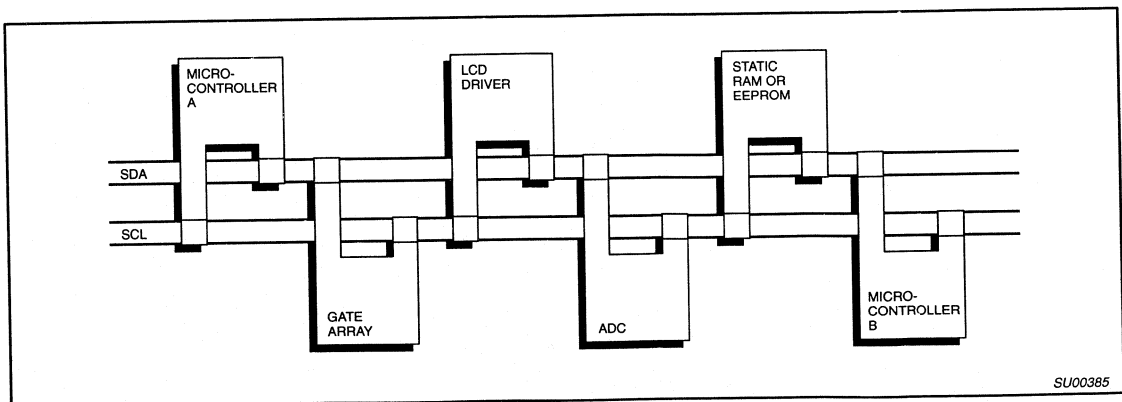
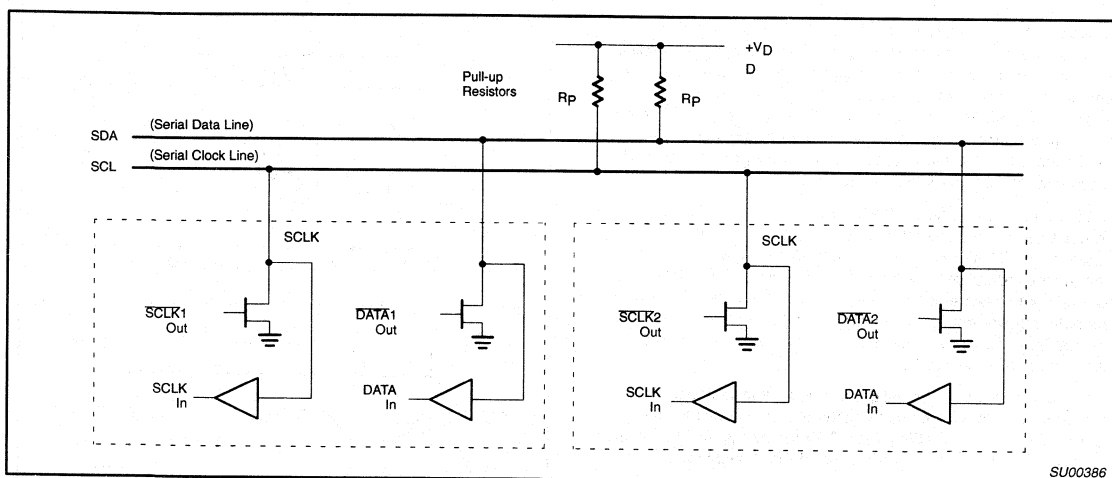


Figure 1. Example of an I²C-bus Configuration

SU00385

Using the 8XC751/752 in multimaster I²C applications

AN430

Figure 2. Connection of I²C-bus Devices to the I²C-bus**THE I²C BUS**

The two lines of the I²C bus are a serial data line (SDA) and a serial clock line (SCL). A typical system configuration is shown in Figure 2. Each device is recognized by a unique address—whether it is a microcomputer, LCD driver, memory or keyboard interface—and can operate as either a transmitter or a receiver, depending on the function of the device. A device generating a message or data is a transmitter, and a device receiving the message or data is a receiver. Obviously, a passive function like an LCD driver could only be a receiver, while a microcontroller or a memory can both transmit and receive data.

Every device connected to the bus must have an open-drain or an open-collector output for both the data (SDA) and the clock (SCL) lines. Each one of the lines is connected to the positive supply via a common pull-up resistor (see Figure 2). This implements a wired-AND function, and each of the bus lines which will have the HIGH level only if all the output transistors tied to it are switched off.

Data on the I²C bus can be transferred at a rate up to 100kbit/s. The number of devices connected to the bus is limited only by the maximum bus capacitance of 400pF. As different technology devices can be connected to the I²C bus, the levels of the logical 0 (Low) and logical 1 (High) are not fixed and depend on the appropriate level of V_{DD}.

MASTERS AND SLAVES

When a data transfer takes place on the bus, a device can be either a master or a slave. The device which initiates the transfer, and generates the clock signals for this transfer is the master. At that time any device addressed is considered a slave. It is important to note that a master could be either a transmitter or a receiver: a master microcontroller may send data to a RAM acting as a transmitter, and then interrogate the RAM for its contents acting as a receiver—in both cases being the master initiating the transfer. In the same manner, a slave could be both a receiver and a transmitter.

The I²C is a multimaster bus. It is possible to have in one system more than one device capable of initiating transfers and controlling

the bus. A microcontroller may act as a master for one transfer, and then be the slave for another transfer, initiated by another processor on the network. The master/slave relationships on the bus are not permanent, and exist per transfer.

As more than one master may be connected to the bus it is possible that two devices will try to initiate transfer at the same time. Obviously, in order to eliminate bus collisions and communications chaos, an arbitration procedure is necessary. The I²C design has an inherent arbitration and clock synchronization procedure relying on the wired-AND connection of the devices on the bus. In a typical multimaster system, a microcontroller program should allow it to gracefully switch between master and slave modes and preserve data integrity upon loss of arbitration.

DATA TRANSFERS

One data bit is transferred during each clock pulse (Figure 3). The data on the SDA line must remain stable during the HIGH period of the clock pulse in order to be valid. Changes in the data line at this time will be interpreted as control signals. A HIGH-to-LOW transition of the data line (SDA) while the clock signal (SCL) is HIGH indicates a Start condition, and a LOW-to-HIGH transition of the SDA while SCL is HIGH defines a Stop condition (Figure 4). The bus is considered to be busy after the Start condition and free again a certain time after the Stop condition. The Start and Stop conditions are always generated by the master.

The number of data bytes transferred between the Start and Stop condition from transmitter to receiver is not limited. Each byte, which must be eight bits long, is transferred serially with the most significant bit first, and is followed by an acknowledge bit (Figure 5). The clock pulse related to the acknowledge bit is generated by the master. The device that acknowledges has to pull down the SDA line during the acknowledge clock pulse, while the transmitting device releases the SDA line (HIGH) during this pulse (Figure 6).

A slave receiver must generate an acknowledge after the reception of each byte, and a master must generate one after the reception of each byte clocked out of the slave transmitter. If a receiving device cannot receive the data byte immediately, it can force the transmitter

Using the 8XC751/752 in multimaster I²C applications

AN430

into a wait state by holding the clock line (SCL) LOW. When designing a system it is necessary to take into account cases when acknowledge is not received. This happens, for example, when the addressed device is busy in a real time operation. In such a case the master, after an appropriate "time-out", should abort the transfer by generating a Stop condition, allowing other transfers to take place. These "other transfers" could be initiated by other masters in a multimaster system or by this same master.

An exception to the "acknowledge after every byte" rule occurs when a master is a receiver: it must signal an end of data to the transmitter by NOT signalling an acknowledge on the last byte that has been clocked out of the slave. The acknowledge related clock, generated by the master, should still take place but the SDA line will not be pulled down. In order to indicate that this is an active and intentional lack of acknowledgement, we shall term this special condition as a "Negative ACK".

The bus design includes special provisions for interfacing to microprocessors which implement all the I²C communications in software only—it is called "Slow Mode". When all the devices on the network have built-in I²C hardware support the Slow Mode is irrelevant.

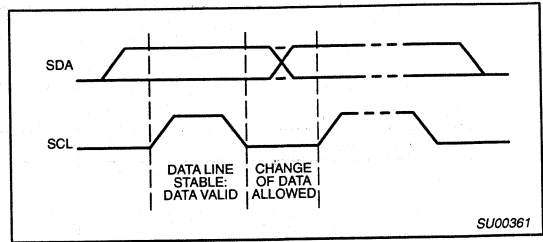


Figure 3. Bit Transfer on the I²C Bus

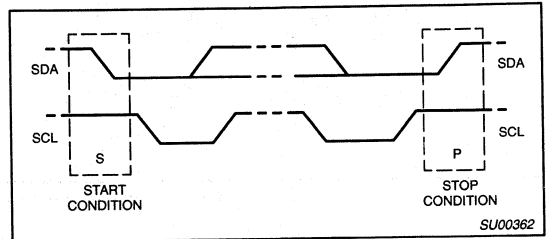


Figure 4. Start and Stop Conditions

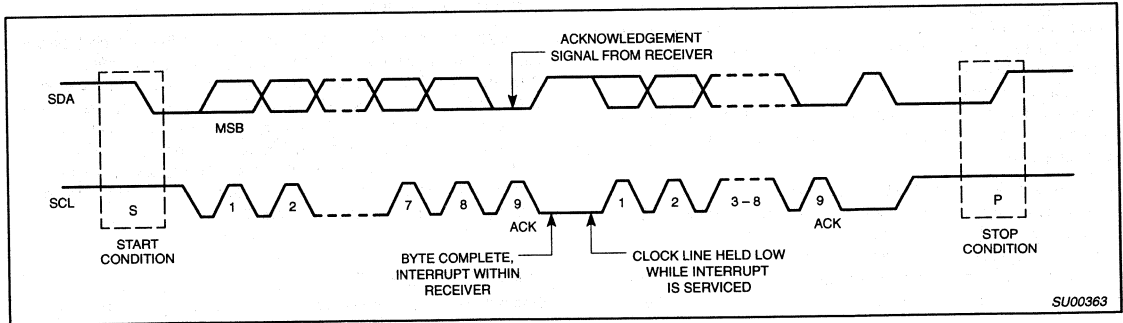


Figure 5. Data Transfer on the I²C Bus

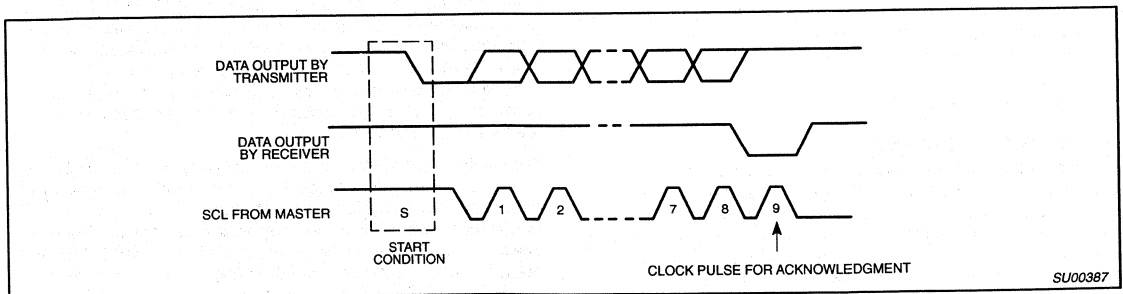
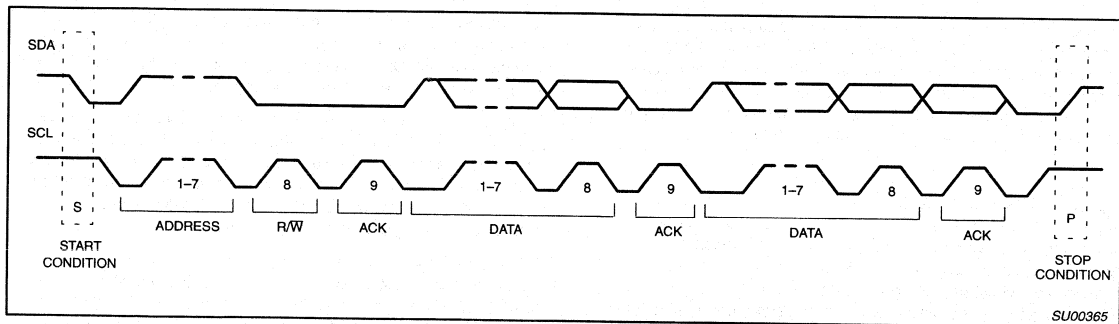
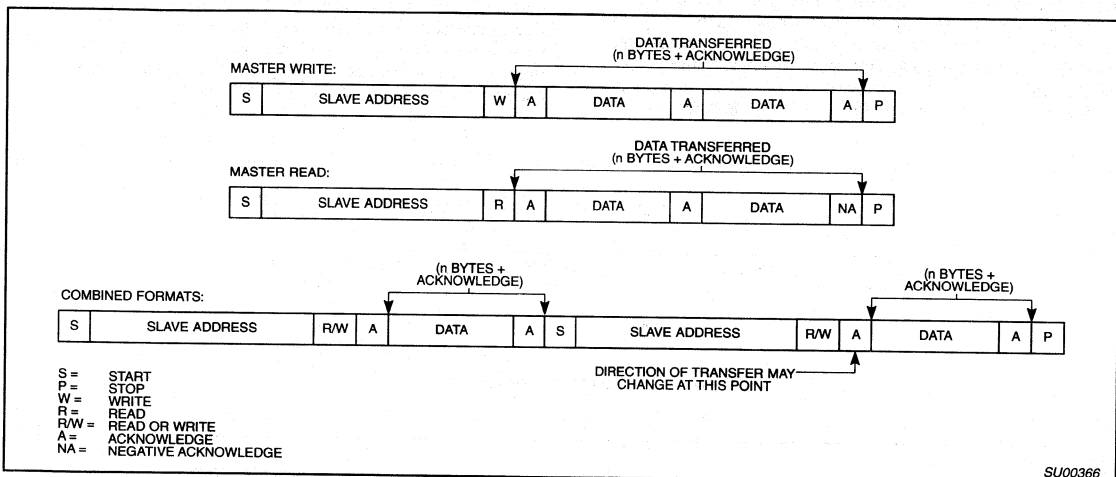


Figure 6. Acknowledge on the I²C Bus

Using the 8XC751/752 in multimaster I²C applications

AN430

Figure 7. A Complete Data Transfer on the I²C-BusFigure 8. I²C Data Formats**ADDRESSING AND TRANSFER FORMATS**

Each device on the bus has its own unique address. Before any data is transmitted on the bus, the master transmits on the bus the address of the slave of this transaction. A well-behaved slave, if it exists on the network, should of course acknowledge the master's addressing. The addressing is done with the first byte transmitted by the master after the Start condition.

An address on the network is seven bits long, appearing as the most significant bits of the address byte. The last bit is a direction (R/W) bit. A zero indicates that the master is transmitting (WRITE) and a one indicates that the master requests data (READ). A complete data transfer, comprised of an address byte indicating a WRITE and two data bytes is shown in Figure 7.

When an address is sent, each device in the system compares the first seven bits after the Start with its own address. If there is a match, the device will consider itself addressed by the master and will send an acknowledge. The device could also determine if in this transaction it is assigned the role of a slave receiver or slave transmitter, depending on the R/W bit.

Each node of the I²C network has a unique seven bit address. The address of a microcontroller is, of course, fully programmable, while peripheral devices usually have fixed and programmable address

portions. In addition to the "standard" addressing discussed here, the I²C bus protocol allows for "general call" addressing and interfacing to CBUS devices.

When the master is communicating with one device only, data transfers follow the format of Figure 8 where the R/W bit could indicate either direction. After completing the transfer and issuing a Stop condition, if a master would like to address some other device on the network, it could start another transaction by issuing a new Start.

Another way for a master to communicate with several different devices would be by using a "repeated start". After the last byte of the transaction was transferred, including its acknowledge (or Negative ACK), the master issues again a Start, followed by address byte and data, without effecting a Stop. The master may communicate with a number of different devices, combining READS and WRITES. Only after the transfer with the last slave took place, the master issues a Stop and releases the bus. Possible data formats are demonstrated in Figure 8. Note that the repeated start allows for both change of a slave and a change of direction, without releasing the bus. We shall see later on that the change of direction feature can come in handy even when dealing with a single device.

Using the 8XC751/752 in multimaster I²C applications

AN430

In a single master system the repeated start mechanism is more efficient than terminating each transfer with a Stop and starting again. In a multimaster environment the determination of which format is more efficient could be more complicated, as when a master is using repeated starts it occupies the bus for a long time and prevents other devices from initiating transfers.

USE OF SUB-ADDRESSES

For some ICs on the I²C bus the device address alone is not sufficient for effective communications and a mechanism for addressing the internals of the device is necessary. A typical example is addressing memories, when we want to access a specific word inside the device or a sequence of memory locations starting at a specific internal address.

A typical I²C memory device like the PCF8570 RAM contains a built-in word address register that is incremented automatically after each read or written data byte. When a master communicates with the PCF8570 it must send a sub-address in the byte following the slave address byte. This sub-address is the internal address of the word the master wants to access for a single byte transfer or the

beginning of a sequence of locations for a multi-byte transfer. A sub-address is an eight bit byte, unlike the device address it does not contain a direction (R/W) bit, and like any byte transferred on the bus it must be followed by an acknowledge.

A memory write cycle is shown in Figure 9(a). The Start is followed by a slave byte with the direction bit set to WRITE, a sub-address byte, a number of data bytes and a Stop signal. The sub-address is loaded into the word address memory. The data bytes which follow will be written one after the other starting with the sub-address location and the register is incremented automatically.

The memory read cycle (Figure 9(b)) commences in a similar manner with the master sending a slave address with the direction bit set to WRITE with a following sub-address. Then, in order to reverse the direction of the transfer, the master issues a repeated Start followed again by the memory device address, but this time with the direction bit set to READ. The data bytes starting at the internal sub-address will be clocked out of the device with each followed by a master-generated acknowledge. The last byte of the read cycle will be followed by a Negative ACK, signalling the end of transfer. The cycle is terminated by a Stop signal.

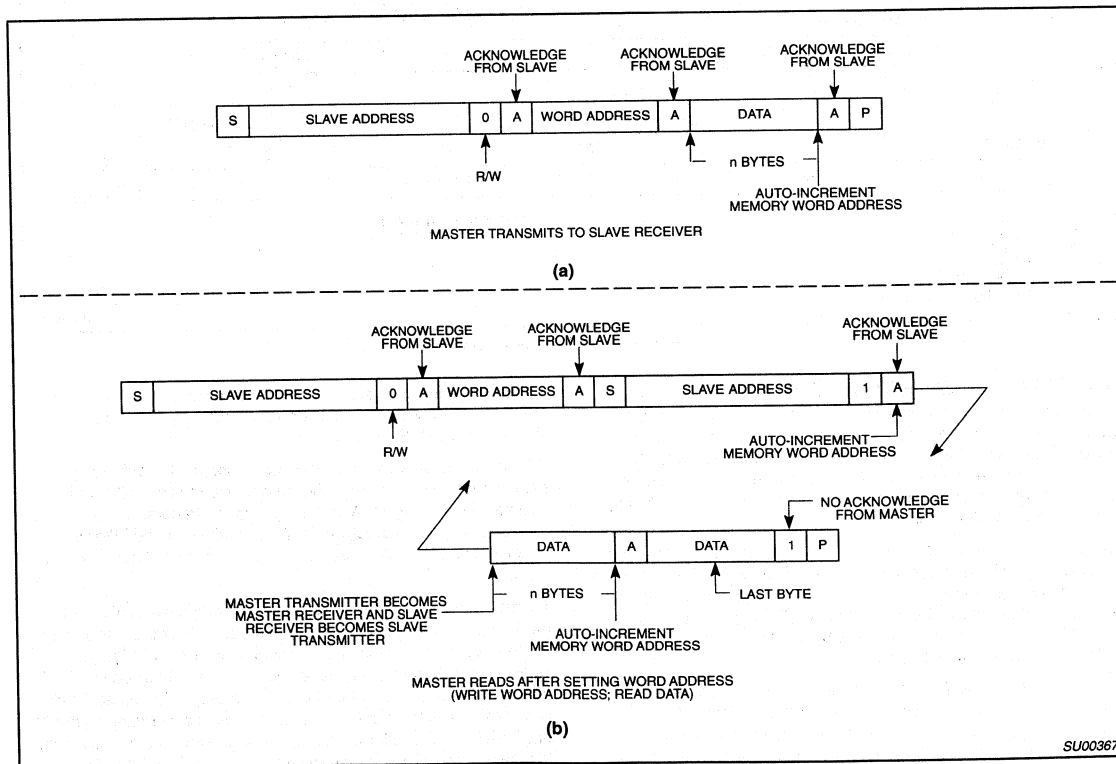


Figure 9. I²C Sub-Address Usage

SU00367

Using the 8XC751/752 in multimaster I²C applications

AN430

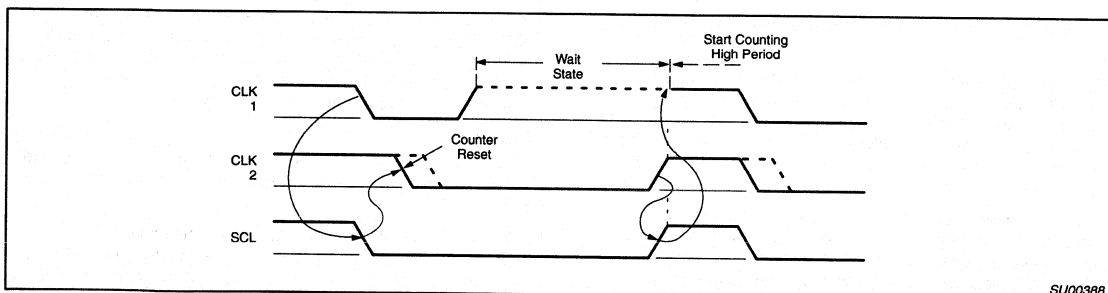


Figure 10. Clock Synchronization During the Arbitration Procedure

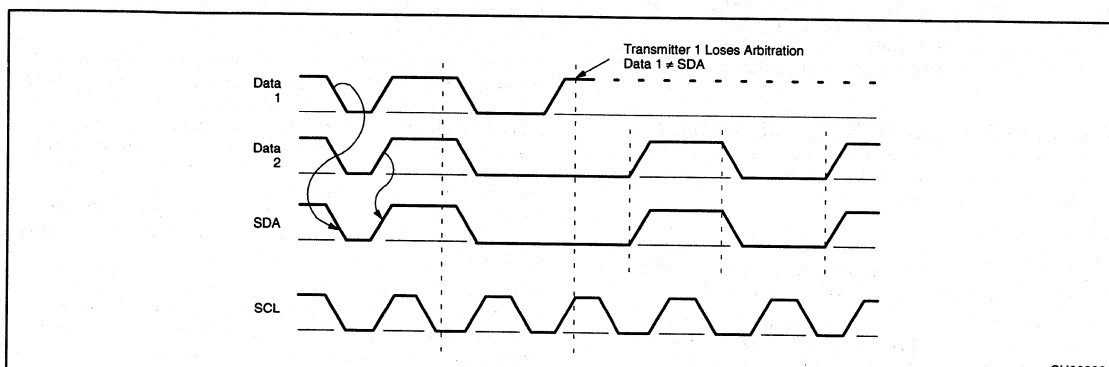


Figure 11. Arbitration Procedure of Two Masters

ARBITRATION IN A MULTIMASTER SYSTEM

The decision about which master has control over the I²C bus is based solely on the address and data sent by competing masters, and there is no central master or any order of device priority on the bus. Any device connected to the I²C bus is allowed to become a master, but devices are not supposed to "steal" the bus from other devices when a transfer is in process. If a device wishing to be a Master is aware that a transaction (initiated by another master) is taking place, it will wait until the transfer is concluded with a Stop condition on the bus—and only then try to seize it by sending its own Start. It is possible, however, that two or more masters may want to start a transfer at exactly the same moment. A scenario that may happen quite frequently in a loaded system: two devices are waiting for a long transaction to be completed, and simultaneously try to get the bus when detecting the Stop condition. An arbitration procedure synchronizes the different clocks, ensuring that the data is not corrupted, and causes all masters except one to withdraw from the bus, so only one master will control the transfer. This procedure applies only when masters initiate transfers simultaneously.

The clock synchronization, illustrated in Figure 10, ensures that only one defined clock is generated on the bus. It occurs naturally, as a result of the wired-AND property of the SCL line. Suppose two masters want to initiate a transfer on the bus. Clk1 and Clk2 in Figure 10 illustrate the desired clock outputs of each device, which would actually occur on the bus if each were the only master. The SCL waveform is the resulting wired-AND of the two clocks. The device that pulls the SCL down first will succeed. The other masters

continuously monitor the clock line, and reset their internal clock counter to start counting their own Low clock period. This way, the first falling edge will synchronize all clock generators to the beginning of the Low time.

Once a device clock has gone Low it will hold the SCL line in this state until its internal clock High state is reached, and then will release the line. The Low to High change in this device will not change the state of the SCL line if another device, which is still within its Low period, is pulling down the line. This way, SCL will be held Low by the device with the longest Low period. A master that has finished its Low time earlier will enter a wait state until SCL is released by the slowest master and goes high. Upon the rising edge of SCL all masters start counting their High period, the first device to complete its High period will pull the SCL Low. In this way a single, synchronized clock is generated on the bus where the rising edge is being defined by the slowest master and the falling edge by the fastest master.

Arbitration between masters takes place on the SDA line. A master which tries to transmit a High while another device transmits a Low will withdraw, shutting off its data output stage and not interfering with the transfer until a Stop condition is detected. Due to the wired-AND property of the SDA line, a device "knows" that it lost arbitration by the fact that the Low SDA is different than its desired High output. Arbitration starts by comparing the address bits. When masters transmit different addresses the one transmitting the address with the lowest binary value wins. If all masters in arbitration transmit to the same address, arbitration continues into

Using the 8XC751/752 in multimaster I²C applications

AN430

the comparison of data. Figure 11 illustrates the arbitration process between two masters.

By definition, the transfer that forces the wired-AND result is the one that wins the arbitration, so the address and data of a winning device are not corrupted and no information is lost in the arbitration process. A master losing arbitration may generate clock pulses until the end of the byte. Thus it may affect the clock speed, but not the data on the bus.

If a master loses arbitration during the addressing stage it is possible that the winning master is trying to address it. In an efficient design, the losing master should switch immediately to its slave receiver mode, receive the data transmitted and acknowledge it—otherwise the message will have to be re-transmitted or is lost. A well designed master will take into account "illegal" protocol situations and will determine that it lost arbitration when it detects a Stop or a Start which are not synchronized with its own transmission. Electrical interference or a malfunctioning device may cause such a situation which actually corrupts the message transfer.

HANDSHAKE BY CLOCK SYNCHRONIZATION

The clock synchronization mechanism as described above actually implements a handshake mechanism, enabling receiving devices to "slow down" fast transfers when necessary.

On the bit level, a slow slave device like a microcontroller that does not have hardware I²C interface port, can extend each clock period and slow down the bus clock. The speed of any master is adapted to the operating rate of this device as long as it is active on the bus.

On the byte level the synchronization mechanism takes effect as a "handshake" mechanism when a slave device that was fast enough to receive or transmit a byte still needs extra time to store the received byte or prepare the next byte for transmission. The slave can hold the SCL line low after the reception and acknowledge of a byte, thus forcing the Master into a wait state—until the slave is ready for the next transfer.

8XC751 I²C HARDWARE

The on-chip I²C bus hardware support of the 8XC751 allows operation on the bus at full speed and simplifies the software needed for effective communications on the network. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing error checks, and takes care of clock stretching and synchronization. The hardware support includes a bus timeout timer, called Timer I. The hardware is synchronized to the software either through polled loops or interrupts.

Two of the port 0 pins are multi-functional. When the I²C is active, the pin associated with P0.0 functions as SCL, and the pin associated with P0.1 functions as SDA. These pins have an open drain output.

Two of the five interrupt sources may be used for I²C support. The I²C interrupt is enabled by the EI2 flag of the interrupt enable register, and its service routine should start at address 023h. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates new data on the bus or a special condition occurs: Start, Stop or arbitration loss. The interrupt is induced by the ATN flag, (see below for the conditions for setting this flag). The Timer I overflow interrupt is enabled by the ETI flag, and the service routine starts at 01Bh.

The I²C port is controlled through four special function registers: I²C Control (I2CON), I²C Configuration (I2CFG), I²C Data (I2DAT) and

I²C Status (I2STA). The register addresses are shown in the 8XC751 section of the Philips Semiconductors Microcontroller Data Handbook (IC20). Although the following discussion of the hardware and register details is not complete, it should give a better understanding of the programming examples.

Timer I

In I²C applications, Timer I is dedicated to the port timing generation and bus monitoring. In non-I²C applications, it is available for use as a fixed rate timer.

For the bus monitoring function, Timer I is being used as a "watchdog timer" for bus hang-ups. It creates an interrupt when the SCL line stays in one state for an extended period of time between a Start condition and a following Stop condition. SCL "stuck low" indicates a faulty master or slave. SCL "stuck high" may mean a faulty device or that noise induced into the I²C caused all masters to withdraw from the I²C arbitration.

The time-out interval of Timer I is fixed: it carries out and interrupts (if enabled) when about 1024 machine cycles have elapsed since a change on SCL within a frame. In other words, whenever I²C is active we let Timer I run, but clear it whenever a frame is not in progress (reset or Stop occurred more recently than the last Start condition) or SCL changes within a frame. (Note: we wrote "about 1024 machine cycles" for the sake of accuracy—this number may slightly change according to the setting of the CT0 and CT1 bits mentioned below. In any case, the exact number of cycles for a time out does not have any practical significance).

In addition to the interrupt upon Timer I overflow, the I²C port hardware is reset. This is useful for multiple master systems in situations where this same 8XC751 caused the bus hang-up due to a lack of software response. SCL will be released and I²C operation between other devices could continue.

I2CON Register

The I²C Control register can be read or written to (see Figure 12).

When writing to the I2CON register one should use bit masks as demonstrated in the examples. Trying to clear or set the bits in the register using the bit addressing capabilities of the 8XC751 may lead to undesirable results. The reason is that a command like CLRFB reads the register, sets the bit and writes it back—and the write-back may affect other bits.

I2CFG Register

The configuration register is a read/write register (see Figure 13).

I2DAT Register

The I²C data register is a read/write register, where the msb represents the data received or data to be sent. The other seven bits are read as 0 (see Figure 14).

I2CSTA Register

The I²C STAatus Register is a read-only register reflecting the internal status of the I²C interface hardware (see Figure 15).

Transmit Active State

The transmit active state—Xmit Active—is an internal state in the I²C interface that is affected by the I²C registers as explained above. The I²C interface will only drive the SDA line low when Xmit Active is set. Xmit Active is set by writing the I2DAT register or by writing I2CON with XSTR = 1 or XSTP = 1. The ARL bit will be set to 1 only when Xmit Active is set—in such a case Xmit Active will be automatically reset upon ARL. Xmit Active is cleared by writing 1 to CXA at I2CON register or by reading the I2DAT register.

Using the 8XC751/752 in multimaster I²C applications

AN430

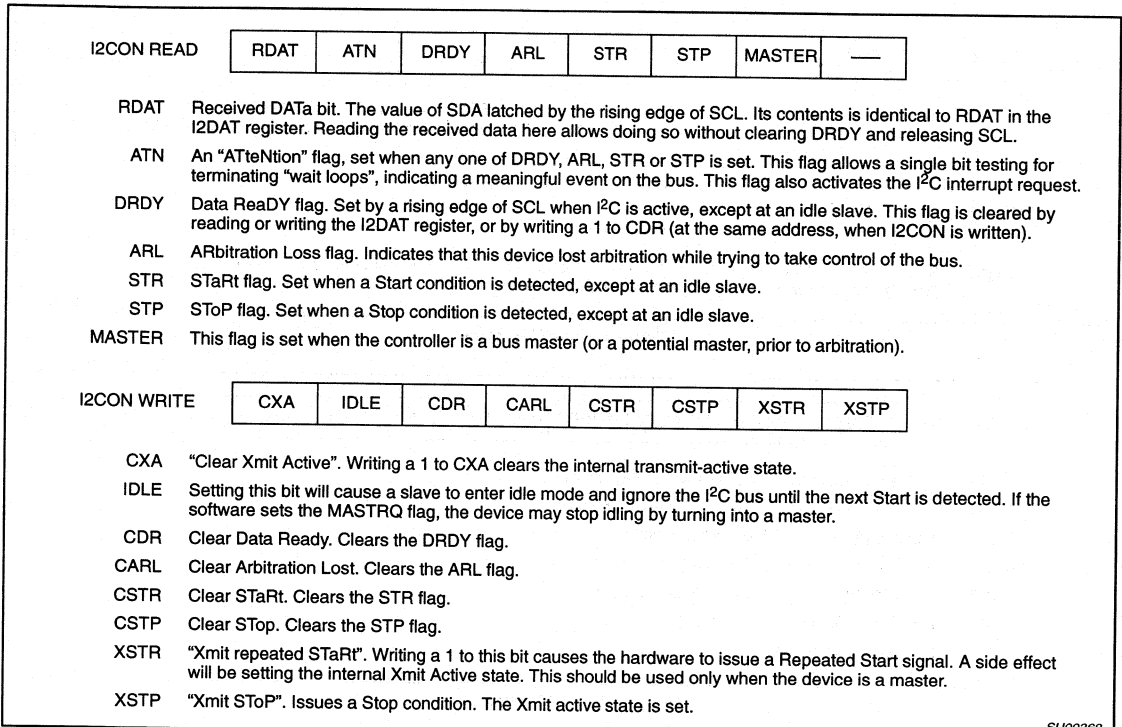


Figure 12. I2CON Register

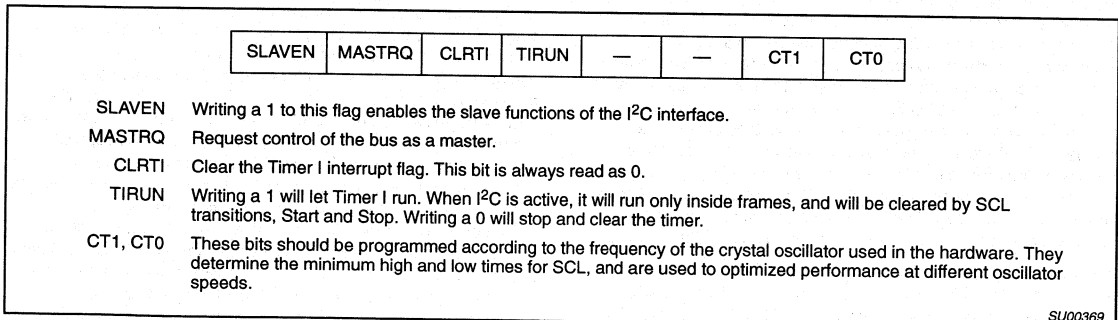


Figure 13. I2CFG Register

Using the 8XC751/752 in multimaster I²C applications

AN430

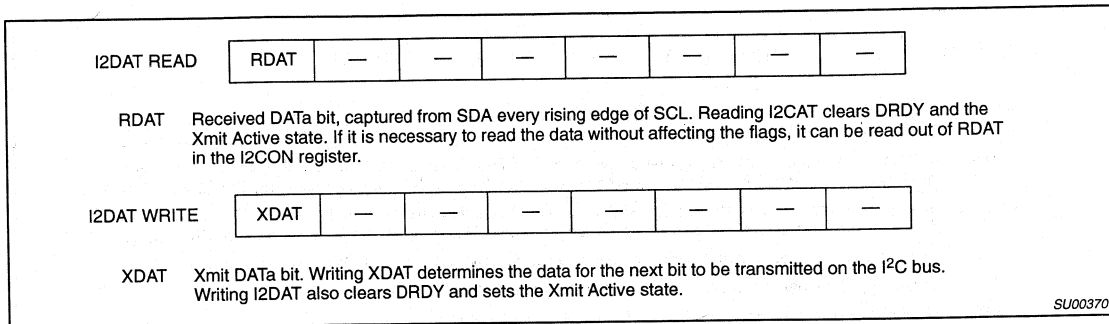


Figure 14. I2DAT Register

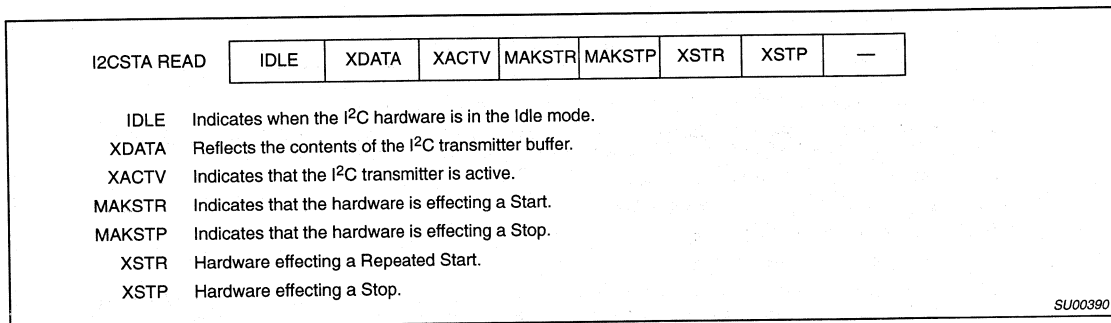


Figure 15. I2CSTA Register

I²C COMMUNICATIONS SOFTWARE

The software listing demonstrates programming the 8XC751/752 for a multimaster I²C environment where the device can be both a Master or a Slave responding to other Masters on the I²C network. The bulk of the software is communications routines which are not only for demonstration but could be ported to other user programs with minimal or no modifications. The routines are quite general and could be useful in most applications. We have tried to design a well-defined software interface, enabling most users to copy the routines as they are, modifying only the pre-defined interface elements to fit the specific applications. We encourage users to use the routines without modifications whenever possible, as the lower levels of the hardware-software integration could be quite involved.

The rest of this application note will relate to the programming example. We shall discuss the general operation of the routines and how they are integrated into an application. Then we shall describe in detail all the software interface elements and how to use them.

I²C COMMUNICATIONS ROUTINES—OVERVIEW

In order to function well in a multimaster environment the microcontroller must be able to take control of the I²C bus as a Master, "tolerate" message transactions between other Masters and other devices, and respond efficiently as a Slave to other bus Masters. The communications routines should allow a Master "graceful" recovery from an arbitration loss and other situations when a message transaction is not completed, allowing for communication re-tries.

For Slave operation the microcontroller must be interrupt driven relative to an I²C frame Start, as any Master on the bus could request a transaction at any moment, not synchronized to the application program executing on the controller. An interrupt service routine monitors the address transmitted on the bus. When the microcontroller is addressed it takes care to either read the data from the bus into a buffer or write buffer data onto the bus. When such a transaction is successfully completed, one of several "Slave Event Routines" is called prior to returning to the main application program. Such an "Event Routine" is a part of the application, allowing an immediate response to the data received, or the fact that data was transmitted to a requesting Master. This allows "synchronization" of the application to a "slave" bus transaction. Typical uses of the Event Routine mechanism will be a computation based on new data, or re-loading the transmit buffer with new data getting ready for the next random request. The actual Event Routines will be programmed differently for different applications, but the names and the calls will remain the same as long as the communications routines are left unmodified.

A transaction as a Master is initiated by the application program. Our implementation uses the interrupt mechanism for the Master communications as well. The application issues a request for the bus by setting the MASTRQ bit of the I²C port control, and when the bus is available an interrupt occurs. This way, if the bus is free there will be an immediate response. If the bus is busy, the application may go on executing (if so programmed) until this controller can get control of the bus. When the microcontroller gets mastership of the bus it initiates a bus transaction according to "directives" set by the

Using the 8XC751/752 in multimaster I²C applications

AN430

application program. The most important directives are the address (and subaddress if relevant) of the slave device addressed, and the length of the message to be transmitted or received.

When a Master transaction is concluded, a Master Event Routine (called MastNext) is called to perform whatever task the application demands. As with the Slave Event Routines it will typically respond to a successful transmission or reception of data. In addition, it could handle situations where a slave does not respond at all, or does not acknowledge a data byte (thus causing data transfer to terminate). A program might react to the fact that a slave does not respond by re-trying to communicate at a later time, by issuing a message to another peripheral device or just ignoring it. The handling of such cases is application dependent, and should be programmed into the routine called "MastNext". The MastNext routine is invoked when the Master terminates the transaction "willingly", but not upon arbitration loss.

The microcontroller operating as a bus Master may lose arbitration to another Master which happens when two Masters transmit in synchronization, commencing with the same Start signal. If arbitration is lost while transmitting or receiving data, our processor withdraws from the bus and turns itself into a slave—an active Slave upon a Start, or returning to the calling program as an idle slave. When the arbitration loss occurs while transmitting an address, our processor turns itself immediately into an active slave, "listening" to the rest of the address transmitted by the new Master. If our processor reads its own address from the bus (as transmitted by the new Master) our processor responds as a willful slave. If this mechanism would not have been implemented, there could be potential inefficiency when a device that happened to be synchronized to another Master loses arbitration, but is not able to respond to the winning device.

Another situation for arbitration loss could be a bus exception resulting from a device operating not according to the bus protocol or interference on the bus lines. In addition to "regular" arbitration loss detected with the ARL hardware flag, such a situation may occur with detecting a Start or a Stop in the middle of transmitting an address or data byte. In such a situation the microcontroller withdraws from the bus as well—active Slave upon a Start detection, or returning as an idle slave in other cases.

When a Master transaction is terminated by an arbitration loss, the Master Request flag (MASTRQ) of the hardware I²C port remains in effect. As a result when the bus gets free, our device will take control, issue a Start, and the transaction that was cut will start again. This restart will happen automatically, without any application involvement (unlike non-acknowledgement, where the MastNext routine determines what shall be done).

The I²C communications routines are structured as an interrupt service routine responding to an I²C port interrupt upon a frame Start. Within a frame the I²C processing is continuous, where the I²C port is polled for hardware response, and the I²C interrupts are disabled. Other interrupts are enabled during the service routine. The set-up requirements from the mainline program are minimal, and interfacing is done via RAM buffers and some pre defined RAM locations. The lower level interface with the hardware is done inside the service routine, and can typically be ignored by the application programmer.

BUS WATCHDOG AND ERROR RECOVERY

A malfunctioning device (in hardware or software) may hold the SCL line low, thus causing the bus to be "stuck". It might even be possible that a transient protocol violation (due to hardware interference, such as a device turning on) may cause some devices (non programmable, or even microcontrollers which were not carefully programmed) to hold the bus. Since within a frame the bus is software-pollled, a "stuck" bus might cause the application software to "hang forever". Here the TIMER1 watchdog comes to the rescue, interrupting when there is no bus activity for a long period of time.

When the I²C service routine is interrupted by the watchdog timer, the processing of the current frame is not completed and the event routines are not called. The software returns to execute the mainline application, and will be interrupted again for the next frame (next Start, received as a slave or induced as a Master). A status flag and a counter report on the watchdog interrupt, so the application program can be made to inhibit the I²C port if there are too many occurrences of a "hanging" bus.

Bus protocol errors and "hangups" might be an issue in systems which are susceptible to noise, temporary bus line shorts, "hot plug in" of devices or even erroneously programmed devices—and a "fail safe" controller program should be able to detect bus problems and possibly assist in resolving them. The RECOVER routine resets the I²C interface of the microcontroller, and attempts to release some other devices on the bus by toggling the clock line. The I²C interface of the 8XC751 is reset by letting Timer1 run and expire, since this circuitry does not feature a software controlled reset. This "extreme" measure is needed in some cases of bus protocol violation.

The bus and interface circuit recovery routine can be automatically invoked whenever Timer1 detects a timeout. In addition, for systems where potential bus failures are a concern and reliability is an issue, one may implement mechanisms to invoke bus and interface recovery from the application code. This may help in cases where the bus gets "stuck" when there is no I²C frame in progress. In such an instance the watchdog timer will not give any timeout indications, as it has not been activated. Another case emanates from a design peculiarity of the interface circuitry on the 8XC751: if the SCL line is externally grounded when there is a Start condition, this Start might be ignored, and the watchdog may not be activated. Our programming example deals with potential failures by testing for transaction completion and retrying transmissions when necessary (these are explicit retries, in addition to an "automatic" retry after a Master's arbitration loss, invoked by the MASTRQ bit). Too many transmission failures activate the RECOVER routine.

Using the 8XC751/752 in multimaster I²C applications

AN430

I²C COMMUNICATIONS ROUTINES—INTERFACE

The I²C service routine deals with the transmission and reception of messages, without any concern for the contents of the message. In order to provide a general interface for different applications the data is transferred via buffers. The service routine does not have to “know” where the data goes to or comes from—as long as the application program specifies the required pointers for these buffers. The interface to the actual application (which “cares” about message contents, timing, addressing and so forth) is done in a well defined manner, allowing usage of the same service routine with different application programs.

The interface is carried out with the use of buffers, pre-defined names for Application Event Routines, interface RAM locations for transferring parameters, pointers and flags, and constants. A more detailed discussion of the interface follows.

Buffers

There are three buffers for data transfers between the I²C bus and the application program.

MasBuf is used for Master transmission and reception. The number of data bytes for each Master message—reception or transmission, is specified by the memory location MASTCNT. The value in MASTCNT should be less than the length of MasBuf. For Master transmission the message is placed in MasBuf before the transmission is initiated. In Master reception, the received message will be contained in the same buffer. There is only one Master message transaction occurring at the same time, so we may use the same buffer both for transmission and reception.

For Slave operation we must accommodate data transfers which may come randomly, asynchronous to each other or to possible operation of the same device as a Master. Therefore it is necessary to allocate additional RAM area as buffers dedicated to Slave operation: SRcvBuf for receiving data, STxBuf for transmission.

The length of the Slave receive buffer is defined by the symbol RBufLen. It is used by the code for protection, avoiding overwriting RAM beyond the allocated buffer size in case a Master sends a message which is too long. There is no need for RAM protection for transmission, but the Master should not request more data than STxBuf can supply.

Interface RAM Locations

RAM location MyAddr contains the address of this processor.

Status flag MSGSTAT is used for reporting to the application on I²C communications status—mainly on the successful, or unsuccessful, completion of a message transaction. The contents of MSGSTAT may be used by the mainline application code or by the Event Routines. The different codes that could be placed by the I²C service routine are described later in the text. When the message processing commences, a code indicating Slave or Master processing is inserted to MSGSTAT, and is updated as we go along.

There could be many applications that will not need to use MSGSTAT contents, as the very fact of calling a certain event routine implies completion of a processing stage.

For Master transactions, in addition to the data buffer MasBuf, there are several RAM locations into which the application inserts Master message “directives”. These directives provide the service routine with the information necessary to carry out the next Master transaction. The one byte RAM locations used for directives are DESTADRW, DESSUBAD, MASTCNT and MASCMD.

DESTADRW contains the destination slave address in bits 7-1, while bit 0 is the R/W bit. Bit 0 contains 0 for a Write operation (the message is to be transmitted to the slave) and 1 for a Read operation (message is being read from the slave and received by this Master).

DESSUBAD contains the 8 bit sub-address of the slave, if necessary. For transactions without a sub-address, the contents of DESSUBAD is ignored.

MASTCNT contains the number of data bytes in the message to be sent from or received into MasBuf. This number should not be bigger than the length of MasBuf.

MASCMD byte contains the bit flags SUBADD, RPSTRT and SETMRQ. SUBADD is 0 (cleared) for a message with a regular address, and 1 (set) when a subaddress is required. When SUBADD is set, the service routine takes care of all the protocol required for sub-addressing, which includes a Repeated Start for Read operations. A message with a subaddress is considered to be a single message, even if it includes a Repeated Start.

The RPSTRT and SETMRQ are kept cleared in regular applications, and will be used only for “tailoring” the bus transfers in special cases. When RPSTRT is cleared the message will terminate, as usually required, with a Stop. When RPSTRT is set a Repeated Start will be sent on the bus, and Master operation will resume. The RPSTRT directive relates to terminating the message after all the data was transferred, and not to the mandatory Repeated Start in the middle of sub-addressed Read operation. A single message with a subaddress will typically have RPSTRT cleared. SETMRQ indicates what will be loaded into the MASTRQ flag of the hardware when Stop is transmitted. Typically it will be cleared. When SETMRQ is 1, MASTRQ will be set, thus trying to issue a new Start immediately following the Stop. In such a case the service routine will not return upon Stop, but will continue as a Master.

TITOCNT is used to count time-outs of the watchdog timer. Whenever such a timeout invokes the TIMER I interrupt service routine the contents of the location TITOCNT are incremented, and the timeout is reported in MSGSTAT. The count is saturated at 0FFh. This mechanism may be used in an application that is very much “concerned” with potential bus failures, allowing some type of “failure monitoring” by the application even for Slave transactions.

Using the 8XC751/752 in multimaster I²C applications

AN430

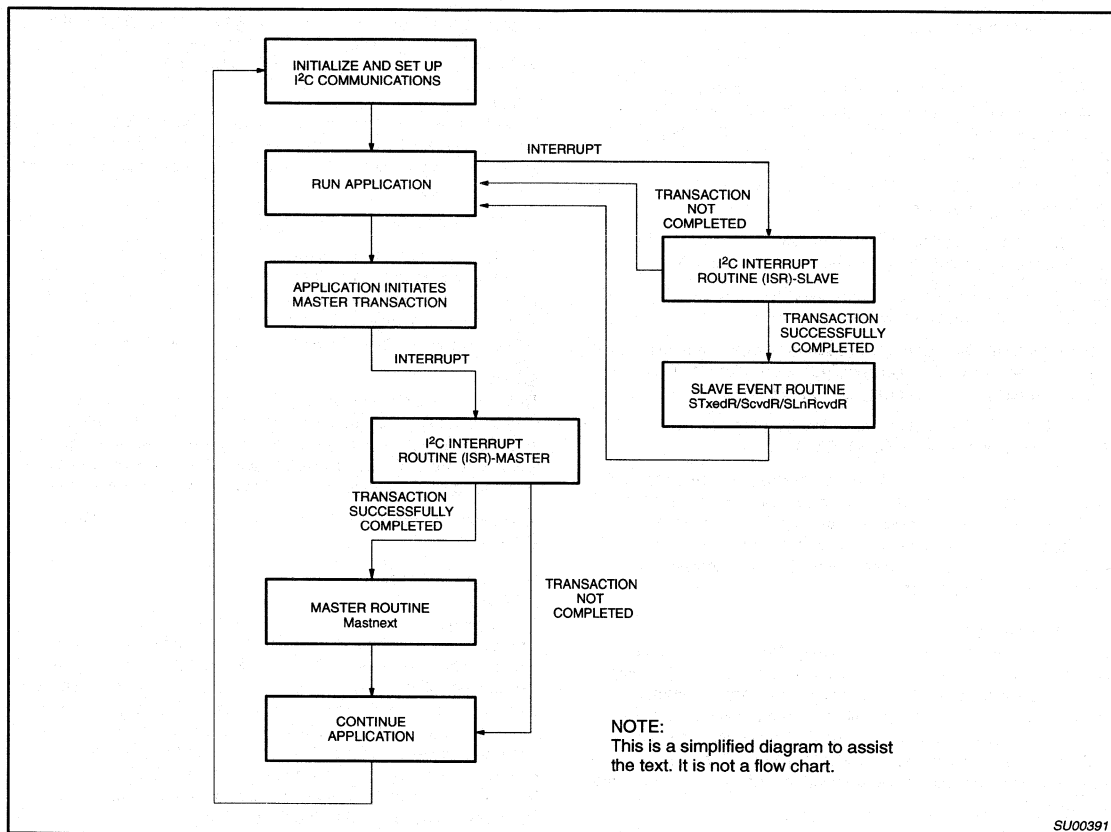


Figure 16. Typical Communications Scenario—A Simplified Diagram

APPLICATION EVENT ROUTINES

The service routine calls Event Routines with pre-defined names (Figure 16), and these routines must be provided by the application program. The actual code of the routines will differ from application to application, but the routine names are being kept the same.

These routines are being called when successful processing of a message (send or receive) is completed. The routines may perform whatever action the application was designed for, which is not necessarily related to the I²C communications mechanism. In addition, the routines may perform the data interface tasks for the I²C port, like emptying buffers from received data or preparing the next message by setting up the buffers.

The mechanism of calling the event routines out of the service routine allows an immediate reaction to the event of message processing completion, before any new activity happens on the bus. In some simple applications this may not be necessary. For example, one may have a main program for a slave which is just a wait loop monitoring a flag set by the service routine when a message transfer, initiated by some master, is completed. In such a case the application could react to the message completion after the interrupt service routine returns. However, in the general case this will not be sufficient. An example could be a slave with an

application which is constantly busy doing another task, in an environment where the communication requests on the I²C bus are frequent. If there is a new message request shortly after the current message is completed, having to wait for the application until it "has time" may result in not reacting, or sending the same data again, or overwriting the received data in the buffer. Another obvious case demanding event routine calls is a Master sending different messages with a Repeated Start—the new data for the following message must be prepared in the interrupt service routine as the current message is completed (there is no return from interrupt prior to the new data transmission).

The programmer has the flexibility to decide where to prepare the next message according to the requirements of the application. This can be done after return from the event routine, in the application code after the return from interrupt, or a combination of both, where the time critical events are performed in the event routines. The application may monitor the MSGSTAT flag for message processing completion. If the event routines are not used, it is recommended to simply code them as a "RET" instruction, thus turning them into dummy routines (this an easier and better practice than changing the service routine itself, eliminating the calls).

Using the 8XC751/752 in multimaster I²C applications

AN430

Master Event Routine:

MastNext

This routine is called by the service routine when the processing of the current Master message is completed. For an indication on the type of message processing completion, MastNext may inspect the contents of MSGSTAT RAM location.

When MastNext is called, MSGSTAT will contain one of the following codes for message processing completion:

MRCVED (= 21h)—a complete message (with number of data bytes indicated by MASTCNT) was received from the slave.

MTXED (= 22h)—the number of data bytes indicated by MASTCNT were successfully sent and acknowledged by the slave.

MTXNAK (= 23h)—the slave did not acknowledge a data byte of the message, even though it had acknowledged its address. The message transmission was terminated upon the NAK.

MTXNOSLV (= 24h)—no slave acknowledged the address indicated by memory location DESTADDR.

The MastNext routine may perform any task(s) necessary for the application. Data handling tasks will typically be dependent on the MSGSTAT indication. One possible task could be setting the directives for the next message. The necessity for executing this task here (versus the main-line code initiating the transfer) is of course application dependent.

Slave Event Routines:

These routines are called when a message transaction as a slave has been completed. In many cases it could be important to utilize the calls to such routines as the requests for message transactions as a slave can come randomly, asynchronous to the application program. The application may demand that new data coming in should immediately initiate some tasks (e.g. control an output port)—and the event routine can be used to process the result of the slave interrupt.

In most cases it will be necessary for a slave to react immediately to a message received simply in order not to lose the data. As a new message may come randomly, it may overwrite the reception buffer before the data has been transferred out of it or acted upon.

For applications in which the reaction for slave events is performed after the return from the service routine, the event is reported by placing an appropriate code in the MSGSTAT flag. The programmer may use event routines, other mainline routines inspecting MSGSTAT, or both. If the event routines are not used, it is recommended to code them as a "RET" instruction.

SRcvdR:

Called by the service routine when a new, complete message has been received into SRcvBuf. When SRcvdR is called, R1 points to the address of the last byte received into the buffer. In a typical application SRcvdR will transfer the new data out of SRcvBuf, so it will not be written over by a subsequent slave reception.

The equivalent MSGSTAT indication for this event is SRCVD (= 11h).

SLnRcvdR:

Called when a slave message has been received into SRcvBuf, but the message was longer than the SRcvBuf buffer (as specified by RBufLen).

The equivalent MSGSTAT indication for this event is SRLNG (= 12h).

If the program is supposed to react to a too long a message the same way as to a message that can be contained in the buffer, one may code SLnRcvdR simply as a call to SRcvdR.

STXedR:

Called by the service routine when data has been transmitted out of the slave STxBuf buffer according to a master's request. This routine may insert new data into the buffer, preparing it for the next slave transmission.

The equivalent MSGSTAT indication for this event is STXED (= 13h).

Note that we do not have a separate routine for the case that the master requested too many bytes—more than STxBuf length—and we sent out meaningless bytes. It is the master's responsibility to specify the message length, and it should be able to request messages with the appropriate length from each slave on the bus.

SRErrR:

This routine relates more to bus communications than to the application itself. It can be called when we positively detect a bus error upon reception as a slave, in case the application is supposed to know about it. In most cases this call will not be used, as dealing with bus communications difficulties is usually left to the Master.

Just prior to calling SRErrR, the code SRERR (= 14h) is placed in MSGSTAT.

I²C Completion Routine:

I2CDONE

This routine is called every time, before returning from the I²C interrupt service routine, whether the transaction was successful or not. It can be used to "safely" monitor MSGSTAT without any risk of a new interrupt modifying the current indication. Simple application programs will not make use of this routine. A more sophisticated application implementing a fail-safe communications protocol may use it to count errors of a certain type in order to determine a recovery scheme. In our programming example, I2CDONE inhibits I²C interrupts when it is evident that as a result of protocol errors interrupts are not caused by legitimate Starts.

CONSTANTS

RBufLen—the length of SRcvBuf, the slave receive buffer. This constant may be used both by the I²C routines and the application program, and it is the responsibility of the application programmer to define the correct buffer length.

MYNUM—This ROM constant is dependent on the application environment. It is a small integer defining a "serial number" of the node, out of all the processors running the same code. This constant is used only when recovering from a timeout, in order to "de-synchronize" masters from each other when trying to recover the bus.

CTVAL1 is a constant defined in ROM. It is used by the application code portion which initializes the I²C, for loading CT0 and CT1 with a value appropriate for the crystal being used.

MYADDR1 is a ROM constant containing the address of the processor's I²C node. This value is used by the application demo to load the RAM location MyAddr.

Using the 8XC751/752 in multimaster I²C applications

AN430

USING THE COMMUNICATIONS SUBROUTINES

In order to use the I²C Communications Routines an application program should take care of the following:

- Upon initialization, load bits CT1, CT0 of I2CFG register according to the clock crystal used (refer to the table of CT1, CT0 values in the 8XC751 section of the Philips Semiconductors Microcontroller Data Handbook (IC20)).
- Load MyAddr RAM location with the address of this node.
- For Slave operation, load STxBuf with the initial data to be transmitted.
- For slave operation, set the SLAVEN bit in the I2CFG register.
- Enable I²C and watchdog interrupts by setting the ETI, EI2 and EA bits of the interrupt enable register.
- For Master operation, set up the next transaction by loading the appropriate directives into MASCMD, DESTADRW, DESSUBAD (if applicable) and MASTCNT, and load MasBuf with the appropriate data if it is a Write message.
- For Master operation, initiate the next transaction by setting MASTRQ bit in I2CFG.
- For both Master and Slave operation, handle data transmission and reception via the buffers in main-line code or the Event Routines.

PROGRAMMING EXAMPLE

The assembler listing includes the I²C Communications Routines and a demo application exercising these routines. In most real-life applications the code of the routines could be used without modifications. For those who follow the coding of the routines, one should note that in many instances code speed and program space have been slightly compromised in order to improve readability. The almost "general purpose" interface to the routines affects efficiency as well, and it is possible to write more compact and somewhat faster code for specific applications. The reader is encouraged, though, to use the code "as is" whenever possible.

The "application" demo is simple—two microcontrollers exchange messages in a "ping-pong" game. In addition to trivial message exchange, the code demonstrates recovery mechanisms from communications errors and bus "hangups". We tried this code with two pairs of controllers exchanging messages on the same bus. The message exchange could repeatedly recover and restart when the SCL and SDA lines were temporarily shorted to ground or between themselves. Simpler versions, without the "protection" mechanisms, could "hang up" under such conditions.

Source Code Available On BBS

The source code file for this program is available for download from the Philips computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PPCODE1      83C751 Multimaster I2C Routines                               4/14/1992      PAGE 1

1      ;
2
3      ;*****
4      ;          Multimaster Code for 83C751/83C752
5      ;          4/14/1992
6      ;*****
7      ; This code was written to accompany an application note. The I2C routines
8      ; are intended to be demonstrative and transportable into different
9      ; application scenarios, and were NOT optimized for speed and/or memory
10     ; utilization.
11     ;
12     ; Yoram Arbel
13
14     $TITLE(83C751 Multi Master I2C Routines)
15     $DATE(4/14/1992)
16     $MOD751
17     $DEBUG
18
19     ;*****
20     ;          8XC751 MULTIMASTER I2C COMMUNICATIONS ROUTINES
21     ;          Symbols and RAM definitions
22     ;*****
23
24     ; Symbols (masks) for I2CFG bits.
25
0010    26     BTIR      EQU      10h      ; TIRUN bit.
0040    27     BMRQ      EQU      40h      ; MASTRQ bit.
28
29
30     ; Symbols (masks) for I2CON bits.
31
0080    32     BCXA      EQU      80h      ; CXA bit.
0040    33     BIDLE     EQU      40h      ; IDLE bit.
0020    34     BCDR      EQU      20h      ; CDR bit.
0010    35     BCARL     EQU      10h      ; CARL bit.
0008    36     BCSTR     EQU      08h      ; CSTR bit.
0004    37     BCSTP     EQU      04h      ; CSTP bit.
0002    38     BXSTR     EQU      02h      ; XSTR bit.
0001    39     BXSTP     EQU      01h      ; XSTP bit.
40
41     ; Note:
42     ;
43     ; Specific bits of the I2CON register are set by writing into this register a
44     ; combination of the masks defined above using the MOV command.
45     ; The SETB command should not be used with I2CON, as it is implemented by
46     ; reading the contents of the register, setting the appropriate bit and
47     ; writing it back into the register. As the functionality of the Read and
48     ; Write portions of the I2CON register is different, using SETB may cause
49     ; unwanted results.
50
51     ; Message transaction status indications in MSGSTAT:
52
0010    53     SGO       EQU      10h      ; Started Slave message processing.

```


Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines			4/14/1992	PAGE 2
0011	54	SRCVD	EQU	11h	; as a slave, received a new message
0012	55	SRLNG	EQU	12h	; received as slave a message which is too
	56				; long for the buffer
0013	57	STXED	EQU	13h	; as slave, completed message transmission.
0014	58	SRERR	EQU	14h	; bus error detected when operating as a slave.
	59				
0020	60	MGO	EQU	20h	; Started Master message processing.
0021	61	MRCVED	EQU	21h	; As Master, received complete message from
	62				; slave.
0022	63	MTXED	EQU	22h	; As Master, completed successful message
	64				; transmission (slave acknowledged all data
	65				; bytes).
0023	66	MTXNAK	EQU	23h	; As Master, truncated message since slave did
	67				; not acknowledge a data byte.
0024	68	MTXNOSLV	EQU	24h	; AS Master, did not receive an acknowledgement
	69				; for the specified slave address.
	70				
0030	71	TIMOUT	EQU	30h	; TIMER1 Timed out.
0032	72	NOTSTR	EQU	32h	; Master did not recognize Start.
	73				
	74				; RAM locations used by I2C interrupt service routines.
	75				
	76				
0020	77	MASCMD	DATA	20h	
0000	78	SUBADD	BIT	MASCMD.0	
0001	79	RPSTRT	BIT	MASCMD.1	
0002	80	SETMRQ	BIT	MASCMD.2	
	81				
0024	82	DSEG	AT	24h	
	83				
0024	84	MSGSTAT:	DS	1	; I2C communications status.
0025	85	MYADDR:	DS	1	; Address of this I2C node.
0026	86	DESTADRW:	DS	1	; Destination address + R/W (for Master).
0027	87	DESSUBAD:	DS	1	; Destination subaddress.
0028	88	MASTCNT:	DS	1	; Number of data bytes in message (Master,
	89				; send or receive).
	90				
0029	91	TITOCNT:	DS	1	; Timer I bus watchdog timeouts counter.
002A	92	StackSave:	DS	1	; SP save location (used when returning from
	93				; bus recovery routine).
	94				
002B	95	MasBuf:	DS	4	; Master receive/transmit buffer, 8 bytes.
002F	96	SRcvBuf:	DS	4	; Slave receive buffer, 8 bytes.
0033	97	STxBuf:	DS	4	; Slave transmit buffer, 8 bytes.
	98				
	99				
	100				
0004	101	RBufLen	EQU	4h	; The length of SRcvBuf
	102				

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PPCODE1      83C751 Multimaster I2C Routines                               4/14/1992      PAGE 3

103 ;*****
104 ;           APPLICATION output pins and RAM definitions
105 ;*****
106
107 ; Outputs used by the application:
108
0090 109 TogLED      BIT      P1.0    ; Toggling output pin, to confirm
110                                     ; that the ping-pong game proceeds fine.
0091 111 ErrLED      BIT      P1.1    ; Error indication.
112
0093 113 OnLED       BIT      P1.3    ;
114
115 ; Application RAM
116
0021 117 APPFLAGS    DATA    21h
0008 118 TRQFLAG     BIT      APPFLAGS.0
119 ; Flag for monitoring I2C transmission success.
0009 120 SErrFLAG    BIT      APPFLAGS.1
121
0037 122 FAILCNT:    DS        1
123
0038 124 TOGCNT:    DS        1          ; Toggle counter.
125
126
127 ;*****
128 ;
129 ;           Program Start
130 ;
131 ;*****
— 132 CSEG
133
134 ; Reset and interrupt vectors.
135
0000 136 AJMP        Reset          ;Reset vector at address 0.
137
138
139 ; A timer I timeout usually indicates a 'hung' bus.
140
001B 141 ORG         1Bh           ; Timer I (I2C timeout) interrupt.
001B D2DD 142 TimerI:    SETB        CLRTI
001D 4111 143 AJMP        TIISR        ; Go to Interrupt Service Routine.
144
145
146
147
148 ;*****
149 ;           I2C Interrupt Service Routine
150 ;*****
151 ;
152 ; Notes on the interrupt mechanism:
153 ;
154 ; Other interrupts are enabled during this ISR upon return from XRETI.
155 ; Limitations imposed on other ISR's:

```

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PCODE1      83C751 Multimaster I2C Routines                                4/14/1992      PAGE 4

156      ; - Should not be long (close to 1000 clock cycles).      A long ISR will cause
157      ;the I2C bus to 'hang', and a TIMERI interrupt to occur.
158      ; - Other interrupts either do not use the same mechanism for allowing
159      ;further interrupts, or if they do - disable TIMERI interrupt beforehand.
160      ;
161      ; The 751 hardware allows only one level of interrupts. We simulate an
162      ; additional level by software: by performing a RETI instruction (at location
163      ; XRETI) the interrupt-in-progress flip-flop is cleared, and other interrupts
164      ; are enabled. The second level of interrupt is a must in our implementation,
165      ; enabling timeout interrupts to occur during "stuck" wait loops in the I2C
166      ; interrupt service routine.
167
168
0023      169      ORG          23h
170
0023 C2AC   171      I2CISR:    CLR          EI2      ; Disable I2C interrupt.
0025 114C   172      ACALL     XRETI      ; Allow other interrupts to occur.
0027 C0D0   173      PUSH      PSW
0029 C0E0   174      PUSH      ACC
002B E8     175      MOV        A,R0
002C C0E0   176      PUSH      ACC
002E E9     177      MOV        A,R1
002F C0E0   178      PUSH      ACC
0031 EA     179      MOV        A,R2
0032 C0E0   180      PUSH      ACC
181
0034 85812A 182      MOV        StackSave, SP
0037 C2DC   183      CLR          TIRUN
0039 D2DC   184      SETB       TIRUN
185
003B 209A09 186      JB          STP,NoGo
003E 30990C 187      JNB        MASTER, GoSlave
0041 752420 188      MOV        MSGSTAT,#MGO
0044 209B76 189      JB          STR,GoMaster
0047 752432 190      NoGo:      MOV        MSGSTAT,#NOTSTR
004A 21AE   191      AJMP       Dismiss      ; Not a valid Start.
192
004C 32     193      XRETI:     RETI
194
195      ;*****
196      ;          Main Transmit and Receive Routines
197      ;*****
198
199      ;          SLAVE CODE -
200      ;          GET THE ADDRESS
201
004D 752410 202      GoSlave:   MOV        MSGSTAT,#SGO
0050 31E2   203      AddrRcv:   ACALL     CIsRcv8
0052 309D5E 204      JNB        DRDY, SMsgEnd ; Must be some strange Start or Stop
205      ; before the address byte was completed.
206      ; Not a valid address.
0055 A2E0   207      STstRW:    MOV        C,ACC.0 ; Save R/W~ bit in carry.
0057 C2E0   208      CLR          ACC.0      ; Clear that bit, leaving "raw" address

```

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 5
0059 6060	209 JZ GoIdle ; If it is a General Address		
	210 ; - ignore it.		
	211		
	212 ; NOTE:		
	213 ; One may insert here a different		
	214 ; treatment for general calls, if		
	215 ; these are relevant.		
	216		
005B 4027	217 JC SlvTx ; It's a Read - (requesting slave		
	218 ; transmit).		
	219		
	220		
	221		
	222		
	223 ; It is a Write (slave should receive the message).		
	224		
	225 ; Check if message is for us		
	226		
005D B5255B	227 SRcv2: CJNE A,MYADDR,GoIdle ; If not my address - ignore the		
	228 ; message.		
0060 792F	229 MOV R1,#SRcvBuf ; Set receive buffer address.		
0062 7A05	230 MOV R2,#RbufLen+1 ;		
0064 8002	231 SJMP SRcv3		
	232		
0066 F7	233 SRcvSto: MOV @R1,A ; Store the byte		
0067 09	234 Inc R1 ; Step address.		
0068 31ED	235 SRcv3: ACALL AckRcv8		
006A 309D09	236 JNB DRDY,SRcvEnd ; Exit loop -end reception.		
006D DAF7	237 DJNZ R2,SRcvSto ; Go to store byte if buffer not full.		
	238		
	239 ; Too many bytes received - do not acknowledge.		
006F 752412	240 MOV MSGSTAT,#SRLNG ; Notify main that (as slave) we		
	241 ; have received too long a message.		
0072 7110	242 ACALL SLnRCvdR ; Handle new data - slave event routine.		
0074 8045	243 SJMP GoIdle		
	244		
	245		
	246		
	247 ; Received a byte, but not DRDY - check if a legitimate message end.		
	248		
0076 B8072E	249 SRcvEnd: CJNE R0,#7,SRcvErr ; If bit count not 7, it was not		
	250 ; a Start or a Stop.		
	251		
	252 ; Received a complete message		
	253		
	254		
0079 752411	255 MOV MSGSTAT,#SRCVD ; Calculate number of bytes received		
	256		
007C E9	257 MOV A,R1		
007D C3	258 CLR C		
007E 942F	259 SUBB A,#SRcvBuf ; number of bytes in ACC		
0080 51EF	260 ACALL SRCvdR ; Handle new data - slave event routine.		
0082 802F	261 SJMP SMsgEnd		

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 6
	262		
	263		
	264		; It is a Read message, check if for us.
	265		
0084 00	266	SivTx: NOP	
	267		
0085 B52533	268	STx2: CJNE A,MYADDR,Goldle	; Not for us.
0088 759900	269	MOV I2DAT,#0	; Acknowledge the address.
008B 309EFD	270	JNB ATN,\$; Wait for attention flag.
008E 309D22	271	JNB DRDY,SMsgEnd	; Exception – unexpected Start
	272		; or Stop before the Ack got out.
0091 7933	273	MOV R1,#STxBuf	; Start address of transmit buffer.
0093 E7	274	STxlp: MOV A,@R1	; Get byte from buffer
0094 09	275	INC R1	
0095 31CE	276	ACALL XmByte	
0097 309D19	277	JNB DRDY,SMsgEnd	; Byte Tx not completed.
009A 309FF6	278	JNB RDAT,STxlp	; Byte acknowledge, proceed trans.
009D 759860	279	MOV I2CON,#BCDR+BIDLE	; Master Nak'ed for msg end.
00A0 752413	280	MOV MSGSTAT,#STXED	
00A3 7110	281	ACALL STXedR	; Slave transmitted event routine.
00A5 21AE	282	AJMP Dismiss	
	283		
	284		
00A7 752414	285	SREvErr: MOV MSGSTAT,#SRERR	; Flag bus/protocol error
00AA 7110	286	ACALL SRErrR	; Slave error event routine.
00AC 8005	287	SJMP SMsgEnd	
00AE 752414	288	StxErR: MOV MSGSTAT,#SRERR	; Flag bus/protocol error
00B1 7110	289	ACALL SRErrR	
	290		
00B3 209903	291	SMsgEnd: JB MASTER,SMsgEnd2	
00B6 209B94	292	JB STR,GoSlave	; If it was a Start, be Slave
00B9	293	SMsgEnd2:	
00B9 21AE	294	AJMP Dismiss	
	295		
	296		
	297		; End of Slave message processing
	298		
00BB	299	GoIdle:	
00BB 21AE	300	AJMP Dismiss	
	301		
	302		
	303		
	304		
	305	;	
	306	;	
	307		
00BD	308	GoMaster:	
	309		
	310		
	311	; Send address & R/W~ byte	
	312		
00BD 792B	313	MOV R1,#MasBuf	; Master buffer address
00BF AA28	314	MOV R2,MASTCNT	; # of bytes, to send or rcv

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 7
00C1 E526	315 MOV A,DESTADRW		; Destination address (including
	316		; R/W~ byte).
00C3 200012	317 JB SUBADD,GoMas2		; Branch if subaddress is needed.
	318		
00C6 31C5	319 ACALL XmAddr		
	320		
00C8 309D03	321 JNB DRDY,GM2		
00CB 309C02	322 JNB ARL,GM3		
00CE 2186	323 GM2: AJMP AdTxArl		; Arbitration loss while transmitting
	324		; the address.
00D0 209F5C	325 GM3: JB RDAT,Noslave		; No Ack for address transmission.
00D3 20E063	326 JB ACC.0, MRcv		; Check R/W~ bit
00D6 211A	327 AJMP MTx		
	328		
	329		; Handling subaddress case:
	330		
00D8 00	331 GoMas2: NOP		; Subaddress needed. Address in ACC.
00D9 C2E0	332 CLR ACC.0		; Force a Write bit with address.
00DB 31C5	333 ACALL XmAddr		
00DD 309D03	334 JNB DRDY,GM4		
00E0 309C02	335 JNB ARL,GM5		
00E3 2186	336 GM4: AJMP AdTxArl		; Arbitration loss while transmitting
	337		; the address.
	338		
00E5 209F47	339 GM5: JB RDAT,Noslave		; No Ack for address transmission.
00E8 E527	340 MOV A,DESSUBAD		
00EA 31CE	341 ACALL XmByte		; Transmit subaddress.
00EC 309DCA	342 JNB DRDY,SMsgEnd2		; Arbitration loss (by Start or Stop)
00EF 209CC7	343 JB ARL,SMsgEnd2		; Arbitration loss occurred.
00F2 209F3F	344 JB RDAT,NoAck		; Subaddress transmission was not ack'ed.
00F5 E526	345 MOV A,DESTADRW		; Reload ACC with address.
00F7 30E020	346 JNB ACC.0, MTx		; It's a Write, so proceed
	347		; by sending the data.
	348		
	349		; Read message, needs rp. Start and add. retransmit.
00FA 759822	350 MOV I2CON,#BCDR+BXSTR		; Send Repeated Start.
00FD 309EFD	351 JNB ATN,\$		
0100 759820	352 MOV I2CON,#BCDR		; Clear useless DRDY while preparing
	353		; for Repeated Start.
0103 309EFD	354 JNB ATN,\$; expecting an STR.
0106 309C02	355 JNB ARL,GM6		
0109 2182	356 AJMP MArlEnd		; oops – lost arbitration.
010B 31C5	357 GM6: ACALL XmAddr		; Retransmit address, this time with the
	358		; Read bit set.
010D 309D03	359 JNB DRDY,GM7		
0110 309C02	360 JNB ARL,GM8		
0113 2186	361 GM7: AJMP AdTxArl		; Arbitration loss while transmitting
	362		; the address.
0115 209F17	363 GM8: JB RDAT,Noslave		; No Ack – the slave disappeared.
0118 801F	364 SJMP MRcv		; Proceed receiving slave's data.
	365		
	366		; A Write message. Master transmits the data.
	367		

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PPCODE1      83C751 Multimaster I2C Routines      4/14/1992      PAGE 8

011A 00      368      MTx:      NOP
              369
011B E7      370      MTxLoop:  MOV      A,@R1      ; Get byte from buffer.
011C 09      371      INC      R1          ; Step the address.
011D 31CE    372      ACALL   XmByte
011F 309D97  373      JNB     DRDY,SMsEnd2 ; Arbitration loss (by Start or Stop)
0122 209C94  374      JB     ARL,SMsEnd2  ; Arbitration loss.
0125 209F0C  375      JB     RDAT,NoAck
0128 DAF1    376      DJNZ   R2,MTxLoop   ; Loop if more bytes to send.
              377
012A 752422  378      MOV     MSGSTAT,#MTXED ; Report completion of buffer
              379      ; transmission.
012D 8025    380      SJMP   MTxStop
012F 752424  381      NoSlave: MOV    MSGSTAT,#MTXNOSLV
0132 8020    382      SJMP   MTxStop
0134 752423  383      NoAck:  MOV    MSGSTAT,#MTXNAK
0137 801B    384      SJMP   MTxStop
              385
              386
              387
              388      ; Master receive – a Read frame
              389
0139 31F6    390      MRcv:   ACALL   ClaRcv8      ; Receive a byte.
013B 8002    391      SJMP   MRcv2
013D 31ED    392      MRcvLoop: ACALL  AckRcv8
013F 309D39  393      MRcv2:  JNB     DRDY,MAr1      ; Other's Start or Stop.
0142 F7      394      MOV     @R1,A          ; Store received byte.
0143 09      395      INC     R1            ; Advance address.
0144 DAF7    396      DJNZ   R2,MRcvLoop
              397
              398      ; Received the desired number of bytes – send Nack.
              399
0146 759980  400      MOV     I2DAT,#80h
0149 309EFD  401      JNB     ATN,$
014C 309D2C  402      JNB     DRDY,MAr1
014F 752421  403      MOV     MSGSTAT,#MRCVED
0152 8000    404      SJMP   MTxStop      ; Go to send Stop or Repeated Start.
              405
              406
              407
              408      ; Conclude this Master message:
0154 300105  409      ; Send Stop, or a Repeated Start
              410
              411
0157 759822  412      MTxStop: JNB     RPSTRT,MTxStop2 ; Check if Repeated Start needed
              413      ; Around if not RPSTRT.
015A 8007    414      MOV     I2CON,#BCDR+BXSTR ; Send Repeated Start.
015C A202    415      SJMP   MTxStop3
015E 92DE    416      MTxStop2: MOV    C,SETMRQ      ; Set new Master Request if demanded
0160 759821  417      MOV     MASTRQ,C      ; by SETMRQ bit of MASCMD.
              418
              419      MOV     I2CON,#BCDR+BXSTP ; Request the HW to send a Stop.
              420

```

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 9
0163 309EFD	421 MTxStop3: JNB ATN,\$; Wait for Attention
0166 759820	422 MOV I2CON,#BCDR		; Clear the useless DRDY, generated
	423		; by SCL going high in preparation
	424		; for thr Stop or Repeated Start.
0169 309EFD	425 JNB ATN,\$; Wait for ARL, STP or STR.
016C 209C13	426 JB ARL,MarlEnd		; Lost arbitration trying to send
	427		; Stop or a ReStart.
	428		
	429 ; Master is done with this message.		May proceed with new messages, if any,
	430 ; or exit.		
	431		
016F 7112	432 ACALL MastNext		; Master Event Routine. May Prepare
	433		; the pointers and data for the
	434 ;		next Master message.
	435		
0171 30DE05	436 JNB MASTRQ,MMsgEnd		; Go end service routine if MASTRQ
	437		; does not indicate that the master
	438		; should continue (was set according
	439		; to SETMRQ bit, or by MastNext).
	440		
0174 309B02	441 JNB STR,MMsgEnd		; Return from the ISR, unless Start
	442		; (avoid danger if we do not return:
	443		; if there was a Stop, the watchdog
	444		; is inactive until next Start).
0177 01BD	445 AJMP GoMaster		; Loop for another Master message
	446 ;		
0179	447 MMsgEnd:		; End of Master messages,
0179 8033	448 SJMP Dismiss		
	449		
	450		
	451		
	452		
	453 ; Terminate mastership due to an arbitration loss:		
	454		
017B	455 MArl:		
	456		
017B 309B02	457 JNB STR,MArl2		; If lost arbitration due to other
	458		; Master's Start, go be a slave.
017E 014D	459 AJMP GoSlave		
	460		
0180	461 MArl2:		
0180 21AE	462 AJMP Dismiss		
	463		
	464		
	465		
	466 ; Switch from Master to Slave due to arbitration loss after completing		
	467 ; transmission of a message. The MASTRQ bit was cleared trying to write a		
	468 ; Stop, and we need to set it again on order to retry transmission when the		
	469 ; bus gets free again.		
	470		
0182	471 MArlEnd:		
0182 D2DE	472 SETB MASTRQ		; Set Master Request – which will get
	473		; into effect when we are done as a
	474		; slave.

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751	Multimaster I2C Routines	4/14/1992	PAGE 10
0184 217B	475	AJMP	MAr1	
	476			
	477			; Handling arbitration loss while transmitting an address
	478			
0186 209BF2	479	AdTxAr1:	JB STR,MAr1	; Non-synchronous Start or Stop.
0189 209AEF	480	JB	STP,MAr1	
	481			
	482			; Switch from Master to Slave due to arbitration loss while transmitting
	483			; an address – complete receiving the address transmitted by the new Master.
	484			
018C B80003	485	CJNE	R0,#0,AdTxAr12	
	486			; Ar1 on last bit of address
	487			; (R0 is 0 on exit from XmAddr).
018F 14	488	DEC	A	; The lsb sent, in which ar1 occurred
	489			; must have been 1. By decrementing
	490			; A we get the address that won.
0190 8012	491	SJMP	AdAr3	
	492			
0192	493	AdTxAr12:		
0192 03	494	RR	A	; Realign partially Tx'ed ACC
0193 F9	495	MOV	R1,A	; and save it in R1
0194 E8	496	MOV	A,R0	; Pointer for lookup table
0195 9001A6	497	MOV	DPTR,#MaskTable	
0198 93	498	MOVC	A,@A+DPTR	
0199 59	499	ANL	A,R1	; Set address bits to be received,
	500			; and the bit on which we lost
	501			; arbitration to 0
	502			; Now we are ready to receive the rest
	503			; of the address.
	504			
	505			
019A 759890	506	MOV	I2CON,#BCXA+BCARL	; Clear flags and release the clock.
	507			
019D 5108	508	ACALL	RBit3	; Complete the address using reception
	509			; subroutine.
019F 209D02	510	JB	DRDY,AdAr3	; Around if received address OK
01A2 01B3	511	AJMP	SMsgEnd	; Unexpected Start or Stop – end
	512			; as a slave.
01A4 0155	513	AdAr3:	AJMP STstRW	; Proceed to check the address
	514			; as a slave.
	515			
01A6 FF7E3E1E	516	MaskTable:	DB Offh,7Eh,3Eh,1Eh,0Eh,06h,02h,00h;	; Offh is dummy
01AA 0E06200				
	517			
	518			; End I2C Interrupt Service Routine:
	519			
01AE 711E	520	Dismiss:	ACALL I2CDONE	
	521			
01B0 7598F4	522	MOV	I2CON,#BCARL+BCSTP+BCDR+BCXA+BIDLE	
01B3 C2DC	523	CLR	TIRUN	
01B5 D0E0	524	POP	ACC	
01B7 FA	525	MOV	R2,A	
01B8 D0E0	526	POP	ACC	

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PPCODE1      83C751 Multimaster I2C Routines      4/14/1992      PAGE 11

01BA F9      527    MOV      R1,A
01BB D0E0    528    POP      ACC
01BD F8      529    MOV      R0,A
01BE D0E0    530    POP      ACC
01C0 D0D0    531    POP      PSW
01C2 D2AC    532    SETB     EI2
            533
01C4 22      534    RET      ; Return from I2C interrupt Service Routine
            535
            536    ;*****
            537    ;          Byte Transmit and Receive Subroutines
            538    ;*****
            539
            540
            541
            542    ;          XmAddr: Transmit Address and R/W~
            543    ;          XmByte: Transmit a byte
            544
01C5 F599    545    XmAddr:  MOV     I2DAT,A          ; Send first bit, clears DRDY.
01C7 75981C  546    MOV     I2CON,#BCARL+BCSTR+BCSTP
            547    ; Clear status, release SCL.
01CA 7808    548    MOV     R0,#8          ; Set R0 as bit counter
01CC 8004    549    SJMP   XmBit2
01CE 7808    550    XmByte: MOV     R0,#8
01D0 F599    551    XmBit:  MOV     I2DAT,A          ; Send the first bit.
01D2 23      552    XmBit2: RL      A          ; Get next bit.
01D3 309EFD  553    JNB     ATN,$          ; Wait for bit sent.
01D6 309D08  554    JNB     DRDY,XmBex     ; Should be data ready.
01D9 D8F5    555    DJNZ   R0,XmBit       ; Repeat until all bits sent.
01DB 7598A0  556    MOV     I2CON,#BCDR+BCXA    ; Switch to receive mode.
01DE 309EFD  557    JNB     ATN,$          ; Wait for acknowledge bit.
            558    ; flag cleared.
01E1 22      559    XmBex:  RET
            560
            561    ;
            562    ; Byte receive routines.
            563    ;
            564    ; ClsRcv8      clears the status register (from Start condition)
            565    ;              and then receives a byte.
            566    ; AckRcv8     Sends an acknowledge, and then receives a new byte.
            567    ;              If a Start or Stop is encountered immediately after the
            568    ;              ack, AckRcv8 returns with 7 in R0.
            569    ; ClaRcv8     clears the transmit active state and releases clock
            570    ;              (from the acknowledge).
            571    ;
            572    ;              A contains the received byte upon return.
            573    ;              R0 is being used as a bit counter.
            574    ;
            575
01E2 75989C  576    ClsRcv8: MOV     I2CON,#BCARL+BCSTR+BCSTP+BCXA
            577    ; Clear status register.
01E5 309EFD  578    JNB     ATN,$
01E8 309D22  579    JNB     DRDY,RCVex

```

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PCCODE1      83C751 Multimaster I2C Routines      4/14/1992      PAGE 12

01EB 800F      580      SJMP          Rcv8
                    581
01ED 759900    582      AckRcv8:      MOV      I2DAT,#0      ; Send Ack (low)
01F0 309EFD    583      JNB          ATN,$
01F3 309D18    584      JNB          DRDY,RCVerr      ; Bus exception – exit.
01F6 7598A0    585      ClaRcv8:      MOV      I2CON,#BCDR+BCXA      ; clear status, release clock
                    586      ;from writing the Ack.

01F9 309EFD    587      JNB          ATN,$
                    588
01FC 7807      589      Rcv8:         MOV      R0,#7      ; Set bit counter for the first seven
                    590      ; bits.
01FE E4        591      CLR          A      ; Init received byte to 0.
01FF 4599      592      RBit:        ORL      A,I2DAT      ; Get bit, clear ATN.
0201 23        593      RBit2:       RL      A      ; Shift data.
0202 309EFD    594      JNB          ATN,$      ; Wait for next bit.
0205 309D05    595      JNB          DRDY,RCVex      ; Exit if not a data bit (could be Start/
                    596      ; Stop, or bus/protocol error)
0208 D8F5      597      RBit3:       DJNZ     R0,RBit      ; Repeat until 7 bits are in.
020A A29F      598      MOV          C,RDAT      ; Get last bit, don't clear ATN.
020C 33        599      RLC          A      ; Form full data byte.
020D 22        600      RCVex:       RET
                    601
020E 7809      602      RCVerr:      MOV      R0,#9      ; Return non legitimate bit count
0210 22        603      RET
                    604
                    605
                    606      ;*****
                    607      ;           Timer I Interrupt Service Routine
                    608      ;           I2C us Timeout
                    609      ;*****
                    610
                    611      ; In addition to reporting the timeout in MSGSTAT, we update a failure
                    612      ; counter, TITOCNT. This allows different types of timeout handling by the
                    613      ; main program.
                    614
0211 C2DE      615      TIISR:       CLR      MASTRQ      ; "Manual" reset.
0213 759801    616      MOV          I2CON,#BXSTP      ;
0216 7598BC    617      MOV          I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP
                    618
0219 752430    619      TI1:         MOV      MSGSTAT,#TIMOUT      ; Status Flag for Main.
021C 74FF      620      TI2:         MOV      A,#0FFh
021E B52902    621      CJNE        A,TITOCNT,TI3      ; Increment TITOCNT, saturating
0221 8002      622      SJMP        TI4      ; at FFh.
0223 0529      623      TI3:         INC      TITOCNT
                    624
0225 5130      625      TI4:         ACALL   RECOVER
                    626
0227 D2DD      627      SETB        CLRTI      ; Clear TI interrupt flag.
0229 114C      628      ACALL      XRETI      ; Clear interrupt pending flag (in
                    629      ; order to re-enable interrupts).
022B 852A81    630      MOV          SP,StackSave      ; Realign stack pointer, re-doing
                    631      ; possible stack changes during
                    632      ; the I2C interrupt service routine.

```

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PCODE1      83C751 Multimaster I2C Routines                4/14/1992      PAGE 13

633                ;TimerI interrupts in other ISR's
634                ;were not allowed !
022E 21AE      635      AJMP      Dismiss      ; Go back to the I2C service routine,
636                ; in order to return to the (main)
637                ; program interrupted.
638
639
640      ;*****
641      ;          Bus recovery attempt subroutine
642      ;*****
643
0230 C2AF      644      RECOVER:  CLR      EA
0232 C2DE      645      CLR      MASTRQ          ; "Manual" reset.
0234 7598FC    646      MOV      I2CON,#BCXA+BIDLE+BCDR+BCARL+BCSTR+BCSTP
0237 C2DF      647      CLR      SLAVEN          ; Non I2C TimerI mode
0239 D2DC      648      SETB     TIRUN          ; Fire up TimerI. When it overflows, it
649                ; will cause I2C interface hardware reset.
023B 79FF      650      MOV      R1,#0fh
023D 00        651      DLY5:   NOP
023E 00        652      NOP
023F 00        653      NOP
0240 D9FB      654      DJNZ     R1,DLY5
0242 C2DC      655      CLR      TIRUN
0244 D2DD      656      SETB     CLRTI
657
0246 D280      658      SETB     SCL          ; Issue clocks to help release other devices.
0248 D281      659      SETB     SDA
024A 7908      660      MOV      R1,#08h
024C C280      661      RC7:   CLR      SCL
024E 00000000  662      DB      0,0,0,0
0252 00
0253 D280      663      SETB     SCL
0255 00000000  664      DB      0,0,0,0
0259 00
025A D9F0      665      DJNZ     R1,RC7
025C C280      666      CLR      SCL
025E 0000      667      DB      0,0
0260 C281      668      CLR      SDA
0262 0000      669      DB      0,0
0264 D280      670      SETB     SCL
0266 00000000  671      DB      0,0,0,0
026A 00
026B D281      672      SETB     SDA
026D 00000000  673      DB      0,0,0,0          ; Issue a Stop.
0271 00
674
0272 7598BC    675      Rex:   MOV      I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP ; clear flags
0275 D2AF      676      SETB     EA
0277 22        677      RET
678

```

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PPCODE1      83C751 Multimaster I2C Routines                                4/14/1992      PAGE 14

679          ;*****
680          ;
681          ;           Main Program
682          ;
683          ;*****
684
685          ; Message ping pong game. Each message is transmitted by
686          ; a processor that is a master on the I2C bus, and it contains one byte
687          ; of data. A processor that receives this data byte as a slave increments
688          ; the data by one and transmits it back as a master. The data received is
689          ; confirmed to be a one increment of the data formerly sent, unless
690          ; it is a "reset" value, chosen to be 00h.
691          ; The two participating processors have similar code, where the node
692          ; address of the second processor is the destination address of this
693          ; one, and vice versa.
694          ; The first data byte each processor tries to send is 00h. One of the
695          ; processors will acquire the bus first, and the second processor that will
696          ; receive this "resetting" 00h will not attempt to confirm it against an
697          ; expected value. It will simply increment and transmit it. Subsequent
698          ; receptions will be confirmed against the expected value, until 0ffh data
699          ; bytes are sent and the game is effectively reset by the 00h resulting from
700          ; the next increment.
701          ; A toggling output (TogLED) tells the outer world that the "ping pong"
702          ; proceeds well. If something unexpected happens we temporarily activate
703          ; another output, ErrLED.
704          ; The different tasks of the code are performed in a combination of main-
705          ; line program and event routines called from the I2C interrupt service
706          ; routine.
707
708
709          ; Initial set-ups:
710          ; Load CT1,CT0 bits of I2CFG register, according to the clock
711          ; crystal used.
712          ; Load RAM location MYADDR with the I2C address of this processor.
713          ; We load these values out of ROM table locations (R_CTVAL and R_MYADDR).
714          ; One may, instead, load with a MOV <immediate> command.
715
0278 758107 716  Reset:      MOV      SP,#07h ;Set stack location.
027B E4     717  CLR        A
027C 90032D 718  MOV        DPTR,#R_CTVAL
027F 93     719  MOVC      A,@A+DPTR
0280 F5D8   720  MOV        I2CFG,A          ; Load CT1,CT0 (I2C timing, crystal
                                ; dependent).
0282 E4     722  CLR        A
0283 90032C 723  MOV        DPTR,#R_MYADDR
0286 93     724  MOVC      A,@A+DPTR          ; Get this node's address from ROM table
0287 F525   725  MOV        MYADDR,A          ; into MYADDR RAM location.
                                726
0289 C293   727  CLR        OnLED
                                728
                                729
028B C291   730  Reset2:    CLR        ErrLED ; Flash LED.
028D 51E6   731  ACALL     LDELAY

```

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751	Multimaster I2C Routines	4/14/1992	PAGE 15
028F D291	732	SETB ErrLED		
0291 C209	733	CLR SErrFLAG		
0293 C208	734	CLR TRQFLAG		
0295 753750	735	MOV FAILCNT,#50h		
0298 D290	736	SETB TogLED		
029A 753850	737	MOV TOGCNT,#050h		; Initialize pin-toggling counter
	738			
	739			; Enable slave operation.
	740			; The Idle bit is set here for a restart situation – in normal
	741			; operation this is redundant, as this bit is set upon power_up reset.
029D 759840	742	MOV I2CON,#BIDLE		; Slave will idle till next Start.
02A0 D2DF	743	SETB SLAVEN		; Enable slave operation.
	744			
	745			; Enable interrupts.
	746			; This is necessary for both Slave and Master operations.
02A2 D2AB	747	SETB ETI		; Enable timer I interrupts.
02A4 D2AC	748	SETB EI2		; Enable I2C port interrupts.
02A6 D2AF	749	SETB EA		; Enable global interrupts.
	750			
	751			; Set up Master operation.
	752			
02A8 752000	753	MOV MASCMD,#0h		; "Regular" master transmissions.
02AB 90032E	754	MOV DPTR,#PongADDR		
02AE E4	755	CLR A		
02AF 93	756	MOVC A,@A+DPTR		
02B0 F526	757	MOV DESTADRW,A		; The partner address. The LSB is
	758			; low, for a Write transaction.
02B2 752801	759	MOV MASTCNT,#01h		; Message length – a single byte.
	760			
02B5	761	PPSTART:		
02B5 752B00	762	MOV MasBuf,#00h		
	763			
	764			; "Ping" transmission:
	765			
02B8	766	PP2:		
02B8 D208	767	SETB TRQFLAG		
02BA D2DE	768	SETB MASTRQ		
02BC 79FF	769	MOV R1,#0ffh		
02BE 300809	770	PP22: JNB TRQFLAG,PP3		; Transmitted OK
02C1 D9FB	771	DJNZ R1,PP22		
02C3 D537F2	772	MFAIL1: DJNZ FAILCNT,PP2		
02C6 5130	773	ACALL RECOVER		
02C8 80C1	774	SJMP Reset2		
	775			
	776			; "Pong" reception:
	777			
02CA 78FF	778	PP3: MOV R0,#0ffh		; Software timeout loop count.
02CC 79FF	779	PP31: MOV R1,#0ffh		
02CE 2008E7	780	PP32: JB TRQFLAG,PP2		; Rcvd ok as slave, go transmit.
02D1 200908	781	JB SErrFLAG,PP5		
02D4 D9F8	782	DJNZ R1,PP32		
02D6 D8F4	783	DJNZ R0,PP31		
02D8 5130	784	PPTO: ACALL RECOVER		; Software timeout.

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PPCODE1      83C751 Multimaster I2C Routines                               4/14/1992      PAGE 16

02DA 418B    785    AJMP      Reset2
              786
02DC C291    787    PP5:      CLR      ErrLED ; Receive error.
02DE 51E6    788    ACALL    LDELAY
02E0 D291    789    SETB    ErrLED
02E2 C209    790    CLR     SErrFLAG
02E4 41B5    791    AJMP    PPSTART
              792
02E6 7A30    793    LDELAY:  MOV     R2,#030h
02E8 79FF    794    LDELAY1: MOV    R1,#0ffh
02EA D9FE    795    DJNZ    R1,$
02EC DAFA    796    DJNZ    R2,LDELAY1
02EE 22      797    RET
              798
              799    ;*****
800          ;                               Slave and Master Event Routines.
801          ;*****
802
803          ;
804          ;Invoked upon completion of a message transaction.
805          ;This is the part of the application program actually dealing
806          ;with the data communicated on the I2C bus, by responding to
807          ;new data received and/or preparing the next transaction.
808
809
810          ; Slave Event Routines
811          ;
812          ; These routines are invoked by the I2C interrupt service routine when a
813          ; message transaction as a slave has been completed. Our "application"
814          ; reacts to a message received as a slave with the routine SRCvdR.
815          ; The calls that indicate erroneous reception are treated the same way as
816          ; erroneous data reception in the "ping pong" game.
817
818          ;SRCvdR
819          ;Invoked when a new message has been received as a Slave.
820
02EF 00      821    SRCvdR:  NOP
02F0 E52F    822    MOV     A,SRCvBuf
02F2 7005    823    JNZ     SR2
02F4 752B01  824    MOV     MasBuf,#01h ; It was ping-pong reset value
02F7 800F    825    SJMP    SR3
              826
02F9 052B    827    SR2:    INC     MasBuf ; The expected data.
02FB B52B0F  828    CJNE   A,MasBuf,ErrSR
02FE 052B    829    INC     MasBuf ; Data for next transmission - the data
              830          ; received incremented by 1.
              831
              832          ;A successful two way data exchange. Let the outside world know by
              833          ;toggling an output pin driving a LED. We actually toggle only
              834          ;when a number of such exchanges is completed, in order to
              835          ;slow down the changes for a good visual indication.
              836

```

Using the 8XC751/752 in multimaster I²C applications

AN430

```

PCODEE1      83C751 Multimaster I2C Routines      4/14/1992      PAGE 17

0300 D53805   837   DJNZ       TOGCNT,SR3
0303 B290     838   CPL         TogLED       ; Toggle output
0305 753850   839   MOV         TOGCNT,#050h   ;
           840
0308 C209     841   SR3:        CLR         SErrFLAG
030A D208     842   SETB       TRQFLAG       ; Request main to transmit
030C 22       843   RET
           844
030D D209     845   ErrSR:     SETB         SErrFLAG
030F 22       846   RET
           847
           848
           849   ;SLnRcvdR
           850   ;Invoked when a message received as a Slave is too long
           851   ;for the receive buffer.
           852
           853   ;STXedR
           854   ;Invoked when a Slave completed transmission of its buffer.
           855   ;We do not expect to get here, since we do not plan to have
           856   ;in our system a master that will request data from this node.
           857   ;
           858
           859   ;SRErrR
           860   ;Slave error event subroutine.
           861   ;In most applications it will not be used.
           862   ;
           863
0310          864   SLnRcvdR:
0310          865   STXedR:
0310 80FB      866   SRErrR:    JMP         ErrSR
           867
           868
           869   ;
           870   ;MastNext – Master Event Routine.
           871   ;
           872   ;Invoked when a Master transaction is completed, or terminated
           873   ;"willingly" due to lack of acknowledge by a slave.
           874   ;
           875
0312          876   MastNext:
0312 E524      877   MOV         A,MSGSTAT
0314 B42206   878   CJNE       A,#MTXED,MN1
0317 753750   879   MOV         FAILCNT,#50h
031A C208     880   CLR         TRQFLAG
031C 22       881   RET
031D          882   MN1:
031D 22       883   RET
           884
           885   ;I2CDONE
           886   ;Called upon completion of the I2C interrupt service routine.
           887   ;In this example it monitors exceptions, and invokes the bus
           888   ;recovery routine when too many occurred.
           889

```


Using the 8XC751/752 in multimaster I²C applications

AN430

```

PCODE1      83C751 Multimaster I2C Routines                                4/14/1992      PAGE 18

031E      890      I2CDONE:
031E E524  891      MOV          A,MSGSTAT
0320 B43208 892      CJNE         A,#NOTSTR,I2CD1
0323 D53705 893      DJNZ          FAILCNT,I2CD1
0326 753701 894      MOV          FAILCNT,#01h      ; Too many "illegal" i2c interrupts
0329 C2AC   895      CLR          EI2              ; - shut off.
032B 22     896      I2CD1:         RET
           897
           898
           899      ;*****
           900      ;                      I2C Communications Table:
           901      ;*****
           902
           903
           904
           905      ; We used table driven values for clarity. one may use immediate to load
           906      ; these values and save several lines of code.
           907
           908      ; Contents is used in the beginning of the main program to load
           909      ; RAM location MYADDR and the I2CFG register.
           910      ; The node address, in R_MYADDR, is application specific, and unique for
           911      ; each device in the I2C network.
           912      ; R_CTVAL depends on the crystal clock frequency.
           913
032C 4E    914      R_MYADDR:  DB          4Eh      ; This node's address
           915
032D 02    916      R_CTVAL:   DB          02h      ; CT1, CT0 bit values
           917
           918      ;*****
           919      ;                      Application Code Definitions
           920      ;*****
           921
032E 4A    922      PongADDR:  DB          4Ah      ; The address of the "partner" in
           923      ; the ping-pong game.
           924
           925
           926
           927
           928      END
           929

VERSION 1.2h ASSEMBLY COMPLETE, 0 ERRORS FOUND

```

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 19
ACC	D ADDR	00E0H	PREDEFINED
ACKRCV8	C ADDR	01EDH	
ADAR3	C ADDR	01A4H	
ADDRRCV	C ADDR	0050H	NOT USED
ADTXARL	C ADDR	0186H	
ADTXARL2	C ADDR	0192H	
APPFLAGS	D ADDR	0021H	
ARL	B ADDR	009CH	PREDEFINED
ATN	B ADDR	009EH	PREDEFINED
BCARL	NUMB	0010H	
BCDR	NUMB	0020H	
BCSTP	NUMB	0004H	
BCSTR	NUMB	0008H	
BCXA	NUMB	0080H	
BIDLE	NUMB	0040H	
BMRQ	NUMB	0040H	NOT USED
BTIR	NUMB	0010H	NOT USED
BXSTP	NUMB	0001H	
BXSTR	NUMB	0002H	
CLARCV8	C ADDR	01F6H	
CLRTI	B ADDR	00DDH	PREDEFINED
CLSRCV8	C ADDR	01E2H	
DESSUBAD	D ADDR	0027H	
DESTADRW	D ADDR	0026H	
DISMISS	C ADDR	01AEH	
DLY5	C ADDR	023DH	
DRDY	B ADDR	009DH	PREDEFINED
EA	B ADDR	00AFH	PREDEFINED
EI2	B ADDR	00ACH	PREDEFINED
ERRLED	B ADDR	0091H	
ERRSR	C ADDR	030DH	
ETI	B ADDR	00ABH	PREDEFINED
FAILCNT	D ADDR	0037H	
GM2	C ADDR	00CEH	
GM3	C ADDR	00D0H	
GM4	C ADDR	00E3H	
GM5	C ADDR	00E5H	
GM6	C ADDR	010BH	
GM7	C ADDR	0113H	
GM8	C ADDR	0115H	
GOIDLE	C ADDR	00BBH	
GOMAS2	C ADDR	00D8H	
GOMASTER	C ADDR	00BDH	
GOSLAVE	C ADDR	004DH	
I2CD1	C ADDR	032BH	
I2CDONE	C ADDR	031EH	

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 20
I2CFG	D ADDR	00D8H	PREDEFINED
I2CISR	C ADDR	0023H	NOT USED
I2CON	D ADDR	0098H	PREDEFINED
I2DAT	D ADDR	0099H	PREDEFINED
LDELAY	C ADDR	02E6H	
LDELAY1	C ADDR	02E8H	
MARL	C ADDR	017BH	
MARL2	C ADDR	0180H	
MARLEND	C ADDR	0182H	
MASBUF	D ADDR	002BH	
MASCMD	D ADDR	0020H	
MASKTABLE	C ADDR	01A6H	
MASTCNT	D ADDR	0028H	
MASTER	B ADDR	0099H	PREDEFINED
MASTNEXT	C ADDR	0312H	
MASTRQ	B ADDR	00DEH	PREDEFINED
MFAIL1	C ADDR	02C3H	NOT USED
MGO	NUMB	0020H	
MMSGEND	C ADDR	0179H	
MN1	C ADDR	031DH	
MRCV	C ADDR	0139H	
MRCV2	C ADDR	013FH	
MRCVED	NUMB	0021H	
MRCVLOOP	C ADDR	013DH	
MSGSTAT	D ADDR	0024H	
MTX	C ADDR	011AH	
MTXED	NUMB	0022H	
MTXLOOP	C ADDR	011BH	
MTXNAK	NUMB	0023H	
MTXNOSLV	NUMB	0024H	
MTXSTOP	C ADDR	0154H	
MTXSTOP2	C ADDR	015CH	
MTXSTOP3	C ADDR	0163H	
MYADDR	D ADDR	0025H	
NOACK	C ADDR	0134H	
NOGO	C ADDR	0047H	
NOSLAVE	C ADDR	012FH	
NOTSTR	NUMB	0032H	
ONLED	B ADDR	0093H	
P1	D ADDR	0090H	PREDEFINED
PONGADDR	C ADDR	032EH	
PP2	C ADDR	02B8H	
PP22	C ADDR	02BEH	
PP3	C ADDR	02CAH	
PP31	C ADDR	02CCH	
PP32	C ADDR	02CEH	

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 21
PP5	C ADDR 02DCH		
PPSTART	C ADDR 02B5H		
PPTO	C ADDR 02D8H	NOT USED	
PSW	D ADDR 00D0H	PREDEFINED	
RBIT	C ADDR 01FFH		
RBIT2	C ADDR 0201H	NOT USED	
RBIT3	C ADDR 0208H		
RBUFLN	NUMB 0004H		
RC7	C ADDR 024CH		
RCV8	C ADDR 01FCH		
RCVERR	C ADDR 020EH		
RCVEX	C ADDR 020DH		
RDAT	B ADDR 009FH	PREDEFINED	
RECOVER	C ADDR 0230H		
RESET	C ADDR 0278H		
RESET2	C ADDR 028BH		
REX	C ADDR 0272H	NOT USED	
RPSTRT	B ADDR 0001H		
R_CTVAL	C ADDR 032DH		
R_MYADDR	C ADDR 032CH		
SCL	B ADDR 0080H	PREDEFINED	
SDA	B ADDR 0081H	PREDEFINED	
SERRFLAG	B ADDR 0009H		
SETMRQ	B ADDR 0002H		
SGO	NUMB 0010H		
SLAVEN	B ADDR 00DFH	PREDEFINED	
SLNRCVDR	C ADDR 0310H		
SLVTX	C ADDR 0084H		
SMSGEND	C ADDR 00B3H		
SMSGEND2	C ADDR 00B9H		
SP	D ADDR 0081H	PREDEFINED	
SR2	C ADDR 02F9H		
SR3	C ADDR 0308H		
SRCV2	C ADDR 005DH	NOT USED	
SRCV3	C ADDR 0068H		
SRCVBUF	D ADDR 002FH		
SRCVD	NUMB 0011H		
SRCVDR	C ADDR 02EFH		
SRCVEND	C ADDR 0076H		
SRCVERR	C ADDR 00A7H		
SRCVSTO	C ADDR 0066H		
SRERR	NUMB 0014H		
SRERRR	C ADDR 0310H		
SRLNG	NUMB 0012H		
STACKSAVE	D ADDR 002AH		
STP	B ADDR 009AH	PREDEFINED	

Using the 8XC751/752 in multimaster I²C applications

AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 22
STR.	B ADDR 009BH	PREDEFINED	
STSTRW.	C ADDR 0055H		
STX2	C ADDR 0085H	NOT USED	
STXBUF.	D ADDR 0033H		
STXED	NUMB 0013H		
STXEDR.	C ADDR 0310H		
STXERR.	C ADDR 00AEH	NOT USED	
STXLP	C ADDR 0093H		
SUBADD.	B ADDR 0000H		
TI1.	C ADDR 0219H	NOT USED	
TI2.	C ADDR 021CH	NOT USED	
TI3.	C ADDR 0223H		
TI4.	C ADDR 0225H		
TIISR	C ADDR 0211H		
TIMERI.	C ADDR 001BH	NOT USED	
TIRUN	B ADDR 00DCH	PREDEFINED	
TITOCNT.	D ADDR 0029H		
TOGCNT.	D ADDR 0038H		
TOGLED.	B ADDR 0090H		
TRQFLAG.	B ADDR 0008H		
XMADDR.	C ADDR 01C5H		
XMBEX	C ADDR 01E1H		
XMBIT	C ADDR 01D0H		
XMBIT2	C ADDR 01D2H		
XMBYTE	C ADDR 01CEH		
XRETI	C ADDR 004CH		

I²C slave routines for the 83C751

AN433

Author: Greg Goodhue

The S83C751/S87C751 Microcontroller combines in a small package the benefits of a high-performance microcontroller with on-board hardware supporting the Inter-Integrated Circuit (I²C) bus interface.

The 8XC751 can be programmed both as an I²C bus master, a slave, or both. An overview of the I²C bus and description of the bus support hardware in the 8XC751 microcontrollers appears in application note AN422, "Using the 8XC751 Microcontroller as an I²C Bus Master." That application note includes a programming example, demonstrating a bus-master code. Here we show an example of programming the microcontroller as an I²C slave.

The code listing demonstrates communications routines for the 8XC751 as a slave on the I²C bus. It compliments the program in AN422 which demonstrates the 8XC752 as an I²C bus master. One may demonstrate two 8XC751 devices communicating with each other on the I²C bus, using the AN422 code in one, and the program presented here in the other. The examples presented here and in AN422 allow the 751 to be either a master or a slave, but not both. Switching between master and slave roles in a multimaster environment is described in application note AN435.

The software for a slave on the bus is relatively simple, as the processor plays a relatively passive role. It does not initiate bus transfers on its own, but responds to a master initiating the communications. This is true whether the slave receives or transmits data—transmission takes place only as a response to a bus master's request. The slave does not have to worry about arbitration or about devices which do not acknowledge their address. As the slave is not supposed to take control of the bus, we do not demand it to resolve bus exceptions or "hangups". If the bus becomes inactive the processor simply withdraws, not interfering with the master (or masters) on the bus which should (hopefully) try to resolve the situation.

The 8XC751 has a single bit I²C hardware interface where the registers may directly affect the levels on the bus, and the software interacting with the hardware registers takes part in the protocol implementation. The hardware and the low level routines dealing with the registers are tightly coupled. We repeat here the warning from the 751 bus-master application note: one should take extra care if trying to modify these lower level routines.

The service routine for the I²C slave is interrupt driven per message. This allows for master communication requests which are

not synchronized with the application program running on the slave. It is possible to write simple slave application programs which will not be interrupt driven, taking care not to lose master transmissions while doing something else, but the user should be discouraged from doing so. As the slave should respond to asynchronous requests of masters on the bus, an interrupt driven service routine makes sense—and, as the code demonstrates, is simple to implement.

DEMONSTRATION CODE

The main program operation, intended for demonstration only, is simple. There are two data buffers, one for data reception and one for data transmission. When new data has been received from the I²C bus into the receive buffer, the program writes it into the transmit buffer. The first and second bytes of received data are also copied to Port 1 and Port 3, respectively. When a bus master requests to read data, Port 1 and Port 3 will be returned for the first two bytes of requested data, while the remaining bytes will come from the transmit buffer. This allows for simple testing of a master and slave system by having the master compare data received to data sent. This scheme also allows the 8XC751 to be used as a two-byte I²C I/O port.

The program begins at address 0, where the microprocessor begins execution after a hardware reset. This location contains a jump instruction to the main program, which starts at the label *Reset* (towards the end of the listing). Upon reset, the program initializes the stack pointer, the I²C address of the slave processor (*MyAddr*) and clears the data buffers and software flags. In this program the receive and transmit buffers are each eight bytes long—the maximum number of bytes is defined by the label *MaxBytes*. One may easily change the program to handle longer messages by changing the value of *MaxBytes* and allocating more data memory to the buffers.

The I²C interface is configured to operate as a slave by setting the msb of register *I²CFG*. This is done simultaneously with loading the appropriate value of *CTVAL*—bits *CT0* and *CT1*, which are determined by the frequency of the microprocessor's crystal. The interface hardware is explicitly instructed to get into the slave idle mode by setting the appropriate bit in the *I²CON* register. *Timer 1*, which operates as a "watchdog" timer detecting bus hangups, is activated and its interrupts are enabled.

After the initialization, the program gets to the label *MainLoop*. Most of the time the program

will "hang" in a wait loop at this label, simply waiting for an I²C interrupt to occur. When there is an I²C bus request there will be an interrupt, the service routine will be executed and we shall return to the *MainLoop* label. If the service routine receives new data, it sets a flag, *DatFlag*, signalling that data has been updated. This flag will allow us to leave the *MainLoop* label, and execute a short routine copying the updated input buffer to the output (transmit) buffer.

If a new bus interrupt comes before overwriting of the old read buffer data is completed, and an undesirable "mix" of old and new data might occur. This type of situation is avoided by disabling the I²C interrupts (clearing the *IE2* bit in the *Interrupt Enable Register*) just before copying the data to the transmit buffer, and re-enabling the interrupts when the copy operation is completed.

When the copy routine is completed the *DatFlag* is cleared and we jump back to *MainLoop*, waiting for the next interrupt to occur. If the interrupt is for data transmission the service routine will not set *DatFlag*, and upon return we shall remain at the *MainLoop* label.

THE INTERRUPT SERVICE ROUTINE

The service routine is interrupt driven with respect to the start of each I²C frame, but within each frame the interaction with the hardware is based on polling. An occurrence of a Start on the bus will cause an interrupt that will initiate the service routine which starts at address 23H. After saving registers, all interrupts except the I²C interrupt itself are enabled, as we want to allow response to other interrupts during the routine. The philosophy behind this is that the I²C may be a lower priority than some other operations in the system. Since the I²C hardware will stretch the clock until the program responds, an interrupt of reasonable duration will not have a harmful effect on the data transfer.

Since we intend to react to the I²C hardware by polling the *ATN* flag in wait loops, we do not want the expected changes on the bus to take us again to the beginning of the routine. Therefore, the *EI2* flag is cleared, masking further I²C interrupts even when interrupts are re-enabled (by the *ACALL* to a *RETI* instruction).

At the label *Slave*, the routine starts receiving the address on the bus. Each new address bit is read after a software wait loop detects that the *ATN* flag is set by the hardware. Note that with the single bit implementation of the

I²C slave routines for the 83C751

AN433

I²C port on the 83C751 the software must closely support the hardware: for example, we need to explicitly clear the Start status before we enter a wait loop for the next bit. If the software does not clear the Start flag, the hardware will stretch the low period of the clock (SCL line) on the bus—and the first address bit will simply not occur. (Such a state will not go on forever—eventually the processor will release the bus as a result of a Timer I timeout.)

Reception of the eight bits of Address + R/W is completed using part of the receive byte subroutines. The address received is compared to MyAddr, the address of this specific slave. If the address is different the processor goes idle and leaves the service routine. If the message is intended for this processor (received address matches MyAddr) the Read/Write bit is tested, and the program jumps to the appropriate labels. When the R/W bit is low the master requests a Write—and this slave should receive the data written into it. When the R/W bit is high the master is requesting a Read and this slave should transmit the data (at code label Read).

For "Master Write" we send an acknowledge for the address byte and proceed with receiving the data bytes, responding with an acknowledge for each and transferring them into the receive buffer. For long messages, when the buffer is full (we have received MaxByte bytes) we read from the bus one additional byte and then send a negative acknowledge, letting the master know it

should stop sending us data. Then we set DatFlag to signal the mainline program that new data has been received, and jump to MsgEnd. At the MsgEnd label we wait for the next Stop or Repeated Start. On a Stop we resume the idle mode (Goldle) and return from the service routine. On a Restart the slave process starts again with reception of the new address at the label Slave.

If the message is short enough so that the receive buffer is not filled up, the RcvByte subroutine (called after WrtLoop) will return due to the Stop condition, DRDY will not be set, and we shall exit the loop via label WLEx—setting the DatFlag and proceeding to MsgEnd.

For "Master Read" the transmit buffer is sent on the bus byte by byte in the RdLoop, using the XmitByte subroutine. We exit the loop when all the buffer is transmitted, or the Master does not respond with an acknowledge. Note that lack of acknowledgement for slave transmission does not necessarily indicate a problem or that the receiving master is busy. This could very well be a normal operation of the protocol, which defines that a receiving master signals the transmitting slave to end its message by explicitly transmitting a negative acknowledge as a response to the last byte the master is interested in. The protocol does not include inherent means for specifying in advance the length of a requested message.

SUBROUTINES

The lower level subroutines closely interact with the hardware and the activity on the bus. The XmitByte subroutine transmits one byte and receives the acknowledge bit that comes in response. The byte receive routine, which one may use from different entry points, receives a data or an address byte, and takes care of acknowledgements. When a Start or Stop is detected the subroutine returns immediately—the calling routine is expected to check the flags to determine whether a whole byte has been received (DRDY will be set), or a Start or a Stop condition has occurred.

Close inspection of RcvByte code shows that a total of nine bits are being read off the bus. The first bit does not belong to the received byte, but is the acknowledge this processor sent in response of the former byte or address. Reading the Ack bit from the I2DAT register clears the Transmit Active state and DRDY, thus releasing SCL and allowing the bus activity to proceed to the next data bit. Upon return the Ack bit is left in the Carry flag, and the actual data byte received is returned in the Acc register.

Upon Timer I interrupt code execution commences at address 1BH, where there is a jump to the service routine TimerI. This interrupt is caused by the watchdog timer, as a result of an I²C bus that is "hanging" without activity in the middle of a transmission for too long a period of time. The slave simply clears the bus interface, and starts all over again at the label Reset.

I²C slave routines for the 83C751

AN433

```

;*****
;           Sample I2C Slave Routines for the 8XC751 and 8XC752
;
; This program demonstrates I2C slave functions for the 8XC751 and 8XC752
; microcontrollers. The program uses separate transmit and receive data
; buffers that are each eight bytes deep. The sample main program
; copies received data to the transmit buffer such that transmitted data can
; be read back by a bus master. Buffer addresses 0 and 1 are mapped to port 1
; and 3 respectively, such that an I2C write will affect the port outputs, and
; an I2C read will return port pin data. The 751 will accept only eight data
; bytes in any one I2C transmission, additional bytes will not be
; acknowledged. Similarly, only eight data bytes may be read from the 751 in
; any one I2C transmission. This program does not support subaddressing for
; buffer access.
;*****

$TITLE(8XC751 I2C Slave Routines)
$DATE(11/23/92)
$MOD752
;*****

; Value definitions.

CTVAL      EQU      02h          ; CT1, CT0 bit values for I2C.
MaxBytes   EQU      8           ; Maximum # of bytes to be sent or
;                               received.

; Masks for I2CFG bits.

BTIR       EQU      10h          ; Mask for TIRUN bit.
BMRQ       EQU      40h          ; Mask for MASTRQ bit.

; Masks for I2CON bits.

BCXA       EQU      80h          ; Mask for CXA bit.
BIDLE      EQU      40h          ; Mask for IDLE bit.
BCDR       EQU      20h          ; Mask for CDR bit.
BCARL      EQU      10h          ; Mask for CARL bit.
BCSTR      EQU      08h          ; Mask for CSTR bit.
BCSTP      EQU      04h          ; Mask for CSTP bit.
BXSTR      EQU      02h          ; Mask for XSTR bit.
BXSTP      EQU      01h          ; Mask for XSTP bit.

; RAM locations used by I2C routines.

RcvDat     DATA    10h          ; I2C receive data buffer (8 bytes).
;                               addresses 10h through 17h.

XmtDat     DATA    18h          ; I2C transmit data buffer (8 bytes).
;                               addresses 18h through 1Fh.

Flags      DATA    20h          ; I2C software status flags.
NoAck      BIT      Flags.7      ; Holds negative acknowledge flag.
DatFlag    BIT      Flags.6      ; Tells whether an I2C write operation
;                               has occurred.

BitCnt     DATA    21h          ; I2C bit counter.
ByteCnt    DATA    22h          ; Send/receive byte counter.
TDAT       DATA    23h          ; Temporary holding register.
MyAddr     DATA    24h          ; Holds address of THIS slave.

AdrRcvd    DATA    25h          ; Holds received slave address + R/W.
RWFlag     BIT      AdrRcvd.0    ; Slave read/write flag.

```


I²C slave routines for the 83C751

AN433

```

;*****
;
;                               Begin Code
;*****

; Reset and interrupt vectors.

        AJMP    Reset            ; Reset vector at address 0.

; A timer I timeout usually indicates a 'hung' bus.

        ORG     1Bh              ; Timer I (I2C timeout) interrupt.
        AJMP    TimerI

; I2C interrupt is used to detect a start while the slave is idle.

        ORG     23h              ; I2C interrupt.
        PUSH    PSW              ; Save status.
        PUSH    ACC              ; Save accumulator.
        CLR     ES               ; Disable I2C interrupt.
        ACALL   ClrInt           ; Re-enable interrupts.

;*****
;
;                               Main Transmit and Receive Routines
;*****

Slave:   MOV     I2CON,#BCARL+BCSTP+BCSTR+BCXA ; Clear start status.
        JNB     ATN,$            ; Wait for next data bit.
        MOV     BitCnt,#7        ; Set bit count.

        ACALL   RcvB2            ; Get remainder of slave address.
        MOV     AdrRcvd,A        ; Save received address + R/W bit.
        CLR     ACC.0
        CJNE   A,MyAddr,GoIdle   ; Enter idle mode if not our address.

        JB      RWFlag,Read       ; Read or Write?
        MOV     R0,#RcvDat        ; Set up receive buffer pointer.
        MOV     ByteCnt,#MaxBytes ; Max 4 bytes can be received.

WrtLoop: ACALL   SendAck          ; Send acknowledge.
        ACALL   RcvByte          ; Get data byte from master.
        JNB     DRDY,WLEx        ; Must be end of frame?
        MOV     @R0,A            ; Save data.
        INC     R0               ; Advance buffer pointer.
        DJNZ   ByteCnt,WrtLoop   ; Back to receive if buffer not full.
        ACALL   SendAck          ; Send acknowledge.
        ACALL   RcvByte          ; Get, but do not store add'l data.
        MOV     I2DAT,#80h        ; Send negative acknowledge.
        JNB     ATN,$            ; Wait for acknowledge sent.

WLEx:    SETB   DatFlag          ; Flag main that data has been received.
        SJMP   MsgEnd           ; Buffer full, enter idle mode.

Read:    MOV     R0,#XmtDat        ; Set up transmit buffer pointer.
        MOV     ByteCnt,#MaxBytes ; Max bytes to be sent.
        ACALL   SendAck          ; Send address acknowledge.

RdLoop:  MOV     A,@R0            ; Get data byte from buffer.
        CJNE   R0,#XmtDat,RdL1   ; Return port 1 value instead of buffer
        MOV     A,P1              ; data if this is buffer address 0.
RdL1:    CJNE   R0,#XmtDat+1,RdL2 ; Return port 3 value instead of buffer
        MOV     A,P3              ; data if this is buffer address 1.

RdL2:    INC     R0               ; Advance buffer pointer.
        ACALL   XmitByte         ; Send data byte.
        JB      NoAck,RLEx       ; Exit if NAK.
        DJNZ   ByteCnt,RdLoop    ; Back if more data requested & avail.

```

I²C slave routines for the 83C751

AN433

```

RLEx:      SJMP      MsgEnd          ; Done, enter idle mode.

MsgEnd:    JNB      ATN,$            ; Wait for stop or repeated start.
           JB       STR,Slave        ; If repeated start, go to slave mode,
           ;         else enter idle mode.

GoIdle:    MOV      I2CON,#BCSTP+BCXA+BCDR+BCARL+EBIDLE ; Enter slave idle mode.
           POP      ACC              ; Restore accumulator.
           POP      PSW              ; Restore status.
           SETB     ES               ; Re-enable I2C interrupts.
           RET

;*****
;
;                               Subroutines
;*****

; Byte transmit routine.
;   Enter with data in ACC.

XmitByte:  MOV      BitCnt,#8        ; Set 8 bits of data count.
XmBit:     MOV      I2DAT,A          ; Send this bit.
           RL       A                ; Get next bit.
           JNB      ATN,$            ; Wait for bit sent.
           DJNZ     BitCnt,XmBit     ; Repeat until all bits sent.
           MOV      I2CON,#BCDR+BCXA ; Switch to receive mode.
           JNB      ATN,$            ; Wait for acknowledge bit.
           MOV      Flags,I2DAT      ; Save acknowledge bit.
           RET

; Byte receive routines.
;   SendAck : sends an I2C acknowledge.
;   RcvByte : receives a byte of data.
;   RcvB2   : receives a partial byte of I2C data, used to allow reception of
;             7 bits of slave address information.
;   Data is returned in the ACC.

SendAck:   MOV      I2DAT,#0         ; Send receive acknowledge.
           JNB      ATN,$            ; Wait for acknowledge sent.
           RET

RcvByte:   MOV      BitCnt,#8        ; Set bit count.
RcvB2:     CLR      A                ; Init received byte to 0.
RBit:      ORL      A,I2DAT          ; Get bit, clear ATN.
           RL       A                ; Shift data.
           JNB      ATN,$            ; Wait for next bit.
           JNB      DRDY,RBEx        ; Exit if not a data bit.
           DJNZ     BitCnt,RBit      ; Repeat until 7 bits are in.
           MOV      C,RDAT           ; Get last bit, don't clear ATN.
           RLC      A                ; Form full data byte.

RBEx:      RET

; Timer I timeout interrupt service routine.

TimerI:    SETB     CLRTI            ; Clear timer I interrupt.
           MOV      I2CFG,#0         ; Turn off I2C.
           MOV      I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP ; Reset I2C flags.
           ACALL   ClrInt            ; Clear interrupt pending flag.
           AJMP    Reset            ; Return to mainline.

ClrInt:    RETI

```

I²C slave routines for the 83C751

AN433

```

;*****
;                               Main Program
;*****
Reset:   MOV     SP,#2Fh           ; Set stack location.
         MOV     IE,#90h         ; Enable I2C interrupt.

         MOV     R0,#RcvDat      ; Set up pointer to data area.
         MOV     R1,#2*MaxBytes  ; Set up buffer length counter.
RLoop:   MOV     @R0,#0          ; Clear buffer memory.
         INC     R0              ; Advance to next buffer position.
         DJNZ   R1,RLoop        ; Repeat until done.

         MOV     MyAddr,#40h     ; Set slave address.
         MOV     Flags,#0        ; Clear system flags.
         MOV     I2CFG,#80h+CTVAL ; Enable slave functions.
         MOV     I2CON,#BIDLE    ; Put slave into idle mode.
         SETB   ETI             ; Enable timer I interrupts.
         SETB   TIRUN           ; Turn on timer I.

; This sample mainline program copies the first two received bytes to Port 1
; and Port 3 whenever there is an I2C write operation. It also copies the
; rest of the input buffer to the output buffer at the same time.

MainLoop: JNB    DatFlag,$        ; Wait for data sent from I2C.
         CLR    EA              ; Turn off interrupts during data move.

         MOV    P1,RcvDat        ; First buffer location goes to port 1.
         MOV    P3,RcvDat+1      ; Second buffer location goes to port 3.

         MOV    R0,#RcvDat       ; Set input buffer start pointer.
         MOV    R1,#XmtDat       ; Set output buffer start pointer.
         MOV    R2,#MaxBytes     ; Set buffer length counter.
ML2:     MOV    A,@R0            ; Get data from input buffer.
         MOV    @R1,A            ; Store data in output buffer.
         INC    R1               ; Increment input buffer pointer.
         INC    R0               ; Increment output buffer pointer.
         DJNZ  R2,ML2           ; Repeat until entire buffer is updated.
         CLR    DatFlag         ; Clear I2C transmission flag.

         SETB  EA               ; Data move done, re-enable interrupts.
         SJMP  MainLoop         ; Wait for next I2C transmission.

END

```

Connecting a PC keyboard to the I²C-bus

AN434

CONNECTING A PC KEYBOARD TO THE I²C BUS

This application note illustrates the use of a low-cost 8-bit microcontroller—the 8XC751—to interface a standard PC/AT keyboard to the I²C bus. The 8XC751 (83C751 = ROM-version, 87C751 = EPROM-version) is ideally suited for the task thanks to its built-in I²C interface, small form-factor (24-pin DIP or 28-pin PLCC) and low power consumption (11mA typical @ 12 MHz; see Figure 1). The application software easily fits within the 2K bytes code and 64 bytes data memory provided on the 8XC751.

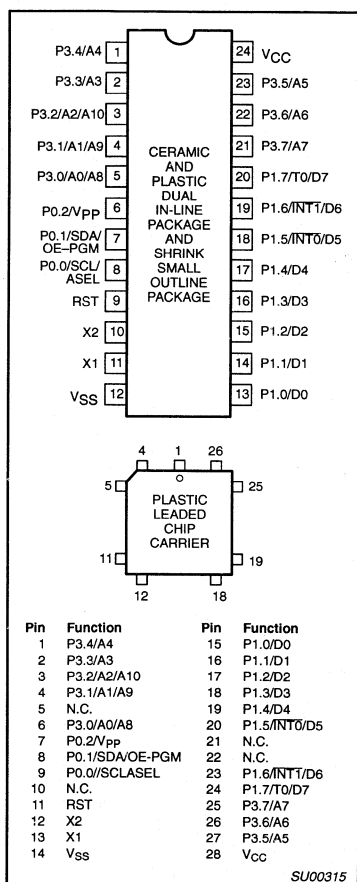


Figure 1. Pin Configuration

The PC/AT Keyboard

The PC/AT keyboard transmits data in a clocked serial format consisting of a start bit, 8 data bits (LSB first), an odd parity bit and a stop bit as shown in Figure 2. Besides clock and data, the 5-pin connector (Figure 3) also includes power, ground and a no connect. Note that the PS/2 keyboard interface is logically equivalent, though it uses a different connector. (A sixth pin provides an additional no connect).

When a key is pressed, the PC/AT keyboard transmits a 'make' code and, when the key is released, a 'break' code. The make code consists of an 8-bit 'scan' code denoting the key pressed. The 'break' code (key released) consists of the same 8-bit scan code preceded by a special code—0F0H.

A notable difference from a regular ASCII keyboard is the way SHIFT, CTRL, ALT, etc. control keys work. For an ASCII keyboard, the control keys directly modify the code output. For example, a 61H (ASCII code for 'a') is output if the 'A' key is pressed by itself, while a 41H (ASCII code for 'A') is output if the SHIFT and 'A' keys are pressed simultaneously.

The PC/AT keyboard handles such a key combination as two separate key presses, i.e., SHIFT-MAKE, 'A'-MAKE, SHIFT-BREAK, 'A'-BREAK. The 'A' scan code (1CH) is the same for both the shifted and unshifted state. To determine whether the 'A' scan code is interpreted as 'A' or 'a' the PC must keep track of the presence or absence of a prior SHIFT-MAKE.

Keyboard-to-I²C Hardware (Figure 4)

The 8XC751 on-chip I²C interface allows direct connection of the SDA (Serial Data) and SCL (Serial Clock) pins to the corresponding I²C bus lines. Since the I²C bus is open collector (allowing multimasters), 10K resistors are used to pull the lines to the idle state between keypresses.

The PC/AT keyboard interface is equally simple. The CLK output from the keyboard is used to generate an interrupt (INT0). In response, the 8XC751 interrupt service routine samples the keyboard serial DATA connected to port 0 bit 2 (P0.2).

When used with a PC, the keyboard implements a bidirectional communication

protocol by exploiting the fact that both the keyboard and PC can drive the open collector CLK and DATA lines. However, bidirectional communication is not required for basic keyboard operation and in this application, the keyboard is treated as an 'input-only' device.

Keyboard-to-I²C Software

The keyboard-to-I²C software performs three major functions:

- Capture the clocked serial data from the keyboard
- Translate the keyboard data to the corresponding ASCII code
- Send the ASCII code as an I²C message.

When a key is pressed, the CLK output from the keyboard generates an interrupt via INT0. The 8XC751 shifts in the DATA from the keyboard on P0.2 (port 0, bit 2) and extracts the 8-bit scan code from the 11-bit packet.

Next, the scan code is interpreted and converted to the corresponding ASCII code using a look-up table. Keyboard multi-code outputs are converted to single ASCII codes by tracking the state (i.e. shifted vs. unshifted) of the keyboard and using separate look-up tables for each. For example, a keyboard SHIFT-MAKE, 'A'-MAKE, SHIFT-BREAK, 'A'-BREAK sequence is converted to the ASCII code for uppercase 'A' (41H). The flowchart in Figure 5 depicts the keyboard data capture and code conversion process.

The 8XC751 operates as an I²C slave. When the master issues a read command, the 8XC751 returns the converted ASCII character. The seven least significant bits are used for the ASCII code, while the most significant bit is used as a NEW flag (0 = new, 1 = old). The key code remains marked as new until the master issues a write to the 8XC751 at which point it is marked as old and will be overwritten by the next key processed.

The keyboard-to-I²C software is shown immediately following Figure 5. Less than half the code space available on the 8XC751 is used, leaving room for extra features such as parity checking and more complete keyboard control state mapping using additional look-up tables.

Connecting a PC keyboard to the I²C-bus

AN434

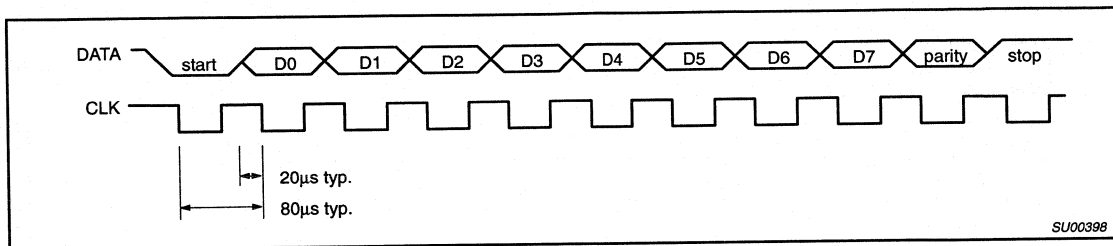


Figure 2. PC/AT Keyboard Timing

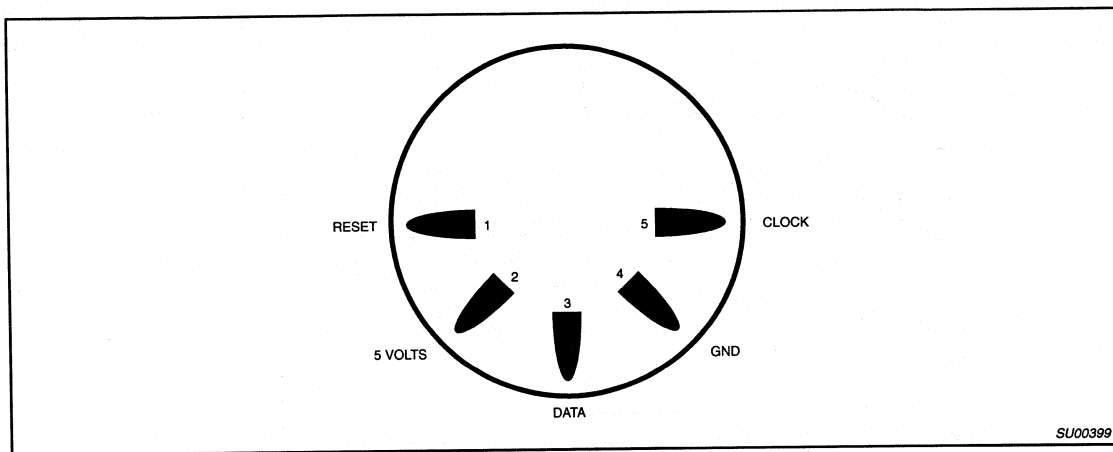


Figure 3. Keyboard Connections (looking into the connector)

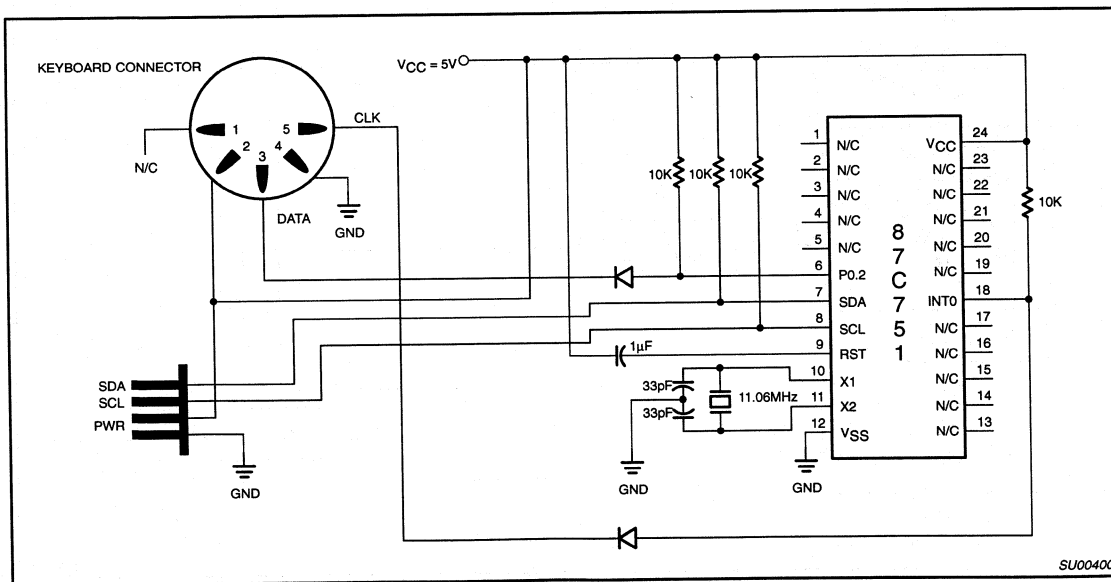
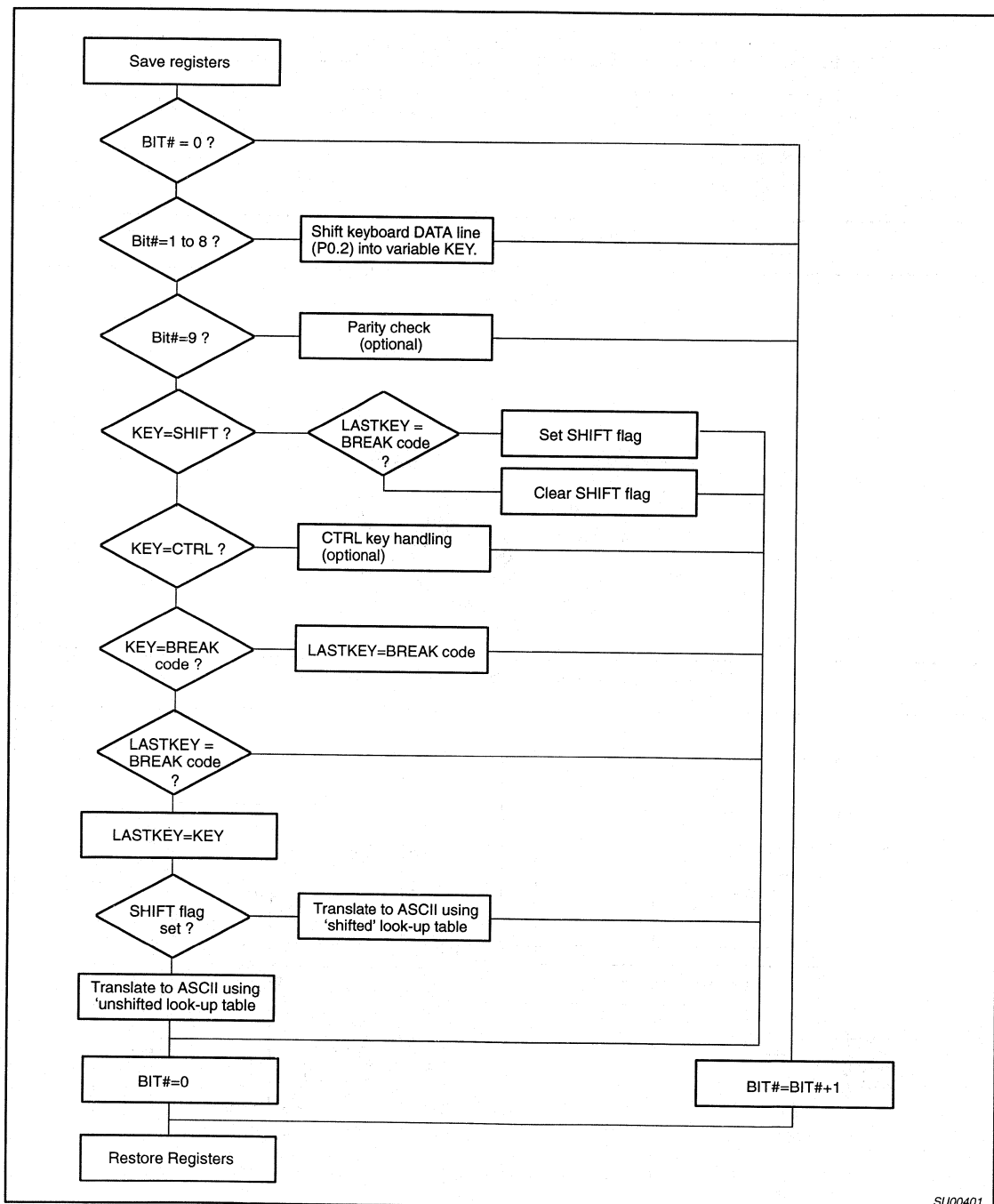


Figure 4. IBM Keyboard to I²C Bus Format Using the 8XC751

Connecting a PC keyboard to the I²C-bus

AN434



SU00401

Figure 5. Keyboard Data Capture and Conversion

Connecting a PC keyboard to the I²C-bus

AN434

```

0001 0000 ;*****
0002 0000 ;
0003 0000 ; Copyright Micro AMPS Ltd
0004 0000 ; & Philips Semiconductors
0005 0000 ; Dec 1990
0006 0000 ;
0007 0000 ;*****
0008 0000
0009 0000 ; Read data under interrupt from an IBM keyboard
0010 0000 ; Hardware resources:
0011 0000 ; Kbd clock on interrupt INT0 P1.5
0012 0000 ; Kbd data on pin P0.2
0013 0000
0014 0000
0015 0000 ; This program reads keys in from the keyboard
0016 0000 ; and translates them to ASCII
0017 0000
0018 0000
0019 0000
0020 0000 #include equates.51
0001+ 0000 ; direct addresses for the standard 8051 processor
0002+ 0000
0003+ 0000 p0 .equ 80h ; port 0
0004+ 0000 sp .equ 81h ; stack pointer
0005+ 0000 dpl .equ 82h ; data pointer low
0006+ 0000 dph .equ 83h ; data pointer high
0007+ 0000
0008+ 0000 pcon .equ 87h ; power control
0009+ 0000 tcon .equ 88h ; timer control
0010+ 0000 tmod .equ 89h ; timer mode
0011+ 0000 tl0 .equ 8ah ; timer 0 low
0012+ 0000 th0 .equ 8ch ; timer 0 high
0013+ 0000
0014+ 0000 th1 .equ 8dh ; timer 1 high
0015+ 0000
0016+ 0000 p1 .equ 90h ; port 1
0017+ 0000 scon .equ 98h ; serial control
0018+ 0000 s0con .equ 98h ; serial control
0019+ 0000 s0buf .equ 99h ; serial data
0020+ 0000
0021+ 0000 p2 .equ 0a0h ; port 2
0022+ 0000 p3 .equ 0b0h ; port 3
0023+ 0000 ien0 .equ 0a8h ; interrupt enable
0024+ 0000 ie .equ 0a8h
0025+ 0000
0026+ 0000 psw .equ 0d0h ; program status word
0027+ 0000 acc .equ 0e0h ; accumulator
0028+ 0000 b .equ 0f0h ; b register
0029+ 0000
0030+ 0000 ; bit addressed flags
0031+ 0000
0032+ 0000 it0 .equ 88h ; int 0 edge/level trigger
0033+ 0000 ie0 .equ 89h ; int 0 edge detect
0034+ 0000 it1 .equ 8ah ; int 1 edge/level trigger
0035+ 0000 iel .equ 8bh ; int 1 edge detect
0036+ 0000 tr0 .equ 8ch ; timer 0 enable/disable
0037+ 0000 tf0 .equ 8dh ; timer 0 overflow detect
0038+ 0000 tr1 .equ 8eh ; timer 1 enable/disable
0039+ 0000 tf1 .equ 8fh ; timer 1 overflow detect
0040+ 0000
0041+ 0000 ri .equ 98h
0042+ 0000 ti .equ 99h
0043+ 0000
0044+ 0000 ien0.7 .equ 0afh ; global int enable/disable
0045+ 0000
0046+ 0000 p0.0 .equ 080h ; port 0 bit 0
0047+ 0000

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0048+ 0000  b.0      .equ 0f0h          ; b reg bits
0049+ 0000  b.1      .equ 0f1h
0050+ 0000  b.2      .equ 0f2h
0051+ 0000  b.3      .equ 0f3h
0052+ 0000  b.4      .equ 0f4h
0053+ 0000  b.5      .equ 0f5h
0054+ 0000  b.6      .equ 0f6h
0055+ 0000  b.7      .equ 0f7h
0056+ 0000
0057+ 0000  a.0      .equ 0e0h          ; accumulator bits
0058+ 0000  a.1      .equ 0e1h
0059+ 0000  a.2      .equ 0e2h
0060+ 0000  a.3      .equ 0e3h
0061+ 0000  a.4      .equ 0e4h
0062+ 0000  a.5      .equ 0e5h
0063+ 0000  a.6      .equ 0e6h
0064+ 0000  a.7      .equ 0e7h
0065+ 0000
0066+ 0000  rth      .equ 8dh          ; timer 0 reload high
0067+ 0000  rtl      .equ 8bh          ; timer 0 reload low
0068+ 0000
0021  0000  #include kbd.h
0001+ 0000  #define reg .equ
0002+ 0000
0003+ 0000  ;
0004+ 0000  ; 8xc751 special register set
0005+ 0000  ;
0006+ 0000  ; 751 I2C byte registers
0007+ 0000
0008+ 0000  I2CON    .equ 098h          ; I2C control
0009+ 0000  I2CFG    .equ 0d8h          ; I2C configuration
0010+ 0000  I2DAT    .equ 099h          ; I2C data
0011+ 0000  I2STA    .equ 0f8h          ; I2C status
0012+ 0000
0013+ 0000  IE       .equ 0a8h          ; interrupt enable
0014+ 0000
0015+ 0000  TCON     .equ 088h          ; timer/counter control
0016+ 0000
0017+ 0000  TL       .equ 08ah          ; timer 0 low
0018+ 0000  TH       .equ 08ch          ; timer 0 high
0019+ 0000  RTL      .equ 08bh          ; timer reload low
0020+ 0000  RTH      .equ 08dh          ; timer reload high
0021+ 0000
0022+ 0000  ; 751 I2C bit registers
0023+ 0000
0024+ 0000  ;I2CNFG
0025+ 0000
0026+ 0000  SLAVEN   .equ 0dfh
0027+ 0000  MASTRQ   .equ 0deh
0028+ 0000  TIRUN    .equ 0dcch
0029+ 0000  CT1      .equ 0d9h
0030+ 0000  CT0      .equ 0d8h
0031+ 0000  CLRTI    .equ 0ddh
0032+ 0000
0033+ 0000  RDATA    .equ 09fh
0034+ 0000  ATN      .equ 09eh
0035+ 0000  DRDY     .equ 09dh
0036+ 0000  ARL      .equ 09ch
0037+ 0000  STR       .equ 09bh
0038+ 0000  STP       .equ 09ah
0039+ 0000  MASTER   .equ 099h
0040+ 0000
0041+ 0000  ; I2CON
0042+ 0000
0043+ 0000  CXA      .equ 09fh
0044+ 0000  IDLE     .equ 09eh
0045+ 0000  CDR       .equ 09dh

```


Connecting a PC keyboard to the I²C-bus

AN434

```

0046+ 0000  CARL      .equ 09ch
0047+ 0000  CSTR      .equ 09bh
0048+ 0000  CSTP      .equ 09ah
0049+ 0000  XSTR      .equ 099h
0050+ 0000  XSTP      .equ 098h
0051+ 0000
0052+ 0000  ;I2STA
0053+ 0000
0054+ 0000  XDATA     .equ 0fdh
0055+ 0000  XACTV     .equ 0fch
0056+ 0000  MAKSTR    .equ 0fbh
0057+ 0000  MAKSTP    .equ 0fah
0058+ 0000
0059+ 0000  ; IE bit registers
0060+ 0000
0061+ 0000  EA        .equ 0afh ; clr to disable all interrupts
0062+ 0000  EI2       .equ 0ach ; set to enable iic interrupt
0063+ 0000  ETI       .equ 0abh ; set to enable timer 1 overflow interrupt
0064+ 0000  EX1       .equ 0aah ; set to enable ext int 1
0065+ 0000  ETO       .equ 0a9h ; set to enable timer 0 overflow interrupt
0066+ 0000  EX0       .equ 0a8h ; set to enable ext int 0
0067+ 0000
0068+ 0000  ; Value definitions.
0069+ 0000
0070+ 0000  CTVAL     .equ 02h ;CT1, CT0 bit values for I2C.
0071+ 0000
0072+ 0000
0073+ 0000  ; Masks for I2CFG bits.
0074+ 0000
0075+ 0000  BTIR     .equ 10h ; mask for TIRUN bit.
0076+ 0000  BMRQ     .equ 40h ; mask for MASTRQ bit.
0077+ 0000
0078+ 0000
0079+ 0000  ; Masks for I2CON bits.
0080+ 0000
0081+ 0000  BCXA     .equ 80h ; mask for CXA bit.
0082+ 0000  BIDLE    .equ 40h ; mask for IDLE bit.
0083+ 0000  BCDR     .equ 20h ; mask for CDR bit.
0084+ 0000  BCARL    .equ 10h ; mask for CARL bit.
0085+ 0000  BCSTR    .equ 08h ; mask for CSTR bit.
0086+ 0000  BCSTP    .equ 04h ; mask for CSTP bit.
0087+ 0000  BXSTR    .equ 02h ; mask for XSTR bit.
0088+ 0000  BXSTP    .equ 01h ; mask for XSTP bit.
0089+ 0000  ;
0090+ 0000
0091+ 0000
0092+ 0000  SCL      .equ p0.0 ; port bit for I2C serial clock line.
0093+ 0000  SDA      .equ p0.1 ; port bit for I2C serial data line.
0094+ 0000
0022 0000
0023 0000  IICADD   .equ 088h ; our I2C slave address
0024 0000  MAXBYTES .equ 1 ; max bytes to recv or trans
0025 0000
0026 0000  rcvdat   .equ 04h ; I2C received data buffer
0027 0000  xmtdat   .equ 06h ; I2C transmitter buffer
0028 0000
0029 0000  STACK    .equ 08h
0030 0000
0031 0000  flags    .equ 020h ; byte used as flags
0032 0000  noack    .equ (flags-20h) ; I2C flags.0, ...1, ...2, etc
0033 0000  recvd    .equ (flags-20h)+1 ;
0034 0000  sent_flag .equ (flags-20h)+2 ;
0035 0000  i2c_busy .equ (flags-20h)+3 ;
0036 0000
0037 0000  Cntrl    .equ (flags-20h)+8 ; control key flag
0038 0000  Shift    .equ (flags-20h)+9 ; shift key flag
0039 0000

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0040 0000  bitcnt    .equ flags+2
0041 0000  bytecnt   .equ flags+3
0042 0000
0043 0000  adrrcvd   .equ flags+4
0044 0000  rwflag    .equ (adrrcvd-20h)*8 ; adrrcvd.0
0045 0000
0046 0000  tick      .equ 025h          ; count 10mS ticks to give 1sec tick
0047 0000  i2ctime   .equ 027h          ; I2C timeout - used on slow I2C bus
0048 0000
0049 0000
0050 0000  NBits     .equ 29h          ; # bits read so far
0051 0000  NBytes    .equ NBits+1      ; # bytes in buffer
0052 0000  lastkey   .equ NBytes+1      ; last key was?
0053 0000  keytemp   .equ lastkey+1    ; used to build the key bit by bit
0054 0000  keybuff   .equ keytemp+1    ; store the chars here
0055 0000
0056 0000
0057 0000  INMAX     .equ 8            ; size of keyboard buffer
0058 0000  KEYCLK    .equ p1.5          ; keyboard clock signal on ext int 0
0059 0000  KEYDAT    .equ 82h          ; keyboard data line
0060 0000
0061 0000  EDGEINT   .equ 08ah
0062 0000
0063 0000    ; reset and interrupt vectors.
0064 0000    ;
0065 0000    .org 0            ; reset vector
0066 0000 01 50  ajmp start
0067 0002
0068 0003    .org 0003h        ; external interrupt 0
0069 0003 01 B5  ajmp kbd
0070 0005
0071 000B    .org 0bh          ; counter/timer 0
0072 000B 21 EA  ajmp badint
0073 000D
0074 0013    .org 013h        ; external interrupt 1
0075 0013 21 EA  ajmp badint
0076 0015
0077 001B    .org 01bh        ; timer 1 - I2C timeout
0078 001B 21 DA  ajmp timerI
0079 001D
0080 0023    .org 023h        ; I2C interrupt
0081 0023 21 38  ajmp i2cint
0082 0025
0083 0025
0084 0048    .org 48h
0085 0048
0086 0048  done:
0087 0048 01 48  ajmp $          ; main routine waiting for key presses
0088 004A
0089 0050    .org 50h
0090 0050
0091 0050  start:
0092 0050 78 FF  mov r0,#0ffh          ; power supply settling time
0093 0052 79 FF  mov r1,#0ffh
0094 0054 7A 04  mov r2,#04h
0095 0056
0096 0056 D8 FE  dly1:djnz r0,$
0097 0058 D9 FC  djnz r1,dly1
0098 005A DA FA  djnz r2,dly1
0099 005C
0100 005C  reset:
0101 005C 75 80 FF  mov p0,#0ffh
0102 005F 75 90 FF  mov p1,#0ffh
0103 0062 75 B0 FF  mov p3,#0ffh
0104 0065
0105 0065 75 81 08  mov sp,#STACK          ; initialize stack pointer
0106 0068 D2 8A  setb EDGEINT          ; make ext int 0 edge activated

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0107 006A
0108 006A C2 09      clr Shift                ; clear keyboard shift flag
0109 006C
0110 006C 78 29      mov r0,#NBits           ; clear the input buffers
0111 006E 79 10      mov r1,#10h
0112 0070 75 E0 00   mov acc,#0
0113 0073
0114 0073 F6         clr rp:mov @r0,a        ; do the clearing
0115 0074 08         inc r0
0116 0075 D9 FC     djnz r1,clr rp
0117 0077
0118 0077 ;          mov xmdat,#'. ' ; transmit buffer filled with
0119 0077 ;          mov xmdat+1,#0ffh ; $ff when empty
0120 0077
0121 0077 75 20 00   mov flags,#0
0122 007A 75 27 00   mov i2ctime,#0
0123 007D
0124 007D          restart:
0125 007D
0126 007D 75 D8 82   mov I2CFG,#80h+CTVAL   ; enable slave functions
0127 0080 75 98 40   mov I2CON,#IDLE       ; place in idle state
0128 0083
0129 0083 75 A8 91   mov ien0,#91h         ; enable external & IIC interrupts
0130 0086
0131 0086
0132 0086 ;***** Main loop *****
0133 0086
0134 0086          main:
0135 0086 E5 2A      mov a,NBytes           ; if data in keybuff then
0136 0088 60 0C     jz empty              ; copy to I2C xmt buffer
0137 008A
0138 008A 75 A8 00   mov ien0,#0h          ; disable all ints temporarily
0139 008D 85 2D 06   mov xmdat,keybuff
0140 0090 75 2A 00   mov NBytes,#0         ; clear keyboard buffer full flag
0141 0093 75 A8 91   mov ien0,#91h         ; enable external&IIC interrupts
0142 0096
0143 0096          empty:
0144 0096
0145 0096 30 01 0A   jnb recvd,notread    ; recvd flag tells 751 to clear
0146 0099 ; I2C xmt buffer when I2C master
0147 0099 85 06 E0   mov acc,xmdat         ; reads the data from the 751
0148 009C 44 80     orl a,#80h           ; the master writes any data
0149 009E 85 E0 06   mov xmdat,acc        ; back which will set the MSB of
0150 00A1 ; the data buffer. This is reqd.
0151 00A1 ; to sync the two processors.
0152 00A1 C2 01     clr recvd            ; reset I2C received flag
0153 00A3
0154 00A3          notread:
0155 00A3
0156 00A3 E5 2B     mov a,lastkey        ; detect alt key for special
0157 00A5 C3        clr c                ; for special functions
0158 00A6 94 11     subb a,#11h
0159 00A8 70 01     jnz notalt
0160 00AA
0161 00AA 00        nop                ; alt code goes here
0162 00AB
0163 00AB          notalt:
0164 00AB E5 2A     mov a,NBytes
0165 00AD C3        clr c
0166 00AE 94 08     subb a,#INMAX        ; limit the input buffer to INMAX
0167 00B0 40 01     jc notdone          ; if data is buffered
0168 00B2 ; then buffer overflow
0169 00B2 00        nop                ; code goes here
0171 00B3
0172 00B3          notdone:
0173 00B3
0174 00B3 80 D1     sjmp main           ; go back to start

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0175 00B5
0176 00B5 ;***** End of Main loop *****
0177 00B5
0178 00B5
0179 00B5 ; ***** External int 0 ISR *****;
0180 00B5 ;
0181 00B5 ; keyboard interrupt service routine ;
0182 00B5 ;
0183 00B5 ; *****;
0184 00B5
0185 00B5
0186 00B5 kbd:
0187 00B5 C0 D0 push psw ; save .equ during ISR
0188 00B7 C0 E0 push acc
0189 00B9
0190 00B9 85 29 E0 mov acc,NBits ; NBits=bit number next expected
0191 00BC ; from the keyboard
0192 00BC B4 00 02 cjne a,#0,bit1_8 ; if not bit 0 then bit 1 to 8
0193 00BF
0194 00BF
0195 00BF
0196 00BF ;***** Keyboard Bit 0 *****
0197 00BF
0198 00BF bit0: ; discard bit 0 - Start bit
0199 00BF 80 70 sjmp bump
0200 00C1
0201 00C1
0202 00C1 ;***** Keyboard Bit 1-8 *****
0203 00C1
0204 00C1 bit1_8:
0205 00C1 B4 09 00 cjne a,#9,$+3 ; CY flag is set if acc < 9
0206 00C4 50 0C jnc bit9
0207 00C6
0208 00C6 A2 82 mov c,KEYDAT ; read data for keyboard data line
0209 00C8 E5 2C mov a,keytemp ; data arrives least sig bit 1st
0210 00CA 03 rr a ; hence old value is rotated and new
0211 00CB 92 E7 mov a.7,c ; bit is or'ed to the msb
0212 00CD 85 E0 2C mov keytemp,acc
0213 00D0 80 5F sjmp bump
0214 00D2
0215 00D2
0216 00D2 ;***** Bit 9 *****
0217 00D2
0218 00D2 bit9:
0219 00D2 B4 09 02 cjne a,#9,bit10
0220 00D5
0221 00D5 80 5A sjmp bump ; parity check code would go here
0222 00D7
0223 00D7
0224 00D7
0225 00D7 ;***** Bit 10 *****
0226 00D7
0227 00D7 ; The stop bit - Key Scan is now complete so convert to ASCII
0228 00D7
0229 00D7 bit10:
0230 00D7 85 2C E0 mov acc,keytemp ; get next key
0231 00DA
0232 00DA B4 12 14 cjne a,#12h,notls ; is it the left shift char?
0233 00DD
0234 00DD
0235 00DD ;***** Left Shift has Been Pressed *****
0236 00DD
0237 00DD 85 2B E0 mov acc,lastkey ; if last key was
0238 00E0 B4 F0 07 cjne a,#0f0h,makels ; $f0 then shift is released
0239 00E3
0240 00E3 C2 09 clr Shift ; next keys will be unshifted
0241 00E5 75 2B 12 mov lastkey,#12h ; copy left shift key to last key

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0242 00E8 80 3F      sjmp tidy
0243 00EA
0244 00EA      makels:
0245 00EA D2 09      setb Shift                ; next keys will be shifted
0246 00EC 75 2B 12  mov lastkey,#12h        ; copy left shift key to last key
0247 00EF 80 38      sjmp tidy
0248 00F1
0249 00F1      ;***** End of Shift Routine *****
0250 00F1
0251 00F1      notls:
0252 00F1      ;      mov acc,keytemp          ; get next key
0253 00F1 B4 14 03  cjne a,#14h,notctrl    ; is it a control char?
0254 00F4
0255 00F4
0256 00F4      ;***** Control State *****
0257 00F4
0258 00F4 00      nop                      ; control state goes here
0259 00F5 80 32      sjmp tidy
0260 00F7
0261 00F7      ;***** End of Control State *****
0262 00F7
0263 00F7      notctrl:
0264 00F7
0265 00F7 B4 F0 04  cjne a,#0f0h,notbreak ; if current key $f0 then break
0266 00FA
0267 00FA      ;***** Key Break *****
0268 00FA
0269 00FA F5 2B      mov lastkey,a            ; record break code in last key
0270 00FC 80 2B      sjmp tidy                ; but don't store in the buffer
0271 00FE
0272 00FE      notbreak:
0273 00FE
0274 00FE 85 2B E0  mov acc,lastkey          ; if last key was $f0 then
0275 0101 B4 F0 05  cjne a,#0f0h,not_f0    ; ignore the next scan code
0276 0104
0277 0104 75 2B 00  mov lastkey,#0            ; which is a break code
0278 0107 80 20      sjmp tidy
0279 0109
0280 0109      not_f0:
0281 0109
0282 0109
0283 0109      ;***** Normal Key Press *****
0284 0109
0285 0109~      #ifdef buffered
0286 0109~
0287 0109~      ;***** Buffered Code *****
0288 0109~
0289 0109~      ; buffered code
0290 0109~      push 0                    ; r0 used as an indirect pointer
0291 0109~      ; so save it
0292 0109~      mov acc,#keybuff         ; copy data into keyboard
0293 0109~      add a,NBytes
0294 0109~      mov r0,a
0295 0109~
0296 0109~      mov a,keytemp            ; get current key
0297 0109~      mov lastkey,a           ; & copy to lastkey
0298 0109~
0299 0109~      push dp                  ; dp used to point to xlat tables
0300 0109~      push dpl                 ; since in ISR save dp contents
0301 0109~
0302 0109~      jb Shift,shifted        ; if in unshifted state
0303 0109~      mov dpnr,#unshift       ; use the unshift table
0304 0109~      sjmp skip1
0305 0109~
0306 0109~      shifted:
0307 0109~      mov dpnr,#shift         ; else use the shift table
0308 0109~

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0309 0109~ skip1:
0310 0109~      movc a,@a+dptr          ; translate char in Acc to Ascii
0311 0109~
0312 0109~      pop dpl                ; restore the data pointer
0313 0109~      pop dph
0314 0109~
0315 0109~      cjne a,#0,Not0       ; if data is zero discard
0316 0109~
0317 0109~      ; sjmp NoSave          ; discard code goes here
0318 0109~
0319 0109~ Not0:
0320 0109~      mov r0,#keybuff      ; Save ascii value in buffer
0321 0109~      mov @r0,a          ; buffered keyboard entry
0322 0109~      inc NBytes
0323 0109~
0324 0109~ NoSave:
0325 0109~      pop 0          ; restore r0
0326 0109~
0327 0109~      ;**** End of Buffered Code
0328 0109~
0329 0109~ #endif
0330 0109
0331 0109
0332 0109 #define unbuffered 1
0333 0109
0334 0109 #ifdef unbuffered
0335 0109
0336 0109 E5 2C      mov a,keytemp      ; get current key
0337 010B F5 2B      mov lastkey,a      ; & copy to lastkey
0338 010D
0339 010D C0 83      push dph          ; dp used to point to xlat tables
0340 010F C0 82      push dpl          ; since in ISR save dp contents
0341 0111
0342 0111 20 09 05    jb Shift,shifted      ; if in unshifted state
0343 0114 90 01 EB    mov dptr,#unshift      ; use the unshift table
0344 0117 80 03      sjmp skip1
0345 0119
0346 0119 shifted:
0347 0119 90 02 6B    mov dptr,#shift        ; else use the shift table
0348 011C
0349 011C skip1:
0350 011C 93      movc a,@a+dptr          ; translate char in Acc to Ascii
0351 011D
0352 011D D0 82      pop dpl                ; restore the data pointer
0353 011F D0 83      pop dph
0354 0121
0355 0121 B4 00 00    cjne a,#0,Not0       ; if data is zero discard
0356 0124
0357 0124      ; sjmp tidy          ; discard code goes here
0358 0124
0359 0124 Not0:
0360 0124 F5 2D      mov keybuff,a        ; store in keyboard buffer
0361 0126 75 2A 01    mov NBytes,#1       ; mark byte read
0362 0129
0363 0129 tidy:
0364 0129 75 29 00    mov NBits,#0        ; clear flags ready for next key
0365 012C 75 2C 00    mov keytemp,#0
0366 012F 80 02      sjmp intdone
0367 0131
0368 0131      ;***** End of Keyboard Translation and Save *****
0369 0131
0370 0131
0371 0131      ;***** Normal unfinished key exit *****
0372 0131
0373 0131 bump:
0374 0131 05 29      inc NBits          ; inc number of bits read so far
0375 0133

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0376 0133   intdone:
0377 0133 D0 E0   pop  acc
0378 0135 D0 D0   pop  psw
0379 0137 32     reti
0380 0138
0381 0138   ;***** End of Ext Int 0 ISR   *****
0382 0138
0383 0138
0384 0138
0385 0138
0386 0138   ;***** I2C CODE SLAVE   *****
0387 0138
0388 0138   i2cint:                               ; I2C interrupt entry point
0389 0138 D2 03   setb  i2c_busy                          ; semaphore on xmtdata buffer
0390 013A
0391 013A C0 D0   push psw                               ; save registers used in ISR
0392 013C C0 E0   push  acc
0393 013E C0 00   push  0                               ; R0 no bank switching
0394 0140
0395 0140 C2 AC   clr   EI2                               ; make I2C ISR interruptable
0396 0142 31 E9   acall  clrint                          ; execute a reti
0397 0144
0398 0144   slave:
0399 0144 75 27 03  mov  i2ctime,#3                          ; set up I2C timeout watchdog 30 mS
0400 0147
0401 0147 75 98 9C  mov  I2CON,#BCARL+BCSTP+BCSTR+BCXA
0402 014A                               ; clear start status
0403 014A
0404 014A 30 9E FD  jnb  ATN,$                          ; wait for next data bit
0405 014D 75 22 07  mov  bitcnt,#7
0406 0150
0407 0150 31 C9   acall  rcvrb2                          ; get remainder of slave address
0408 0152 F5 24   mov  adrrcvd,a
0409 0154 C2 E0   clr   a.0                               ; mask r/w bit to check address
0410 0156 B4 88 3B  cjne  a,#IICADD,goidle                          ; idle again if not for us
0411 0159
0412 0159 20 20 1F  jb   rwflag,read                          ; test for read or write
0413 015C
0414 015C
0415 015C
0416 015C   ;***** I2C Receive Code   *****
0417 015C
0418 015C 78 04   mov  r0,#rcvdat                          ; r0 points to data buffer
0419 015E 75 23 01  mov  bytecnt,#MAXBYTES
0420 0161
0421 0161   rcvloop:
0422 0161 31 BF   acall  sendack                          ; acknowledge the address
0423 0163 31 C6   acall  rcvbyte                          ; wait for the next data byte
0424 0165 30 9D 0F  jnb  DRDY,exitwr                          ; end of frame
0425 0168 F6     mov  @r0,a
0426 0169 08     inc  r0
0427 016A D5 23 F4  djnz  bytecnt,rcvloop
0428 016D
0429 016D   ; no more room
0430 016D
0431 016D 31 BF   acall  sendack                          ; ack last byte
0432 016F 31 C6   acall  rcvbyte                          ; get but discard next one
0433 0171 75 99 80  mov  I2DAT,#80h                          ; send neg ack
0434 0174 30 9E FD  jnb  ATN,$                          ; wait till gone
0435 0177   exitwr:
0436 0177
0437 0177 D2 01   setb  rcvrd
0438 0179 80 13   sjmp  msgend
0439 017B
0440 017B   ;***** End of Receive Routine   *****
0441 017B
0442 017B

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0443 017B ;***** I2C Transmit Code *****
0444 017B
0445 017B read:
0446 017B 78 06 mov r0,#xmtdat ; r0 points to data buffer
0447 017D 75 23 01 mov bytecnt,#MAXBYTES
0448 0180 31 BF acall sendack ; acknowledge address
0449 0182
0450 0182 txloop:
0451 0182 E6 mov a,@r0 ; get next data byte
0452 0183 08 inc r0 ; bump buffer pointer
0453 0184 31 A7 acall xmitbyte ; transmit the byte to the I2C
0454 0186 20 00 03 jb noack,exitrd ; if not acknowledged then exit
0455 0189 D5 23 F6 djnz bytecnt,txloop
0456 018C
0457 018C 80 00 exitrd: sjmp msgend
0458 018E
0459 018E ;***** End of I2C transmit *****
0460 018E
0461 018E
0462 018E ;***** Repeated start state *****
0463 018E msgend:
0464 018E 30 9E FD jnb ATN,$ ; wait for stop or repeated start
0465 0191 20 9B B0 jb STR,slave ; if repeat start do again
0466 0194
0467 0194 ; stop so enter idle mode
0468 0194
0469 0194 goidle:
0470 0194 75 27 00 mov i2ctime,#0 ; stop I2C timeout
0471 0197 75 98 F4 mov I2CON,#BCSTP+BCXA+BCDR+BCARL+BIDLE
0472 019A
0473 019A D0 00 pop 0 ; restore state before I2C ISR
0474 019C D0 E0 pop acc
0475 019E D0 D0 pop psw
0476 01A0
0477 01A0 D2 AC setb EI2
0478 01A2 D2 02 setb sent_flag ; flag to say data has been sent
0479 01A4 C2 03 clr i2c_busy ; flag denotes exiting I2C routine
0480 01A6
0481 01A6 22 ret
0482 01A7
0483 01A7
0484 01A7 ;***** General I2C routines *****
0485 01A7
0486 01A7 xmitbyte: ; transmit data in acc to I2C
0487 01A7 75 22 08 mov bitcnt,#8
0488 01AA
0489 01AA xmitbit:
0490 01AA F5 99 mov I2DAT,a
0491 01AC 23 rl a
0492 01AD 30 9E FD jnb ATN,$
0493 01B0 D5 22 F7 djnz bitcnt,xmitbit
0494 01B3 75 98 A0 mov I2CON,#BCDR+BCXA ; switch to rcv mode
0495 01B6 30 9E FD jnb ATN,$ ; wait for ack
0496 01B9 85 99 20 mov flags,I2DAT ; save ack bit
0497 01BC 22 ret
0498 01BD
0499 01BD
0500 01BD
0501 01BD rdack: ; receives data byte then sends ack
0502 01BD 31 C6 acall rcvbyte ; I2C receive, data returned in acc
0503 01BF
0504 01BF sendack:
0505 01BF 75 99 00 mov I2DAT,#0 ; I2C ack = data low and clock high
0506 01C2 30 9E FD jnb ATN,$
0507 01C5 22 ret
0508 01C6
0509 01C6 rcvbyte: ; I2C receive, data returned in acc

```


Connecting a PC keyboard to the I²C-bus

AN434

```

0510 01C6 75 22 08  mov bitcnt,#8
0511 01C9
0512 01C9 E4      recvb2: clr a
0513 01CA
0514 01CA 45 99      rbit:  orl a,I2DAT
0515 01CC 23        rl a
0516 01CD 30 9E FD  jnb ATN,$
0517 01D0 30 9D 06  jnb DRDY,rbex          ; exit if not a data bit
0518 01D3 D5 22 F4  djnz bitcnt,rbit
0519 01D6
0520 01D6 A2 9F      mov c,RDAT            ; get last bit - do not clear ATN
0521 01D8 33        rlc a                ; shift into byte
0522 01D9
0523 01D9 rbex:
0524 01D9 22        ret
0525 01DA
0526 01DA ;         IIC timer interrupt service
0527 01DA timerI:
0528 01DA 75 A8 00  mov ien0,#0          ; break point address in ICE751
0529 01DD D2 DD      setb CLRTI          ; clear the interrupt
0530 01DF fixup:
0531 01DF 75 D8 00  mov I2CFG,#0        ; turn off I2C
0532 01E2 75 98 BC  mov I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP
0533 01E5           ; reset I2C flags
0534 01E5 31 E9      acall clrnt
0535 01E7 01 5C      ajmp reset          ; restart program
0536 01E9
0537 01E9
0538 01E9 ;***** call here to make code interruptible *****
0539 01E9
0540 01E9 clrnt:
0541 01E9 32        reti
0542 01EA
0543 01EA ;***** unused interrupts are vectored to here *****
0544 01EA badint:
0545 01EA 32        reti
0546 01EB
0547 01EB #include attable.h
0001+ 01EB unshift          ; scan code
0002+ 01EB 00       .byte 0          ; 0
0003+ 01EC 00       .byte 0          ; 1 - f9
0004+ 01ED 00       .byte 0          ; 2 - f7
0005+ 01EE 00       .byte 0          ; 3 - f5
0006+ 01EF 00       .byte 0          ; 4 - f3
0007+ 01F0 00       .byte 0          ; 5 - f1
0008+ 01F1 00       .byte 0          ; 6 - f2
0009+ 01F2 00       .byte 0          ; 7 - f2
0010+ 01F3 00       .byte 0          ; 8 -
0011+ 01F4 00       .byte 0          ; 9 - f10
0012+ 01F5 00       .byte 0          ; a - f8
0013+ 01F6 00       .byte 0          ; b - f6
0014+ 01F7 00       .byte 0          ; c - f4
0015+ 01F8 09       .byte 09h        ; d - tab
0016+ 01F9 60       .byte ''         ; e - '
0017+ 01FA 00       .byte 0          ; f -
0018+ 01FB
0019+ 01FB 00       .byte 0          ; 10
0020+ 01FC 00       .byte 0          ; 11 - left shift
0021+ 01FD 00       .byte 0          ; 12
0022+ 01FE 00       .byte 0          ; 13
0023+ 01FF 00       .byte 0          ; 14
0024+ 0200 71       .byte 'q'        ; 15
0025+ 0201 31       .byte '1'        ; 16
0026+ 0202 00       .byte 0          ; 17
0027+ 0203 00       .byte 0          ; 18
0028+ 0204 00       .byte 0          ; 19
0029+ 0205 7A       .byte 'z'        ; 1a

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0030+ 0206 73      .byte 's' ; 1b
0031+ 0207 61      .byte 'a' ; 1c
0032+ 0208 77      .byte 'w' ; 1d
0033+ 0209 32      .byte '2' ; 1e
0034+ 020A 00      .byte 0 ; 1f
0035+ 020B
0036+ 020B 00      .byte 0 ; 20
0037+ 020C 63      .byte 'c' ; 21
0038+ 020D 78      .byte 'x' ; 22
0039+ 020E 64      .byte 'd' ; 23
0040+ 020F 65      .byte 'e' ; 24
0041+ 0210 34      .byte '4' ; 25
0042+ 0211 33      .byte '3' ; 26
0043+ 0212 00      .byte 0 ; 27
0044+ 0213 00      .byte 0 ; 28
0045+ 0214 20      .byte ' ' ; 29
0046+ 0215 76      .byte 'v' ; 2a
0047+ 0216 66      .byte 'f' ; 2b
0048+ 0217 74      .byte 't' ; 2c
0049+ 0218 72      .byte 'r' ; 2d
0050+ 0219 35      .byte '5' ; 2e
0051+ 021A 00      .byte 0 ; 2f
0052+ 021B
0053+ 021B 00      .byte 0 ; 30
0054+ 021C 6E      .byte 'n' ; 31
0055+ 021D 62      .byte 'b' ; 32
0056+ 021E 68      .byte 'h' ; 33
0057+ 021F 67      .byte 'g' ; 34
0058+ 0220 79      .byte 'y' ; 35
0059+ 0221 36      .byte '6' ; 36
0060+ 0222 00      .byte 0 ; 37
0061+ 0223 00      .byte 0 ; 38
0062+ 0224 00      .byte 0 ; 39
0063+ 0225 6D      .byte 'm' ; 3a
0064+ 0226 6A      .byte 'j' ; 3b
0065+ 0227 75      .byte 'u' ; 3c
0066+ 0228 37      .byte '7' ; 3d
0067+ 0229 38      .byte '8' ; 3e
0068+ 022A 00      .byte 0 ; 3f
0069+ 022B
0070+ 022B 00      .byte 0 ; 40
0071+ 022C 2C      .byte ',' ; 41
0072+ 022D 6B      .byte 'k' ; 42
0073+ 022E 69      .byte 'i' ; 43
0074+ 022F 6F      .byte 'o' ; 44
0075+ 0230 30      .byte '0' ; 45
0076+ 0231 39      .byte '9' ; 46
0077+ 0232 00      .byte 0 ; 47
0078+ 0233 00      .byte 0 ; 48
0079+ 0234 2E      .byte '.' ; 49
0080+ 0235 2F      .byte '/' ; 4a
0081+ 0236 6C      .byte 'l' ; 4b
0082+ 0237 00      .byte ';' ; 4c
0083+ 0238 70      .byte 'p' ; 4d
0084+ 0239 2D      .byte '-' ; 4e
0085+ 023A 00      .byte 0 ; 4f
0086+ 023B
0087+ 023B 00      .byte 0 ; 50
0088+ 023C 2C      .byte ',' ; 51
0089+ 023D 00      .byte 0 ; 52
0090+ 023E 00      .byte 0 ; 53
0091+ 023F 5B      .byte '[' ; 54
0092+ 0240 3D      .byte '=' ; 55
0093+ 0241 00      .byte 0 ; 56
0094+ 0242 00      .byte 0 ; 57
0095+ 0243 00      .byte 0 ; 58
0096+ 0244 00      .byte 0 ; 59

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0097+ 0245 0D      .byte 13      ; 5a
0098+ 0246 5D      .byte ']'     ; 5b
0099+ 0247 00      .byte 0       ; 5c
0100+ 0248 5C      .byte 92      ; 5d
0101+ 0249 00      .byte 0       ; 5e
0102+ 024A 00      .byte 0       ; 5f
0103+ 024B
0104+ 024B 00      .byte 0       ; 60
0105+ 024C 00      .byte 0       ; 61
0106+ 024D 00      .byte 0       ; 62
0107+ 024E 00      .byte 0       ; 63
0108+ 024F 00      .byte 0       ; 64
0109+ 0250 00      .byte 0       ; 65
0110+ 0251 08      .byte 8       ; 66
0111+ 0252 00      .byte 0       ; 67
0112+ 0253 00      .byte 0       ; 68
0113+ 0254 00      .byte 0       ; 69
0114+ 0255 00      .byte 0       ; 6a
0115+ 0256 00      .byte 0       ; 6b
0116+ 0257 00      .byte 0       ; 6c
0117+ 0258 00      .byte 0       ; 6d
0118+ 0259 00      .byte 0       ; 6e
0119+ 025A 00      .byte 0       ; 6f
0120+ 025B
0121+ 025B 00      .byte 0       ; 70
0122+ 025C 7F      .byte 127     ; 71
0123+ 025D 00      .byte 0       ; 72
0124+ 025E 00      .byte 0       ; 73
0125+ 025F 00      .byte 0       ; 74
0126+ 0260 1B      .byte 27      ; 75
0127+ 0261 00      .byte 0       ; 76
0128+ 0262 00      .byte 0       ; 77
0129+ 0263 00      .byte 0       ; 78
0130+ 0264 2B      .byte '+'     ; 79
0131+ 0265 00      .byte 0       ; 7a
0132+ 0266 2D      .byte '-'     ; 7b
0133+ 0267 2A      .byte '*'     ; 7c
0134+ 0268 00      .byte 0       ; 7d
0135+ 0269 00      .byte 0       ; 7e
0136+ 026A 00      .byte 0       ; 7f
0137+ 026B
0138+ 026B      shift:      ; scan code
0139+ 026B 00      .byte 0       ; 0
0140+ 026C 00      .byte 0       ; 1 - f9
0141+ 026D 00      .byte 0       ; 2 - f7
0142+ 026E 00      .byte 0       ; 3 - f5
0143+ 026F 00      .byte 0       ; 4 - f3
0144+ 0270 00      .byte 0       ; 5 - f1
0145+ 0271 00      .byte 0       ; 6 - f2
0146+ 0272 00      .byte 0       ; 7 - f2
0147+ 0273 00      .byte 0       ; 8 -
0148+ 0274 00      .byte 0       ; 9 - f10
0149+ 0275 00      .byte 0       ; a - f8
0150+ 0276 00      .byte 0       ; b - f6
0151+ 0277 00      .byte 0       ; c - f4
0152+ 0278 00      .byte 0       ; d - tab
0153+ 0279 7E      .byte '~'     ; e - ~
0154+ 027A 00      .byte 0       ; f -
0155+ 027B
0156+ 027B 00      .byte 0       ; 10
0157+ 027C 00      .byte 0       ; 11 -
0158+ 027D 00      .byte 0       ; 12
0159+ 027E 00      .byte 0       ; 13
0160+ 027F 00      .byte 0       ; 14
0161+ 0280 51      .byte 'Q'     ; 15
0162+ 0281 21      .byte '!'     ; 16
0163+ 0282 00      .byte 0       ; 17

```

Connecting a PC keyboard to the I²C-bus

AN434

```

0164+ 0283 00      .byte 0      ; 18
0165+ 0284 00      .byte 0      ; 19
0166+ 0285 5A      .byte 'Z'    ; 1a
0167+ 0286 53      .byte 'S'    ; 1b
0168+ 0287 41      .byte 'A'    ; 1c
0169+ 0288 57      .byte 'W'    ; 1d
0170+ 0289 40      .byte '@'    ; 1e
0171+ 028A 00      .byte 0      ; 1f
0172+ 028B
0173+ 028B 00      .byte 0      ; 20
0174+ 028C 43      .byte 'C'    ; 21
0175+ 028D 58      .byte 'X'    ; 22
0176+ 028E 44      .byte 'D'    ; 23
0177+ 028F 45      .byte 'E'    ; 24
0178+ 0290 24      .byte '$'    ; 25
0179+ 0291 23      .byte '#'    ; 26
0180+ 0292 00      .byte 0      ; 27
0181+ 0293 00      .byte 0      ; 28
0182+ 0294 20      .byte ' '    ; 29
0183+ 0295 56      .byte 'V'    ; 2a
0184+ 0296 46      .byte 'F'    ; 2b
0185+ 0297 54      .byte 'T'    ; 2c
0186+ 0298 52      .byte 'R'    ; 2d
0187+ 0299 25      .byte ' '    ; 2e
0188+ 029A 00      .byte 0      ; 2f
0189+ 029B
0190+ 029B 00      .byte 0      ; 30
0191+ 029C 4E      .byte 'N'    ; 31
0192+ 029D 42      .byte 'B'    ; 32
0193+ 029E 48      .byte 'H'    ; 33
0194+ 029F 47      .byte 'G'    ; 34
0195+ 02A0 59      .byte 'Y'    ; 35
0196+ 02A1 5E      .byte '^'    ; 36
0197+ 02A2 00      .byte 0      ; 37
0198+ 02A3 00      .byte 0      ; 38
0199+ 02A4 00      .byte 0      ; 39
0200+ 02A5 4D      .byte 'M'    ; 3a
0201+ 02A6 4A      .byte 'J'    ; 3b
0202+ 02A7 55      .byte 'U'    ; 3c
0203+ 02A8 26      .byte '&'    ; 3d
0204+ 02A9 2A      .byte '*'    ; 3e
0205+ 02AA 00      .byte 0      ; 3f
0206+ 02AB
0207+ 02AB 00      .byte 0      ; 40
0208+ 02AC 3C      .byte '<'    ; 41
0209+ 02AD 4B      .byte 'K'    ; 42
0210+ 02AE 49      .byte 'I'    ; 43
0211+ 02AF 4F      .byte 'O'    ; 44
0212+ 02B0 29      .byte ')'    ; 45
0213+ 02B1 28      .byte '('    ; 46
0214+ 02B2 00      .byte 0      ; 47
0215+ 02B3 00      .byte 0      ; 48
0216+ 02B4 3E      .byte '>'    ; 49
0217+ 02B5 3F      .byte '?'    ; 4a
0218+ 02B6 4C      .byte 'L'    ; 4b
0219+ 02B7 3A      .byte ':'    ; 4c
0220+ 02B8 50      .byte 'P'    ; 4d
0221+ 02B9 5F      .byte '_'    ; 4e
0222+ 02BA 00      .byte 0      ; 4f
0223+ 02BB
0224+ 02BB 00      .byte 0      ; 50
0225+ 02BC 00      .byte 0      ; 51
0226+ 02BD 22      .byte '"'    ; 52
0227+ 02BE 00      .byte 0      ; 53
0228+ 02BF 7B      .byte '{'    ; 54
0229+ 02C0 2B      .byte '+'    ; 55
0230+ 02C1 00      .byte 0      ; 56

```

Connecting a PC keyboard to the I²C-bus

AN434

```
0231+ 02C2 00      .byte 0      ; 57
0232+ 02C3 00      .byte 0      ; 58
0233+ 02C4 00      .byte 0      ; 59
0234+ 02C5 0D      .byte 13     ; 5a
0235+ 02C6 7D      .byte '}'    ; 5b
0236+ 02C7 00      .byte 0      ; 5c
0237+ 02C8 7C      .byte '|'    ; 5d
0238+ 02C9 00      .byte 0      ; 5e
0239+ 02CA 00      .byte 0      ; 5f
0240+ 02CB
0241+ 02CB 00      .byte 0      ; 60
0242+ 02CC 00      .byte 0      ; 61
0243+ 02CD 00      .byte 0      ; 62
0244+ 02CE 00      .byte 0      ; 63
0245+ 02CF 00      .byte 0      ; 64
0246+ 02D0 00      .byte 0      ; 65
0247+ 02D1 08      .byte 8      ; 66
0248+ 02D2 00      .byte 0      ; 67
0249+ 02D3 00      .byte 0      ; 68
0250+ 02D4 31      .byte '1'    ; 69
0251+ 02D5 00      .byte 0      ; 6a
0252+ 02D6 34      .byte '4'    ; 6b
0253+ 02D7 37      .byte '7'    ; 6c
0254+ 02D8 00      .byte 0      ; 6d
0255+ 02D9 00      .byte 0      ; 6e
0256+ 02DA 00      .byte 0      ; 6f
0257+ 02DB
0258+ 02DB 30      .byte '0'    ; 70
0259+ 02DC 2E      .byte '.'    ; 71
0260+ 02DD 32      .byte '2'    ; 72
0261+ 02DE 35      .byte '5'    ; 73
0262+ 02DF 36      .byte '6'    ; 74
0263+ 02E0 38      .byte '8'    ; 75
0264+ 02E1 1B      .byte 27     ; 76
0265+ 02E2 00      .byte 0      ; 77
0266+ 02E3 00      .byte 0      ; 78
0267+ 02E4 2B      .byte '+'    ; 79
0268+ 02E5 33      .byte '3'    ; 7a
0269+ 02E6 2D      .byte '-'    ; 7b
0270+ 02E7 00      .byte 0      ; 7c
0271+ 02E8 39      .byte '9'    ; 7d
0272+ 02E9 00      .byte 0      ; 7e
0273+ 02EA 00      .byte 0      ; 7f
0274+ 02EB
0275+ 02EB
0548 02EB
0549 02EB      .end
tasm: Number of errors = 0
```

I²C routines for 8XC528

AN438

Philips Semiconductors Application Note EIE/AN90015

Summary

This application note presents a set of software routines to drive the I²C interface in 8xC528 type of microcontrollers. A description of the I²C interface is given. Examples show how to use these routines in PL/M-51, C and assembly source code.

1.0 INTRODUCTION

This application note describes the I²C interface of the 8xC528 microcontroller and gives a set of routines in application programs to drive this interface.

Chapter 2.0 gives a hardware description of the bit level I²C. It gives an overview of what functions are done in hardware by the interface and the functions that should be implemented by software. The registers described are accessible with software and control the I²C interface.

Chapter 3.0 gives a description of the routines that may be used by the application program. The routines are written in such a way that the I²C interface becomes transparent to the user. the slave program is described in more detail, because this routine may be adapted by the user for his specific application.

Chapter 4.0 gives simple example programs that show how to use the routines in assemble, PL/M and C application programs.

References:

- The I²C-bus specification 9398 358 10011
- 80C51-based 8-bit Microcontrollers Data handbook IC20
- PLM51 I²C Software Interface IIC51 ETV/AN89004

2.0 THE I²C INTERFACE

2.1 Characteristics of I²C Interface

The Block diagram of the bit-level I²C interface is shown on page 165. P1.6/SCL and P1.7/SDA are the serial I/O pins. These two pins meet the I²C specification concerning the input levels and output drive capability. Consequently, these pins have an open drain configuration. All four modes of the I²C bus can be used:

- Master transmitter
- Master receiver
- Slave transmitter
- Slave receiver

The advantages of using the bit-level I²C hardware compared with a full software implementation are:

- Higher bit rate
- No critical software timing requirements
- Less software overhead
- More reliable data transfer

The bit-level I²C hardware can perform the following functions:

- Filtering the incoming serial data and clock signals. Glitches shorter than 4 XTAL periods are rejected.
- Recognition of a START or STOP condition.
- Generating an interrupt request after reception of a START condition.
- Setting the Bus Busy flag when a START condition is detected.
- Clearing the Bus Busy flag when a STOP condition is detected.
- Recognition of a serial clock pulse on the SCL line.
- Latching the serial data bit on the SDA line at every rising edge on the SCL line.
- Stretching the LOW period of the serial clock SCL to synchronizer with external master devices.
- Setting the Read Bit Finished (RBF) or Write Bit Finished (WBF) flag is an error free bit transfer has occurred.
- Setting a Clock LOW-to-HIGH (CLH) flag when a leading edge is detected on the SCL line.
- Generation of serial clock pulse on SCL in master mode.

The following functions must be done with software:

- Handling the I²C interrupt caused by a detected START condition.
- Conversion of serial to parallel data when receiving.
- Conversion of parallel to serial data when transmitting.
- Comparing received slave address with own slave address.
- Interpretation of acknowledge information.
- Guarding the I²C status if the RBF and WBF flags indicate a not regular bit transfer.
- Generating START/STOP conditions when in master mode.
- Handling bus arbitration when in master mode.

I²C routines for 8XC528

AN438

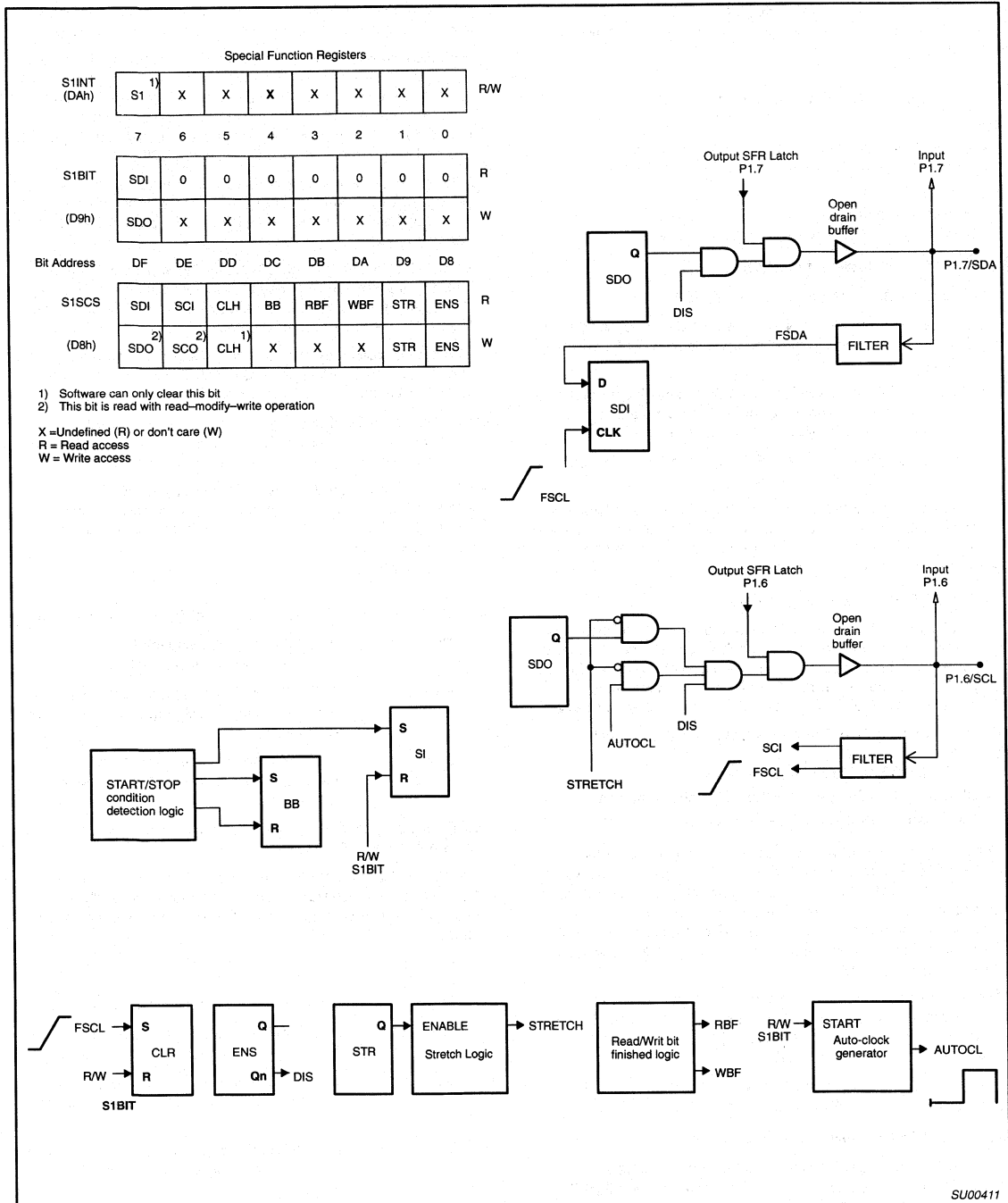


Figure 1.

I²C routines for 8XC528

AN438

2.2 Control and Status Registers

Control of the I²C bus hardware is done via 3 Special Function Registers:

S1INT

This register contains the serial interrupt flag SI.

S1BIT

For read, this register contains the received bit SDI.

For write, this register contains bit SDO to be transmitted.

S1SCS

For read, this register contains status information.

For write, this register is used as control register.

2.2.1 S1INT: I²C Interrupt Register

- **S1INT.7** is the Serial Interrupt request flag (SI).

If the serial I/O is enabled (ENS = 1), then a START condition will be detected and the SI flag is set on the falling edge of the filtered SCL signal.

Provided that EA (global enable) and ES1 (enable I²C interrupt) are set (in the interrupt enable IE register), SI generates an interrupt that will start the slave address receive routine.

SI is cleared by accessing the S1BIT register or by writing '00H' to S1INT. SI cannot be set by software.

After reception of a START condition, the LOW period of the SCL pulse is stretched, suspending serial transfer to allow the software to take appropriate action. This clock stretching is ended by accessing the S1BIT register.

2.2.2 S1BIT: Single Bit Data Register

- **S1BIT.7** contains two physical latches: the Serial Data Output (SDO) latch for a write operation, and the filtered Serial Data Input (SDI) latch for a read operation. SDI data is latched on the rising edge of the filtered SCL pulse. S1BIT.7 accesses the same physical latches as S1SCS.7, but S1BIT.7 is not bit addressable.

Reading or writing S1BIT register starts the next additional actions:

- SI, CLH, RBF and WBF flags are cleared.
- Stretching the LOW period of the SCL clock is finished.
- Auto-clock pulse is started if enabled.

The auto-lock is an active HIGH SCL pulse that starts 28 Xtal periods after an access to S1BIT. SCL remains high for 100 Xtal periods. If the SCL line is kept LOW by any device that wants to hold up the bus transfer, the auto-clock counter still runs for 20 Xtal periods to try to make SCL high and then go into a wait-state. This will result in a minimum SCL HIGH time of 80 Xtal periods (5 μ s at $f_{Xtal} = 16$ MHz).

The auto-clock signal will be inhibited if the SCO flag in the S1SCS register is set to '1'. SCL pulses must then be generated by software. In this situation, access to S1BIT may be used to clear the SI, CLH, RBF and WBF flags.

A quick check on a successful bit transfer from/to SDO/SDI is carried out by testing only the RBF or WBF flag (see 2.2.3).

2.2.3 S1SCS: Control and Status Register

- **S1SCS.7** represents two physical latches, the Serial Data Output (SDO) latch for write operations and the Serial Data Input (SDI) latch for read operations. S1SCS.7 accesses the same physical latches as S1BIT.7, but S1SCS.7 is bit addressable. However, a

read or write operation of S1SCS.7 does not start an auto-lock pulse, with not finish clock stretching, and will not clear flags.

- **S1SCS.6** represents two physical latches, the Serial Clock Output (SCO) latch for write operations and the Serial Clock Input (SCI) latch for read operations. The output of SCO is "OR-ed" with the auto-clock pulse. If SCO = '1' the auto-clock generation is disabled and its output is LOW. Internal clock stretching logic and external devices can then pull the SCL line LOW.

If the auto-clock is not used, the SCL line has to be controlled by setting SCO = '1', waiting for CLH to become '1' and setting SCO = '0' after the specified SCL HIGH time. Data access should be done via S1SCS.7.

- **S1SCS.5** is the serial Clock LOW-to-HIGH transition flag (CLH). This flag is set by a rising edge of the filtered serial clock. CLH = '1' indicates that no devices are stretching SCL LOW, and since the last CLH reset, a new valid data bit has been latched in SDI.

CLH can be cleared by writing '0' to S1SCS.5 or by a read or write operation to the S1BIT register. Clearing CLH also clears RBF and WBF. Writing a '1' to S1SCS.5 will not affect CLH.

- **S1SCS.4** is the Bus Busy flag (BB). BB is set or cleared by hardware only. If set, it indicates that a START condition has been detected on the I²C bus. A STOP condition clears the BB flag.

- **S1SCS.3** is the Read Bit Finished flag (RBF). If RBF = 1, it indicates that a serial bit has been received and latched into SDI successfully. If during a bit transfer RBF is '0', the cause is indicated as follows:

SCI = '1'	
and	
CLH = '1'	The SCL pulse is not finished and still HIGH.
CLH = '0'	A bus device is delaying the transfer by stretching the LOW level on the SCL line.
BB = '0'	A STOP-condition has been detected during the bit transfer. This should be considered as a bus-error.
SI = '1'	A START-condition has been detected during the bit transfer. This should be considered as a bus-error.

RBF can be cleared by clearing CLH or by a read or write operation to the S1BIT register.

- **S1SCS.2** is the Write Bit Finished flag (WBF). If set, it indicates that a serial bit in SDO has been transmitted successfully. If during bit transfer WBF is '0', the following conditions may be the cause:

SCI = '1'	
and	
CLH = '1'	The SCL pulse is not finished and still HIGH.
CLH = '0'	A bus device is delaying the transfer by stretching the LOW level on the SCL line.
BB = '0'	A STOP-condition has been detected during the bit transfer. This should be considered as a bus-error.
SI = '1'	A START-condition has been detected during the bit transfer. This should be considered as a bus-error.

WBF can be cleared by clearing CLH or access to the S1BIT register.

I²C routines for 8XC528

AN438

- **S1SCS.1** is the STRetch control flag (STR). STR can be set or cleared by software only. Setting STR enables the stretching of SCL LOW periods. Stretching will occur after a falling edge on the filtered serial clock. This allows synchronization with the SCL clock signal of an external master device.

If STR is cleared, no stretching of the SCL LOW period will occur after the transfer of a serial bit.

The LOW level on the SCL line is also stretched after a START condition is received, regardless of the STR contents. The stretching of the SCL LOW period is finished by a read or write operation of the S1BIT register.

- **S1SCS.0** is the ENable Serial I/O flag (ENS).

ENS can be set or cleared by software only.

ENS = '0' disables the serial I/O. The I/O signals P1.6/SCL and P1.7/SDA are determined by the port latches of P1.6 and P1.7 (open drain). If P1.6 and P1.7 are connected to an I²C bus, then the flags SDI, SCI, CLH, and BB still monitor the I²C bus status, but will not influence the I/O lines, nor will they request an interrupt.

ENS = '1' enables the START detection and clock stretching logic. Note that the P1.6 and P1.7 latches and the SDO and SCO control flags must be set to '1' before ENS is set to avoid SCL and/or SDA to pull the lines LOW.

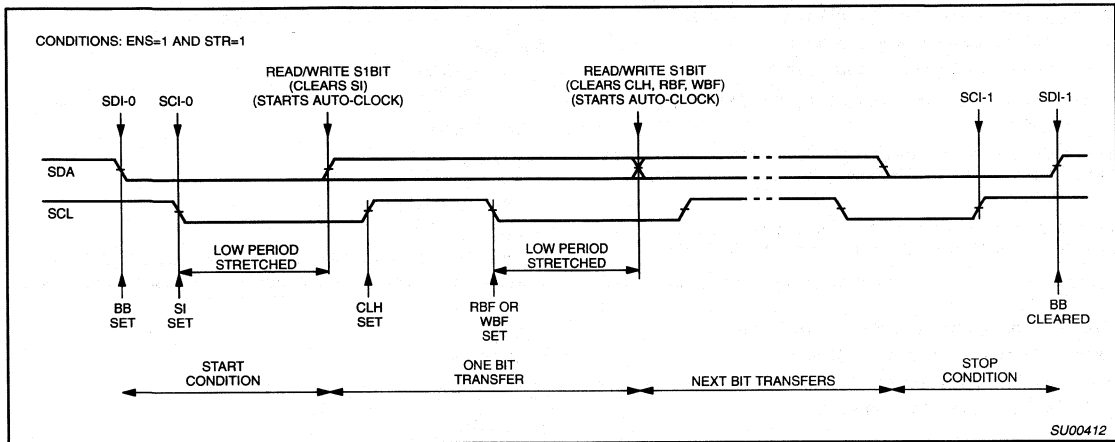


Figure 2. Example of a Serial Transfer

I²C routines for 8XC528

AN438

3.0 I²C ROUTINES

3.1 Introduction

A set of routines is written for the I²C interface that supports multi-master and slave operation. The routines are placed in a library I2C_DR.LIB. If I2C_DR.LIB is linked to an application program, only the needed object modules are linked in the output file.

The routines can be used as device driver for PL/M-51, C and 8051-assembly code. By using these routines the bit-level I²C interface is fully transparent for the user.

The routines use the following 8xC528 resources:

- Exclusive use of Register_Bank_1. Only R7 of this register bank contains static data (Own Slave Address). R0..R6 may be used by the application program when the I²C routine is finished.
- 7 bytes DATA used for parameter passing.
- 1 byte Bit-Addressable DATA for status flags.

When using routines from this library DPH, DPL, PSW (except CY) and B are not altered.

An n-bytes data buffer is used as destination or source buffer for the bytes to be received/transmitted and reside in DATA or IDATA memory space.

The code is written to generate the highest transfer rate on the I²C bus. At $f_{Xtal} = 16\text{MHz}$ this will result in a bit rate of 87.5kbit/sec.

The following software tools from BSO/Tasking are used for program development:

- OM4142 Cross Assembler 8051 for DOS: V3.0b
- OM4144 PL/M 8051 Compiler for DOS: V3.0a
- OM4136 C8051 Compiler for DOS: V1.1a
- OM4129 XRAY51 debugger: V1.4c

3.2 Functional Description

When using these routines in a PL/M application program, they must be declared EXTERNAL. In this declaration the user can specify the type returned by each procedure. All procedures (except Init_IIC and Dis_IIC) can return a BIT or BYTE, depending on the chosen EXTERNAL declaration. The BIT or BYTE returned is '0' if the I²C was successful. If a BYTE is returned, the following check bits are available for the user:

BYTE.0	An I ² C error has been detected.
BYTE.1	No ACK received.
BYTE.2	Arbitration lost.
BYTE.3	Time out error. This may be caused by an external device pulling SCL LOW.
BYTE.4	A bus error has occurred. This may be a spurious START/STOP during a bit transfer.
BYTE.5	No access to I ² C bus.
BYTE.6	0
BYTE.7	0

Note that typed procedures must be called using an expression. If the result of an I²C procedure is to be ignored, a dummy assignment must be done for a typed procedure. The examples in the following section assume that the procedures are called from a PL/M program. Examples will be given later how to use these routines with C and assembly application programs.

3.2.1 Init_IIC

Declaration

```
Init_IIC:
  PROCEDURE (Own_Slave_Address, Slave_Sub_Address)
  EXTERNAL;
  DECLARE (Own_Slave_Address, Slave_Sub_Address) BYTE;
  END;
```

Description

Init_IIC must be called after RESET, before any procedure is called. The I²C interface and I²C interrupt will be enabled. The global enable interrupt flag, however, will not be affected. This should be done afterwards. Own_Slave_Address is passed to Init_IIC for use as slave. Slave_Sub_Address is the pointer to a DATA buffer that is used for data transfer in slave mode. When used as master in a single master system, these parameters are not used.

Example

```
CALL Init_IIC (54h, Slave_Data_Buffer);
ENABLE; /* Enable Interrupts; EA=1 */
```

3.2.2 Dis_IIC

Declaration

```
Dis_IIC:
  PROCEDURE EXTERNAL;
```

Description

Dis_IIC will disable the I²C-interface and the I²C-interrupt. The I²C interface will still monitor the bus, but will not influence the SDA and SCL lines.

Example

```
CALL Dis_IIC;
```

3.2.3 IIC_Test_Device

Declaration

```
IIC_Test_Device:
  PROCEDURE (Slave_Address) [BIT|BYTE] EXTERNAL;
  DECLARE (Slave_Address) BYTE;
  END;
```

Description

IIC_Test_Device just sends the slave address to the I²C bus. It can be used to check the presence of a device on the I²C bus.

I²C Protocol

```
S-SlW-A-P      : Device is present, IIC_Error=0
S-SlW-N-P      : Device is not present, IIC_Error=1
```

Example

```
DECLARE IIC_Error BIT;
.....
IIC_Error=IIC_Test_Device(8Ch);
IF (IIC_Error) THEN
  "Device not acknowledging on slave address"
ELSE
  "Device acknowledges on slave address"
```

I²C routines for 8XC528

AN438

3.2.4 IIC_Write

Declaration

IIC_Write:

```
PROCEDURE (Slave_Address, Count, Source_Ptr)
  [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr) BYTE;
END;
```

Description

IIC_Write is the most basic procedure to write a message to a slave device.

I²C Protocol

```
L           =Count
D1[0..L-1]   BASED by Source_Ptr
```

S-SlW-A-D1[0]-A....A-D1[L-1]-A-P

Example

```
DECLARE Data_Buffer(4) BYTE;
.....
CALL IIC_Write(02Ch, LENGTH(Data_Buffer),Data_Buffer);
```

3.2.5 IIC_Write_Sub

Declaration

IIC_Write_Sub:

```
PROCEDURE (Slave_Address, Count, Source_Ptr,
  Sub_Address) [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address)
  BYTE;
END;
```

Description

IIC_Write_Sub writes a message preceded by a sub-address to a slave device.

I²C Protocol

```
L           =Count
Sub         =Sub_Address
D1[0..L-1]   BASED by Source_Ptr
```

S-SlW-A-Sub-A-D1[0]-A-D1[1]-A....A-D1[L-1]-A-P

Example

```
DECLARE Data_Buffer(8) BYTE;
.....
CALL IIC_Write_Sub (48h,LENGTH(Data_Buffer),Data_Buffer,2);
```

3.2.6 IIC_Write_Sub_SWInc

Declaration

IIC_Write_Sub_SWInc:

```
PROCEDURE (Slave_Address, Count, Source_Ptr,
  Sub_Address) [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address)
  BYTE;
END;
```

Description

Some I²C devices addressed with a sub-address do not automatically increment the sub-address after reception of each byte. IIC_Write_Sub_SWInc can be used for such devices the same way as IIC_Write_Sub is used. IIC_Write_Sub_SWInc splits up the message in smaller messages and increments the sub-address itself.

I²C Protocol

```
L           =Count
Sub         =Sub_Address
D1[0..L-1]   BASED by Source_Ptr
```

S-SlW-A- (Sub+0) – A-D1[0] – A-P

S-SlW-A- (Sub+1) – A-D1[1] – A-P

.....

S-SlW-A- (Sub+L-1)-A-D1[L-1]-A-P

Example

```
DECLARE Data_Buffer(6) BYTE;
.....
CALL IIC_Write_Sub_SWInc(80h,LENGTH
  (Data_Buffer),Data_Buffer,2);
```

3.2.7 IIC_Write_Memory

Declaration

IIC_Write_Memory:

```
PROCEDURE (Slave_Address, Count, Source_Ptr,
  Sub_Address) [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address)
  BYTE;
END;
```

Description

I²C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. IIC_Write_Memory can be used to write to such devices the same way IIC_Write_Sub is used. IIC_Write_Memory splits up the message in smaller messages and increments the sub-address itself. After transmission of each message a delay of 40 milliseconds ($f_{xtal} = 16 \text{ MHz}$) is inserted.

I²C Protocol

```
L           =Count
Sub         =Sub_Address
D1[0..L-1]   BASED by Source_Ptr
```

S-SlW-A- (Sub+0) – A-D1[0] – A-P

Delay 40ms

S-SlW-A- (Sub+1) – A-D1[1] – A-P

Delay 40ms

.....

S-SlW-A- (Sub+L-1)-A-D1[L-1]-A-P

Delay 40ms

Example

```
DECLARE Data_Buffer(10) BYTE;
.....
CALL IIC_Write_Memory(0A0h,LENGTH
  (Data_Buffer),Data_Buffer,0F0h);
```

I²C routines for 8XC528

AN438

3.2.8 IIC_Write_Sub_Write

Declaration

```
IIC_Write_Sub_Write:
  PROCEDURE (Slave_Address, Count1, Source_Ptr1,
    Sub_Address, Count2, Source_Ptr2)
    [BITIBYTE] EXTERNAL;
  DECLARE (Slave_Address, Count1, Source_Ptr1,
    Sub_Address, Count2, Source_Ptr2) BYTE;
  END;
```

Description

IIC_Write_Sub_Write writes 2 data blocks preceded by a sub-address in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol

```
L           =Count1
M           =Count2
Sub         =Sub_Address
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Source_Ptr2
```

```
S-SlVW-A-Sub-A-D1[0]-A-D1[1]-A-....
-A-D1[L-1]-A-D2[0]-A-D2[1]-A-....
-A-D2[M-1]-A-P
```

Example

```
PROCEDURE Write_CCT_Memory
  (Chapter, Row, Column, Data_Buf, Data_Count);
DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE;
```

```
/*
  The extended address (CCT-Cursor) is formed by Chapter, Row
  and Column. These three bytes are written after the sub-address
  (=8) followed by the actual data that will be stored relative to the
  extended address.
*/
```

```
CALL IIC_Write_Sub_Write (22h, 3, .Chapter, 8, Data_Buf,
  Data_Count);
END Write_CCT_Memory;
```

3.2.9 IIC_Write_Sub_Read

Declaration

```
IIC_Write_Sub_Read:
  PROCEDURE (Slave_Address, Count1, Source_Ptr1,
    Sub_Address, Count2, Dest_Ptr2)
    [BITIBYTE] EXTERNAL;
  DECLARE (Slave_Address, Count1, Source_Ptr1,
    Sub_Address, Count2, Dest_Ptr2) BYTE;
  END;
```

Description

IIC_Write_Sub_Read writes a data block preceded by a sub-address, generates an I²C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT.

I²C Protocol

```
L           =Count1
M           =Count2
Sub         =Sub_Address
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Source_Ptr2
```

```
S-SlVW-A-Sub-A-D1[0]-A-D1[1]-A-....
-A-D1[L-1]-A-S-SlVR-A-D2[0]-A-D2[1]-A-....
-A-D2[M-1]-N-P
```

Example

```
PROCEDURE Read_CCT_Memory
  (Chapter, Row, Column, Data_Buf, Data_Count);
DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE;
```

```
/*
  The extended address (CCT-Cursor) is formed by Chapter, Row
  and Column. These three bytes are written after the sub-address
  (8). After that the actual data will be read relative to the extended
  address.
*/
```

```
CALL IIC_Write_Sub_Write (22h, 3, .Chapter, 8, Data_Buf,
  Data_Count);
END Read_CCT_Memory;
```

I²C routines for 8XC528

AN438

3.2.10 IIC_Write_Com_Write**Declaration**

```
IIC_Write_Com_Write:
  PROCEDURE (Slave_Address, Count1, Source_Ptr1, Count2,
    Source_Ptr2) [BITIBYTE] EXTERNAL;
  DECLARE (Slave_Address, Count1, Source_Ptr1, Count2,
    Source_Ptr2) BYTE;
  END;
```

Description

IIC_Write_Com_Write writes two data blocks from different data buffers in one message to a slave receiver. This procedure can be used for devices where the message consists of 2 different data blocks. Such devices are, for instance, LCD-drivers, where the first part of the message consists of addressing and control information, and the second part is the data string to be displayed.

I²C Protocol

```
L           =Count1
M           =Count2
D1[0..L-1]  BASED by Source_Ptr1
D2[0..M-1]  BASED by Source_Ptr2
```

```
S-SlW-A-D1[0]-A-D1[1]-A-....
-A-D1[L-1]-A-D2[0]-A-D2[1]-A-....
-A-D2[M-1]-A-P
```

Example

```
DECLARE Control_Buffer(2) BYTE;
DECLARE Data_Buffer(20) BYTE;
.....
CALL IIC_Write_Com_Write(74h, LENGTH(Control_Buffer),
  .Control_Buffer, LENGTH(Data_Buffer), .Data_Buffer);
```

3.2.11 IIC_Write_Rep_Write**Declaration**

```
IIC_Write_Rep_Write:
  PROCEDURE (Slave_Address1, Count1, Source_Ptr1,
    Slave_Address2, Count2, Source_Ptr2)
    [BITIBYTE] EXTERNAL;
  DECLARE (Slave_Address1, Count1, Source_Ptr1,
    Slave_Address2, Count2, Source_Ptr2) BYTE;
  END;
```

Description

Two data strings are sent to separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

```
L           =Count1
M           =Count2
SlwW1       =Slave_Address1
SlwW2       =Slave_Address2
D1[0..L-1]  BASED by Source_Ptr1
D2[0..M-1]  BASED by Source_Ptr2
```

```
S-SlW-A-D1[0]-A-D1[1]-....
-A-D1[L-1]-A-S-SlW-A-D2[0]-A-D2[1]-....
-A-D2[M-1]-A-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Write_Rep_Write (48h, LENGTH(Data_Buffer_1),
  .Data_Buffer_1, 50h, LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

I²C routines for 8XC528

AN438

3.2.12 IIC_Write_Rep_Read**Declaration**

IIC_Write_Rep_Read:

```

PROCEDURE (Slave_Address1, Count1, Source_Ptr1,
           Slave_Address2, Count2, Dest_Ptr2)
  [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address1, Count1, Source_Ptr1,
         Slave_Address2, Count2, Dest_Ptr2) BYTE;
END;
```

Description

A data string is sent and received to/from two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

```

L           =Count1
M           =Count2
SivW1      =Slave_Address1
SivW2      =Slave_Address2
D1[0..L-1]   BASED by Source_Ptr1
D2[0..M-1]   BASED by Dest_Ptr2
```

```

S-SivW-A-D1[0]-A-D1[1]-....
-A-D1[L-1]-A-S-SivR-A-D2[0]-A-D2[1]-....
-A-D2[M-1]-N-P
```

Example

```

DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Write_Rep_Read (48h, LENGTH(Data_Buffer_1),
                        .Data_Buffer_1, 57h, LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

3.2.13 IIC_Read**Declaration**

IIC_Read:

```

PROCEDURE (Slave_Address, Count, Dest_Ptr)
  [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Dest_Ptr) BYTE;
END;
```

Description

IIC_Read is the most basic procedure to read a message from a slave device.

I²C Protocol

```

M           =Count
D2[0..M-1]   BASED by Dest_Ptr
```

```
S-SivR-A-D2[0]-A-D2[1]-A.....A-D2[M-1]-N-P
```

Example

```

DECLARE Data_Buffer(4) BYTE;
.....
CALL IIC_Read (0B5, LENGTH(Data_Buffer), .Data_Buffer);
```

3.2.14 IIC_Read_Status**Declaration**

IIC_Read_Status:

```

PROCEDURE (Slave_Address, Dest_Ptr)
  [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Dest_Ptr) BYTE;
END;
```

Description

Several I²C devices can send a one byte status-word via the bus. IIC_Read_Status can be used for this purpose. IIC_Read_Status works the same way as IIC_Read but the user does not have to pass a count parameter.

I²C Protocol

```
Status          BASED by Dest_Ptr
```

```
S-SivR-A-Status-N-P
```

Example

```

DECLARE Status_Byte BYTE;
.....
CALL IIC_Read_Status (84h, .Status_Byte);
```

3.2.15 IIC_Read_Sub**Declaration**

IIC_Read_Sub:

```

PROCEDURE (Slave_Address, Count, Dest_Ptr, Sub_Address)
  [BITIBYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Dest_Ptr, Sub_Address)
  BYTE;
END;
```

Description

IIC_Read_Sub reads a message from a slave device, preceded by a write of the sub-address. Between writing the sub-address and reading the message, an I²C restart condition is generated without releasing the bus. This prevents other masters from accessing the slave device in between and overwriting the sub-address.

I²C Protocol

```

M           =Count
Sub         =Sub_Address
D2[0..M-1]   BASED by Dest_Ptr
```

```
S-SivW-A-Sub-A-S-SivR-D2[0]-A-D2[1]-A.....A-D2[M-1]-N-P
```

Example

```

DECLARE Data_Buffer(5) BYTE;
.....
CALL IIC_Read_Sub (0A3h, LENGTH(Data_Buffer), .Data_Buffer, 2);
```

I²C routines for 8XC528

AN438

3.2.16 IIC_Read_Rep_Read

Declaration

```
IIC_Read_Rep_Read:
  PROCEDURE (Slave_Address1, Count1, Dest_Ptr1,
    Slave_Address2, Count2, Dest_Ptr2)
    [BIT|BYTE] EXTERNAL;
  DECLARE (Slave_Address1, Count1, Dest_Ptr1,
    Slave_Address2, Count2, Dest_Ptr2) BYTE;
  END;
```

Description

Two data strings are read from separate slave device, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

```
L           =Count1
M           =Count2
SlvW1      =Slave_Address1
SlvW2      =Slave_Address2
D1[0..L-1] BASED by Dest_Ptr1
D2[0..M-1] BASED by Dest_Ptr2
```

```
S-Slvr-A-D1[0]-A-D1[1]-.....
-A-D1[L-1]-N-S-Slvr-A-D2[0]-A-D2[1]-.....
-A-D2[M-1]-N-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Read_Rep_Read (49h, LENGTH(Data_Buffer_1),
  .Data_Buffer_1, 51h,
  LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

3.2.17 IIC_Read_Rep_Write

Declaration

```
IIC_Read_Rep_Write:
  PROCEDURE (Slave_Address1, Count1, Dest_Ptr1,
    Slave_Address2, Count2, Source_Ptr2)
    [BIT|BYTE] EXTERNAL;
  DECLARE (Slave_Address1, Count1, Dest_Ptr1,
    Slave_Address2, Count2, Source_Ptr2) BYTE;
  END;
```

Description

A data string is received and sent from/to two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

```
L           =Count1
M           =Count2
SlvW1      =Slave_Address1
SlvW2      =Slave_Address2
D1[0..L-1] BASED by Dest_Ptr1
D2[0..M-1] BASED by Source_Ptr2
```

```
S-Slvr-A-D1[0]-A-D1[1]-.....
-A-D1[L-1]-N-S-Slvr-A-D2[0]-A-D2[1]-.....
-A-D2[M-1]-A-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Read_Rep_Write(49h, LENGTH(Data_Buffer_1),
  .Data_Buffer_1, 58h,
  LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

I²C routines for 8XC528

AN438

3.2.18 Slave Mode Routines

There are two ways for the I²C interface to enter the slave-mode:

- After an I²C interrupt the software must enter the slave-receiver mode to receive the slave address. This address will then be compared with its own address. If there is a match either slave-transmitter or slave-receiver mode will be entered. If no match occurs, the interrupted program will be continued.
- During transmission of a slave-address in master-mode, arbitration is lost to another master. The interface must then switch to slave-receiver mode to check if this other master wants to address the 8xC528 interface.

The slave-mode protocol is very application dependent. In this note the basic slave-receive and slave-transmit routines are given and should be considered as examples. The user may for instance send NO_ACK after receiving a number of bytes to signal to the master-transmitter that a data buffer is full. A description of the code will be given later.

Slave parameters are given with the Init_IIC procedure. The passed parameters are the own-slave-address and a source/destination-pointer to a data buffer.

The slave-routine will be suspended at the following conditions:

- Interrupts with higher priority. Slave-routine will be resumed again after interrupt is handled.
- If a NO_ACKNOWLEDGE is received from a master-receiver.
- If a STOP condition is detected from a master transmitter.

Constraints for user software.

- The user must control the global enable (EA) bit.
- The user must control the priority level of the I²C interrupt. If the slave routine is interrupted by a higher priority interrupt, the SCL line will be stretched to postpone bus transfer until the higher interrupt is finished.

3.3 The Slave Routine: SLAVE.ASM

The listing of the slave routine can be seen on page 175. The routine is written in such a way that stretching of SCL is minimized. Application code can be inserted in this routine and this will increase stretching time.

The routine has 2 entry points.

Entry via MST_ENTRY happens when an arbitration error has occurred when transmitting a slave address in master mode.

Auto-clock generation will be disabled and SCL stretching enabled. The byte will be continued to be received and can later be compared with the own slave address.

The second entry point is via an interrupt when a START condition is detected. At _PIPOA the context of the interrupted program is stored. Next Auto-clock generation is disabled and SCL stretching enabled. Reception of the slave address can now begin by calling RCV_SL_BY. When the received slave-address is compared with the own-slave-address the R/W-bit is ignored. If there is no match between the 2 addresses, a negative ACK bit is sent and the slave routine is left via EXIT. If there was a match the R/W bit is checked to enter the slave-receiver or slave-transmitter mode.

The slave-transmitter mode starts at NXT_TRX. After getting the byte from the data buffer via BUF_POINT and initializing the bit counter BIT_CNT the transmission loop is entered. A bit is written via access to S1BIT because this will automatically reset the CLH and WBF status flags, and also SCL stretching. Now WBF must be tested until the transmission is successful. When WBF becomes true, SCL will be stretched again. When 8 bits are sent, the SDA line is released and RBF is tested until the ACK bit is received. The ACK bit is read by reading SDI instead of S1BIT to maintain SCL stretching. If ACK was false, no more bytes have to be sent and the routine is left. If another byte has to be transmitted, BUF_POINT is updated and transmission will continue.

The slave-receiver mode starts at RCV_SLAVE. A byte is received by calling RCV_SL_BY. This routine will clear the CY-flag when a STOP condition has been received. This means that the master will send no more bytes to this slave and the slave routine will be left. When no STOP condition was detected, the received byte will be stored @BUF_POINT and an ACK bit will be sent. After this, a new byte can be received.

When calling RCV_SL_BY the bit counter BIT_CNT will be initialized and the SCL stretching stopped by a dummy access to S1BIT. In the receive loop both BB and RBF will be checked. When BB is cleared, a STOP condition is detected and the routine will be left with CY=0.

The first 7 bits are received via S1BIT because this will release stretching. The 8th bit is accessed via SDI because stretching must be maintained.

If the slave routine is left via EXIT, the STR bit is cleared (to disable stretching on SCL edges when the 8xC528 is not addressed as slave) and a dummy access to S1BIT is done to finish current SCL stretching. If the slave routine was entered via an interrupt the previous context is restored.

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 1

```

LOC   OBJ           LINE  SOURCE
                                           1  $TITLE(Slave interrupt routine)
                                           2  $DEBUG
                                           3  $NOLIST
                                           6  ;
                                           7  ;This routine handles I2C interrupts.
                                           8  ;8xC528 I2C interface enters in slave mode.
                                           9  ;After testing R/W bit, 8xC528 will go in slave-transmit or
010   ;slave-receive mode.
                                           11 ;Source or destination buffer for data uses pointer SLAVE_SUB_ADDRESS
                                           12 ;Slave routine will use register bank 01
                                           13 ;
                                           14 ;*****
                                           15 ;Interrupt entry point
                                           16
----- 17             CSEG AT 53H
                                           18
0053: 020000  R 19             LJMP __PIP0A    ;Vector to interrupt handler
                                           20 ;
                                           21 ;*****
                                           22
                                           23             I2C_DRIVER SEGMENT CODE INBLOCK
----- 24             RSEG I2C_DRIVER
                                           25
                                           26             PUBLIC MST_ENTRY
                                           27             EXTRN  DATA(SLAVE_SUB_ADDRESS)
                                           28             EXTRN  BIT(ARB_LOST)
                                           29
REG END 30             BUF_POINT SET R0
REG END 31             OWN_SLAVE SET R7
REG END 32             BIT_CNT  SET R2
                                           33
                                           34 ;*****
                                           35
0000: C0E0   R 36  __PIP0A:PUSH ACC    ;Push CPU status on stack
0002: C0D0   37             PUSH PSW
0004: 75D008 38             MOV PSW,#08H    ;Select registerbank 01
                                           39
                                           40 ;*****
                                           41 ;Check slave address
                                           42 ;*****
                                           43
0007: 43D842 44             ORL S1SCS,#01000010B ;Disable SCL generation and enable SCL
                                           ;stretching stretching
000A: 1142   R 45             ACALL RCV_SL_BY ;Receive slave address, on exit SCL is
                                           ; stretched
000C: A2E0   46  PROC:  MOV C,ACC.0    ;Store R/W bit in F0
000E; 92D5   47             MOV F0,C
0010: 6F     48             XRL A,OWN_SLAVE ;Compare received slave address

```

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 2

```

LOC   OBJ           LINE  SOURCE
-----
0011: C2E0          49      CLR ACC.0      ;Ignore R/W bit
0013: 7050          50      JNZ NO_MATCH   ;Leave slave-routine if there is no match
0015: C3            51      CLR C          ;Send ACK
0016: 115C          R 52      ACALL SEND_ACK
0018: A800          R 53      MOV BUF_POINT,SLAVE_SUB_ADDRESS ;Get buffer pointer
001A: A2D5          54      MOV C,F0       ;Restore R/W bit
001C: 5019          55      JNC RCV_SLAVE ;Test R/W bit
                    56
                    57
                    58      ;*****
                    59      ;Slave transmitter mode
                    60      ;*****
                    61
                    62
001E: E6           63      NXT_TRX:MOV A,@BUF_POINT;Get byte to send
001F: 7A08          64      MOV BIT_CNT,#08 ;Init bit counter
                    65
0021:              66      NXT_TRX_BIT:
0021: F5D9          67      MOV S1BIT,A    ;Trx bit and stretch after transmission
0023: 23            68      RL A          ;Prepare next bit to send
0024: 30DAFD        69      JNB WBF,$      ;Test if bit is sent
0027: DAF8          70      DJNZ BIT_CNT,NXT_TRX_BIT ;Test if all bits are sent
                    71
0029: D2DF          72      SETB SDO      ;Release SDA line for NO_ACK/ACK reception
002B: E5D9          73      MOV A,S1BIT   ;Stop stretching
002D: 30DBFD        74      JNB RBF,$      ;Test if ACK bit is received
0030: A2DF          75      MOV C,SDI     ;Read bit, SCL remains stretched
0032: 4040          76      JC EXIT       ;NO_ACK received. Exit slave routine
0034: 08            77      INC BUF_POINT ;ACK received. Update pointer for next byte to
                    ;send
0035: 80E7          78      SJMP NXT_TRX
                    79
                    80      ;*****
                    81      ;Slave receiver mode
                    82      ;*****
                    83
0037:              84      RCV_SLAVE:   ;Entry in slave-receiver mode
0037: 1142          R 85      ACALL RCV_SL_BY ;Receive byte
0039: 5039          86      JNC EXIT     ;If STOP is detected, then exit
003B: F6            87      MOV @BUF_POINT,A ;Store received byte
003C: C3            88      CLR C        ;Send ACK
003D: 115C          R 89      CALL SEND_ACK
003F: 08            90      INC BUF_POINT ;Update pointer
0040: 80F5          91      SJMP RCV_SLAVE ;Receive next byte
                    92
                    93

```

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 3

```

LOC   OBJ           LINE  SOURCE
                                           94  ;*****
                                           95  ;Receive byte routine
                                           96  ;On exit, received byte in accu
                                           97  ;On exit CY=0 if STOP is detected
                                           98  ;*****
0042:                                           99  RCV_SL_BY:
0042: 7A08           100      MOV BIT_CNT,#08
0044: E5D9           101      MOV A,S1BIT      ;Disable stretching from START or previous ACK
0046: E4             102      CLR A
0047:                                           103  RCV_BIT:
0047: 30DC10          104      JNB BB,STOP_RCV ;Test if STOP-condition is received
004A: 30DBFA          105      JNB RBF,RCV_BIT ;Wait till received bit is valid
004D: BA0105          106      CJNE BIT_CNT,#01,ASSEM_BIT ;Check if last bit is to be received
                                           107
0050: A2DF           108      MOV C,SDI        ;Get last bit without stopping stretching
0052: 33             109      RLC A
0053: D3             110      SETB C          ;No STOP detected
0054: 22             111      RET
                                           112
0055:           113  ASSEM_BIT:
0055: 45D9           114      ORL A,S1BIT     ;Receive bit; release RBF,CLH and SCL stretching
0057: 23             115      RL A
0058: DAED           116      DJNZ BIT_CNT,RCV_BIT
                                           117
005A:           118  STOP_RCV:
005A: C3             119      CLR C          ;STOP detected
005B: 22             120      RET
                                           121
                                           122  ;*****
                                           123  ;Send ACK/NO_ACK. Value of ACK in Carry
                                           124  ;*****
005C:           125  SEND_ACK:
005C: 13             126      RRC A
005D: F5D9           127      MOV S1BIT,A     ;Carry to SDA line
005F: 30DAFD          128      JNB WBF,$      ;Test if ACK/NO_ACK is sent
0062: D3DF           129      SETB SDO       ;Release SDA line
0064: 22             130      RET
                                           131
                                           132  ;*****
                                           133  ;No match between received slave-address and own-slave-address
                                           134  ;*****
0065:           135  NO_MATCH:
0065: D3             136      SETB C          ;Send NO_ACK
0066: 115C           R 137      ACALL SEND_ACK
0068: 800A           138      SJMP EXIT

```

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 4

```

LOC   OBJ           LINE  SOURCE
                                139  ;*****
                                141  ;Entry point when an arbitration-lost condition is detected in
                                ;master-mode.
                                142  ;*****
006A: 143  MST_ENTRY:
006A: 23            144      RL A           ;Restore slave address  sofar
006B: C2E0         145      CLR ACC.0
006D: 43D842      146      ORL S1SCS,#01000010B ;Disable SCL generation and enable SCL
                                ;stretching
0070: 1147        R    147      ACALL RCV_BIT   ;Proceed with receiving rest of slave address
0072: 8098         148      SJMP PROC
                                149
                                150
                                151  ;*****
                                152  ;Exit from interrupt routine
                                153  ;*****
                                154
0074: C2D9         155  EXIT:  CLR STR           ;Disable stretching on next falling SCL edges
0076: E5D9         156      MOV A,S1BIT    ;Stop current SCL stretching
0078: 30001       R    157      JNB ARB_LOST,EX_SL
007B: 22           158      RET           ;Exit when entered from master mode
007C: D0D0         159  EX_SL: POP PSW        ;Restore old CPU status
007E: D0E0         160      POP ACC
0080: 32           161      RETI
                                162
0081:             163      END

```

I²C routines for 8XC528

AN438

4.0 EXAMPLES

4.1 Introduction

Some examples are given how to use the I²C routines in an application program. Examples are given for assembly, PL/M and C program.

The program displays time from the PCF8583P clock/calendar/RAM on an LCD display driven by the PCF8577.

The example can be executed on the OM4151 I²C evaluation board.

4.2 Using the Routines with Assembly Sources

The listing of the example program is shown on page 180. The most important aspect when using the I²C routines is preparing the input parameters before the sub-routine call. When, for example, the IIC_Write routine must be called, the parameters must be called in the following order:

```
MOV _IIC_READ_BYTE,#SLAVE_ADR
MOV _IIC_READ_BYTE+1,#COUNT_1
MOV _IIC_READ_BYTE+2,#SOURCE_PTR_1
CALL _IIC_READ
```

Note that the order of defining the parameters is the same as in a PL/M-call. An easier way to call the routines is making a macro that includes the initializing of the parameters. The example program makes use of macros.

IIC_Read is then called in the following way:

```
%IIC_Read(Slave_Adr,Count_1,Source_Ptr_1);
```

Note that in the listing the contents of the macro are shown, instead of the call.

The macro must be written as follows:

```
%* DEFINE
(IIC_Read(SLAVE_ADR,COUNT_1,SOURCE_PTR_1))
(MOV _IIC_READ_BYTE,#%SLAVE_ADR
MOV _IIC_READ_BYTE+1,#%COUNT_1
MOV _IIC_READ_BYTE+2,#%SOURCE_PTR_1
LCALL _IIC_READ)
```

Macros for the I²C CALLs are found in I2C.MAC. This file should be included in all modules making use of the macros. One of the modules should also include the variable definitions needed by the I²C routines. These are found in file VAR_DEF.ASM. If the program consists of more than 1 module, then these modules should also include EXT_VAR.ASM. This file contains the EXTRN- definitions of the I²C routines.

When an I²C routine is called, the accumulator contains status information and the CY-bit is set if an error has occurred. The contents of the accumulator are the same as the returned byte when using PL/M.

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Assembly example program

PAGE 1

```

LOC   OBJ           LINE  SOURCE
                                           1  $TITLE(Assembly example program)
                                           2  $DEBUG
                                           3
                                           4  ;Hours and minutes will be displayed on LCD display
                                           5  ;Dot between hours and minutes will blink
                                           6
                                           7  $                                           ;Include I2C var. definitions
                                           8  # 1 "C:\USER\VAR_DEF.ASM"
74    # 8 "DEMO_ASM.ASM"
                                           8  $                                           ;Include I2C macro's
                                           9  # 1 "C:\USER\I2C.MAC"
35    # 9 "DEMO_ASM.ASM"
00A2  10  CLOCK_ADR   EQU 0A2h           ;Address of PCF8583
0001  11  CL_SUB_ADR EQU 01h           ;Sub address for reading time
0074  12  LCD_ADR    EQU 74h           ;Address of PCF8577
                                           13
                                           14  RAMVAR  SEGMENT DATA           ;Segment for variables
                                           15  USER   SEGMENT DATA           ;Segment for application
                                           ;program
                                           16
----- 17          RSEC RAMVAR
0000:   R 18  STACK:      DS 10           ;Stack area
000A:   19  TIME_BUFFER:DS 4           ;Buffer for I2C strings
000E:   20  LCD_BUFFER: DS 5
                                           21
----- 22          CSEG AT 00
0000: 020000 R 23          LJMPL APL_START
                                           24
                                           25
----- 26          RSEG USER
                                           27
0000:   R 28  APL_START:
0000: 900073 R 29          MOV DPTR,#LCD_TAB           ;Pointer to segment table
0003: 7581FF R 30          MOV SP,#STACK-1           ;Initialize stack
0006: 750E00 R 31          MOV LCD_BUFFER,#00           ;Control word for LCD driver
                                           32
0009: 750022 R 33          MOV _Init_IIC_Byte ,#22h
000C: 75010A R 34          MOV _Init_IIC_Byte+1,#TIME_BUFFER
000F: 120000 R 35          LCALL _Init_IIC
                                           36          ;Initialize I2C interface
0012: E4          37          CLR A                                           ;Prepare buffer for clock int.
0013: F50A   R 38          MOV TIME_BUFFER,A
0015: F50B   R 39          MOV TIME_BUFFER+1,A
                                           40
0017: 7500A2 R 41          MOV _IIC_Write_Byte ,#CLOCK_ADR
001A: 750102 R 42          MOV _IIC_Write_Byte+1,#2
001D: 75020A R 43          MOV _IIC_Write_Byte+2,#TIME_BUFFER
0020: 120000 R 44          LCALL _IIC_Write

```

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Assembly example program

PAGE 2

```

LOC   OBJ           LINE   SOURCE
                                45       ;Initialize clock
                                46
0023:                                47 REPEAT:
0023: 7500A2   R   48       MOV _IIC_Read_Sub_Byte ,#CLOCK_ADR
0026: 750104   R   49       MOV _IIC_Read_Sub_Byte+1,#4
0029: 75020A   R   50       MOV _IIC_Read_Sub_Byte+2,#TIME_BUFFER
002C: 750301   R   51       MOV _IIC_Read_Sub_Byte+3,#CL_SUB_ADR
002F: 120000   R   52       LCALL _IIC_Read_Sub
                                53       ;Read time
                                54
                                55 ;Time has been read. Order: hundreds of sec's, sec's, min's and hr's
0032: E50D     R   56       MOV A,TIME_BUFFER+3           ;Mask of hour counter
0034: 543F     57       ANL A,#3Fh
0036: F50D     R   58       MOV TIME_BUFFER+3,A
                                59
0038: 120054   R   60       CALL CONVERT                 ;Convert time data to LCD
                                ;segment data
                                61
                                62 ;Check if dot has to be switched on
003B: 431101   R   63       ORL LCD_BUFFER+3,#01h
                                64 ;If lsb of seconds is '0', then switch on dp
003E: E50B     R   65       MOV A,TIME_BUFFER+1           ;Get seconds
0040: 13        66       RRC A
0041: 4003     67       JC PROCEED
0043: 430F01   R   68       ORL LCD_BUFFER+1,#01         ;Switch on dp
                                69
                                70 ;Display new time
0046:                                71 PROCEED:
0046: 750074   R   72       MOV _IIC_Write_Byte ,#LCD_ADR
0049: 750105   R   73       MOV _IIC_Write_Byte+1,#5
004C: 75020E   R   74       MOV _IIC_Write_Byte+2,#LCD_BUFFER
004F: 120000   R   75       LCALL _IIC_Write
                                76
0052: 80CF     77       SJMP REPEAT                 ;Read new time
                                78
                                79
                                80 ;CONVERT converts BCD data of time to segment data
0054: 780F     R   81       CONVERT:MOV R0,#LCD_BUFFER+1 ;R0 is pointer
0056: E50D     R   82       MOV A,TIME_BUFFER+3           ;Get hours
0058: C4        83       SWAP A                       ;Swap nibbles
0059: 12006D   R   84       CALL LCD_DATA                 ;Convert 10's of hours
005C: E50D     R   85       MOV A,TIME_BUFFER+3
005E: 12006D   R   86       CALL LCD_DATA                 ;Convert hours
0061: E50C     R   87       MOV A,TIME_BUFFER+2           ;Get minutes
0063: C4        88       SWAP A
0064: 12006D   R   89       CALL LCD_DATA                 ;Convert 10's of minutes
0067: E50C     R   90       MOV A,TIME_BUFFER+2
0069: 12006D   R   91       CALL LCD_DATA                 ;Convert minutes

```

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Assembly example program

PAGE 3

```

LOC   OBJ           LINE   SOURCE

006C: 22           92           RET
                               93
                               94   ;LCD_DATA gets data from segment table and stores it in LCD_BUFFER
006D:           95   LCD_DATA:
006D: 540F         96           ANL A,#0FH           ;Mask off LS-nibble
006F: 93          97           MOVC A,@A+DPTR      ;Get segment data
0070: F6          98           MOV @R0,A           ;Save segment data
0071: 08          99           INC R0
0072: 22         100          RET
                               101   ;
                               102   ;Conversion table for LCD
0073:           103   LCD_TAB:
0073: FC60DA      104          DB 0FCH,60H,0DAH      ;'0','1','2'
0076: F266B6      105          DB 0F2H,66H,0B6H      ;'3','4','5'
0079: 3EE0FE      106          DB 3EH,0E0H,0FEH      ;'6','7','8'
007C: E6          107          DB 0E6H               ;'9'
                               108   ;
007D:           109          END

```

4.3 Using the Routines with PL/M-51 Sources

The following listing shows the listing of the clock program in PL/M-51. The procedures are untyped. The routines are used the same way as in the examples of chapter 3.2.

```

$OPTIMIZE(4)
$DEBUG
$CODE

/* Hours and minutes will be displayed on LCD display
   Dots between hours and minutes will blink */

Demo_plm: Do;

/* External declarations */

Init_IIC: Procedure(Own_Adr,Slave_Ptr) External;
          Declare (Own_Adr,Slave_Ptr) Byte Main;
End Init_IIC;

IIC_Write: Procedure(Sl_Adr,Nr_Bytes,Source_Ptr) External;
          Declare (Sl_Adr,Nr_Bytes,Source_Ptr) Byte Main;
End IIC_Write;

IIC_Read_Sub: Procedure(Sl_Adr,Nr_Bytes,Dest_Ptr,Sub_Adr) External;
          Declare (Sl_Adr,Nr_Bytes,Dest_Ptr,Sub_Adr) Byte Main;
End IIC_Read_Sub;

```


I²C routines for 8XC528

AN438

```

Clock: Do;
    /* Variable and constant declarations */

    Declare LCD_TAB(*) Byte Constant (0FCh,60H,0DAH,0F2H,66H,
                                     0B6H,3EH,0E0H,0FEH,0E6H);

    Declare Time_Buffer(4) Byte Main;
    Declare LCD_Buffer(5) Byte Main;
    Declare Tab_Point Word Main;
    Declare (LCD_Point,Time_Point) Byte Main;
    Declare Segment Based LCD_Point Byte Main;
    Declare Time Based Time_Point Byte Main;
    Declare Tab_Value Based Tab_Point Byte Constant;

    Declare clock_Adr Literally '0A2h';
    Declare LCD_Adr Literally '74h';
    Declare Cl_Sub_Adr Literally '01h';

    Call Init_IIC(22h,.Time_Buffer);
    LCD_Buffer(0)=0; /* LCD control word */
    Time_Buffer(0)=0;
    Time_Buffer(1)=0;
    Call IIC_Write(Clock_Adr,2,.Time_Buffer); /* Initialize clock */

    Do While LCD_Buffer(0)=0; /* Program loop */
        Call IIC_Read_Sub(Clock_Adr,4,.Time_Buffer,Cl_Sub_Adr);
                                /* Get time */

        LCD_Point=.LCD_Buffer+1; /* Initialize pointers */
        Time_point=.Time_Buffer(3);
        Tab_Point=.LCD_Tab(0)+SHR(Time,4); /* 10-HR's */
        Segment=Tab_Value;

        LCD_Point=LCD_Point+1;
        Tab_Point=.LCD_Tab(0)+(Time AND 0FH); /* HR's */
        Segment=Tab_Value;
        Time_Point=Time_Point-1;
        LCD_Point=LCD_Point+1;
        Tab_Point=.LCD_Tab+SHR(Time,4); /* 10-MIN's */
        Segment=(Tab_Value OR 01H); /* dp */
        LCD_Point=LCD_Point+1;
        Tab_Point=.LCD_Tab+(Time AND 0FH); /* MIN's */
        Segment=Tab_Value;
        Time_Point=.Time_Buffer(1)+1; /*Check sec's for blinking */
        LCD_Point=.LCD_Buffer+1;
        If (Time AND 01H)>0 then Segment=(Segment OR 01H);
        Call IIC_Write(LCD_Adr,5,.LCD_Buffer); /* Display time */
    End;

End Clock;

End Demo_plm;

```

I²C routines for 8XC528

AN438

4.4 Using the Routines with C Sources

An example of a C program using the I²C routines follows. Function prototypes are found in header file "i2c.h". In this example the function prototypes are written in such a way that not value is returned by the function. If the STATUS byte is needed, the header file may be changed to return a byte. Note that the function calls are written in upper-case. This is due to the fact that the used version of the assembler/linker is case sensitive.

```
#include <C:\USER\i2c.h>

rom char          LCD_Tab[]={0xFC,0x60,0xDA,0xF2,0x66,0xB6,0x3E,
                             0xE0,0xFE,0xE6};

void main()

{

#define Clock_Adr      0xA2
#define LCD_Adr        0x74
#define Cl_Sub_Adr     0x01

rom char           * Tab_Ptr;
data char          Time_Buffer[4];
data char          * Time_Ptr;
data char          LCD_Buffer[5];
data char          * LCD_Ptr;

INIT_IIC(0x22,&Time_Buffer);
LCD_Buffer[0]=0; /* LCD control word */
Time_Buffer[0]=0;
Time_Buffer[1]=0;
IIC_WRITE(Clock_Adr,2,&Time_Buffer); /* Initialize clock */

while (1)          /* Program loop */
{
    IIC_READ_SUB(Clock_Adr,4,&Time_Buffer,Cl_Sub_Adr);
                                     /* Get time */
    LCD_Ptr = &LCD_Buffer[1]; /* Initialize pointers */
    Time_Ptr = &Time_Buffer[3];
    Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10-HR's */
    *(LCD_Ptr++) = *Tab_Ptr;
    Tab_Ptr = (LCD_Tab+(*Time_Ptr-- & 0x0F)); /* HR's */
    *(LCD_Ptr++) = *Tab_Ptr;
    Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10-MIN's */
    *(LCD_Ptr++) = (*Tab_Ptr | 0x01); /* dp */
    Tab_Ptr = (LCD_Tab+(*Time_Ptr & 0x0F)); /* MIN's */
    *LCD_Ptr = *Tab_Ptr;
    Time_Ptr = &Time_Buffer[1]; /* Check sec's for blinking */
    LCD_Ptr = &LCD_Buffer[1];
    if ((*Time_Ptr & 0x01)>0)
        *LCD_Ptr = (*LCD_Ptr | 0x01);
    IIC_WRITE(LCD_Adr,5,&LCD_Buffer); /* Display time */
}
}
```

I²C routines for 8XC528

AN438

5.0 CONTENTS OF DISK

A disk contains the following 3 directories:

1: \USER

This director contains the files that may be used in the user program.

I2C_DR.LIB	Library with I ² C routines.
I2C.H	Header file for C applications.
I2C.MAC	Macro's for the I ² C routine calls in assembly programs.
VAR_DEF.ASM	Include file with variable definitions for assembly programs.
EXT_VAR.ASM	Include file with external definitions for assembly programs.
LIB.BAT	Example batch file to create I2C_DR.LIB.
ASM.BAT	Example batch file to assemble source modules for library.

2: \EXAMPLE

This directory contains the source files of the examples described in chapter 4.0.

DEMO_ASM.*	Assembly example.
DEMO_PLM.*	PL/M example.
HEAD_51.SRC	Example of environment file for PL/M example.
DEMO_C.*	C example.
CSTART.ASM	Example of environment file for C example.

3: \SOURCE

This directory contains the source files of the modules in the library.

Using the P82B715 I²C extender on long cables

AN444

Author: Don Sherman, Sunnyvale

The P82B715 I²C Buffer was designed to extend the range of the local I²C bus out to 50 Meters. This application note describes the results of testing the buffer on several different types of cables to determine the maximum operating distances possible. The results are summarized in a table for easy reference.

The I²C bus was originally conceived as a convenient 2 wire communication method between Integrated Circuits located within a common chassis, such as inside a TV set or inside a VCR. The serial protocol contains an address, or identifying code, for each type of device and additional internal addresses, if needed within the addressed device. Each device has its own decoding circuitry to allow it to recognize its own unique address or identifying code. To communicate, a device watches the bus activity and jumps in when it sees a stop. Once a Master gets control of the bus, it sends the address of the particular device with which it wants to communicate. Communication will then transpire between the Master and the Slave device. The existence of many types of ICs which have built-in I²C interface capabilities makes system design almost as easy as drawing a block diagram. Real-time clocks, RAM, A/D converters, EEPROMs, Microcontrollers, Keyboard encoders, LCD display drivers, and many other I²C supported chips all communicate over two wires rather than needing 16 Address lines, 8 data lines and Address decoders along with handshake signals, which more conventional designs would require to be routed all over the Printed Circuit board.

Now, with the introduction of the I²C buffer chip, it is easy to branch out beyond the single chassis mode and use this convenient local area network to tie together whole systems without the need to convert from the "internal" I²C protocol to an external communication medium such as RS-232 and then RS-485. By using the new Philips I²C buffer, the external systems' components can be accessed as easily as the internal I²C connected components.

The P82B715 is an 8 pin IC which contains 2 identical amplifier sections to allow for the current amplification and buffering of both the SDA and the SCL signals on the I²C bus. Each section in the P82B715 contains a bipolar times 10 current amplifier which senses the direction of current flow through an internal 30 ohm series resistor in the I²C line. The P82B715 then boosts the current, while keeping the voltage gain at unity, and continues to maintain the voltage drop direction across the resistor. This

configuration results in different waveforms as the P82B715 starts to do its job. If the driving source has a strong current sink capability, then it will start to drive the buffered I²C line immediately through the 30 ohm resistor. A microsecond later the P82B715's amplified pull down current kicks in and pulls the line down even harder. If the driving IC is only capable of the I²C specified 3 milliamp pull down current, the buffered bus will fall a little and then just wait at that voltage level for the propagation delay of the amplifier to finally turn on and bring the buffered bus down to a logic low. Thus, there will always be some form of a step in the falling edge of the buffered output waveform, see Figure 1. A weak source will have a step (plateau) up near 4 volts and a strong source, such as the Philips Semiconductors 87C751 microcontroller, will have the step occur below 2 volts. The position of the step will be determined by the current sink capability of the I²C bus driver versus the value of the pull-up resistor which is used on the buffered I²C bus, $V_{step} = 5V - (I_{sink} \times R_{buf})$. For example: $V_{step} = 5V - (3mA \times .165k\text{ ohms}) = 5 - .495 = 4.5\text{Volts}$; another example: $V_{step} = 5V - (20mA \times .165k\text{ ohms}) = 5 - 3.3 = 1.7\text{Volts}$.

Running the I²C signals over long distances poses several problems. The I²C SDA and SCL lines are monitored by all of the ICs connected on the I²C bus. These ICs each have their own circuitry to decipher the information on the bus. In normal operation, a Start occurs when there is a high to low transition on the SDA line while SCL is high. Obviously, if any external noise is coupled into the SDA line, it could be mistakenly perceived as a Start. Because of this, some form of shielding will be preferred to protect the two I²C signals from external noise sources. During the transmission of data there are signals which are active on both SDA and SCL. If these normal signals are cross-coupled, then data can be corrupted. Thus, although the standard telephone twisted pair cable is the most commonly available built in cable, it is not recommended for long I²C runs. This cable maximizes crosstalk, due to the twisted pair configuration and, since there is no shielding, is very vulnerable to adjacent wire telephone signal coupling and to any stray external electromagnetic interference. This effect can be somewhat reduced by running a signal wire and a grounded wire as adjacent pairs.

Long distance cables present capacitive loading which must be overcome with the driver chips. The limiting factor is the amount of pull-up current which is available to charge the line capacitance. With the simple resistor

pull-up recommended by I²C standards, three milliamps is available for charging this line capacitance. The rise time of the signal will increase linearly with the increase in capacitive loading and the specified maximum capacitive loading is only 400 Pico Farads for guaranteed 100kHz communication rates. The P82B715 current buffer allows for 30 milliamps of pull-up current, with a resulting maximum capacitive loading of 4,000 Pico Farads (4 Nano Farads).

The I²C hardware inputs look at the I²C signals and act when those signals pass through the active linear region at about 1.2 to 1.4 volts, and are detected as digital levels. Thus, there is a delay between when an output transistor turns off and when the rising signal is detected as a logic one at the receiver. This time depends on the value of the pull-up resistor, the perceived capacitance at the transmitting end, the delay through the cable, and finally the delay through the receiver's amplifier to its output stage. The maximum allowable time is limited by the characteristic that the I²C master provides the clock signal which must travel down the cable and be received by the slave. This slave must act on the clock signal and produce data information which is sent back to the master with an additional set of delays. Upon reception the data must be put in its proper place before the master starts its next clock signal, or an error will occur.

Different types of cable were tested and the results are shown in Table 1. Keep in mind that the results are based on cable runs in a low electrical noise environment. If reliable operation is desired in a high electrical noise environment, shielded cable must be used. For "short" runs, flat cable with every other conductor grounded, seems to provide a good, low capacitance medium for I²C transmission, otherwise, the shielded audio cable seemed to provide the best price/performance. Note that for long runs, it is desirable to have a separate power supply at each end of the cable, and the shield or ground wire will provide a common reference between the two supplies. The voltage drop due to the resistance of the wire usually is the limiting factor for very long runs of cable where the power to the remote system must also come through the cable. Table 1 shows the results of testing with longer and longer cable lengths until failures were detected. The values in the table represent the maximum cable lengths which still provided error free code from a modified version of the Ping-pong program which is listed in Application Note AN430.

Using the P82B715 I²C extender on long cables

AN444

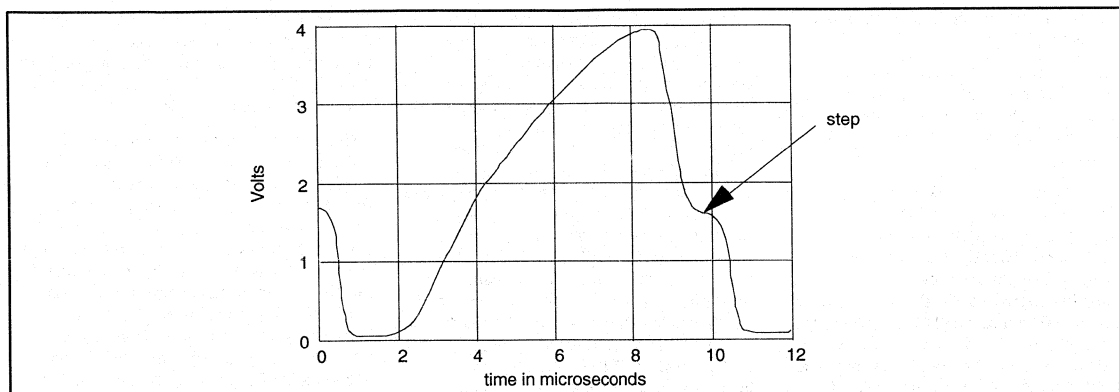


Figure 1. P82B715 Output Waveform on Long Cable

Table 1. Test Results with P82B715 Over Long Cables

CABLE TYPE	Ohms/m	pF/m	Total Length	Total Ohms	Total Cap.
Belden 8723 45 Ohm Audio 2 each 2—24AWG wire stranded Beldfoil Aluminum-polyester shielded with common drain wire SDA & ground on one pair; SCL & ground on other pair	.049	115	305M (1000')	11.5	48.2nF
Belden 8723 45 Ohm Audio using 1 shielded pair, SDA on Red, SCL on Black	.049	115	330M (1100')	12.7	53nF
RG-174/U 50 Ohm Video Cable SDA and grounded shield in one cable SCL and grounded shield in one cable	.318	101	150M (500')	47.7	15.2nF
"Telephone Cable" 22&24 AWG Solid Copper Twisted Pair, Level 3 LAN & Medium Speed Data SDA and ground in one twisted pair SCL and ground in one twisted pair	.0286	66	95M (310')	2.7	6.4nF
Flat "Ribbon" Cable, every other conductor grounded	.20	52	400M (1320')	80.5	21nF

In all of the tests, the power supply voltage was 4.5 volts. The ground for the remote test fixture was through the long cable. Since 4.5 volts is the recommended minimum voltage for both the 87C751 and the P82B715, it was not possible to operate the remote unit on power supplied through the long cable, since any ohmic drop would place the ICs out of their specified range. However, it is necessary to connect the grounds between the two units for the best noise immunity.

The P82B715 is designed to drive a 4 nF capacitive load at 100kHz. However, the actual total capacitances of the long cables which worked were substantially greater than this. The loading did effect the software driven hardware part of the 87C751. To achieve a true 100kHz data rate, it was necessary to shorten the '751 Timer values for the I²C drivers. This resulted in an asymmetrical waveform, but did achieve a 10 microsecond period (100kHz). This

asymmetry in duty cycle can be easily seen in the Figure 1 waveform.

The test with the Belden 8723 Audio Cable worked if one of the shielded pair was connected to a signal and the other was connected to ground or +5volts. When both wires were connected in parallel as signal wires, the capacitance to ground doubled and the test failed. Also note that the adjacent wire mutual inductive coupling of the SDA and SCL signals did not seem to cause any problems even out to 1000 feet. This indicated that possibly the Belden 9452 45 ohm beldfoil shielded audio cable with a single set of twisted pair wires would be a good candidate to also try.

Flat ribbon cable provided a good compromise between shielding and reasonable capacitance. It is possible to increase the shielding effect by using flat cable with an etched copper foil layer on the back side of the cable. Noise can be induced

into the cable by folding it back over itself for mutual induction effects, and also by operating a noise source close to the cable. A transformer type of soldering iron and fluorescent light transformers seemed to be good noise sources.

The P82B715 can drive multiple P82B715 remote units. The line should have some form of pull-up resistor at each driver. If only two drivers are used, as shown in Figure 2, the load should be split between the two drivers. For example, if the pull-up current is to be 30 milliamps and the voltage is 5 volts, the pull-up resistance should be: $5V / .030 \text{ amps} = 165 \text{ ohms}$. This should be implemented by placing a 330 ohm resistor at each end of the cable so that the parallel resistance is 165 ohms and each end of the line is terminated. Remembering that the current gain can be as low as 8 and that most runs will not be to the maximum possible distance, lower values of pull-up current can

Using the P82B715 I²C extender on long cables

AN444

be used with the appropriate modifications to the above equations.

For larger fan-out with fixed locations, the load resistance should also be evenly divided so that the parallel combination of all of the pull-up resistors will provide the desired D.C. pull-up current.

If some of the remote units will be pluggable, it will be necessary to divide the pull-up load to accommodate all of the possible combinations of possible fanout. Figure 3 shows an example of driving up to 30 remote, pluggable peripherals. On the 3 milliamp side of the P82B715 a complete I²C system may exist. In Figure 3, a local I²C network cluster could be joined to other local network clusters through the P82B715 buffered bus so that hundreds of I²C devices could potentially be interconnected.

The ease of connecting I²C clusters into a complete LAN opens the door for many new uses of components which have an I²C bus connection. Now an electronic instrument can have access to remote keyboards and remote sensors by using the I²C bus. The instrument's output can easily be shown on multiple remote displays all connected with the I²C bus. Multiple instruments can also pass data back and forth over the I²C bus. Thus, we see that the I²C bus can become an effective and inexpensive Local Area Network by using the P82B715 I²C bus extender.

THE TEST SETUP

These tests were run on two identical test boards which each use a Philips Semiconductors 87C751 microcontroller that drives the I²C buffer which has a 330 ohm pull-up resistor. The schematic is shown in Figure 4. The software is a modified version of the "Ping-Pong" program which is described in the Philips Semiconductors Application Note, AN430, "Using the 8XC751/752 in Multimaster applications". This program sends a number down the I²C line and, when received, the receiving unit becomes a master and increments the number and sends it back to the first unit where it is checked and then the process

repeats itself. The software has extensive error detection capability and monitors for corruption of data, false starts, over run of data, stuck lines and about anything else which might indicate a problem. If any errors did occur, a software counter was incremented. In this setup, the counter was stopped at Hex 07F to prevent wrap around and the contents of the counter are displayed on a bank of 8 LEDs. The MSB of the counter register was used as an indicator that the unit was working. The MSB LED flashes at about a 1 Hz rate when the unit is operating normally. When a cable length was reached which was too long, the MSB LED would stop flashing and the counter would rapidly fill up and stop with all 7 LEDs on (LED on indicates a logic "1" in this application).

THE TEST HARDWARE

A general purpose test rig was designed so that future needs of a general I²C platform could also be met. All of the port pins on the '751 were used. The inputs to the system were a toggle switch with a pull-up resistor connected to P0.2 (because this pin is Open Drain) and an octal DIP switch connected to port 1 (the internal pull ups of the port were used, so no external pull-up resistors were needed). The output is displayed through an octal buffer connected to port 3. A logical "1" on the pin will light up the LED. The I²C signals, SDA and SCL, are connected to the I²C buffer chip and the outputs of the buffer are pulled up by 330 ohm resistors. The parallel combination of the buffered transmitting end pull-up and the receiving end pull-up resistors is 330/2 ohms, which results in a pull-up load current of 30 milliamps. This current from the two pull-up resistors must be sunk by the single driving transistor of the acting sender. The effective loading seen by the '751 is the I²C buffer's load divided by 10. Thus, the '751's I²C outputs will sink 3 milliamps when driving the I²C buffer which is sinking 30 milliamps on the buffered bus.

The software monitor routine allows the user to monitor any internal '751 RAM location and display the contents on the LEDs. The monitor routine also allows the user to modify the contents of any RAM location including

SFR space. The Ping-Pong program needed the first 8 locations in RAM, so the stack pointer for this application was changed from the default location of 07H to location 09H. This starts the stack at 0AH.

To read the contents of RAM, set the DIP switches to the desired RAM address. The toggle switch is set to a "1". Pressing the Reset switch causes the microprocessor to reset and then enter the monitor program where the program then waits until the toggle switch is changed. Upon closing the toggle switch (a "1" to "0" transition) the program loads the DIP switch selection into R0 of bank 1 (RAM location 08H). The program then loads the contents of the RAM location pointed to by R0 (bank 1) and copies it into port 3, where it is displayed on the 8 LEDs. Thus, the Address is seen by looking at the DIP switches and the contents pointed to are displayed on the LEDs. Note that this indirect Address latch location (R0, bank 1) would have been the normal beginning of the stack, had it not been changed.

The contents of an internal RAM location can also be modified with this program. First, set the DIP switches to the desired Address and set the toggle switch to "0". Reset the processor and then set the toggle switch to "1". This transfers the address to R0 (bank 1). Next, load the desired new data, which is to be stored in RAM, into the DIP switches, and then set the toggle switch to "0". At this time the LEDs will now show the Address of RAM and the DIP switches show what was written into the selected RAM location. To verify that the data was actually written into the RAM, follow the read RAM sequence.

Although this may seem to be a bit cumbersome, it is a workable way to see what is happening inside of the '751. Remember that it is necessary to re-enter the monitor program, or at least to duplicate the read RAM of R0 (bank 1) and output to port 3, to see the latest version of the contents of the RAM location. Since this experiment only looked at the contents of one RAM location, the above method was easy to use and the display always showed the current status of the desired RAM location because it is updated often by the software.

Using the P82B715 I²C extender on long cables

AN444

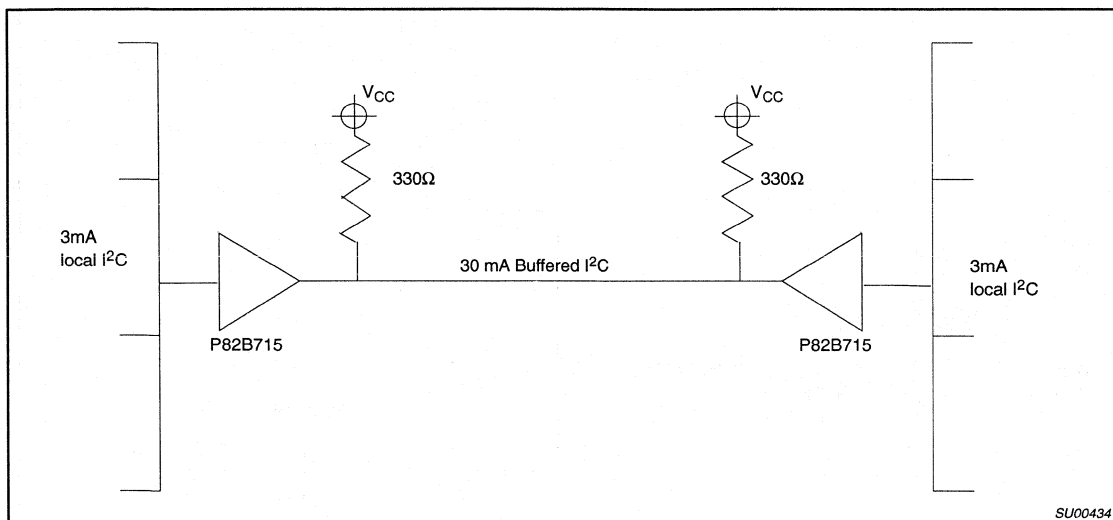


Figure 2. P82B715 Driving Long Line

Using the P82B715 I²C extender on long cables

AN444

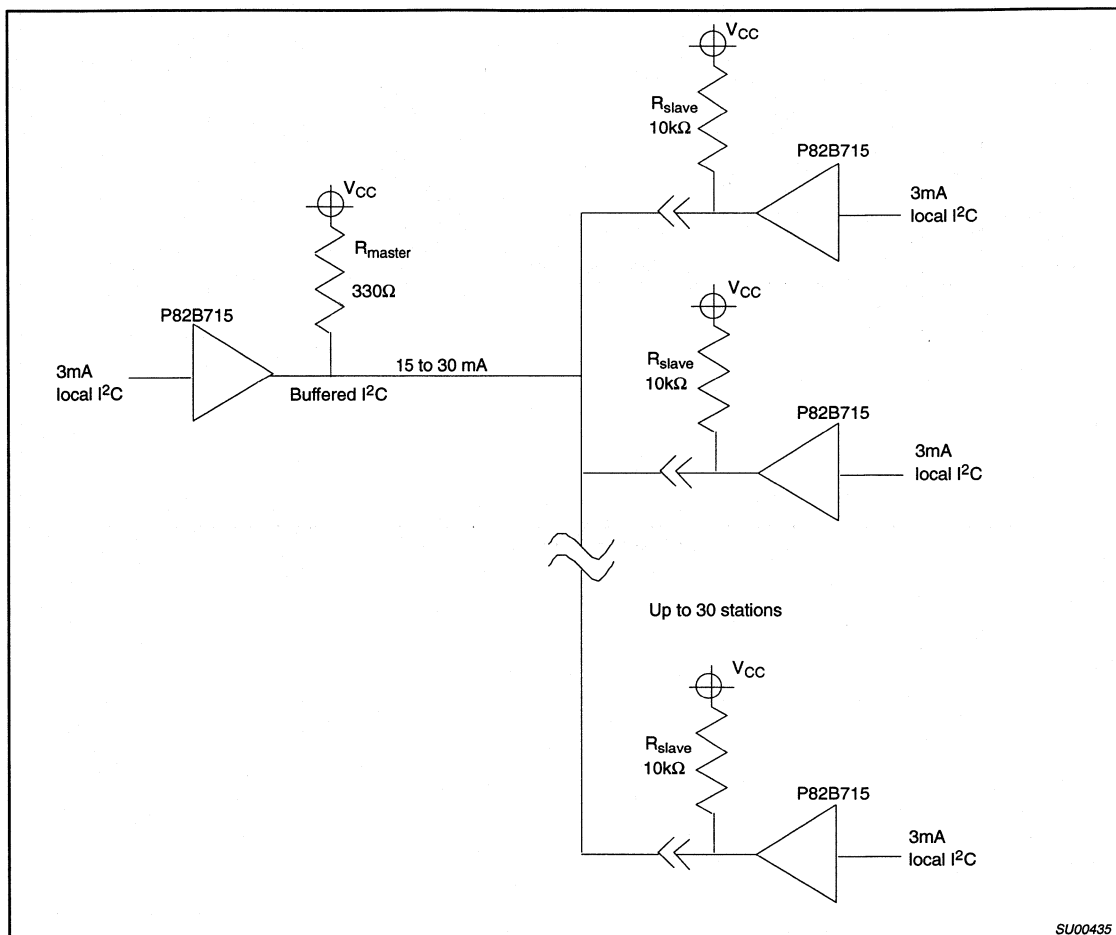


Figure 3. Large Fan-Out Configuration for P82B715

Note that V_{CC} is 5 volts for these values of load resistors. If a different voltage is desired, the calculations are as follows:

$$R_{\text{master}} = \frac{V_{\text{CC}}}{15\text{mA}} \quad \text{example: } R_{\text{master}} = \frac{5\text{V}}{15\text{mA}} = 0.33\text{k} = 330\Omega$$

The pluggable units would be calculated as follows:

$$\text{Parallel combination of } R_{\text{slave}} = R_{\text{master}}$$

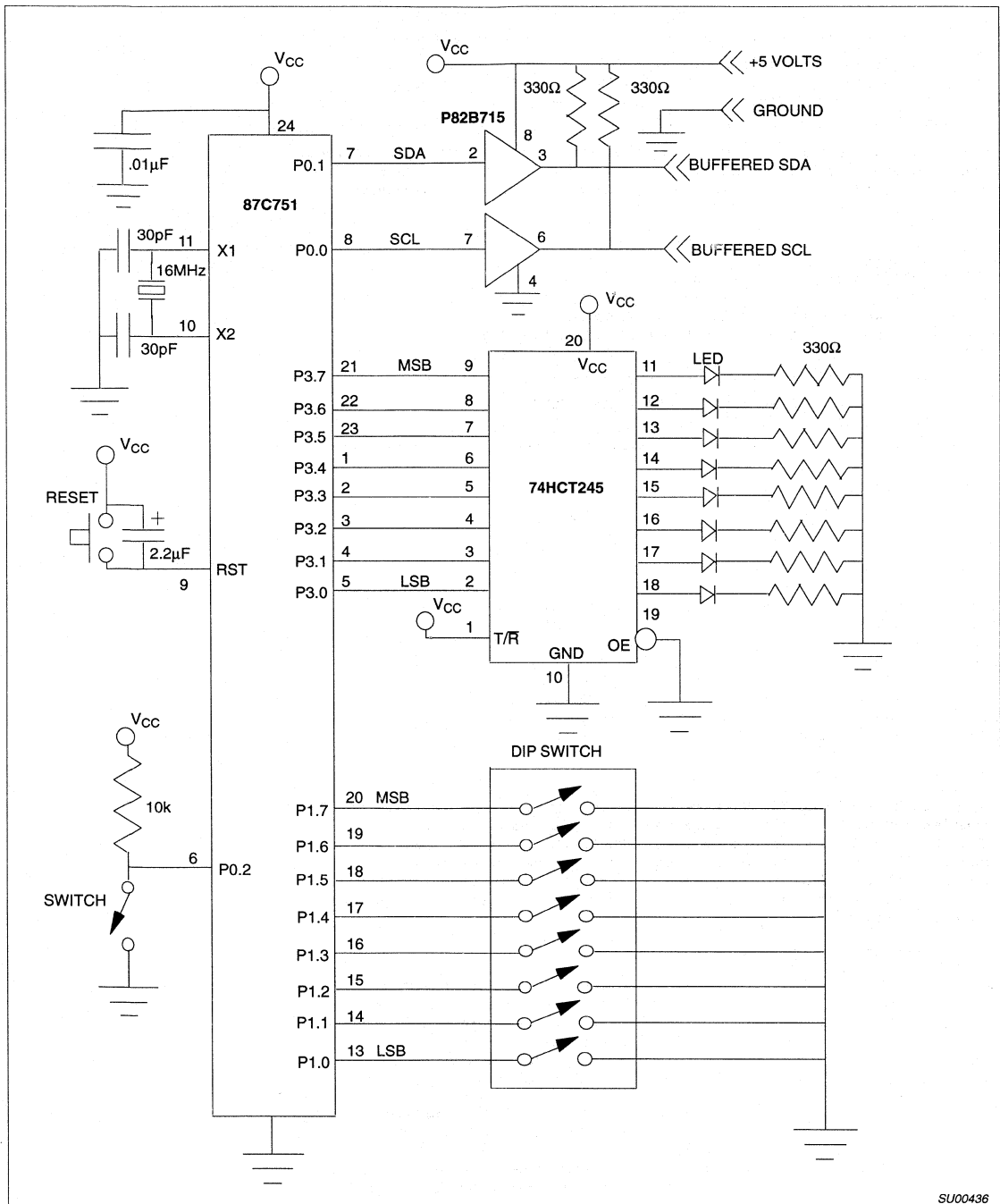
$$R_{\text{slave}} = R_{\text{master}} \times \text{Fan out}$$

$$\text{example: } R_{\text{slave}} = 330\Omega \times 30 = 9900\Omega = 10\text{k}$$

SU00435

Using the P82B715 I²C extender on long cables

AN444



SU00436

Figure 4. Schematic

Using the P82B715 I²C extender on long cables

AN444

```

;
;*****
;      Multimaster Code for 83C751/83C752
;      4/14/1992      MODIFIED BY DON SHERMAN 5-21-92
;      ;; is used to show where original code was modified
;*****
; This code was written to accompany an application note. The I2C routines
; are intended to be demonstrative and transportable into different
; application scenarios, and were NOT optimized for speed and/or memory
; utilization.
;
; Yoram Arbel

$TITLE(83C751 Multi Master I2C Routines)
$DATE(4/14/1992)
$MOD751      ;;NEED TO USE $MOD752 FOR 752 EMULATOR
;EI2        EQU      ES          NEED ENABLE FOR EMULATOR
$DEBUG

;*****
;      8XC751 MULTIMASTER I2C COMMUNICATIONS ROUTINES
;      Symbols and RAM definitions
;*****

; Symbols (masks) for I2CFG bits.

BTIR      EQU      10h          ; TIRUN bit.
BMRQ      EQU      40h          ; MASTRQ bit.

; Symbols (masks) for I2CON bits.

BCXA      EQU      80h          ; CXA bit.
BIDLE     EQU      40h          ; IDLE bit.
BCDR      EQU      20h          ; CDR bit.
BCARL     EQU      10h          ; CARL bit.
BCSTR     EQU      08h          ; CSTR bit.
BCSTP     EQU      04h          ; CSTP bit.
BXSTR     EQU      02h          ; XSTR bit.
BXSTP     EQU      01h          ; XSTP bit.

; Note:
;
; Specific bits of the I2CON register are set by writing into this register a
; combination of the masks defined above using the MOV command.
; The SETB command should not be used with I2CON, as it is implemented by
; reading the contents of the register, setting the appropriate bit and
; writing it back into the register. As the functionality of the Read and
; Write portions of the I2CON register is different, using SETB may cause
; unwanted results.

; Message transaction status indications in MSGSTAT:

SGO        EQU      10h          ; Started Slave message processing.
SRCVD      EQU      11h          ; as a slave, received a new message
SRLNG      EQU      12h          ; received as slave a message which is too
; long for the buffer
STXED      EQU      13h          ; as slave, completed message transmission.
SRERR      EQU      14h          ; bus error detected when operating as a slave.

MGO        EQU      20h          ; Started Master message processing.
MRCVD      EQU      21h          ; As Master, received complete message from
; slave.
MTXED      EQU      22h          ; As Master, completed successful message
; transmission (slave acknowledged all data
; bytes).
MTXNAK     EQU      23h          ; As Master, truncated message since slave did
; not acknowledge a data byte.

```

Using the P82B715 I²C extender on long cables

AN444

```

MTXNOSLV EQU 24h ; AS Master, did not receive an acknowledgement
; for the specified slave address.

TIMOUT EQU 30h ; TIMERI Timed out.
NOTSTR EQU 32h ; Master did not recognize Start.

; RAM locations used by I2C interrupt service routines.

MASCMD DATA 20h
SUBADD BIT MASCMD.0
RPSTRT BIT MASCMD.1
SETMRQ BIT MASCMD.2

DSEG AT 24h

MSGSTAT: DS 1 ; I2C communications status.
MYADDR: DS 1 ; Address of this I2C node.
DESTADR: DS 1 ; Destination address + R/W (for Master).
DESSUBAD: DS 1 ; Destination subaddress.
MASTCNT: DS 1 ; Number of data bytes in message (Master,
; send or receive).

TITOCNT: DS 1 ; Timer I bus watchdog timeouts counter.
StackSave: DS 1 ; SP save location (used when returning from
; bus recovery routine).

MasBuf: DS 4 ; Master receive/transmit buffer, 8 bytes.
SRcvBuf: DS 4 ; Slave receive buffer, 8 bytes.
STxBuf: DS 4 ; Slave transmit buffer, 8 bytes.

RBufLen EQU 4h ; The length of SRcvBuf
;*****
; APPLICATION output pins and RAM definitions
;*****

; Outputs used by the application:

;;TogLED BIT P1.0 ; Toggling output pin, to confirm
; that the ping-pong game proceeds fine.
;;ErrLED BIT P1.1 ; Error indication.
;;OnLED BIT P1.3 ;

; Application RAM
APPFLAGS DATA 21h
TRQFLAG BIT APPFLAGS.0
; Flag for monitoring I2C transmission success.
SErrFLAG BIT APPFLAGS.1

FAILCNT: DS 1
TOGCNT: DS 1 ; Toggle counter.

```

Using the P82B715 I²C extender on long cables

AN444

```

;*****
;
;                               Program Start
;
;*****
CSEG
; Reset and interrupt vectors.

        AJMP    DONMON           ;;JUMP TO MONITOR
                                   ;Reset vector at address 0.

; A timer I timeout usually indicates a 'hung' bus.

TimerI:  ORG     1Bh              ; Timer I (I2C timeout) interrupt.
          SETB   CLRTI
          AJMP   TIISR           ; Go to Interrupt Service Routine.
;*****
;                               I2C Interrupt Service Routine
;*****
;
; Notes on the interrupt mechanism:
;
; Other interrupts are enabled during this ISR upon return from XRETI.
; Limitations imposed on other ISR's:
; - Should not be long (close to 1000 clock cycles). A long ISR will cause
;   the I2C bus to 'hang", and a TIMERI interrupt to occur.
; - Other interrupts either do not use the same mechanism for allowing
;   further interrupts, or if they do - disable TIMERI interrupt beforehand.
;
; The 751 hardware allows only one level of interrupts. We simulate an
; additional level by software: by performing a RETI instruction (at location
; XRETI) the interrupt-in-progress flip-flop is cleared, and other interrupts
; are enabled. The second level of interrupt is a must in our implementation,
; enabling timeout interrupts to occur during "stuck" wait loops in the I2C
; interrupt service routine.

I2CISR:  ORG     23h

        CLR     EI2              ; Disable I2C interrupt.
        ACALL  XRETI           ; Allow other interrupts to occur.
        PUSH   PSW
        PUSH   ACC
        MOV    A,R0
        PUSH   ACC
        MOV    A,R1
        PUSH   ACC
        MOV    A,R2
        PUSH   ACC

        MOV    StackSave, SP
        CLR   TIRUN
        SETB  TIRUN

        JB     STP,NoGo
        JNB   MASTER, GoSlave
        MOV   MSGSTAT,#MGO
        JB    STR,GoMaster
NoGo:     MOV   MSGSTAT,#NOTSTR
        AJMP  Dismiss           ; Not a valid Start.

XRETI:   RETI

```

Using the P82B715 I²C extender on long cables

AN444

```

;*****
;                               Main Transmit and Receive Routines
;*****

; SLAVE CODE -
; GET THE ADDRESS

GoSlave:  MOV     MSGSTAT,#SGO
AddrRcv:  ACALL   C1sRcv8
          JNB     DRDY, SMsgEnd      ; Must be some strange Start or Stop
                                           ; before the address byte was completed.
                                           ; Not a valid address.

STstRW:   MOV     C,ACC.0           ; Save R/W- bit in carry.
          CLR     ACC.0            ; Clear that bit, leaving "raw" address
          JZ      GoIdle          ; If it is a General Address
                                           ; - ignore it.

                                           ; NOTE:
                                           ; One may insert here a different
                                           ; treatment for general calls, if
                                           ; these are relevant.

          JC      SlvTx           ; It's a Read - (requesting slave
                                           ; transmit).

; It is a Write (slave should receive the message).

; Check if message is for us

SRcv2:    CJNE   A,MYADDR,GoIdle   ; If not my address - ignore the
                                           ; message.
          MOV     R1,#SRcvBuf      ; Set receive buffer address.
          MOV     R2,#RbufLen+1
          SJMP   SRcv3

SRcvSto:  MOV     @R1,A            ; Store the byte
          INC     R1              ; Step address.

SRcv3:    ACALL   AckRcv8
          JNB     DRDY,SRcvEnd     ; Exit loop -end reception.
          DJNZ   R2,SRcvSto       ; Go to store byte if buffer not full.

; Too many bytes received - do not acknowledge.
          MOV     MSGSTAT,#SRLNG   ; Notify main that (as slave) we
                                           ; have received too long a message.
          ACALL   SLnRCvdR        ; Handle new data - slave event routine.
          SJMP   GoIdle

; Received a byte, but not DRDY - check if a legitimate message end.

SRcvEnd:  CJNE   R0,#7,SRcvErr    ; If bit count not 7, it was not
                                           ; a Start or a Stop.

; Received a complete message

          MOV     MSGSTAT,#SRCVD   ; Calculate number of bytes received

          MOV     A,R1
          CLR     C
          SUBB   A,#SRcvBuf       ; number of bytes in ACC
          ACALL   SRCvdR          ; Handle new data - slave event routine.
          SJMP   SMsgEnd

; It is a Read message, check if for us.

SlvTx:    NOP

STx2:     CJNE   A,MYADDR,GoIdle   ; Not for us.
          MOV     I2DAT,#0        ; Acknowledge the address.
          JNB     ATN,$           ; Wait for attention flag.

```

Using the P82B715 I²C extender on long cables

AN444

```

JNB     DRDY, SMsgEnd      ; Exception - unexpected Start
                               ; or Stop before the Ack got out.
MOV     R1, #STxBuf       ; Start address of transmit buffer.
STxlp:  MOV     A, @R1      ; Get byte from buffer
        INC     R1
        ACALL  XmByte
        JNB    DRDY, SMsgEnd ; Byte Tx not completed.
        JNB    RDAT, STxlp  ; Byte acknowledge, proceed trans.
        MOV    I2CON, #BCDR+BIDLE ; Master Nak'ed for msg end.
        MOV    MSGSTAT, #STXED
        ACALL  STXedR      ; Slave transmitted event routine.
        AJMP  Dismiss

SRcvErr: MOV    MSGSTAT, #SRERR ; Flag bus/protocol error
        ACALL  SRerrR      ; Slave error event routine.
        SJMP  SMsgEnd

StxErr:  MOV    MSGSTAT, #SRERR ; Flag bus/protocol error
        ACALL  SRerrR

SMsgEnd: JB     MASTER, SMsgEnd2
        JB     STR, GoSlave  ; If it was a Start, be Slave

SMsgEnd2:
        AJMP  Dismiss

; End of Slave message processing

GoIdle:
        AJMP  Dismiss

;
;

GoMaster:
; Send address & R/W~ byte

        MOV    R1, #MasBuf   ; Master buffer address
        MOV    R2, MASTCNT   ; # of bytes, to send or rcv
        MOV    A, DESTADRW   ; Destination address (including
                               ; R/W~ byte).
        JB     SUBADD, GoMas2 ; Branch if subaddress is needed.

        ACALL  XmAddr

        JNB    DRDY, GM2
        JNB    ARL, GM3
GM2:     AJMP  AdTxAr1        ; Arbitration loss while transmitting
                               ; the address.
GM3:     JB     RDAT, Noslave  ; No Ack for address transmission.
        JB     ACC.0, MRcv    ; Check R/W~ bit
        AJMP  MTx

; Handling subaddress case:

GoMas2:  NOP                 ; Subaddress needed. Address in ACC.
        CLR    ACC.0         ; Force a Write bit with address.
        ACALL  XmAddr
        JNB    DRDY, GM4
        JNB    ARL, GM5
GM4:     AJMP  AdTxAr1        ; Arbitration loss while transmitting
                               ; the address.
GM5:     JB     RDAT, Noslave  ; No Ack for address transmission.
        MOV    A, DESSUBAD
        ACALL  XmByte        ; Transmit subaddress.
        JNB    DRDY, SMsgEnd2 ; Arbitration loss (by Start or Stop)
        JB     ARL, SMsgEnd2 ; Arbitration loss occurred.
        JB     RDAT, NoAck    ; Subaddress transmission was not ack'ed.
        MOV    A, DESTADRW   ; Reload ACC with address.
        JNB    ACC.0, MTx    ; It's a Write, so proceed
                               ; by sending the data.
        ; Read message, needs rp. Start and add. retransmit.

```

Using the P82B715 I²C extender on long cables

AN444

```

MOV     I2CON,#BCDR+BXSTR ; Send Repeated Start.
JNB     ATN,$
MOV     I2CON,#BCDR      ; Clear useless DRDY while preparing
                                ; for Repeated Start.
JNB     ATN,$            ; expecting an STR.
JNB     ARL,GM6
AJMP    MARlEnd         ; oops - lost arbitration.
GM6:    ACALL    XmAddr   ; Retransmit address, this time with the
                                ; Read bit set.
JNB     DRDY,GM7
JNB     ARL,GM8
GM7:    AJMP    AdTxAr1   ; Arbitration loss while transmitting
                                ; the address.
GM8:    JB      RDAT,Noslave ; No Ack - the slave disappeared.
SJMP    MRcv           ; Proceed receiving slave's data.

; A Write message. Master transmits the data.
MTx:    NOP
MTxLoop: MOV     A,@R1      ; Get byte from buffer.
INC     R1                ; Step the address.
ACALL   XmByte
JNB     DRDY,SMsgEnd2     ; Arbitration loss (by Start or Stop)
JB      ARL,SMsgEnd2     ; Arbitration loss.
JB      RDAT,NoAck
DJNZ   R2,MTxLoop       ; Loop if more bytes to send.
MOV     MSGSTAT,#MTXED   ; Report completion of buffer
                                ; transmission.
SJMP    MTxStop
NoSlave: MOV     MSGSTAT,#MTXNOSLV
SJMP    MTxStop
NoAck:  MOV     MSGSTAT,#MTXNAK
SJMP    MTxStop

; Master receive - a Read frame
MRcv:   ACALL   ClaRcv8    ; Receive a byte.
SJMP    MRcv2
MRcvLoop: ACALL   AckRcv8
MRcv2:  JNB     DRDY,Marl  ; Other's Start or Stop.
MOV     @R1,A            ; Store received byte.
INC     R1                ; Advance address.
DJNZ   R2,MRcvLoop

; Received the desired number of bytes - send Nack.
MOV     I2DAT,#80h
JNB     ATN,$
JNB     DRDY,Marl
MOV     MSGSTAT,#MRCVED
SJMP    MTxStop         ; Go to send Stop or Repeated Start.

; Conclude this Master message:
; Send Stop, or a Repeated Start
MTxStop: JNB     RPSTRT,MTxStop2 ; Check if Repeated Start needed
                                ; Around if not RPSTRT.
MOV     I2CON,#BCDR+BXSTR ; Send Repeated Start.
SJMP    MTxStop3
MTxStop2: MOV     C,SETMRQ    ; Set new Master Request if demanded
MOV     MASTRQ,C          ; by SETMRQ bit of MASCMD.
MOV     I2CON,#BCDR+BXSTP ; Request the HW to send a Stop.

MTxStop3: JNB     ATN,$      ; Wait for Attention
MOV     I2CON,#BCDR      ; Clear the useless DRDY, generated
                                ; by SCL going high in preparation
                                ; for the Stop or Repeated Start.
JNB     ATN,$            ; Wait for ARL, STP or STR.
JB      ARL,MarlEnd     ; Lost arbitration trying to send
                                ; Stop or a ReStart.

```

Using the P82B715 I²C extender on long cables

AN444

```

; Master is done with this message. May proceed with new messages, if any,
; or exit.

        ACALL   MastNext           ; Master Event Routine. May Prepare
                                   ; the pointers and data for the
                                   ; next Master message.

        JNB     MASTRQ,MMsgEnd     ; Go end service routine if MASTRQ
                                   ; does not indicate that the master
                                   ; should continue (was set according
                                   ; to SETMRQ bit, or by MastNext).

        JNB     STR,MMsgEnd        ; Return from the ISR, unless Start
                                   ; (avoid danger if we do not return:
                                   ; if there was a Stop, the watchdog
                                   ; is inactive until next Start).

        AJMP    GoMaster           ; Loop for another Master message
                                   ;
MMsgEnd:                               ; End of Master messages,
        SJMP    Dismiss

; Terminate mastership due to an arbitration loss:

Mar1:

        JNB     STR,Mar12         ; If lost arbitration due to other
                                   ; Master's Start, go be a slave.

        AJMP    GoSlave

Mar12:

        AJMP    Dismiss
; Switch from Master to Slave due to arbitration loss after completing
; transmission of a message. The MASTRQ bit was cleared trying to write a
; Stop, and we need to set it again on order to retry transmission when the
; bus gets free again.

Mar1End:

        SETB    MASTRQ           ; Set Master Request - which will get
                                   ; into effect when we are done as a
                                   ; slave.

        ACALL   MORERR           ;; INCREASE ERROR COUNT

        AJMP    Mar1

; Handling arbitration loss while transmitting an address

AdTxAr1:  JB     STR,Mar1         ; Non-synchronous Start or Stop.
           JB     STP,Mar1

; Switch from Master to Slave due to arbitration loss while transmitting
; an address - complete receiving the address transmitted by the new Master.

        CJNE    R0,#0,AdTxAr12   ; Arl on last bit of address
                                   ; (R0 is 0 on exit from XmAddr).

        DEC     A                 ; The lsb sent, in which arl occurred
                                   ; must have been 1. By decrementing
                                   ; A we get the address that won.

        SJMP    AdAr3

AdTxAr12:

        RR      A                 ; Realign partially Tx'ed ACC
        MOV     R1,A              ; and save itin R1
        MOV     A,R0              ; Pointer for lookup table
        MOV     DPTR,#MaskTable
        MOVC    A,@A+DPTR
        ANL    A,R1              ; Set address bits to be received,
                                   ; and the bit on which we lost
                                   ; arbitration to 0
                                   ; Now we are ready to receive the rest
                                   ; of the address.

```


Using the P82B715 I²C extender on long cables

AN444

```

MOV     I2CON,#BCXA+BCARL    ; Clear flags and release the clock.

ACALL  RBit3                ; Complete the address using reception
                                ; subroutine.
JB     DRDY,AdAr3           ; Around if received address OK
AJMP   SMsgEnd              ; Unexpected Start or Stop - end
                                ; as a slave.
AdAr3: AJMP   STstRW         ; Proceed to check the address
                                ; as a slave.

MaskTable: DB      0ffh,7Eh,3Eh,1Eh,0Eh,06h,02h,00h, ; 0ffh is dummy
; End I2C Interrupt Service Routine:

Dismiss: ACALL  I2CDONE
MOV      I2CON,#BCARL+BCSTP+BCDR+BCXA+BDLE
CLR      TIRUN
POP      ACC
MOV      R2,A
POP      ACC
MOV      R1,A
POP      ACC
MOV      R0,A
POP      ACC
POP      PSW
SETB    EI2

RET      ; Return from I2C interrupt Service Routine
;*****
;
;      Byte Transmit and Receive Subroutines
;*****
; XmAddr: Transmit Address and R/W-
; XmByte: Transmit a byte

XmAddr: MOV     I2DAT,A      ; Send first bit, clears DRDY.
MOV     I2CON,#BCARL+BCSTR+BCSTP
                                ; Clear status, release SCL.
MOV     R0,#8                ; Set R0 as bit counter
SJMP   XmBit2
XmByte: MOV     R0,#8
XmBit:  MOV     I2DAT,A      ; Send the first bit.
XmBit2: RL     A            ; Get next bit.
JNB     ATN,$                ; Wait for bit sent.
JNB     DRDY,XmBex          ; Should be data ready.
DJNZ   R0,XmBit            ; Repeat until all bits sent.
MOV     I2CON,#BCDR+BCXA    ; Switch to receive mode.
JNB     ATN,$                ; Wait for acknowledge bit.
                                ; flag cleared.

XmBex:  RET

;
; Byte receive routines.
;
; ClsRcv8 clears the status register (from Start condition)
; and then receives a byte.
; AckRcv8 Sends an acknowledge, and then receives a new byte.
; If a Start or Stop is encountered immediately after the
; ack, AckRcv8 returns with 7 in R0.
; ClaRcv8 clears the transmit active state and releases clock
; (from the acknowledge).
;
; A contains the received byte upon return.
; R0 is being used as a bit counter.
;

ClsRcv8: MOV     I2CON,#BCARL+BCSTR+BCSTP+BCXA
                                ;Clear status register.
JNB     ATN,$
JNB     DRDY,RCVex
SJMP   Rcv8

```

Using the P82B715 I²C extender on long cables

AN444

```

AckRcv8:  MOV     I2DAT,#0           ; Send Ack (low)
          JNB     ATN,$             ;
          JNB     DRDY,RCVerr       ; Bus exception - exit.
ClaRcv8:  MOV     I2CON,#BCDR+BCXA  ; clear status, release clock
          ; from writing the Ack.
          JNB     ATN,$
Rcv8:     MOV     R0,#7             ; Set bit counter for the first seven
          ; bits.
          CLR     A                 ; Init received byte to 0.
RBit:     ORL     A,I2DAT           ; Get bit, clear ATN.
RBit2:    RL      A                 ; Shift data.
          JNB     ATN,$             ; Wait for next bit.
          JNB     DRDY,RCVex       ; Exit if not a data bit (could be Start/
          ; Stop, or bus/protocol error)
RBit3:    DJNZ   R0,RBit           ; Repeat until 7 bits are in.
          MOV     C,RDAT           ; Get last bit, don't clear ATN.
          RLC     A                 ; Form full data byte.
RCVex:    RET
RCVerr:   MOV     R0,#9             ; Return non legitimate bit count
          RET
;*****
;           Timer I Interrupt Service Routine
;           I2C us Timeout
;*****
; In addition to reporting the timeout in MSGSTAT, we update a failure
; counter, TITOCNT. This allows different types of timeout handling by the
; main program.

TIISR:    CLR     MASTRQ            ; "Manual" reset.
          MOV     I2CON,#BXSTP      ;
          MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP
TI1:      MOV     MSGSTAT,#TIMOUT    ; Status Flag for Main.
TI2:      ACALL  MORERR             ;;INC TITOCNT
TI4:      ACALL  RECOVER
          SETB    CLRTI             ; Clear TI interrupt flag.
          ACALL  XRETI             ; Clear interrupt pending flag (in
          ; order to re-enable interrupts).
          MOV     SP,StackSave      ; Realign stack pointer, re-doing
          ; possible stack changes during
          ; the I2C interrupt service routine.
          ; TimerI interrupts in other ISR's
          ; were not allowed !
          AJMP   Dismiss            ; Go back to the I2C service routine,
          ; in order to return to the (main)
          ; program interrupted.

;*****
;           Bus recovery attempt subroutine
;*****
RECOVER:   CLR     EA               ; "Manual" reset.
          CLR     MASTRQ            ;
          MOV     I2CON,#BCXA+BDLE+BCDR+BCARL+BCSTR+BCSTP
          CLR     SLAVEN           ; Non I2C TimerI mode
          SETB    TIRUN           ; Fire up TimerI. When it overflows, it
          ; will cause I2C interface hardware reset.
          MOV     R1,#0ffh
DLY5:     NOP
          NOP
          NOP
          DJNZ   R1,DLY5
          CLR     TIRUN
          SETB    CLRTI
          SETB    SCL               ; Issue clocks to help release other devices.
          SETB    SDA
          MOV     R1,#08h

```

Using the P82B715 I²C extender on long cables

AN444

```

RC7:      CLR      SCL
          DB       0,0,0,0,0
          SETB     SCL
          DB       0,0,0,0,0
          DJNZ    R1,RC7
          CLR      SCL
          DB       0,0
          CLR      SDA
          DB       0,0
          SETB     SCL
          DB       0,0,0,0,0
          SETB     SDA
          DB       0,0,0,0,0 ; Issue a Stop.

Rex:      MOV      I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP ; clear flags
          SETB     EA
          RET

;*****
;
;                               Main Program
;
;*****

; Message ping pong game. Each message is transmitted by
; a processor that is a master on the I2C bus, and it contains one byte
; of data. A processor that receives this data byte as a slave increments
; the data by one and transmits it back as a master. The data received is
; confirmed to be a one increment of the data formerly sent, unless
; it is a "reset" value, chosen to be 00h.
; The two participating processors have similar code, where the node
; address of the second processor is the destination address of this
; one, and vice versa.
; The first data byte each processor tries to send is 00h. One of the
; processors will acquire the bus first, and the second processor that will
; receive this "resetting" 00h will not attempt to confirm it against an
; expected value. It will simply increment and transmit it. Subsequent
; receptions will be confirmed against the expected value, until 0ffh data
; bytes are sent and the game is effectively reset by the 00h resulting from
; the next increment.
; A toggling output (TogLED) tells the outer world that the "ping pong"
; proceeds well. If something unexpected happens we temporarily activate
; another output, ErrLED.
; The different tasks of the code are performed in a combination of main-
; line program and event routines called from the I2C interrupt service
; routine.

; Initial set-ups:
; Load CTL,CT0 bits of I2CFG register, according to the clock
; crystal used.
; Load RAM location MYADDR with the I2C address of this processor.
; We load these values out of ROM table locations (R_CTVAL and R_MYADDR).
; One may, instead, load with a MOV <immediate> command.

;;Reset:  MOV      SP,#07h ;Set stack location.
RESET:   CLR      A
          MOV      DPTR,#R_CTVAL
          MOVC     A,@A+DPTR
          MOV      I2CFG,A ; Load CTL,CT0 (I2C timing, crystal
                           ; dependent).

          CLR      A
          MOV      DPTR,#R_MYADDR
          MOVC     A,@A+DPTR ; Get this node's address from ROM table
          MOV      MYADDR,A ; into MYADDR RAM location.

;;
CLR      OnLED

```

Using the P82B715 I²C extender on long cables

AN444

```

;;Reset2:   CLR      ErrLED                ; Flash LED.
RESET2:    ACALL   LDELAY
;;         SETB    ErrLED
           CLR     SErrFLAG
           CLR     TRQFLAG
           MOV     FAILCNT,#50h
;;         SETB    TogLED
           MOV     TOGCNT,#050h        ; Initialize pin-toggling counter

; Enable slave operation.
; The Idle bit is set here for a restart situation - in normal
; operation this is redundant, as this bit is set upon power_up reset.
           MOV     I2CON,#BIDLE        ; Slave will idle till next Start.
           SETB    SLAVEN              ; Enable slave operation.

; Enable interrupts.
; This is necessary for both Slave and Master operations.
           SETB    ETI                 ; Enable timer I interrupts.
           SETB    EI2                 ; Enable I2C port interrupts.
           SETB    EA                   ; Enable global interrupts.

; Set up Master operation.
           MOV     MASCMD,#0h          ; "Regular" master transmissions.
           MOV     DPTR,#PongADDR
           CLR     A
           MOVC    A,@A+DPTR
           MOV     DESTADRW,A         ; The partner address. The LSB is
                                       ; low, for a Write transaction.
           MOV     MASTCNT,#01h       ; Message length - a single byte.

PPSTART:   MOV     MasBuf,#00h

; "Ping" transmission:
PP2:
           SETB    TRQFLAG
           SETB    MASTRQ
           MOV     R1,#0ffh
PP22:      JNB     TRQFLAG,PP3         ; Transmitted OK
           DJNZ    R1,PP22
MFAIL1:    DJNZ    FAILCNT,PP2
           ACALL   MORERR              ; INCREMENT TITOCNT
           ACALL   RECOVER
           SJMP   Reset2

; "Pong" reception:
PP3:       MOV     R0,#0ffh           ; Software timeout loop count.
PP31:      MOV     R1,#0ffh
PP32:      JB      TRQFLAG,PP2        ; Rcvd ok as slave, go transmit.
           JB      SErrFLAG,PP5
           DJNZ    R1,PP32
           DJNZ    R0,PP31
PPTO:      ACALL   RECOVER            ; Software timeout.
           AJMP   Reset2

;;PP5:     CLR     ErrLED              ; Receive error.
;;         ACALL   LDELAY
;;         SETB    ErrLED
PP5:       CLR     SErrFLAG
           AJMP   PPSTART

LDELAY:    MOV     R2,#030h           ;LONG DELAY
LDELAY1:   MOV     R1,#0ffh
           DJNZ    R1,$
           DJNZ    R2,LDELAY1
           RET

```

Using the P82B715 I²C extender on long cables

AN444

```

;*****
; Slave and Master Event Routines.
;*****
;
; Invoked upon completion of a message transaction.
; This is the part of the application program actually dealing
; with the data communicated on the I2C bus, by responding to
; new data received and/or preparing the next transaction.
; Slave Event Routines
;
; These routines are invoked by the I2C interrupt service routine when a
; message transaction as a slave has been completed. Our "application"
; reacts to a message received as a slave with the routine SRCvdR.
; The calls that indicate erroneous reception are treated the same way as
; erroneous data reception in the "ping pong" game.
; SRCvdR
; Invoked when a new message has been received as a Slave.
SRCvdR:  NOP
        MOV     A,SRcvBuf
        JNZ     SR2
        MOV     MasBuf,#01h      ; It was ping-pong reset value
        SJMP    SR3
SR2:     INC     MasBuf           ; The expected data.
        CJNE   A,MasBuf,ErrSR
        INC     MasBuf           ; Data for next transmission - the data
                                   ; received incremented by 1.
;
; A successful two way data exchange. Let the outside world know by
; toggling an output pin driving a LED. We actually toggle only
; when a number of such exchanges is completed, in order to
; slow down the changes for a good visual indication.
        DJNZ   TOGCNT,SR3
;;      CPL     TogLED           ; Toggle output
        XRL   TITOCNT, #80H     ;;TOGGLE MSB LED
        MOV   TOGCNT,#050h      ;
        SETB  PSW.3             ;;RS TO 1
        MOV   LED, @R0          ;;RAM POINTED TO BY R0
        CLR   PSW.3             ;;RS BACK TO 0
SR3:     CLR   SErrFLAG
        SETB  TRQFLAG          ; Request main to transmit
        RET
ErrSR:   SETB  SErrFLAG
        RET
; SLnRcvdR
; Invoked when a message received as a Slave is too long
; for the receive buffer.
; STXedR
; Invoked when a Slave completed transmission of its buffer.
; We do not expect to get here, since we do not plan to have
; in our system a master that will request data from this node.
;
; SRErrR
; Slave error event subroutine.
; In most applications it will not be used.
;
SLnRcvdR:
STXedR:
SRErrR:  JMP     ErrSR

```

Using the P82B715 I²C extender on long cables

AN444

```

;
; MastNext - Master Event Routine.
;
; Invoked when a Master transaction is completed, or terminated
; "willingly" due to lack of acknowledge by a slave.
;
MastNext:
    MOV     A,MSGSTAT
    CJNE   A,#MTXED,MN1
    MOV     FAILCNT,#50h
    CLR    TRQFLAG
    RET

MN1:
    RET

; I2CDONE
; Called upon completion of the I2C interrupt service routine.
; In this example it monitors exceptions, and invokes the bus
; recovery routine when too many occurred.

I2CDONE:
    MOV     A,MSGSTAT
    CJNE   A,#NOTSTR,I2CD1
    ACALL  MORERR          ;; INCREMENT TITOCNT
    DJNZ   FAILCNT,I2CD1
    MOV     FAILCNT,#01h   ; Too many "illegal" i2c interrupts
    CLR    EI2            ; - shut off.
I2CD1:   RET

;*****
;           I2C Communications Table:
;*****

; We used table driven values for clarity. One may use immediates to load
; these values and save several lines of code.

; Contents is used in the beginning of the main program to load
; RAM location MYADDR and the I2CFG register.
; The node address, in R_MYADDR, is application specific, and unique for
; each device in the I2C network.
; R_CTVAL depends on the crystal clock frequency.

R_MYADDR:  DB    4Ah          ; This node's address
           ; ;NOTE THAT R_MYADDR AND PongADDR
           ; ;MUST BE SWITCHED ON THE OTHER
           ; ;'751
R_CTVAL:   DB    02h          ; CT1, CT0 bit values

;*****
;           Application Code Definitions
;*****

PongADDR:  DB    4Eh          ; The address of the "partner" in
           ; ; the ping-pong game.

;;I2CMON   THIS PROGRAM RUNS THE MONITOR ON
;;         THE SMALL TEST BOARD DESIGNED TO
;;         TEST THE I2C DRIVER CHIP.
;;         IT USES A '751.
;
;
LED        EQU     P3
LDEL       EQU     022H
HDEL       EQU     LDEL + 1
    
```

Using the P82B715 I²C extender on long cables

AN444

```

SWITCH EQU P1
TOG EQU P0.2 ;TOGGLE SWITCH
RNAME EQU R0 ;R0 RAM POINTER
;
;
;
DONMON: MOV SP, #09H ;SP=09, STARTS AT 0AH
        SETB PSW.3 ;RS = 01
        CLR PSW.1 ;PSW.1 FLAG=0
        JB TOG, ONLYAD ;IF TOG 1, PSW1=0
        SETB PSW.1 ;WRITE DESIRED
ONLYAD: JNB TOG, ONLYAD ;WAIT FOR HI
HIWAIT: JB TOG, HIWAIT ;NOW WAIT FOR LOW
        MOV LDEL, #0 ;DELAY TIMER
        MOV HDEL, #0
SDELAY: DJNZ LDEL, SDELAY ;DELAY LOOP
        DJNZ HDEL, SDELAY ;UPPER DELAY
        JB TOG, HIWAIT ;FALSE ALARM, GO BACK
        MOV RNAME, SWITCH ;VALID HI TO LO
        MOV LED, @RNAME ;DISPLAY CONTENTS OF
        ; RAM OF RNAME
        JNB PSW.1, DONE ;PSW1 FLAG, 0=DONE
STAYLO: JNB TOG, STAYLO ;NOW WAIT FOR HI
HDELAY: DJNZ LDEL, HDELAY ;LDEL=HDEL=0
        DJNZ HDEL, HDELAY
        JNB TOG, STAYLO ;FALSE ALARM
        MOV @RNAME, SWITCH ;SUCCESSFUL LO TO HI
        ; SWITCH TO RAM
        MOV LED, RNAME ;DISPLAY WHICH RAM
        ;LOCATION FOR SWITCH
DONE: CLR PSW.3 ;RS BANK BACK TO 0
      AJMP RESET ;STARTS PING PONG
;
;
MORERR: PUSH ACC
        MOV A, #7FH ;;INCREMENT TITOCNT
        ANL A, TITOCNT
        XRL A, #7FH ;;STOP AT 7F
        JZ NOUP
        INC TITOCNT
        SETB PSW.3 ;;RS TO 1
        MOV LED, @R0 ;;DISPLAY NEW TITOCNT
        CLR PSW.3 ;;RS BACK TO 0
NOUP: POP ACC
      RET
;
      END

```

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

Author: R.C.J. Brink, Eindhoven

1. INTRODUCTION

1.1. Purpose

This document is a user manual for the I²C software module IIC51. It is intended for Intel PLM51 users who need to control an I²C bus. This document assumes some basic knowledge about I²C and Intel PLM51.

1.2. Scope

IIC51 is a software module to provide an Intel PLM51 user with a set of procedures to control a bi-directional I²C bus. These procedures have been coded in Intel ASM51 and have been optimized for speed. IIC51 supports all common used I²C master transmitter and master receiver protocols. Each different protocol corresponds to one of the procedures in IIC51. IIC51 is available in two different versions:

IIC51S:

IIC51S is a module for singlemaster I²C to be used on microcontrollers of the 8XC51 family. It directly controls the microcontroller I/O pins by software without the need of any specific hardware. No other I²C masters are allowed on the bus. Note that the electrical characteristics of this microcontroller family are not conform the I²C specifications.

IIC51M:

IIC51M is a module for multimaster I²C to be used on microcontrollers of the PCB8XC552/C652 family. It makes use of the built-in I²C interface hardware (SIO1) of these microcontrollers. Since this hardware is a multimaster interface, other I²C masters are allowed on the bus.

All I²C transfer procedures in IIC51S are fully software interface compatible with IIC51M. This allows a single PLM51 program using I²C to be written for both mentioned microcontroller families.

1.3. Definitions, Acronyms and Abbreviations

I ² C	Inter-IC bus
PLM51	High level Program Language for 8051 family Microcontrollers
ASM51	Assembly Language for 8051 family Microcontrollers
RL51	Relocating Linker for 8051 family Microcontrollers
S	I ² C Message Start Condition
P	I ² C Message Stop Condition
A	I ² C Message Acknowledge
N	I ² C Message Negative Acknowledge
SIVW	I ² C Message Slave Address + Write
SIVR	I ² C Message Slave Address + Read
Sub	I ² C Slave Subaddress

NV-Memory I²C Controlled Non Volatile Memory (E²PROM)

1.4. References

- I²C Specification
I²C-bus compatible ICs
Philips Components Data Handbook IC12a 1989
- PL/M-51 User's Guide for DOS Systems
Intel Corporation 1986
- MSC-51 Macro Assembler User's Guide for DOS Systems
Intel Corporation 1986
- MSC-51 Utilities User's Guide for DOS Systems
Intel Corporation 1986
- Single-chip 8-bit microcontrollers PCB83C552/PCB80C552, PCB83C652/PCB80C652 etc.
I²C-bus compatible ICs
Philips Components Data Handbook IC12a 1989
- Single-chip 8-bit microcontroller PCB80C51
Integrated circuits Book IC14 1987

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

2. GENERAL DESCRIPTION

2.1. Perspective

IIC51 is designed for use in stand-alone microcontroller I²C systems. It is mainly written to provide a standard set of procedures for computer controlled television/teletext concepts based on 8051 family microcontrollers.

2.2. Functions

IIC51 contains the following functions:

- Initialisation of the I²C interface (software and hardware)
- Transfer of I²C messages to and from an I²C slave device
- Error detection
- Automatic retrying if an error occurs during a transfer (up to 5 attempts)
- Error recovery if the bus is held by a slave device that is out of bit-sync
- Optional slave receiver/transmitter function (IIC51M only)

2.3. User Characteristics

IIC51 is designed to be an easy to use package. All needed code and data is defined in a single object module (IIC51M.OBJ or IIC51S.OBJ). The PLM51 user needs only to link this module to his own application object modules, using Intel's RL51. Procedures and data of concern to the user can be declared EXTERNAL by including the file IIC51.DCL.

2.4. General Constraints

IIC51 is coded for and translated by the Intel MSC-51 Macro Assembler. It is tested together with Intel PLM51 modules. Intel utilities used for testing:

- MSC-51 Macro Assembler, ASM51.EXE, Version V2.3
- PL/M-51 Compiler, PLM51.EXE, Version V1.2 and V1.3
- MSC-51 Relocator and Linker, RL51.EXE, Version V3.1

Development is done on an IBM-PC/AT running DOS.

IIC51S needs:

- 350 Bytes CODE (approx.)
- 6 Bytes DATA
- 1 Byte Bit-Addressable DATA
- 1 Bit

IIC5MS needs:

- 400 Bytes CODE (approx.)
- 6 Bytes DATA
- 1 Byte Bit-Addressable DATA
- 1 Bit
- Exclusive use of Register Bank 1

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

3. FUNCTIONAL DESCRIPTION

3.1. Master Mode Functions

This section describes the available functions in IIC51 on a procedure by procedure basis.

Each procedure must be declared EXTERNAL by the PLM51 user. In this declaration the user can specify the type returned by each procedure. All procedures (except Init_IIC) can return a BIT or a BYTE (depending on the chosen EXTERNAL declaration). The BIT or BYTE returned is 0 if the I²C transmission was successful. If the user decides to declare a procedure untyped, the result of the previous I²C transmission can always be checked by examining the static BIT variable IIC_Error. Note that typed procedures must be called using an expression. If the result of an I²C procedure is to be ignored, a dummy assignment must be done for a typed procedure. An untyped procedure can be called by the PLM51 CALL statement, without any additional overhead. The examples in the following section assume the procedures to be declared untyped.

Note that the least significant bit of all slaveaddresses passed to the I²C procedures must be 0.

3.1.1. Init_IIC

Declaration:

```
Init_IIC:
  PROCEDURE ( Own_Slave_Address ) EXTERNAL ;
  DECLARE ( Own_Slave_Address ) BYTE ;
  END ;
```

Description:

Init_IIC must be called once after reset, before any other procedure is used. It initialises all I²C internal static data and hardware. The Own_Slave_Address is passed to Init_IIC for the optional slave function in a multimaster I²C system (IIC51M). In a singlemaster I²C system (IIC51S), the Own_Slave_Address is ignored. Note that Init_IIC does not effect the global interrupt enable flag (EA). IIC51M requires the user to enable interrupts afterwards (see example).

Example:

```
CALL Init_IIC ( 54h ) ;
ENABLE ;                /*Enable Interrupts; EA = 1 */
```

3.1.2. IIC_Test_Device

Declaration:

```
IIC_Test_Device:
  PROCEDURE ( Slave_Address ) [ BIT | BYTE ] EXTERNAL ;
```

Description:

IIC_Test_Device just sends the slaveaddress on the I²C bus. It can be used to check the presence of a device on the I²C bus.

I²C Protocol:

S	SlvW	A	P	(Device is Present, IIC_Error = 0)
---	------	---	---	--------------------------------------

OR

S	SlvW	N	P	(Device is Not Present, IIC_Error = 1)
---	------	---	---	--

Example:

```
DECLARE IIC_Error BIT EXTERNAL ;
.....
CALL IIC_Test_Device ( 8Ch ) ;
IF ( IIC_Error ) THEN
  "Device is Not Present Handling"
ELSE
  "Device is Present Handling"
```

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

3.1.3. IIC_Write**Declaration:**

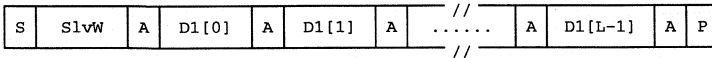
```
IIC_Write:
  PROCEDURE ( Slave_Address, Count, Source_Ptr ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Count, Source_Ptr ) BYTE ;
  END ;
```

Description:

IIC_Write is the most basic procedure to write a message to a slave device.

I²C Protocol:

```
L           = Count
D1[0..L-1] BASED by Source_Ptr
```

**Example:**

```
DECLARE Data_Buffer ( 4 ) BYTE ;
.....
CALL IIC_Write ( 0C2h, LENGTH ( Data_Buffer ), .Date_Buffer ) ;
```

3.1.4. IIC_Write_Sub**Declaration:**

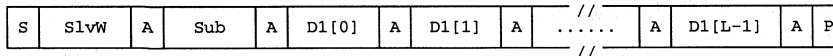
```
IIC_Write_Sub:
  PROCEDURE ( Slave_Address, Count, Source_Ptr, Sub_Address ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Count, Source_Ptr, Sub_Address ) BYTE ;
  END ;
```

Description:

IIC_Write_Sub writes a message preceded by a subaddress to a slave device.

I²C Protocol:

```
L           = Count
Sub         = Sub_Address
D1[0..L-1] BASED by Source_Ptr
```

**Example:**

```
DECLARE Data_Buffer ( 8 ) BYTE ;
.....
CALL IIC_Write_Sub ( 48h, LENGTH ( Data_Buffer ), .Date_Buffer, 2 ) ;
```

3.1.5. IIC_Write_Sub_SWInc**Declaration:**

```
IIC_Write_Sub_SWInc:
  PROCEDURE ( Slave_Address, Count, Source_Ptr, Sub_Address ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Count, Source_Ptr, Sub_Address ) BYTE ;
  END ;
```

Description:

Some I²C devices addressed with a subaddress do not automatically increment the subaddress after reception of each byte. IIC_Write_Sub_SWInc can be used for such devices the same way IIC_Write_Sub is used. IIC_Write_Sub_SWInc splits up the message in smaller messages and increments the subaddress itself.

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

I²C Protocol:

L = Count
 Sub = Sub_Address
 D1[0..L-1] BASED by Source_Ptr

S	SlvW	A	Sub	A	D1[0]	A	P
---	------	---	-----	---	-------	---	---

S	SlvW	A	Sub+1	A	D1[1]	A	P
---	------	---	-------	---	-------	---	---

.....

S	SlvW	A	Sub+L-1	A	D1[L-1]	A	P
---	------	---	---------	---	---------	---	---

Example:

```
DECLARE Data_Buffer ( 6 ) BYTE ;
.....
CALL IIC_Write_Sub_SWinc ( 80h, LENGTH ( Data_Buffer ), .Date_Buffer, 0 ) ;
```

3.1.6. IIC_Write_Memory

Declaration:

```
IIC_Write_Memory:
PROCEDURE ( Slave_Address, Count, Source_Ptr, Sub_Address) [ BIT| BYTE ] EXTERNAL ;
DECLARE ( Slave_Address, Count, Source_Ptr, Sub_Address ) BYTE ;
END ;
```

Description:

I²C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. IIC_Write_Memory can be used to write to such devices the same way IIC_Write_Sub is used. IIC_Write_Memory splits up the message in smaller messages and increments the subaddress itself. After transmission of each small message a delay of 40 milliseconds is inserted.

I²C Protocol:

L = Count
 Sub = Sub_Address
 D1[0..L-1] BASED by Source_Ptr

S	SlvW	A	Sub	A	D1[0]	A	P	< Delay 40 ms >
---	------	---	-----	---	-------	---	---	-----------------

S	SlvW	A	Sub+1	A	D1[1]	A	P	< Delay 40 ms >
---	------	---	-------	---	-------	---	---	-----------------

.....

S	SlvW	A	Sub+L-1	A	D1[L-1]	A	P	< Delay 40 ms >
---	------	---	---------	---	---------	---	---	-----------------

Example:

```
DECLARE Data_Buffer ( 10 ) BYTE ;
.....
CALL IIC_Memory ( 0A0h, LENGTH ( Data_Buffer ), .Date_Buffer, 0F0h ) ;
```

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

3.1.7. IIC_Write_Sub_Write**Declaration:**

```

IIC_Write_Sub_Write:
  PROCEDURE ( Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Source_Ptr2 )
    [ BIT | BYTE ] EXTERNAL ;
  DECLARE ( Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Source_Ptr2 ) BYTE ;
  END ;

```

Description:

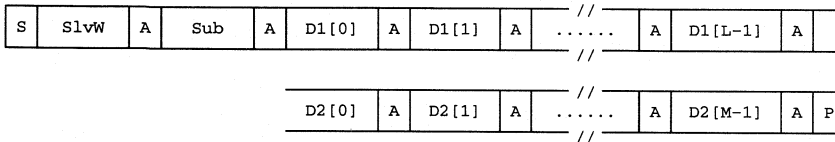
IIC_Write_Sub_Write write 2 data blocks preceded by a subaddress in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol:

```

L           = Count1
M           = Count2
Sub         = Sub_Address
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Source_Ptr2

```

**Example:**

```

PROCEDURE Write_CCT_Memory ( Chapter, Row, Column, Data_Buf, Data_Count ) ;
DECLARE ( Chapter, Row, Column, Data_Buf, Data_Count ) BYTE;

/*
  The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These
  three bytes are written after the subaddress (8) followed by the actual data which
  will be stored relative to the extended address.
*/
CALL IIC_Write_Sub_Write ( 22h, 3, .Chapter, 8, Data_Buf, Data_Count ) ;

END Write_CCT_Memory ;

```

3.1.8. IIC_Read**Declaration:**

```

IIC_Read:
  PROCEDURE ( Slave_Address, Count, Dest_Ptr ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE ( Slave_Address, Count, Dest_Ptr ) BYTE ;
  END ;

```

Description:

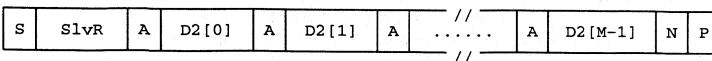
IIC_Read is the most basic procedure to read a message from a slave device.

I²C Protocol:

```

M           = Count
D2[0..M-1] BASED by Dest_Ptr

```

**Example:**

```

DECLARE Data_Buffer ( 4 ) BYTE ;
.....
CALL IIC_Read ( 0B4h, LENGTH ( Data_Buffer ), .Data_Buffer ) ;

```

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

3.1.9. IIC_Read_Status**Declaration:**

```
IIC_Read_Status:
  PROCEDURE ( Slave_Address, Dest_Ptr ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Dest_Ptr ) BYTE ;
  END ;
```

Description:

A lot of I²C devices have only a one status byte that can be read via I²C. IIC_Read_Status can be used for this purpose. IIC_Read_Status works the same as IIC_Read but the user does not have to pass a count parameter.

I²C Protocol:

```
M           = Count
Status      BASED by Dest_Ptr
```

S	SlvR	A	Status	N	P
---	------	---	--------	---	---

Example:

```
DECLARE Status_Byte BYTE ;
.....
CALL IIC_Read_Status ( 84h, .Status_Byte ) ;
```

3.1.10. IIC_Read_Sub**Declaration:**

```
IIC_Read_Sub:
  PROCEDURE ( Slave_Address, Count, Dest_Ptr, Sub_Address ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Count, Dest_Ptr, Sub_Address ) BYTE ;
  END ;
```

Description:

IIC_Read_Sub reads a message from a slave device preceded by a write of the subaddress. Between writing the subaddress and reading the message, an I²C restart condition is generated without surrendering the bus. This prevents other masters from accessing the slave device in between and overwriting the subaddress.

I²C Protocol:

```
M           = Count
Sub         = Sub_Address
D2[0..M-1] BASED by Dest_Ptr
```

S	SlvW	A	Sub	A	S	SlvR	A	D2[0]	A	D2[1]	A	A	D2[M-1]	N	P
---	------	---	-----	---	---	------	---	-------	---	-------	---	-------	---	---------	---	---

Example:

```
DECLARE Data_Buffer ( 5 ) BYTE ;
.....
CALL IIC_Write_Sub ( 0A2h, LENGTH ( Data_Buffer ), .Data_Buffer, 2 ) ;
```

3.1.11. IIC_Write_Sub_Read**Declaration:**

```
IIC_Write_Sub_Read:
  PROCEDURE ( Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Dest_Ptr2 )
    [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Dest_Ptr2 ) BYTE ;
  END ;
```

Description:

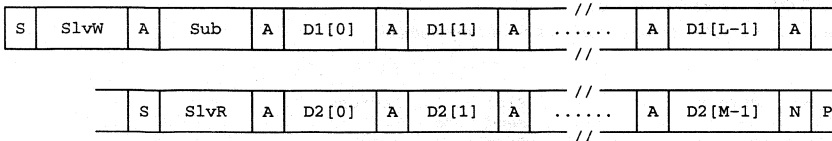
IIC_Write_Sub_Read writes a data block preceded by a subaddress, generates an I²C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT (I²C controlled teletext device; see example).

PLM51 I²C software interface IIC51 (version 0.5)

ETV/AN89004

I²C Protocol:

L = Count1
M = Count2
Sub = Sub_Address
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Dest_Ptr2

Example:

```
PROCEDURE Read_CCT_Memory ( Chapter, Row, Column, Data_Buf, Data_Count );
DECLARE ( Chapter, Row, Column, Data_Buf, Data_Count ) BYTE ;

/*
The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These
three bytes are written after the subaddress (8). After that the actual data will be
read relative to the extended address.
*/
CALL IIC_Write_Sub_Read ( 22h, 3, .Chapter, 8, Data_Buf, Data_Count ) ;

END Read_CCT_Memory ;
```

3.2. Slave Mode Functions

I²C slave mode is provided by IIC51M only. All slave mode actions (except initialisation) take place in the SIO1 interrupt procedure. Slave mode I²C protocol is very application dependent. If a specific slave mode is required, the user have to modify three procedures in IIC51M at source level. The following sections describe these procedures. The program examples of the procedures implement an I²C slave protocol to read and write the microcontroller's on chip RAM via I²C. This can be a useful feature during program development and debugging.

3.2.1. Init_Slave

This procedure is called from IIC_Init. In this procedure the user can initialise all static data concerning slave mode functions (if any).

Example:

```
Slave_Sub_Address: db 1
.....
Init_Slave:      mov Slave_Sub_Address,#0 ; Initialise Sub Address
                ret
```

3.2.2. Receive_Slave

Receive_Slave is a procedure called from the SIO1 interrupt procedure each time a byte is received from another I²C master. The procedure can make use of the bit "IICntrl.BYTE1EXPECTED", as defined in IIC51M. This bit is set to logic 1, every time the first data byte of an I²C message is about to be received. Receive_Slave can use this bit to detect the start of a new message.

Normally all bytes received from the other master will be acknowledged (i.e., SIO1 control bit Assert Acknowledge is set, AA = 1). If AA is cleared by Receive_Slave subsequent bytes in the message will be ignored and a negative acknowledge will be transmitted after reception of each byte. Note that the example does not make use of this feature.

Constraints:

- Receive_Slave **must** read the S1DAT register.
- Receive_Slave may clear the SIO1 control bit AA, to stop acknowledging data.
- Receive_Slave may not effect any other SIO1 hardware registers/bits.
- Receive_Slave is only allowed to use the accumulator and register R0 in the current registerbank.

Example:

```
Receive_Slave:  mov a,S1DAT                ; Pick up data
                mov r0,#Slave_Sub_Address ; Prepare for 1st byte
                jbc IICntrl.BYTE1EXPECTED,Save_Byte ; Jump if 1st byte
                mov r0,Slave_Sub_Address ; Else data byte
                inc Slave_Sub_Address    ; Postincrement Sub.
Save_Byte:     mov @r0,a                ; Save Data
                ret                    ; Exit
```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.2.3. Send_Slave

Send_Slave is a procedure called during the SIO1 interrupt procedure each time a byte has to be transmitted to another I²C master. This occurs after reception of I²C startcondition followed by the microcontroller's own slaveaddress (as passed to Init_IIC) with read-bit. Send_Slave will be called again after transmission of each subsequent byte, until a negative acknowledge is received from the reading I²C master.

Constraints:

- Send_Slave **must** write to the S1DAT register.
- Send_Slave may not effect any other SIO1 hardware registers/bits.
- Send_Slave is only allowed to use the accumulator and register R0 in the current registerbank.

Example:

```
Send_Slave:   mov r0,Slave_Sub_Address      ; Pick up Sub Address
              mov S1DAT,@r0                ; Send Data
              inc Slave_Sub_Address        ; Postincrement Sub.
              ret
```


I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

Author: J.C.P.M. Pijenburg, Eindhoven

1. INTRODUCTION

This report describes the I²C drivers which are written for the 8xC751/2. The report describes not only how to use the routines, but also the structure of the software. The software is written around a set of basic routines and a message handler. The message handler does not contain any specific 8xC751 code, so the software can be easily rewritten for any other bit level I²C interface by rewriting the set of basic routines. In the rest of this report, when 8xC751 is written, it means 8xC751/2.

The package supports also the multimaster features of the I²C bus.

The maximum bit rate possible when using those routines is approximately 70Kbit/sec.

References:

- The I²C-bus specification: 9398 358 10011
- 80C51-based 8-bit microcontrollers Data Handbook IC20
- PLM51 I²C Software interface I2C51: ETV/AN89004

2. GENERAL

2.1 Memory Usage & File Structure

The driver software consists of 3 main parts:

- I²C message handler
- I²C basic routines
- I²C slave routines

During I²C usage it claims register bank 1, however, register bank 1 does not contain any static I²C data and can be used by the application program outside the I²C routines (this data will be destroyed by I²C routines). The accumulator is also modified during I²C transfer.

The message handler uses a Message Control Block which consists of 8 bytes RAM. In those bytes, the following parameters are stored:

For block 1: I2C_ADDR_1, BUF_LEN and BUF_PTR_1

For block 2: I2C_ADDR_2, BUF_LEN and BUF_PTR_2

2 bytes of bit addressable RAM for STATUS and CONTROL information

The STATUS byte is returned into the accumulator. If you do not need a detailed status, you can test the carry bit, this is a copy of the I2C_ERROR bit of the status register (returned in the acc.). The status register contains the following information:

BIT	NAME	FUNCTION
0	RETRY_0	
1	RETRY_1	— Retry counter (0..7), as given during I2C_INIT
2	RETRY_2	
3	I2C_ERR	I2C error if set (also available in carry)
4	TIME_ERR	Bus timeout occurred if set
5	RECOVER	— (no value for user) always 0
6	BUS_RECOVERED	If set, bus K recovered after timeout
7	NO_ACK	No acknowledge received

The slave function uses 2 bytes of RAM, those contain the own slave address (OWN_SLV_ADDR) and a pointer the slave transmit/receive buffer of the 8xC751. This is the buffer from/in which the 8xC751 gets/stores the data bytes in slave mode.

The I²C module is built around a message handler which calls basic functions such as I2C_TRX_BYTE and I2C_START. Each function calls the message handler after loading the correct mask into the I2C_CTRL byte.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

FILENAME	FUNCTION	INCLUDE/LINK	CODE SIZE (BYTE)
I2C_DATA.GLO	I ² C global data definitions	I, each I ² C function and assembler main	0
I2C_DATA.LOC	I ² C local data definitions	I, each I ² C function	0
I2C_CODE.GLO	I ² C global function definitions	I, assembler main	0
I2C_INIT.ASM	Init_I2C (does not use message handler)	Link	22
I2C_DEF.ASM	Define MCB & _I2C_xxx_BYTEs	Link	0
I2C_HAND.ASM	I ² C Message handler	Link	144
I2C_BASI.ASM	I ² C basic functions, and T1 interrupt handling	Link	293
I2C_TDEV.ASM	I ² C Test_Device	Link, if used	5
I2C_WRIT.ASM	I ² C Write	Link, if used	5
I2C_WSUB.ASM	I ² C Write_Sub	Link, if used	5
I2C_WSWI.ASM	I ² C Write_Sub_SWinc & Write_Mem	Link, if used	36
I2C_WSUW.ASM	I ² C Write_Sub_Write	Link, if used	5
I2C_WSUR.ASM	I ² C Write_Sub_Read	Link, if used	5
I2C_WCOW.ASM	I ² C Write_Com_Write	Link, if used	11
I2C_WREW.ASM	I ² C Write_Rep_Write	Link, if used	5
I2C_WRER.ASM	I ² C Write_Rep_Read	Link, if used	5
I2C_READ.ASM	I ² C Read and Read_Status	Link, if used	11
I2C_RSUB.ASM	I ² C Read_Sub	Link, if used	5
I2C_RRER.ASM	I ² C Read_Rep_Read	Link, if used	5
I2C_RREW.ASM	I ² C Read_Rep_Write	Link, if used	5
I2C_SLAV.ASM	I ² C Slave Function	Link	56

The total memory usage for the full package is

ROM	:	520 (single function) to 623 (all functions) bytes
RAM	byte addressable	: 8 bytes
	bit addressable	: 2 bytes
	register bank 1	: 8 bytes

The message handler, causes the other functions to be very small, to further reduce the code, all functions are placed in separate modules, which are put into a library I2C_751.LIB. If this library is linked to an application program, only the object modules which are used by the application program are linked in the output file.

The I2C_CODE.H file contains the references to the separate functions (EXTRN CODE definitions). The use must not include this file into main, but only copy the definitions which he needs into the source file. If this file is included, all functions will be linked, the library approach is of no use in this case.

2.2 Retries

During initialization, the user defines whether he wants to use retries or not. If an I²C message fails, and retries >=0, the program restarts the message. This is done for at most 7 times. If the message remains unsuccessful, the message handler returns to the main program, indicating that the message has failed (carry set).

2.3 Error Handling

In case of an error while operating as master, the program returns to the message handler. The message handler decides whether to invoke a retry or to return to the main program.

The I²C interface of the 8xC751 generates a timeout interrupt if the bus hangs for more than 1022 cycles, in this case, if the 8xC751 is master (RECOVER = 1), a bus recover routine is started, if the 8xC751 is not master, the I²C bus is released. Retries are only invoked in the master situation.

2.4 Development Tools

The following software tools from Tasking/BSO are used for program development:

- OM4142 Cross Assembler 8051 for DOS: V3.0b
- OM4144 PL/M 8051 Compiler for DOS: V3.0a
- OM4136 C8051 Compiler for DOS: V1.1a
- OM4129 XRAY51 debugger: V1.4c

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3. MASTER ROUTINES

3.1 Message Handler

To make the I²C protocols as described in paragraphs 3.2 to 3.15, and I²C message handler is written. The message control block (I2C_MCB) together with the I²C control byte (I2C_CTRL) form the input for the message handler. The I2C_MCB includes 6 bytes of data, containing:

- I2C_ADDR1, first address in the protocol
- BUF_LEN_1, length of the first data buffer
- BUF_PTR_1, pointer to the first data buffer
- I2C_ADDR2, second address in the protocol
- BUF_LEN_2, length of the second data buffer
- BUF_PTR_2, pointer to the second data buffer

The I2C_CTRL byte is bit addressable. It contains 8 bits that determine the flow through the message handler. This byte must be loaded with the corresponding mask before starting the message handler.

The I2C_CTRL byte contains the following bits:

- REP_STRT_BLK1 must we send a repeated start before the first data block? (0=NO, 1=YES)
- RWN_BLK1 read (1) or write (0) the first block of data
- ADDR2 is there a second address in the protocol? (0=NO, 1=YES)
- ADDR2_SUB is the 2nd address a sub address, only relevant if ADDR2=1.
- BLOCK2 is there a second block of data in the protocol? (0=NO, 1=YES)
- RWN_BLK1 read (1) or write (0) the first block of data
- REP_STRT_BLK1 must we send a repeated start before the second data block?
- TEST_DEVICE is it the test device protocol

When an I²C protocol is handled successfully by the message handler, it returns control to the main program; if not it can do a retry by resending the message (maximum 5 retries are possible).

The message handler return value is stored in the I2C_STAT byte. The I2C_ERROR bit indicates whether the transfer has succeeded.

NOTE: It is better to copy this byte into Acc before returning the control to the main program, this way a byte can be saved (I2C_STAT can be placed in register bank 1) and the main program can do a JZ/JNZ test (must be changed).

3.2 I2C_INIT

Description

Init_I2C must be called after RESET, before any procedure is called. The I²C interface and I²C interrupt will be enabled (STEB ET1, EI2 and EA). Own_Slave_Address is passed to Init_I2C for use as slave. Slave_Sub_Address is the pointer to a DATA buffer that is used for data transfer in slave mode. When used as master in a single master system, these parameters are not used. Retry is the number of retries on messages when an error occurs. 0 means no retry (just 1 attempt to send a message), while the maximum amount of retries is 7.

I²C Protocol

none (no action at I²C bus)

Calling Sequence

```
C          : I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry);
PL/M51    : I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry);
Assembler : %I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry);
           : (macro call)
```

Parameters

```
Own_Slave_Adr   : 8xC751 own slave address
Slave_Buffer_Adr : Base address of buffer, to transmit data from, or receive data in, when 8xC751 is in slave mode.
Retry           : Number of times to do a retry in case of an error. 0 = No Retry, maximum retries is 7.
```

NOTE:

The Init_I2C function enables the I²C watchdog timer interrupt (TI). This watchdog generates an interrupt when during and I²C transfer, SCL is held longer than 1022 machine cycles (ca. 760µs @ 16MHz). If this time is too short for your application, you can disable the TI (CLR ETI). In this case, the main program must check if a bus hangup occurs, and take proper action when the bus is hangup.

I²C driver routines for 8XC751/2 microcontrollers

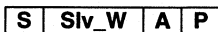
EIE/AN91007

3.3 I2C_TEST_DEVICE**Description**

I2C_Test_Device just sends the slave address to the I²C bus. It can be used to check the presence of a device on the I²C bus.

I²C Protocol

Slv_W : Slave_Adr + Write bit



Device is present



Device is not present

Calling Sequence

C : I2C_TEST_DEVICE(Slv_Adr);
 PL/M51 : I2C_TEST_DEVICE(Slv_Adr);
 Assembler : %I2C_TEST_DEVICE(Slv_Adr);
 (macro call)

Parameters

Slave_Adr : Slave address of the device to be tested.

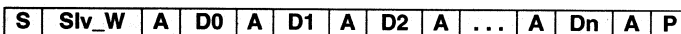
3.4 I2C_WRITE**Description**

I2C_Write is the most basic procedure to write a message to a slave device.

I²C Protocol

Slv_W : Slave_Adr + Write bit

D0..Dn : Data bytes

**Calling Sequence**

C : I2C_WRITE(Slv_Adr,Count,Source_Ptr);
 PL/M51 : I2C_WRITE(Slv_Adr,Count,Source_Ptr);
 Assembler : %I2C_WRITE(Slv_Adr,Count,Source_Ptr);

Parameters

Slave_Adr : Slave address of the device to write to.
 Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
 Source_Ptr : Pointer to data buffer, to transmit bytes from.
 (macro call)

I²C driver routines for 8XC751/2 microcontrollers

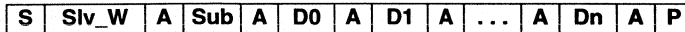
EIE/AN91007

3.5 I2C_WRITE_SUB**Description**

I2C_Write_Sub writes a message preceded by a sub-address to a slave device.

I²C Protocol

Slv_W : Slave_Adr + Write bit
 Sub : Sub_Adr
 D0..Dn : Data bytes

**Calling Sequence**

C : I2C_WRITE_SUB(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 PL/M51 : I2C_WRITE_SUB(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 Assembler : %I2C_WRITE_SUB(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 (macro call)

Parameters

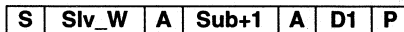
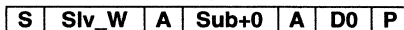
Slave_Adr : Slave address of the device to write to.
 Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
 Source_Ptr : Pointer to data buffer, to transmit bytes from.
 Sub_Adr : Sub address.

3.6 I2C_WRITE_SUB_SWINC**Description**

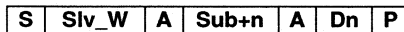
Some I²C devices addressed with a sub-address do not automatically increment the sub-address after reception of each byte. I2C_Write_Sub_SWinc can be used for such devices the same way as I2C_Write_Sub is used. I2C_Write_Sub_SWinc splits up the message in smaller messages and increments the sub-address itself.

I²C Protocol

Slv_W : Slave_Adr + Write bit
 Sub+x : Sub_Adr+x
 D0..Dn : Data bytes



.....

**Calling Sequence**

C : I2C_WRITE_SUB_SWINC(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 PL/M51 : I2C_WRITE_SUB_SWINC(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 Assembler : %I2C_WRITE_SUB_SWINC(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 (macro call)

Parameters

Slave_Adr : Slave address of the device to write to.
 Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
 Source_Ptr : Pointer to data buffer, to transmit bytes from.
 Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

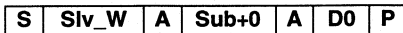
3.7 I2C_WRITE_MEMORY

Description

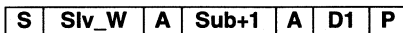
I²C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. I2C_Write_Memory can be used to write to such devices the same way I2C_Write_Sub is used. I2C_Write_Memory splits up the message in smaller messages and increments the sub-address itself. After transmission of each message, a delay of 40 milliseconds ($f_{XTAL} = 16\text{MHz}$) is inserted.

I²C Protocol

Slv_W : Slave_Adr + Write bit
 Sub+x : Sub_Adr+x
 D0..Dn : Data bytes



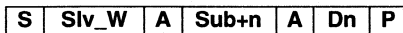
40 mS



40 mS

.

40 mS



40 mS

Calling Sequence

C : I2C_WRITE_MEMORY(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 PL/M51 : I2C_WRITE_MEMORY(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 Assembler : %I2C_WRITE_MEMORY(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 (macro call)

Parameters

Slave_Adr : Slave address of the device to write to.
 Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
 Source_Ptr : Pointer to data buffer, to transmit bytes from.
 Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

3.8 I2C_WRITE_SUB_WRITE**Description**

I2C_Write_Sub_Write writes 2 data blocks preceded by a sub-address in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol

Slv_W : Slave_Adr + Write bit
 Sub : Sub_Adr
 D1.0..D1.n : Data bytes in first block
 D2.0..D2.p : Data bytes in second block

S	Slv_W	A	Sub	A	D1.0	A	D1.1	A	...	A	D1.n	A	D2.0	A	...	A	D2.p	A	P
---	-------	---	-----	---	------	---	------	---	-----	---	------	---	------	---	-----	---	------	---	---

Calling Sequence

C : I2C_WRITE_SUB_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Sub_Adr,Count_2,Source_Ptr_2);
 PL/M51 : I2C_WRITE_SUB_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Sub_Adr,Count_2,Source_Ptr_2);
 Assembler : %I2C_WRITE_SUB_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Sub_Adr,Count_2,Source_Ptr_2);
 (macro call)

Parameters

Slave_Adr_1 : Slave address of the device to write to.
 Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n = count_1 - 1)
 Source_Ptr_1 : Pointer to first block of data to transmit.
 Sub_Adr : Sub address.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
 Source_Ptr_2 : Pointer to second block of data to transmit.

3.9 I2C_WRITE_SUB_READ**Description**

I2C_Write_Sub_Read writes a data block preceded by a sub-address, generates an I²C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT.

I²C Protocol

Slv_W : Slave_Adr + Write bit
 Slv_R : Slave_Adr + Read bit
 Sub : Sub_Adr
 D1.0..D1.n : Data bytes in first block (write)
 D2.0..D2.p : Data bytes in second block (read)

S	Slv_W	A	Sub	A	D1.0	A	D1.1	A	...	A	D1.n	A	S	Slv_R	A	D2.0	A	D2.1	A	...	A	D2.p	N	P
---	-------	---	-----	---	------	---	------	---	-----	---	------	---	---	-------	---	------	---	------	---	-----	---	------	---	---

Calling Sequence

C : I2C_WRITE_SUB_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count,Dest_Ptr);
 PL/M51 : I2C_WRITE_SUB_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count,Dest_Ptr);
 Assembler : %I2C_WRITE_SUB_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count,Dest_Ptr);
 (macro call)

Parameters

Slave_Adr_1 : Slave address of the device to write and read to/from.
 Count_1 : Number of bytes to transmit (D1.0 .. D1.n, n = count - 1)
 Source_Ptr_1 : Pointer to first block of data to transmit.
 Sub_Adr : Sub address.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
 Dest_Ptr_2 : Pointer buffer to receive second block of data in.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

3.10 I2C_WRITE_COM_WRITE**Description**

I2C_Write_Com_Write writes two data blocks from different data buffers in one message to a slave receiver. This procedure can be used for devices where the message consists of 2 different data blocks. Such devices are, for instance, LCD-drivers, where the first part of the message consists of addressing and control information, and the second part is the data string to be displayed.

I²C Protocol

Slv_W : Slave_Adr + Write bit
 D1.0..D1.n : Data bytes in first block (write)
 D2.0..D2.p : Data bytes in second block (write)

S	Slv_W	A	D1.0	A	D1.1	A	D1.2	A	...	A	D1.n	A	D2.0	A	...	A	D2.p	A	P
---	-------	---	------	---	------	---	------	---	-----	---	------	---	------	---	-----	---	------	---	---

Calling Sequence

C : I2C_WRITE_COM_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2);
 PL/M51 : I2C_WRITE_COM_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2);
 Assembler : %I2C_WRITE_COM_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2);
 (macro call)

Parameters

Slave_Adr : Slave address of the device to write to.
 Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n = count_1 - 1)
 Source_Ptr_1 : Pointer to first block of data to transmit.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
 Source_Ptr_2 : Pointer to second block of data to transmit.

3.11 I2C_WRITE_REP_WRITE**Description**

Two data strings are sent to separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

Slv1W : Slave_Adr_1 + Write bit
 Slv2W : Slave_Adr_2 + Write bit
 D1.0..D1.n : Data bytes in first block (write to first slave)
 D2.0..D2.p : Data bytes in second block (write to second slave)

S	Slv1W	A	D1.0	A	D1.1	A	D1.2	A	...	A	D1.n	A	P
---	-------	---	------	---	------	---	------	---	-----	---	------	---	---

Calling Sequence

C : I2C_WRITE_REP_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Sub_Adr,Count_2,Source_Ptr_2);
 PL/M51 : I2C_WRITE_REP_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Sub_Adr,Count_2,Source_Ptr_2);
 Assembler : %I2C_WRITE_REP_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Sub_Adr,Count_2,Source_Ptr_2);
 (macro call)

Parameters

Slave_Adr_1 : Slave address of first device to write to.
 Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n = count_1 - 1)
 Source_Ptr_1 : Pointer to first block of data to transmit.
 Slave_Adr_2 : Slave address of second device to write to.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
 Source_Ptr_2 : Pointer to second block of data to transmit.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

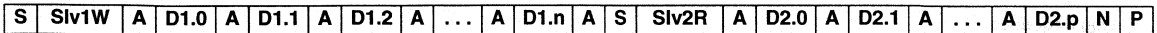
3.12 I2C_WRITE_REP_READ

Description

A data string is sent and received to/from two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

- Slv1W : Slave_Adr_1 + Write bit
- Slv2R : Slave_Adr_2 + Read bit
- D1.0..D1.n : Data bytes in first block (write to first slave)
- D2.0..D2.p : Data bytes in second block (write to second slave)



Calling Sequence

- C : I2C_WRITE_REP_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count_2,Dest_Ptr);
- PL/M51 : I2C_WRITE_REP_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count_2,Dest_Ptr);
- Assembler : %I2C_WRITE_REP_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count_2,Dest_Ptr);
(macro call)

Parameters

- Slave_Adr_1 : Slave address of first device to write to.
- Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n = count_1 - 1)
- Source_Ptr_1 : Pointer to first block of data to transmit.
- Slave_Adr_2 : Slave address of second device to read from.
- Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
- Dest_Ptr_2 : Pointer buffer to receive second block of data in.

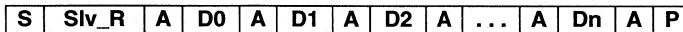
3.13 I2C_READ

Description

I2C_Read is the most basic procedure to read a message from a slave device.

I²C Protocol

- Slv_R : Slave_Adr + Read bit
- D0 .. Dn : Data bytes



Calling Sequence

- C : I2C_READ(Slv_Adr,Count,Dest_Ptr);
- PL/M51 : I2C_READ(Slv_Adr,Count,Dest_Ptr);
- Assembler : %I2C_READ(Slv_Adr,Count,Dest_Ptr);
(macro call)

Parameters

- Slave_Adr : Slave address of the device to be tested.
- Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
- Dest_Ptr : Pointer to data buffer, to receive bytes in.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

3.14 I2C_READ_STATUS**Description**

Several I²C devices can send a one byte status-word via the bus. I2C_Read_Status can be used for this purpose. I2C_Read_Status works the same way as I2C_Read, but the user does not have to pass a count parameter.

I²C Protocol

Slv_R : Slave_Adr + Read bit
 Status : Status bytes

S	Slv_R	A	Status	A	P
---	-------	---	--------	---	---

Calling Sequence

C : I2C_READ_STATUS(Slv_Adr, Dest_Ptr);
 PL/M51 : I2C_READ_STATUS(Slv_Adr, Dest_Ptr);
 Assembler : %I2C_READ_STATUS(Slv_Adr, Dest_Ptr);
 (macro call)

Parameters

Slave_Adr : Slave address of the device to be tested.
 Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
 Dest_Ptr : Pointer to data buffer, to receive status byte in.

3.15 I2C_READ_SUB**Description**

I2C_Read_Sub reads a message from a slave device, preceded by a write of the sub-address. Between writing the sub-address and reading the message, an I²C restart condition is generated without releasing the bus. This prevents other masters from accessing the slave device in between and overwriting the sub-address.

I²C Protocol

Slv_W : Slave_Adr + Write bit
 Slv_R : Slave_Adr + Read bit
 Sub : Sub_Adr

S	Slv_W	A	Sub	A	S	Slv_R	A	D1	A	...	A	Dn	N	P
---	-------	---	-----	---	---	-------	---	----	---	-----	---	----	---	---

Calling Sequence

C : I2C_READ_SUB(Slv_Adr, Count, Dest_Ptr, Sub_Adr);
 PL/M51 : I2C_READ_SUB(Slv_Adr, Count, Dest_Ptr, Sub_Adr);
 Assembler : %I2C_READ_SUB(Slv_Adr, Count, Dest_Ptr, Sub_Adr);
 (macro call)

Parameters

Slave_Adr : Slave address of the device to be tested.
 Count : Number of bytes to transmit (D0 .. Dn, n = count - 1)
 Dest_Ptr : Pointer to data buffer, to receive bytes in.
 Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

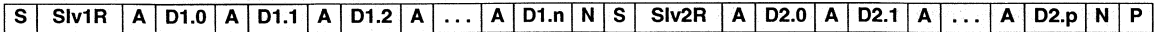
3.16 I2C_READ_REP_READ

Description

Two data strings are read from separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

- Slv1R : Slave_Adr_1 + Read bit
- Slv2R : Slave_Adr_2 + Read bit
- D1.0 .. D1.n : Data bytes in first block (read from first slave)
- D2.0 .. D2.p : Data bytes in second block (read from second slave)



Calling Sequence

- C : I2C_READ_REP_READ(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Dest_Ptr_2);
- PL/M51 : I2C_READ_REP_READ(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Dest_Ptr_2);
- Assembler : %I2C_READ_REP_READ(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Dest_Ptr_2);
(macro call)

Parameters

- Slave_Adr_1 : Slave address of first device to write to.
- Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n = count_1 - 1)
- Dest_Ptr_1 : Pointer buffer to receive first block of data in.
- Sub_Adr_2 : Slave address of second device to read from.
- Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
- Dest_Ptr_2 : Pointer buffer to receive second block of data in.

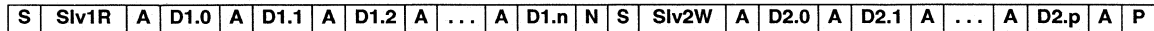
3.17 I2C_READ_REP_WRITE

Description

A data string is received and sent from/to two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

- Slv1R : Slave_Adr_1 + Read bit
- Slv2W : Slave_Adr_2 + Write bit
- D1.0 .. D1.n : Data bytes in first block (read from first slave)
- D2.0 .. D2.p : Data bytes in second block (read from second slave)



Calling Sequence

- C : I2C_READ_REP_WRITE(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Source_Ptr);
- PL/M51 : I2C_READ_REP_WRITE(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Source_Ptr);
- Assembler : %I2C_READ_REP_WRITE(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Source_Ptr);
(macro call)

Parameters

- Slave_Adr_1 : Slave address of first device to write to.
- Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n = count_1 - 1)
- Dest_Ptr_1 : Pointer buffer to receive first block of data in.
- Sub_Adr_2 : Slave address of second device to read from.
- Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p = count_2 - 1)
- Dest_Ptr_2 : Pointer buffer to transmit second block of data from.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

4. SLAVE ROUTINES

The slave-mode protocol is very application dependent. In this note the basic slave-receive and slave-transmit routines are given and should be considered as examples. The user may, for instance, send NO_ACK after receiving a number of bytes to signal to the master-transmitter that a data buffer is full. A listing of the slave routines is given in Appendix III.

The I²C slave function has two entries:

1. The I²C interrupt.

This can only occur at an idle slave, because when a transmission is in progress, the I²C interrupt is disabled.

2. Through the master routines.

During transmission of a slave-address in master-mode, arbitration is lost to another master. The interface must then switch to slave-receiver mode to check if this other master wants to address the 8xC751 I²C interface. If the 8xC751 recognizes his own slave address, the slave mode routines are entered at labels I2C_SLV_TRX or I2C_SLV_RCV.

Interfacing the master routines, if the user wants to adapt the slave routines to his own needs, he has to keep in mind that the master routines use the I2C_SLV_TRX and I2C_SLV_RCV entries. The I²C slave routines are entered after the acknowledge has been sent, therefore the ATN flag will be set when entering the slave routines at I2C_SLV_TRX or I2C_SLV_RVC.

The slave routines, as given, make use of a single data buffer. When addressed as slave transmitter, data bytes from the data buffer are transmitted over the I²C bus until a not acknowledge or stop is received. When addressed as slave receiver, the data from the I²C bus is received into the data buffer until a not acknowledge or a stop is received.

The data buffer is initialized during the Init_I2C function, one of the parameters of this function is the pointer to the data buffer (SLV_BUF_PTR DS 1).

4.1 Slave Transmitter

The slave transmitter function transmits data bytes from the 8xC751 data buffer (ACALL I2C_TRX_BYTE) until a not acknowledge or a stop is received. The function is also exit on an I²C error. The function is exit with the ATN bit set.

4.2 Slave Receiver

The slave receiver function receives data bytes into the 8xC751 data buffer (ACALL I2C_RCV_BYTE) until a stop is received. The function is also exit on an I²C error. The function is exit with the ATN bit set. If a byte has been received, an acknowledge is sent.

5. EXAMPLES

5.1 Introduction

Some examples are given on how to use the I²C routines in an application program. Examples are given for an assembly, PL/M and C programs. The program displays time from the PCF8583P clock/calendar/RAM on an LCD display driven by the PCF8577. The example can be executed on the OM4151 I²C evaluation board.

5.2 Using the Routines in Assembly Sources

Appendix VII shows the listing of the example program. The most important aspect when using the I²C routines is preparing the input parameters before the sub-routine call. The parameters must be transferred to the MCB (Message Control Block). Below are 2 examples of how to transfer the necessary parameters to MCB (_I2C_Read and _I2C_Write_Sub_Read).

```
MOV _I2C_MCB,#Slave_Adr
MOV _I2C_MCB+1,#Count_1
MOV _I2C_MCB+2,#Dest_Ptr_1
ACALL _I2C_READ

MOV _I2C_MCB,#S1_Adr
MOV _I2C_MCB+1,#Cnt_1
MOV _I2C_MCB+2,#S_Ptr_1
MOV _I2C_MCB+3,#Sub_Adr
MOV _I2C_MCB+4,#Cnt_2
MOV _I2C_MCB+5,#S_Ptr_2
ACALL _I2C_WRITE_SUB_READ
```

Note that the order of defining the parameters is the same as in PL/M- and C-calls (Calling sequences in paragraphs 3.2 to 3.17). An easier way to call the routines is to make a macro that includes the transfer of the parameters.

The example program makes use of macros. I2C_Read is then called in the following way:

```
%I2C_READ(Slave_Adr,Count_1,Source_Ptr_1);
```

Note that in the listing the macro call is replaced by the contents of the macro.

The macro must be written as follows:

```
;%* DEFINE (I2C_READ(Slave_Adr,Count_1,Dest_Ptr_1))
(
  MOV _I2C_MCB,##Slave_Adr
  MOV _I2C_MCB+1,##Count_1
  MOV _I2C_MCB+2,##Dest_Ptr_1
  ACALL _I2C_READ
)
```

File I2C_MAC.DEF contains the macro calls for the routines as described in paragraphs 3.2 to 3.17. This file should be included in all assembler modules in which calls to the I²C routines are made.

The file I2C_CODE.GLO contains the global function definitions (EXTRN CODE) of the I²C functions, copy the ones you need into your application. The file I2C_DATA.GLO contains the global data definitions of the I²C functions. Therefore, this file must also be included in all assembler modules in which calls to the I²C routines are made.

All I²C routines return a status into the CY-bit. If the CY-bit is set, an error has occurred.

5.3 Using the Routines in PL/M-51 Sources

Appendix VIII shows the listing of the example program in PL/M-51. All procedures return a BIT value. The file I2C_PL/M.H contains the procedure declarations, this file can be included in the modules which call I²C routines. The routines are used the same way as in the examples of paragraph 5.2.

5.4 Using the Routines in C Sources

Appendix IX shows the listing of the example program in C. All functions return a bit value. The file I2C_C.H contains the function prototypes, this file can be included in the modules which call I²C routines. The routines are used the same way as in the examples of paragraph 5.2.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

Read.Me

```

*-----*
*
*   PACKAGE: I2C drivers for 8xc751/2 microcontroller
*
*   DESCRIPTION: To use the package just link the library: I2c_751.lib
*               to your application program
*
*   NOTES: If you use the package with assembler sources, you must include
*          \USER\INCLUDE\I2C_DATA.GLO and \USER\INCLUDE\I2C_MAC.DEF into
*          your main application(s).
*          \USER\INCLUDE\I2C_CODE.GLO contains external code definitions,
*          select the ones you need and copy them into your main application*
*          If you include this file, the linker assumes that you use all I2C*
*          functions and therefore links the complete package to your
*          application (in this case the library approach is of no use!)
*
*          If you use the package with PLM sources, you must include
*          \USER\INCLUDE\I2C_PLM.H in each file which uses an I2C function
*
*          If you use the package with C sources, you must include
*          \USER\INCLUDE\I2C_PLM.C in each file which uses an I2C function
*
*-----*

```

CONTENTS OF DISK

The disk contains 3 directories:

1:\USER :This directory contains 2 directories:

\INCLUDE :

```

I2C_PLM.H      :PLM header file
I2C_C.H       :C header file
I2C_MAC.DEF   :ASM header file,
               Macro definitions for ASM function calls
I2C_DATA.GLO  :I2C global data (assembler only)
I2C_DATA.LOC  :I2C local data (assembler only, not for user)
I2C_CODE.GLO  :I2C extern code definitions (assembler only)
REG751.H      :8xc751 register file

```

\LIB :

```

LIB.BAT       :example batch file to create library
I2C_751.LIB   :8xc751/2 I2C drive library

```

2:\EXAMPLE :This directory contains 3 directories

```

\DEMO_ASM    :Assembly example
\DEMO_PLM    :PL/M example
\DEMO_C      :C example

```

3:\SOURCE :This directory contains the source files of the modules that are put in library with I2C_751.LIB

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007I²C Master routines

```

$ TITLE(I2C_DEF.ASM)
;-----*
;*
;*      INCLUDE FILE:  I2C_DEF.ASM      *
;*      PACKAGE       :  I2C           *
;*-----*

;-----*
;*      This file must be LINKED to each I2C sub function      *
;-----*

USING(1)

;-----*
;*      GLOBAL DATA DECLARATIONS      *
;-----*
PUBLIC      I2C_MCB
PUBLIC      I2C_CTRL
PUBLIC      I2C_STAT
PUBLIC      OWN_SLV_ADDR
PUBLIC      SLV_BUF_PTR

PUBLIC      I2C_ADDR_1
PUBLIC      BUF_LEN_1
PUBLIC      BUF_PTR_1
PUBLIC      I2C_ADDR_2
PUBLIC      BUF_LEN_2
PUBLIC      BUF_PTR_2

;-----*
;*      GLOBAL FUNCTION DECLARATION    *
;-----*
PUBLIC      _I2C_INIT_BYTE
PUBLIC      _I2C_TEST_DEVICE_BYTE
PUBLIC      _I2C_WRITE_BYTE
PUBLIC      _I2C_WRITE_SUB_BYTE
PUBLIC      _I2C_WRITE_SUB_SWINC_BYTE
PUBLIC      _I2C_WRITE_MEMORY_BYTE
PUBLIC      _I2C_WRITE_SUB_WRITE_BYTE
PUBLIC      _I2C_WRITE_SUB_READ_BYTE
PUBLIC      _I2C_WRITE_COM_WRITE_BYTE
PUBLIC      _I2C_WRITE_REP_WRITE_BYTE
PUBLIC      _I2C_WRITE_REP_READ_BYTE
PUBLIC      _I2C_READ_BYTE
PUBLIC      _I2C_READ_STATUS_BYTE
PUBLIC      _I2C_READ_SUB_BYTE
PUBLIC      _I2C_READ_REP_READ_BYTE
PUBLIC      _I2C_READ_REP_WRITE_BYTE

;-----*
;*      GLOBAL FUNCTION DEFINITIONS    *
;-----*
I2C_MCB_DATA      SEGMENT DATA
RSEG              I2C_MCB_DATA

_I2C_INIT_BYTE:
    OWN_SLV_ADDR:  DS      1
    SLV_BUF_PTR:   DS      1

_I2C_TEST_DEVICE_BYTE:
_I2C_WRITE_BYTE:
_I2C_WRITE_SUB_BYTE:
_I2C_WRITE_SUB_SWINC_BYTE:
_I2C_WRITE_MEMORY_BYTE:
_I2C_WRITE_SUB_WRITE_BYTE:
_I2C_WRITE_SUB_READ_BYTE:
_I2C_WRITE_COM_WRITE_BYTE:
_I2C_WRITE_REP_WRITE_BYTE:
_I2C_WRITE_REP_READ_BYTE:
_I2C_READ_BYTE:
_I2C_READ_STATUS_BYTE:
_I2C_READ_SUB_BYTE:
_I2C_READ_REP_READ_BYTE:
_I2C_READ_REP_WRITE_BYTE:

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```
I2C_MCB:          DS      6
  I2C_ADDR_1      DATA    I2C_MCB+0
  BUF_LEN_1       DATA    I2C_MCB+1
  BUF_PTR_1       DATA    I2C_MCB+2
  I2C_ADDR_2      DATA    I2C_MCB+3
  BUF_LEN_2       DATA    I2C_MCB+4
  BUF_PTR_2       DATA    I2C_MCB+5
```

```
I2C_STAT_DATA    SEGMENT DATA BITADDRESSABLE
RSEG              I2C_STAT_DATA
```

```
I2C_CTRL:        DS      1
I2C_STAT:         DS      1
```

END

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

$ TITLE(I2CDATAG.H)
;-----*
;*
;*          INCLUDE FILE:  I2CDATA.GLO
;*          PACKAGE      :  I2C
;*
;-----*

;-----*
;*          This file must be included into each I2C function,
;*          and into the MAIN ASSEMBLER program (if exists)
;*          It contains the I2C Global data definitions
;-----*

;-----*
;*          I 2 C   F R E Q U E N C Y   S E T T I N G S
;-----*

;-----*
;*          This part contains frequency dependent settings
;*          of the 8xC751/8xC752 I2C interface
;*
;*          The user can adapt this part to his own wishes.
;*          If this part has been changed, the whole I2C package
;*          must be assembled, linked and put into a library
;*          again.
;*
;*          The default setting are made for a 16MHz clock
;*          frequency, and a 40 mS delay for programming EEPROM
;-----*
CT1_CT0      EQU          002H    ;Frequency <= 16.8 MHz
              ; 001H    Frequency <= 14.25 MHz
              ; 000H    Frequency <= 11.7 MHz
              ; 003H    Frequency <= 9.14 MHz

EEPROM_PROG_DELAY EQU      103    ;40 Msec at 16MHz

      1 DELAY = 514 * EEPROM_PROG_DELAY * 12/fosc
;-----*
;*          E N D   I 2 C   F R E Q U E N C Y   S E T T I N G S
;-----*

;-----*
;*          G L O B A L   D A T A   D E F I N I T I O N S
;-----*
EXTRN DATA (_I2C_INIT_BYTE)
EXTRN DATA (OWN_SLV_ADDR)
EXTRN DATA (SLV_BUF_PTR)

EXTRN DATA (I2C_MCB)
EXTRN DATA (I2C_ADDR_1)
EXTRN DATA (BUF_LEN_1)
EXTRN DATA (BUF_PTR_1)
EXTRN DATA (I2C_ADDR_2)
EXTRN DATA (BUF_LEN_2)
EXTRN DATA (BUF_PTR_2)

EXTRN DATA (I2C_CTRL)
REP_STRT_BLK1 BIT    I2C_CTRL.0
REP_BLK1      BIT    I2C_CTRL.1
ADDR2         BIT    I2C_CTRL.2
ADDR2_SUB     BIT    I2C_CTRL.3
BLOCK2        BIT    I2C_CTRL.4
RWN_BLK2      BIT    I2C_CTRL.5
REP_STRT_BLK2 BIT    I2C_CTRL.6
TEST_DEVICE   BIT    I2C_CTRL.7

EXTRN DATA (I2C_STAT)
RETRY_0       BIT    I2C_STAT.0
RETRY_1       BIT    I2C_STAT.1
RETRY_2       BIT    I2C_STAT.2
I2C_ERR       BIT    I2C_STAT.3
TIME_ERR      BIT    I2C_STAT.4
RECOVER       BIT    I2C_STAT.5
BUS_RECOVERED BIT    I2C_STAT.6
NO_ACK        BIT    I2C_STAT.7

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```
*****
; *   G L O B A L   S Y M B O L   D E C L A R A T I O N S   *
; *   *****
I2C_START_CTRL      EQU      0D0H+CT1_CT0
I2C_ENABLE          EQU      080H+CT1_CT0
I2C_RELEASE         EQU      0F4H

C_XMFA              EQU      080H
C_IDLE              EQU      040H
C_DRDY              EQU      020H
C_ARL               EQU      010H
C_STRT              EQU      008H
C_STP               EQU      004H
S_RSTR              EQU      022H
S_STP               EQU      021H
```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$ TITLE(I2C_DATA.LOC)
;=====
;*
;*
;*      INCLUDE FILE:  I2C_DATA.LOC
;*      PACKAGE       :  I2C
;*
;*=====
;-----
;*
;*      This file must be included into each I2C function,
;*      It contains the I2C Local symbol definitions
;*-----
;=====
;*
;*      L O C A L   S Y M B O L   D E F I N I T I O N S
;*=====
;
SDA          BIT          81H
SCL          BIT          80H

BUF_PTR      SET          R0
BUF_LEN      SET          R1
BIT_CNT      SET          R2
MESS_RETRY_CNT SET        R3
BUS_ERR_CLKS SET          R4
MEM_MESS_LEN SET          R5
MEM_DELAY_H  SET          R6
MEM_DELAY_L  SET          R7

```

```

$ TITLE(I2C_CODE.H)
;=====
;*
;*
;*      INCLUDE FILE:  I2C_CODE.H
;*      PACKAGE       :  I2C
;*
;*=====
;-----
;*
;*      This file must be included into the ASSEMBLER MAIN
;*      It contains the EXTERNAL CODE references (Global
;*      function definitions) of the I2C functions
;*-----
;=====
;*
;*      G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*=====
;
EXTRN CODE(_I2C_INIT)
EXTRN CODE(_I2C_TEST_DEVICE)
EXTRN CODE(_I2C_WRITE)
EXTRN CODE(_I2C_WRITE_SUB)
EXTRN CODE(_I2C_WRITE_SUB_SWINC)
EXTRN CODE(_I2C_WRITE_MEMORY)
EXTRN CODE(_I2C_WRITE_SUB_WRITE)
EXTRN CODE(_I2C_WRITE_SUB_READ)
EXTRN CODE(_I2C_WRITE_COM_WRITE)
EXTRN CODE(_I2C_WRITE_REP_WRITE)
EXTRN CODE(_I2C_WRITE_REP_READ)
EXTRN CODE(_I2C_READ)
EXTRN CODE(_I2C_READ_STATUS)
EXTRN CODE(_I2C_READ_SUB)
EXTRN CODE(_I2C_READ_REP_READ)
EXTRN CODE(_I2C_READ_REP_WRITE)

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$ TITLE(I2C_Init command)
;*****
;*
;*          SOURCE FILE :  I2C_INIT.ASM
;*          PACKAGE      :  I2C
;*
;*****
$DEBUG

;*****
;*  I N C L U D E S
;*****

;*****
;*  L O C A L  S Y M B O L  D E C L A R A T I O N S
;*****
RETRIES      SET      R3

$NOLIST
$INCLUDE(REG751.H)
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$LIST

;*****
;*  G L O B A L  F U N C T I O N  D E F I N I T I O N S
;*****
PUBLIC _I2C_INIT

;*****
;*  C O D E  S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF::I2C::I2C_INIT.ASM:I2C_INIT=====
;*
;* FUNCTION NAME:      I2C_INIT
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Initialize I2C interface: set SDA & SCL, enable time out
;* timer, allow 16 MHz (CT1,CT0 = 0). Set the number of
;* retries (max 7) into the I2C_STAT. Bit 7,6 and 5 of the
;* I2C_STAT contain the number of retries. Those bits may
;* not be changed during the I2C routines.
;*
;* INPUT:
;* Before calling I2C_INIT the main program must take care
;* that the correct parameters are available in
;* OWN_SLV_ADDR, SLV_BUF_PTR and _I2C_INIT_BYTE+2, this
;* is done automatically when using C, PL/M or the pre-
;* defined assembler macro (available in I2C_MAC.DEF)
;*
;* OUTPUT:
;* initialized I2C and retry number in I2C_STAT 7..5
;*
;EMP=====
_I2C_INIT:
    MOV     I2CFG,#I2C_ENABLE      ;CLR TIRUN, CLR MASTRO
    SETB   ETI
    SETB   EI2
    SETB   EA                      ;enable interrupts
    ANL    OWN_SLV_ADDR,#0FEH      ;save slv addr bit 0=0
    MOV    I2C_STAT,_I2C_INIT_BYTE+2
    ANL    I2C_STAT,#07H          ;I2C_STAT = retries
    MOV    I2CON,#I2C_RELEASE
    RET

;*****
;*  H I S T O R Y
;*****
;*
;* 03-07-91   J.C. Pijnenburg original version
;*
;*****
END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C age Handler)
;*****
;*
;*          SOURCE FILE : I2C_HAND.ASM
;*          PACKAGE     : I2C
;*
;*****
$DEBUG

;*****
;*          I N C L U D E S
;*
$NOLIST
$INCLUDE(REG751.H)
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$LIST

;*****
;*          G L O B A L   R E F E R E N C E S
;*
EXTRN CODE(I2C_STOP)
EXTRN CODE(I2C_TRX_BYTE)
EXTRN CODE(I2C_TRX_ADDR)
EXTRN CODE(I2C_RCV_BYTE)
EXTRN CODE(I2C_TRX_BLOCK)
EXTRN CODE(I2C_RCV_BLOCK)
EXTRN CODE(I2C_STRT_SLVAD)
EXTRN CODE(I2C_RSTRT_SLVAD)

;*****
;*          G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*
PUBLIC I2C_MESS_HAND

;*****
;*          L O C A L   S Y M B O L   D E C L A R A T I O N S
;*
RWN      BIT      0E0H      ;bit ACC.0
I2C_PSW  EQU      8

;*****
;*          C O D E   S E G M E N T
;*
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF::I2C::I2C_HAND.ASM:I2C_MESS_HAND=====
;*
;* FUNCTION NAME:      I2C_HAND
;* PACKAGE:           I2C
;* DESCRIPTION:
;*   Transmit an I2C age, includes error handling
;*
;* INPUT: age control byte I2C_CTRL (bit addressable)
;*         age control block I2C_MCB, containing:
;*         I2C_ADDR1    (i.e. slave address)
;*         BUF_LEN1     (i.e. number of bytes to trx.)
;*         BUF_PTR1     (i.e. transmit buffer)
;*         I2C_ADDR2    (i.e. sub address)
;*         BUF_LEN2     (i.e. length of second data blk)
;*         BUF_PTR2     (i.e. second transmit buffer)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
I2C_MESS_HAND:
    PUSH    PSW
    MOV     PSW,#I2C_PSW      ;sel RB1
    ANL    I2C_STAT,#07H     ;clr all but retry bits
    MOV    MESS_RETRY_CNT,I2C_STAT
    INC    MESS_RETRY_CNT     ;load retry counter

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

RETRY:
    ANL    I2C_STAT,#07H      ;clr all but retry bits
    MOV    A,I2C_ADDR_1      ;load SLV_ADDR
    CLR    RWN               ;if (subaddress)
    JB    ADDR2_SUB,STRT     ; RWN = 0
    MOV    C,RWN_BLK1        ;else
    MOV    RWN,C             ; RWN = RWN_BLK1

STRT:
    ACALL  I2C_STRT_SLVAD    ;send START+SLV_ADDR+RWN
    JNB   I2C_ERR,CONTINUE  ;branch offset to large
    AJMP  EXIT

CONTINUE:
    JB    TEST_DEVICE,M_STOP
    MOV    BUF_PTR,BUF_PTR_1 ;load pointer block1
    MOV    BUF_LEN,BUF_LEN_1 ;load length block1
    JNB   ADDR2_SUB,BLOCK    ;if (addr2_sub)
    MOV    A,I2C_ADDR_2      ; load sub address
    ACALL  I2C_TRX_BYTE      ; trx_byte(sub address)
    JB    I2C_ERR,EXIT       ; if (error) exit ();
    JNB   REP_STRT_BLK1,BLOCK ; if (rep. start blk1)
    MOV    A,I2C_ADDR_1      ; load slave address
    SETB  RWN               ; read
    ACALL  I2C_RSTRT_SLVAD   ; send RSTART+SLV_ADDR
    JB    I2C_ERR,EXIT       ; if (error) exit();

BLOCK:
    JNB   RWN_BLK1,TRX_1     ;if ((rwn_blk1) == read)
    ACALL  I2C_RCV_BLOCK     ; rcv_block(&data1,cnt1)
    SJMP  END_BLOCK1

TRX_1:
    ACALL  I2C_TRX_BLOCK     ; trx_block(&data1,cnt1)

ENC_BLOCK1:
    JB    I2C_ERR,EXIT       ;if (error) exit();
    JNB   BLOCK2,M_STOP     ;if (2nd block of data)
    MOV    BUF_PTR,BUF_PTR_2 ;{
    MOV    BUF_LEN,BUF_LEN_2 ;
    JNB   ADDR2,DATA2       ; if (addr2)
    JNB   REP_STRT_BLK2,DATA2 ; if (rep. start blk2)
    MOV    A,I2C_ADDR_2      ; ( set address2
    JNB   ADDR2_SUB,SET_RWN  ; if(addr2_sub)
    MOV    A,I2C_ADDR_1      ; set address1

SET_RWN:
    MOV    C,RWN_BLK2        ; /* same slave */
    MOV    RWN,C             ; modify RWN_BLK1
    ACALL  I2C_RSTRT_SLVAD   ; snd RSTART+SLV_ADDR
    JB    I2C_ERR,EXIT       ; if (error) exit();

DATA2:
    JNB   RWN_BLK2,TRX_2     ; if ((rwn_blk2)==read)
    ACALL  I2C_RCV_BLOCK     ; rcv_block(&data1,c1)
    SJMP  BLOCK_ERR

TRX_2:
    ACALL  I2C_TRX_BLOCK     ; else
    ACALL  I2C_TRX_BLOCK     ; trx_block(&data1,c1)

BLOCK_ERR:
    JB    I2C_ERR,EXIT       ; if (error exit());

M_STOP:
    ACALL  I2C_STOP          ;}
    JB    I2C_ERR,EXIT       ; if (error exit());
    AJMP  RESTORE_CONTEXT

```

```

; *MPF:::I2C::I2C_HAND.ASM:EXIT=====
; *
; * FUNCTION NAME:      EXIT
; * PACKAGE:           I2C
; * DESCRIPTION:
; *   Exit an I2C message. This routine is only entered if an
; *   I2C error has occurred. If more retries must be made,
; *   the message is started again. If no retry must be made,
; *   the message handler is left after setting the carry.
; *   Carry is 1 indicates that an error has occurred (return
; *   value for C and PL/M calls). If the routine is entered
; *   at the RESTORE_CONTEXT label, no error has occurred
; *
; * OUTPUT: I2C_ERROR byte (bit addressable)
; *
; *EMP=====
EXIT:
    JNB   TIME_ERR,TO_RETRY
    JNB   BUS_RECOVERED,RESTORE_CONTEXT

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```
TO_RETRY:
    MOV     I2CON,#I2C_RELEASE
    DJNZ   MESS_RETRY_CNT,RETRY ; if (no more retries)

RESTORE_CONTEXT:
    MOV     I2CFG,#I2C_ENABLE ; SLVEN=1,MSTRQ=0,TIRN=0
    MOV     A,I2C_STAT
    POP     PSW                ; restore PSW
    MOV     C,I2C_ERR
END_MESSAGE:
    SETB   EI2                ;label for debugging
    RET    ; enable I2C interrupt
          ; with XRAY

;*=====*
;*  H I S T O R Y  *
;*=====*
;*                *
;* 12-06-91   J.C. Pijenburg  original version  *
;*                *
;*=====*
```

END

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Basic Functions)
;*****
;*
;*          SOURCE FILE : I2C_BASI.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*   I N C L U D E S
;*****
$NOLIST
$INCLUDE(REG751.H)
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$LIST

;*****
;*   G L O B A L   R E F E R E N C E S
;*****
EXTRN CODE(I2C_SLV_TRX)
EXTRN CODE(I2C_SLV_RCV)
EXTRN CODE(ADDR_RECOG)

;*****
;*   G L O B A L   F U N C T I O N   D E F I N I T I O N S *
;*****
PUBLIC I2C_STRT_SLVAD
PUBLIC I2C_RSTRT_SLVAD
PUBLIC I2C_STOP
PUBLIC I2C_TRX_BYTE
PUBLIC I2C_TRX_ADDR
PUBLIC I2C_RCV_BYTE
PUBLIC I2C_RCV_ADDR
PUBLIC I2C_TRX_BLOCK
PUBLIC I2C_RCV_BLOCK
PUBLIC ADDR_COMPARE

;*****
;*   I N T E R R U P T   C O D E   S E G M :   T I M E R 1 *
;*****
CSEG AT 01BH

;MPF:::I2C::I2C_INIT.ASM:I2C_TIME_OUT=====
;*
;* FUNCTION NAME:      I2C_TIME_OUT
;* PACKAGE:           I2C
;* DESCRIPTION:
;* I2C time out routine, clear timer interrupt, set the
;* TIME_ERR bit. If the 8xC751 was I2C bus master while the
;* interrupt occurred (RECOVER = 1), and attempt to recover
;* the bus is made.
;* To recover, SDA and SCL are set, if SCL remains low, the
;* I2C bus cannot be recovered and the routine is left.
;* If SCL is HIGH but SDA is low, 9 additional clocks are
;* generated. If SDA becomes HIGH, a STOP is made
;* If the 8xC751 was not I2C bus master (RECOVER = 0), the
;* bus is released.
;*
;*EMP=====
SETB   CLRTI
SETB   TIME_ERR
CLR    TIRUN
AJMP   TI_INT

;*****
;*   C O D E   S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

```


I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

;*MPP::I2C::I2C_BASI=====
;*
;* SOURCE FILE:      I2C_BASI.ASM
;* PACKAGE:         I2C
;* DESCRIPTION:
;* Generate a start condition on the I2C bus, Set the
;* MASTRQ bit. If 8x751 has not become master on ATN,
;* switch to receive mode and check if the own slave
;* address is received.
;*
;* INPUT: none
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;* OUTPUT CONDITION: SCL is stretched
;*
;*EMP=====
I2C_STRT_SLVAD:
    SETB    TIRUN
    JB     STR,IS_MASTER      ;already started
    CLR    EI2                ;disable I2C interrupt
    MOV    I2CFG,#I2C_START_CTRL
    ACALL  WAIT_ATN
IS_MASTER:
    JB     MASTER,I2C_TRX_ADDR
    MOV    I2CCON,#C_STRT
    ACALL  I2C_RCV_ADDR
    JB     I2C_ERR,END_I2C_START
    ACALL  ADDR_COMPARE
START_ERR:
    SETB   I2C_ERR
END_I2C_START:
    RET

;*MPP::I2C::I2C_BASI.ASM:I2C_REP_STRT=====
;*
;* FUNCTION NAME:    I2C_REP_START
;* PACKAGE:         I2C
;* DESCRIPTION:
;* Generate a repeated start condition on the I2C bus
;* The repeated start is generated by setting the XSTR
;* bit. If STR is not set (by hardware), the I2C bus is
;* released, no check for own slave address is done after a
;* repeated start.
;*
;* INPUT: none
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;* OUTPUT CONDITION: SCL is stretched
;*
;*EMP=====
I2C_RSTRT_SLVAD:
    MOV    I2CON,#S_RSTR
    ACALL  WAIT_ATN          ;wait for rising SCL
    JNB   DRDY,I2C_BASIC_ERR
    MOV    I2CON,#C_DRDY
    ACALL  WAIT_ATN          ;wait for rep start
    JNB   STR,I2C_BASIC_ERR
    SJMP  I2C_TRX_ADDR

;*MPP::I2C::I2C_BASI.ASM:I2C_STOP=====
;*
;* FUNCTION NAME:    I2C_STOP
;* PACKAGE:         I2C
;* DESCRIPTION:
;* Generate a stop condition on the I2C bus
;* The STOP condition is generated by setting the XSTP bit.*
;* If no error occurs, this function is left with I2C bus
;* released and TI stopped. In case of an error the bus is
;* released in the message handler.
;*
;* INPUT: none
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;*
;*EMP=====
I2C_STOP:
    CLR    MASTRQ
    MOV    I2CCON,#S_STP
    ACALL  WAIT_ATN          ;wait for rising SCL
    JNB   DRDY,I2C_BASIC_ERR
    MOV    I2CON,#C_DRDY
    ACALL  WAIT_ATN          ;wait for stop

```

I2C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

MOV     I2CON,#I2C_RELEASE
CLR     TIRUN
RET

;*MPF::I2C::I2C_BASI.ASM:I2C_TRX_ADDR=====*
;*
;* FUNCTION NAME:      I2C_TRX_ADDR
;* PACKAGE:           I2C
;* DESCRIPTION:
;* This function calls I2C_TRX_BYTE to transmit the
;* slave address, if an arbitration is lost before the last
;* bit is transmitted, the function receives the remaining
;* bits (receive mode), and checks whether the own slave
;* address has been received (call ADDR_CMP).
;*
;* INPUT byte to transmit in ACC
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;* OUTPUT CONDITION: SCL is stretched
;*
;*EMP=====
I2C_TRX_ADDR:
ACALL  I2C_TRX_BYTE
JNB    I2C_ERR,END_TRX_ADDR
DJNZ  BIT_CNT,CONTINUE
AJMP  END_TRX_ADDR
CONTINUE:
JNB    ARL,END_TRX_ADDR
JB     STR,END_TRX_ADDR      ;parasitaire START
JB     STR,END_TRX_ADDR      ;parasitaire STOP
CLR    I2C_ERR              ;clr err for slv func
INC    BIT_CNT              ;correct BIT_CNT
CLR    OEOH                 ;RDAT = 0 to ACC.0
RCV_NEXT_BIT:
MOV    I2CON,#C_XMTA+C_DRDY+C_ARL ;rcv mode
ACALL  WAIT_ATN
JNB    DRDY,RESTORE_ERR
MOV    C,RDAT
RLC    A
DJNZ  BIT_CNT,RCV_NEXT_BIT
ACALL  ADDR_COMPARE
RESTORE_ERR:
SETB   I2C_ERR              ;set err for ret main
END_TRX_ADDR:
RET

;*MPF::I2C::I2C_BASI.ASM:I2C_ADDR_COMPARE=====*
;*
;* FUNCTION NAME:      I2C_ADDR_COMPARE
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Compares the contents of the accumulator (received
;* address) with the OWN_SLV_ADDR. If equal and the RWN
;* bit is 0 (master transmit, slave receive) I2C_CLV_RCV is
;* called. If equal and the RWN bit is 1 (master receive,
;* slave transmit) I2C_SLV_TRX is called. If not equal exit*
;* I2C_ADDR_COMPARE
;*
;* INPUT: received address in ACC
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;* OUTPUT CONDITION: SCL is stretched
;*
;*EMP=====
ADDR_COMPARE:
XRL   A,OWN_SLV_ADDR
JZ    SEND_ACK
CJNE  A,#1,END_ADDR_COMPARE
SEND_ACK:
MOV    I2DAT,#0
ACALL  WAIT_ATN
JNB    DRDY,END_ADDR_COMPARE
JZ     SLAVE_RCV
AJMP  I2C_SLV_TRX
SLAVE_RCV:
AJMP  I2C_SLV_RCV
END_ADDR_COMPARE:
RET

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

; *MPF:::I2C::I2C_BASI.ASM:I2C_BASIC_ERR=====
; *
; * FUNCTION NAME:      I2C_BASIC_ERR
; * PACKAGE:           I2C
; * DESCRIPTION:
; * Set the I2C_ERR bit. The message handler tests this bit
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR = 1
; *
; *EMP=====
I2C_BASIC_ERR:
    SETB    I2C_ERR
    RET

; *MPF:::I2C::I2C_BASI.ASM:I2C_TRX_BYTE=====
; *
; * FUNCTION NAME:      I2C_TRX_BYTE
; * PACKAGE:           I2C
; * DESCRIPTION:
; * Transmit a byte over the I2C bus.
; * NOTE: The STR bit is cleared here instead of in the
; *       I2C_START routine, because there must be valid
; *       data in I2DAT before STR may be cleared (also
; *       releases the SCL line).
; *
; * The I2C_TRX_BYTE function transmits a byte over the I2C
; * bus, after the last bit has been transmitted,
; * the function switches to receive mode to receive the
; * acknowledge bit. If NACK is received, the NO_ACK bit is
; * set. If arbitration is lost or an error occurs during
; * I2C_TRX_BYTE the function is exit with the I2C_ERR bit
; * set.
; * INPUT: byte to transmit in ACC
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP=====
I2C_TRX_BYTE:
    MOV     BIT_CNT,#8
TRX_BIT:
    MOV     I2DAT,A           ;release SCL
    MOV     I2CON,#C_STRT     ;if STR clear STR
                                ;else dummy MOV
    ACALL   WAIT_ATN
    JNB     DRDY,I2C_BASIC_ERR
    RL     A
    DJNZ   BIT_CNT,TRX_BIT
    MOV     I2CON,#C_XMTA+C_DRDY ;receive mode
    ACALL   WAIT_ATN
    JNB     DRDY,I2C_BASIC_ERR
    JNB     RDAT,TRX_BYTE_RDY ;stretch SCL
    SETB   NO_ACK
TRX_BYTE_RDY:
    RET

; *MPF:::I2C::I2C_BASI.ASM:I2C_RCV_BYTE=====
; *
; * FUNCTION NAME:      I2C_RCV_BYTE
; * PACKAGE:           I2C
; * DESCRIPTION:
; * Receive a byte from te I2C bus
; * This is one function which receives a byte into acc.
; * RCV_BYTE first releases the SCL and then receives the 8
; * bits. If RCV_ADDR is called, the first bit is already in
; * the RDAT register, this must first be saved before the
; * SCL is released.
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; *         if (! I2C_ERROR) received byte in ACC.
; *
; *EMP=====
I2C_RCV_BYTE:
    MOV     I2CON,#C_XMTA+C_DRDY ;rel. SCL, rcv mode
I2C_RCV_ADDR:
    MOV     BIT_CNT,#8
    CLR    A                 ; rcv first bit

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

RCV_BIT:
    ACALL    WAIT_ATN
    JNB     DRDY,I2C_BASIC_ERR
    DJNZ    BIF_CNT,NOT_LAST_BIT
    MOV     C,RDAT
    RLC
    RET

NOT_LAST_BIT:
    ORL     A,I2DAT                ; save bit, rel. SCL
    RL
    SJMP    RCV_BIT

;MPP:::I2C::I2C_BASI.ASM:I2C_TRX_BLOCK=====
;*
;* FUNCTION NAME:      I2C_TRX_BLOCK
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Transmit a block of bytes over the I2C bus
;* The I2C_TRX_BLOCK function transmits as much bytes as
;* defined in BUF_LEN (set R2), before the message handler *
;* is called, BUF_LEN_1 or BUF_LEN_2 is copied into BUF_LEN*
;*
;* INPUT: pointer to begin of block data in BUF_PTR (R0)
;*         byte counter in BIF_LEN (R2)
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;*
;*EMP=====
I2C_TRX_BLOCK:
    MOV     A,@BUF_PTR            ;load byte
    ACALL   I2C_TRX_BYTE
    JB     I2C_ERR,END_TRX_BLOCK
    JB     NO_ACK,END_TRX_BLOCK
    INC    BUF_PTR
    DJNZ   BUF_LEN,I2C_TRX_BLOCK
END_TRX_BLOCK:
    RET

;MPP:::I2C::I2C_BASI.ASM:I2C_RCV_BLOCK=====
;*
;* FUNCTION NAME:      I2C_RCV_BLOCK
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Receive a block of bytes from the I2C bus
;* The I2C_RCV_BLOCK function receives as much bytes as
;* defined in BUF_LEN (set R2), before the message handler *
;* is called, BUF_LEN_1 or BUF_LEN_2 is copied into BUF_LEN*
;*
;* INPUT: pointer to begin of receive buffer BUF_PTR (R0)
;*         byte counter in BUF_LEN (R2)
;* OUTPUT: I2C_ERROR (0, no error; 1, error)
;*         if (!I2C_ERROR) received bytes in buffer.
;*
;*EMP=====
ACK_RCV_BYTE:
    MOV     I2DAT,#0              ;send ACK
    ACALL   WAIT_ATN
    JNB     DRDY,I2C_BASIC_ERR

I2C_RCV_BLOCK:
    ACALL   I2C_RCV_BYTE
    JB     I2C_ERR,END_RCV_BLOCK
    MOV     @BUF_PTR,A           ;save byte
    INC    BUF_PTR
    DJNZ   BUF_LEN,ACK_RCV_BYTE
    MOV     I2DAT,#80H          ;send NACK
    ACALL   WAIT_ATN
    JNB     DRDY,I2C_BASIC_ERR
END_RCV_BLOCK:
    RET

;MPP:::I2C::I2C_BASI.ASM:WAIT_ATN=====
;*
;* FUNCTION NAME:      WAIT_ATN
;* PACKAGE:           I2C
;* DESCRIPTION:
;* The WAIT_ATN function waits for the ATN bit to be set.
;* The function is left if the ATN bit is set or if the
;* TIME_ERR bit is set. The TIME_ERR bit indicates that a

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

;* bus timeout has occurred. If the 8xC751 enters this
;* function as a master, the RECOVER bit is set,
;* indicating that in case of a timeout, a bus recover
;* action must be started.
;*
;*EMP=====
WAIT_ATN:
    JNB     MASTER, WAIT
    SETB    RECOVER

WIAT:
    JB     TIME_ERR, END_WAIT
    JNB    ATN, WAIT

END_WAIT:
    CLR     RECOVER
    RET

;*MPF:::I2C::I2C_BASI.ASM:TI_INT=====
;*
;* FUNCTION NAME:      TI_INT
;* PACKAGE:           I2C
;* DESCRIPTION:
;* The TI_INT handles the timeout interrupt. It is
;* entered when a time out occurs (during WAIT_ATN).
;* The function is placed here to be sure that it is
;* linked when placed in a library.
;*
;*EMP=====
TI_INT:
    CLR     092H
    ACALL   SCL_DELAY
    SETB    092H
    JNB     RECOVER, RET_INT
    SETB    SDA
    SETB    SCL
    JNB     SCL, RET_INT
    MOV     BUS_ERR_CLKS, #9           ;SCL = 1

RETRY_LOOP:
    CLR     SCL
    ACALL   SCL_DELAY
    SETB    SCL
    ACALL   SCL_DELAY
    JB     SDA, MAKE_STOP             ;if SDA = 1 make stop
    DJNZ   BUS_ERR_CLKS, RETRY_LOOP
    RETI

MAKE_STOP:
    CLR     SCL
    NOP
    CLR     SDA
    ACALL   SCL_DELAY
    SETB    SCL
    ACALL   SCL_DELAY
    SETB    SDA                       ;make stop condition
    ACALL   SCL_DELAY
    SETB    BUS_RECOVERED

RET_INT:
    RETI

SCL_DELAY:
    ;delay of 9 periods (>= 6 micro sec.)
    NOP    ; ACALL(2) + 5 NOP (4) + RET (2)
    NOP
    NOP
    NOP
    NOP
    RET

;*=====
;* H I S T O R Y
;*=====
;*
;* 03-07-91   J.C. Pijnenburg  original version
;*
;*=====
END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Test_Device command)
;*****
;*
;*          SOURCE FILE : I2C_TDEV.ASM          *
;*          PACKAGE      : I2C                  *
;*
;*****
$DEBUG

;*****
;*  I N C L U D E S                               *
;*****
$NOLIST
$INCLUDE (I2C_DATA.GLO)
$LIST

;*****
;*  G L O B A L   R E F E R E N C E S             *
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S *
;*****
PUBLIC _I2C_TEST_DEVICE

;*****
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S   *
;*****
TEST_DEVICE_MASK EQU 80H
;REP_STRT_BLK1 = 0      (NO)
;RWN_BLK1      = 0      (WRITE)
;ADDR2         = 0      (NO)
;ADDR2_SUB     = 0      (--)
;BLOCK2        = 0      (NO)
;RWN_BLK2      = 0      (--)
;REP_STRT_BLK2 = 0      (--)
;T_DEVICE      = 1      (--)

;*****
;*  C O D E   S E G M E N T                       *
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C::I2C_TDEV.ASM:I2C_TEST_DEVICE=====
;*
;* FUNCTION NAME:      I2C_TEST_DEVICE          *
;* PACKAGE:           I2C                      *
;* DESCRIPTION:        *
;*   Address a slave , if ack received slave was present *
;*
;* PROTOCOL:          *
;*   <S><SLV_ADDR><W><A><P>                    *
;*
;* INPUT:  Message control byte I2C_CTRL (bit addressable) *
;*         Message control block I2C_MCB, containing:      *
;*         I2C_ADDR1      (slave address)                 *
;* OUTPUT: I2C_ERROR byte (bit addressable)              *
;*
;*****
;*****
_I2C_TEST_DEVICE:
MOV     I2C_CTRL,#TEST_DEVICE_MASK
AJMP   I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

$TITLE(I2C_Write command)
;*****
;*
;*          SOURCE FILE : I2C_WRIT.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*          I N C L U D E S
;*****
$NOLIST
$INCLUDE (I2C_DATA.GLO)
$LIST

;*****
;*          G L O B A L   R E F E R E N C E S
;*****
        EXTRN CODE(I2C_MESS_HAND)

;*****
;*          G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
        PUBLIC _I2C_WRITE

;*****
;*          L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
        WRITE_MASK EQU 00H
                ;REP_STRT_BLK1 = 0      (NO)
                ;RWN_BLK1      = 0      (WRITE)
                ;ADDR2         = 0      (NO)
                ;ADDR2_SUB     = 0      (--)
                ;BLOCK2        = 0      (NO)
                ;RWN_BLK2      = 0      (--)
                ;REP_STRT_BLK2 = 0      (--)
                ;TEST_DEVICE   = 0      (--)

;*****
;*          C O D E   S E G M E N T
;*****
        I2C_DRIVER SEGMENT CODE
        RSEG I2C_DRIVER

;*****
;*MPF:::I2C::I2C_WRIT.ASM:I2C_WRITE=====
;*
;* FUNCTION NAME:      I2C_WRITE
;* PACKAGE:           I2C
;* DESCRIPTION:
;*   Write n bytes to a slave device.
;*
;* PROTOCOL:
;*   <S><SLV_ADDR><W><A><D0><A><D1><A>...<Dn-1><A><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;*        Message control block I2C_MCB, containing:
;*        I2C_ADDR1      (slave address)
;*        BUF_LEN1       (number of bytes (n) to trx.)
;*        BUF_PTR1       (ptr to transmit buffer)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
;*****

_I2C_WRITE:
        MOV     I2C_CTRL,#WRITE_MASK
        AJMP   I2C_MESS_HAND

        END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Sub command)
;*****
;*
;*          SOURCE FILE : I2C_WSUB.ASM          *
;*          PACKAGE      : I2C                  *
;*
;*****
$DEBUG

;*****
;*   I N C L U D E S   *
;*****
$NOLIST
$INCLUDE (I2C_DATA.GLO)
$LIST

;*****
;*   G L O B A L   R E F E R E N C E S   *
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*   G L O B A L   F U N C T I O N   D E F I N I T I O N S   *
;*****
PUBLIC _I2C_WRITE_SUB

;*****
;*   L O C A L   S Y M B O L   D E C L A R A T I O N S   *
;*****
WRITE_SUB_MASK EQU 00H
;REP_STRT_BLK1 = 0      (NO)
;RWN_BLK1      = 0      (WRITE)
;ADDR2         = 1      (YES)
;ADDR2_SUB     = 1      (YES)
;BLOCK2        = 0      (NO)
;RWN_BLK2      = 0      (--)
;REP_STRT_BLK2 = 0      (--)
;TEST_DEVICE   = 0      (--)

;*****
;*   C O D E   S E G M E N T   *
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;*MPF:::I2C::I2C_WSUB.ASM:I2C_WRITE_SUB=====
;*
;* FUNCTION NAME:      I2C_WRITE_SUB          *
;* PACKAGE:           I2C                    *
;* DESCRIPTION:       *
;*   Write a block of data (a length n) preceded by a *
;*   sub address to a slave device.             *
;*
;* PROTOCOL:          *
;*   <S><SLV_ADDR><W><A><SUB_ADDR><A><Da0><A><Da1><A>..<A> *
;*                                     <Dan-1><A><P> *
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable) *
;*         Message control block I2C_MCB, containing: *
;*         I2C_ADDR1      (slave address) *
;*         BUF_LEN1       (number of bytes in block ) *
;*         BUF_PTR1       (ptr to block ) *
;*         I2C_ADDR2     (sub address) *
;*
;* OUTPUT: I2C_ERROR byte (bit addressable) *
;*
;*EMP=====

_I2C_WRITE_SUB:
MOV     I2C_CTRL,#WRITE_SUB_MASK
AJMP   I2C_MESS_HAND

END

```


I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Sub_SWinc command)
;*****
;*
;*          SOURCE FILE : I2C_WSWI.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*          I N C L U D E S
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$INCLUDE(REG751.H)
$LIST

;*****
;*          G L O B A L   R E F E R E N C E S
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*          G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
PUBLIC _I2C_WRITE_SUB_SWINC
PUBLIC _I2C_WRITE_MEMORY

;*****
;*          L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
WRITE_SUB_SWINC_MASK EQU 00CH
WRITE_MEMORY_MASK   EQU 02CH
;REP_STRT_BLK1      = 0      (NO)
;RWN_BLK1           = 0      (WRITE)
;ADDR2              = 1      (YES)
;ADDR2_SUB          = 1      (YES)
;BLOCK2             = 0      (NO)
;RWN_BLK2           = 0 or 1 (no blk2,
                        used for delay/no delay)
;REP_STRT_BLK2      = 0      (--)
;TEST_DEVICE        = 0      (--)

;*****
;*          C O D E   S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C:::I2C_WSWI.ASM:I2C_WRITE_SUB_SWINC=====
;*
;* FUNCTION NAME:      I2C_WRITE_SUB_SWINC
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Transmit an I2C message, the message is split into
;* sub messages. Each sub message transmits one byte.
;* If the slave is an EEPROM a delay is generated after
;* each sub message. The RWN_BLK2 is not used in the
;* message handler (no block 2) and is therefore free to
;* distinguish between write to EEPROM (1=delay) and other
;* (0= no delay)
;*
;* PROTOCOL:
;* <S><SLV_ADDR><W><A><SUB_ADDR><A><D0><A><P>
;* if (RWN_BLK2) delay 40 ms
;* <S><SLV_ADDR><W><A><SUB_ADDR+1><A><D1><A><P>
;* if (RWN_BLK2) delay 40 ms
;* -- -- -- -->
;* if (RWN_BLK2) delay 40 ms
;* <S><SLV_ADDR><W><A><SUB_ADDR+n-1><A><Dn-1><P>
;* if (RWN_BLK2) delay 40 ms
;*

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

;* INPUT: Message control byte I2C_CTRL (bit addressable) *
;* Message control block I2C_MCB, containing: *
;* I2C_ADDR1 (slave address) *
;* BUF_LEN1 (number of bytes (mess) to trx.) *
;* BUF_PTR1 (ptr to transmit buffer) *
;* I2C_ADDR2 (sub address) *
;* *
;* OUTPUT: I2C_ERROR byte (bit addressable) *
;* *
;*EMP=====

_I2C_WRITE_MEMORY:
    MOV     I2C_CTRL,#WRITE_MEMORY_MASK
    SJMP   SET_MESS_CNT
_I2C_WRITE_SUB_SWINC:
    MOV     I2C_CTRL,#WRITE_SUB_SWINC_MASK

SET_MESS_CNT:
    MOV     MEM_MESS_LEN,I2C_MCB+1
    MOV     I2C_MCB+1,#1 ;set BUF_LEN_1 = 1
SUB_MESS:
    ACALL  I2C_MESS_HAND
    JB     I2C_ERR,END_WSWI
    INC    I2C_MCB+2 ;inc. BUFF_PTR_1
    INC    I2C_MCB+3 ;inc. SUB_ADDR
    JNB   RWN_BLK2,NEXT
    MOV    MEM_DELAY_H,#EEPROM_PROG_DELAY
    MOV    MEM_DELAY_L,#00 ; 40 mS delay at 16MHz
PROGRAM_DELAY:
    DJNZ  MEM_DELAY_L,$
    DJNZ  MEM_DELAY_H,PROGRAM_DELAY
NEXT:
    DJNZ  MEM_MESS_LEN,SUB-MESS
END_WSWI:
    RET
END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Sub_Write command)
;*****
;*
;*          SOURCE FILE : I2C_WSUW.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*          I N C L U D E S
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;*          G L O B A L   R E F E R E N C E S
;*****
          EXTRN CODE(I2C_MESS_HAND)

;*****
;*          G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
          PUBLIC _I2C_WRITE_SUB_WRITE

;*****
;*          L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
          WRITE_SUB_WRITE_MASK EQU 1CH
                                ;REP_STRT_BLK1 = 0      (NO)
                                ;RWN_BLK1      = 0      (WRITE)
                                ;ADDR2        = 1      (YES)
                                ;ADDR2_SUB    = 1      (YES)
                                ;BLOCK2       = 1      (YES)
                                ;RWN_BLK2     = 0      (WRITE)
                                ;REP_STRT_BLK2 = 0      (NO)
                                ;TEST_DEVICE  = 0      (--)

;*****
;*          C O D E   S E G M E N T
;*****
          I2C_DRIVER SEGMENT CODE
          RSEG I2C_DRIVER

;MPF:::I2C::I2C_WSUW.ASM:I2C_WRITE_SUB_WRITE=====
;*
;* FUNCTION NAME:      I2C_WRITE_SUB_WRITE
;* PACKAGE:           I2C
;* DESCRIPTION:
;*   Write 2 blocks of data (a and b, length n and m)
;*   preceded by a sub address into a single slave device
;*
;* PROTOCOL:
;*   <S><SLV_ADDR><W><A><SUB_ADDR><A><Da0><A>...<A><Dan-1><A>
;*   <Db0><A>...<Dbm-1><A><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;*         Message control block I2C_MCB, containing:
;*         I2C_ADDR1 (slave address)
;*         BUF_LEN1 (number of bytes in block a)
;*         BUF_PTR1 (ptr to block a)
;*         I2C_ADDR2 (sub address)
;*         BUF_LEN1 (number of bytes in block b)
;*         BUF_PTR1 (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
;
_I2C_WRITE_SUB_WRITE:
  MOV     I2C_CTRL,#WRITE_SUB_WRITE_MASK
  AJMP   I2C_MESS_HAND

  END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Sub_Read command)
;=====
;*
;*          SOURCE FILE : I2C_WSUR.ASM
;*          PACKAGE     : I2C
;*
;=====
$DEBUG

;=====
;*
;*  I N C L U D E S
;=====
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;=====
;*
;*  G L O B A L   R E F E R E N C E S
;=====
EXTRN CODE(I2C_MESS_HAND)

;=====
;*
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;=====
PUBLIC _I2C_WRITE_SUB_READ

;=====
;*
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;=====
WRITE_SUB_READ_MASK EQU 7CH
;REP_STRT_BLK1 = 0 (NO)
;RWN_BLK1 = 0 (WRITE)
;ADDR2 = 1 (YES)
;ADDR2_SUB = 1 (YES)
;BLOCK2 = 1 (YES)
;RWN_BLK2 = 1 (READ)
;REP_STRT_BLK2 = 1 (YES)
;TEST_DEVICE = 0 (--

;=====
;*
;*  C O D E   S E G M E N T
;=====
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPP::I2C::I2C_WSUR.ASM:I2C_WRITE_SUB_READ=====
;*
;*
;* FUNCTION NAME: I2C_WRITE_SUB_READ
;* PACKAGE: I2C
;* DESCRIPTION:
;* Write a block of data (a length n) preceded by a sub
;* address, generate repeated start and read a second
;* block of data (b length m) from the slave device
;*
;*
;* PROTOCOL:
;* <S><SLV_ADDR><W><A><SUB_ADDR><A><Da0><A>..<A><DaN-1><A>
;* <S><SLV_ADDR><R><A><Db0><A>..<A><Dbm-1><N><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address)
;* BUF_LEN1 (number of bytes in block a)
;* BUF_PTR1 (ptr to block a)
;* I2C_ADDR2 (sub address)
;* BUF_LEN1 (number of bytes in block b)
;* BUF_PTR1 (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====

_I2C_WRITE_SUB_READ:
MOV I2C_CTRL,#WRITE_SUB_READ_MASK
AJMP I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Com_Write command)
;=====
;*
;*          SOURCE FILE : I2C_WCOW.ASM
;*          PACKAGE     : I2C
;*
;=====
$DEBUG

;=====
;*  I N C L U D E S
;=====
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;=====
;*  G L O B A L   R E F E R E N C E S
;=====
EXTRN CODE(I2C_MESS_HAND)

;=====
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;=====
PUBLIC _I2C_WRITE_COM_WRITE

;=====
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;=====
WRITE_COM_WRITE_MASK EQU 10H
;REP_STRT_BLK1 = 0 (NO)
;RWN_BLK1 = 0 (WRITE)
;ADDR2 = 0 (NO)
;ADDR2_SUB = 0 (--)
;BLOCK2 = 1 (YES)
;RWN_BLK2 = 0 (WRITE)
;REP_STRT_BLK2 = 0 (--)
;TEST_DEVICE = 0 (NO)

;=====
;*  C O D E   S E G M E N T
;=====
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF::I2C::I2C_WCOW.ASM:I2C_WRITE_COM_WRITE=====
;*
;* FUNCTION NAME: I2C_WRITE_COM_WRITE
;* PACKAGE: I2C
;* DESCRIPTION:
;* Write a 2 blocks of data (a,b length n,m) in a single
;* message to a slave device
;* another slave device.
;*
;* PROTOCOL:
;* <S><SLV_ADDR1><W><A><Da0><A><Da1><A>...<A><Dan-1><A>
;* <Db0><A><Db1><A>...<A><Dbm-1><A><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address first device)
;* BUF_LEN1 (number of bytes in block a)
;* BUF_PTR1 (ptr to block a)
;* BUF_LEN1 (number of bytes in block b)
;* BUF_PTR1 (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
_I2C_WRITE_COM_WRITE:
MOV I2C_MCB+5,I2C_MCB+4
MOV I2C_MCB+4,I2C_MCB+3
MOV I2C_CTRL,#WRITE_COM_WRITE_MASK
AJMP I2C_MESS_HAND

END

```

I2C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Rep_Write command)
;*****
;*
;*          SOURCE FILE : I2C_WREW.ASM
;*          PACKAGE     : I2C
;*
;*****
$DEBUG

;*****
;*  I N C L U D E S
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;*  G L O B A L   R E F E R E N C E S
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
PUBLIC _I2C_WRITE_REP_WRITE

;*****
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
WRITE_REP_WRITE_MASK EQU 54H
;REP_STRT_BLK1 = 0 (NO)
;RWN_BLK1      = 0 (WRITE)
;ADDR2        = 1 (YES)
;ADDR2_SUB    = 0 (NO)
;BLOCK2       = 1 (YES)
;RWN_BLK2     = 0 (WRITE)
;REP_STRT_BLK2 = 1 (YES)
;TEST_DEVICE  = 0 (--)

;*****
;*  C O D E   S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C::I2C_WREW.ASM:I2C_WRITE_REP_WRITE=====
;*
;* FUNCTION NAME:      I2C_WRITE_REP_WRITE
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Write a block of data (a length n) to a slave device,
;* sent repeated start and write a block (b length m) to
;* another slave device.
;*
;* PROTOCOL:
;* <S><SLV_ADDR1><W><A><Da0><A><Da1><A>..<A><Dan-1><A>
;* <S><SLV_ADDR2><W><A><Db0><A><Db1><A>..<A><Dbm-1><N><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address first device)
;* BUF_LEN1 (number of bytes in block a)
;* BUF_PTR1 (ptr to block a)
;* I2C_ADDR2 (slave address second device)
;* BUF_LEN1 (number of bytes in block b)
;* BUF_PTR1 (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;EMP=====

_I2C_WRITE_REP_WRITE:
MOV     I2C_CTRL,#WRITE_REP_WRITE_MASK
AJMP   I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Write_Rep_Read command)
;*****
;*
;*          SOURCE FILE : I2C_WRER.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*  I N C L U D E S
;*
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;*  G L O B A L   R E F E R E N C E S
;*
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*
;*****
PUBLIC _I2C_WRITE_REP_READ

;*****
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;*
;*****
WRITE_REP_READ_MASK EQU 74H
;REP_STRT_BLK1 = 0 (NO)
;RWN_BLK1 = 0 (WRITE)
;ADDR2 = 1 (YES)
;ADDR2_SUB = 0 (NO)
;BLOCK2 = 1 (YES)
;RWN_BLK2 = 1 (READ)
;REP_STRT_BLK2 = 1 (YES)
;TEST_DEVICE = 0 (--)

;*****
;*  C O D E   S E G M E N T
;*
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C::I2C_WRER.ASM:I2C_WRITE_REP_READ=====
;*
;* FUNCTION NAME: I2C_WRITE_REP_READ
;* PACKAGE: I2C
;* DESCRIPTION:
;* Write a block of data (a length n) to a slave device,
;* sent repeated start and read a block (b length m) from
;* another slave device.
;*
;*
;* PROTOCOL:
;* <S><SLV_ADDR1><W><A><Da0><A><Da1><A>...<A><Dan-1><A>
;* <S><SLV_ADDR2><R><A><Db0><A><Db1><A>...<A><Dbm-1><N><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address first device)
;* BUF_LEN1 (number of bytes in block a)
;* BUF_PTR1 (ptr to block a)
;* I2C_ADDR2 (slave address second device)
;* BUF_LEN1 (number of bytes in block b)
;* BUF_PTR1 (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
_I2C_WRITE_REP_READ:
MOV I2C_CTRL,#WRITE_REP_READ_MASK
AJMP I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Read command)
;*****
;*
;*          SOURCE FILE : I2C_READ.ASM
;*          PACKAGE      : I2C
;*
;*****
SDEBUG

;*****
;*  I N C L U D E S
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;*  G L O B A L   R E F E R E N C E S
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
PUBLIC _I2C_READ
PUBLIC _I2C_READ_STATUS

;*****
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
READ_MASK EQU 02H
;REP_STRT_BLK1 = 0 (NO)
;RWN_BLK1 = 1 (READ)
;ADDR2 = 0 (NO)
;ADDR2_SUB = 0 (--)
;BLOCK2 = 0 (NO)
;RWN_BLK2 = 0 (--)
;REP_STRT_BLK2 = 0 (--)
;TEST_DEVICE = 0 (--)

;*****
;*  C O D E   S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;*MPF:::I2C::I2C_READ.ASM:I2C_READ=====
;*
;* FUNCTION NAME: I2C_READ
;* PACKAGE: I2C
;* DESCRIPTION:
;* Read a block of data from a slave device (READ) or read
;* a single byte from a slave device (READ_STATUS)
;*
;* PROTOCOL:
;* <S><SLV_ADDR><R><A><D0><A><D1><A>..<A><Dn-1><N><P>
;* or
;* <S><SLV_ADDR><R><A><STATUS><N><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address)
;* BUF_LEN1 (number of bytes in block)
;* BUF_PTR1 (ptr to store status)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
_I2C_READ_STATUS:
MOV I2C_MCB+2,I2C_MCB+1
MOV I2C_MCB+1,#1 ;buffer length = 1
_i2c_read:
MOV I2C_CTRL,#READ_MASK
AJMP I2C_MESS_HAND

END

```


I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Read_Sub command)
;*****
;*
;*          SOURCE FILE : I2C_RSUB.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*  I N C L U D E S
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;*  G L O B A L   R E F E R E N C E S
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
PUBLIC _I2C_READ_SUB

;*****
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
READ_SUB_MASK EQU 0FH
                ;REP_STRT_BLK1 = 1      (YES)
                ;RWN_BLK1     = 1      (READ)
                ;ADDR2        = 1      (YES)
                ;ADDR2_SUB    = 1      (YES)
                ;BLOCK2       = 0      (NO)
                ;RWN_BLK2     = 0      (--)
                ;REP_STRT_BLK2 = 0      (--)
                ;TEST_DEVICE   = 0      (--)

;*****
;*  C O D E   S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;*****
;*MPF:::I2C::I2C_RSUB.ASM:I2C_READ_SUB=====
;*
;* FUNCTION NAME:      I2C_READ_SUB
;* PACKAGE:           I2C
;* DESCRIPTION:
;* Read a block of data (a length n) preceded by a
;* sub address from a slave device.
;*
;*
;* PROTOCOL:
;* <S><SLV_ADDR><W><A><SUB_ADDR><A><S><SLV_ADDR><R><A>
;* <Da0><A><Da1><A>...<A><Dan-1><A><P>
;*
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address)
;* BUF_LEN1 (number of bytes in block )
;* BUF_PTR1 (ptr to block )
;* I2C_ADDR2 (sub address)
;*
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
_I2C_READ_SUB:
MOV     I2C_CTRL,#READ_SUB_MASK
AJMP   I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

STITLE(I2C_Read_Rep_Write command)
;*****
;*
;* SOURCE FILE : I2C_RREW.ASM
;* PACKAGE : I2C
;*
;*****
$DEBUG

;*****
;* I N C L U D E S
;*****
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;* G L O B A L R E F E R E N C E S
;*****
EXTRN CODE(I2C_MESS_HAND)

;*****
;* G L O B A L F U N C T I O N D E F I N I T I O N S *
;*****
PUBLIC _I2C_READ_REP_WRITE

;*****
;* L O C A L S Y M B O L D E C L A R A T I O N S *
;*****
READ_REP_WRITE_MASK EQU 56H
;REP_STRT_BLK1 = 0 (NO)
;RWN_BLK1 = 1 (READ)
;ADDR2 = 1 (YES)
;ADDR2_SUB = 0 (NO)
;BLOCK2 = 1 (YES)
;RWN_BLK2 = 0 (WRITE)
;REP_STRT_BLK2 = 1 (YES)
;TEST_DEVICE = 0 (--)

;*****
;* C O D E S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C::I2C_RREW.ASM:I2C_READ_REP_WRITE=====
;*
;* FUNCTION NAME: I2C_READ_REP_WRITE
;* PACKAGE: I2C
;* DESCRIPTION:
;* Read a block of data (a length n) from a slave device,
;* sent repeated start and write a block (b length m) to
;* another slave device.
;*
;* PROTOCOL:
;* <S><SLV_ADDR1><R><A><Da0><A><Da1><A>...<A><Dan-1><N>
;* <S><SLV_ADDR2><W><A><Db0><A><Db1><A>...<A><Dbm-1><A><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;* Message control block I2C_MCB, containing:
;* I2C_ADDR1 (slave address first device)
;* BUF_LEN1 (number of bytes in block a)
;* BUF_PTR1 (ptr to block a)
;* I2C_ADDR2 (slave address second device)
;* BUF_LEN1 (number of bytes in block b)
;* BUF_PTR1 (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;EMP=====

_I2C_READ_REP_WRITE:
MOV I2C_CTRL,#READ_REP_WRITE_MASK
AJMP I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

$TITLE(I2C_Read_Rep_Read command)
;*****
;*
;*          SOURCE FILE : I2C_RRER.ASM
;*          PACKAGE    : I2C
;*
;******
$DEBUG

;*****
;*  I N C L U D E S
;******
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;*****
;*  G L O B A L   R E F E R E N C E S
;******
EXTRN CODE(I2C_MESS_HAND)

;*****
;*  G L O B A L   F U N C T I O N   D E F I N I T I O N S
;******
PUBLIC _I2C_READ_REP_READ

;*****
;*  L O C A L   S Y M B O L   D E C L A R A T I O N S
;******
READ_REP_READ_MASK EQU 076H
;REP_STRT_BLK1    = 0    (NO)
;RWN_BLK1        = 1    (READ)
;ADDR2           = 1    (YES)
;ADDR2_SUB       = 0    (NO)
;BLOCK2         = 1    (YES)
;RWN_BLK2        = 1    (READ)
;REP_STRT_BLK2   = 1    (YES)
;TEST_DEVICE     = 0    (--)

;*****
;*  C O D E   S E G M E N T
;******
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPP::I2C::I2C_RRER.ASM:I2C_READ_REP_READ=====
;*
;* FUNCTION NAME:      I2C_READ_REP_READ
;* PACKAGE:           I2C
;* DESCRIPTION:
;*   Read a block of data (a length n) from a slave device,
;*   sent repeated start and read a block (b length m) from
;*   another slave device.
;*
;* PROTOCOL:
;*   <S><SLV_ADDR1><R><A><Da0><A><Da1><A>..<A><Dan-1><N>
;*   <S><SLV_ADDR2><R><A><Db0><A><Db1><A>..<A><Dbm-1><N><P>
;*
;* INPUT: Message control byte I2C_CTRL (bit addressable)
;*         Message control block I2C_MCB, containing:
;*         I2C_ADDR1    (slave address first device)
;*         BUF_LEN1     (number of bytes in block a)
;*         BUF_PTR1     (ptr to block a)
;*         I2C_ADDR2    (slave address second device)
;*         BUF_LEN1     (number of bytes in block b)
;*         BUF_PTR1     (ptr to block b)
;*
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
_I2C_READ_REP_READ:
MOV     I2C_CTRL, #READ_REP_READ_MASK
AJMP   I2C_MESS_HAND

END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

I²C Slave routines

```

$TITLE(I2C_SLAVE)
;*****
;*
;*          SOURCE FILE : I2C_SLAV.ASM
;*          PACKAGE      : I2C
;*
;*****
$DEBUG

;*****
;*   I N C L U D E S
;*****
$NOLIST
$INCLUDE (I2C_DATA.GLO)
$INCLUDE (REG751.H)
$LIST

;*****
;*   G L O B A L   R E F E R E N C E S
;*****
EXTRN CODE(I2C_TRX_BYTE)
EXTRN CODE(I2C_RCV_BYTE)
EXTRN CODE(I2C_RCV_ADDR)
EXTRN CODE(ADDR_COMPARE)

;*****
;*   G L O B A L   F U N C T I O N   D E F I N I T I O N S
;*****
PUBLIC _I2C_SLV_TRX
PUBLIC _I2C_SLV_RCV
PUBLIC ADDR_RECOG

;*****
;*   L O C A L   S Y M B O L   D E C L A R A T I O N S
;*****
SLV_BUF_PTR_W  SET      R1
BIT_CNT        SET      R3
I2C_RELEASE    EQU      0F4H

;*MPF:::I2C_SLAVE=====
;*
;* FUNCTION NAME:      I2C_SLAVE_ROUTINES
;* DESCRIPTION:        I2C
;* DESCRIPTION:
;* This file contains an example of how to make a slave
;* transmitter and slave receiver function. The slave
;* transmitter functions transmits byte from a buffer,
;* while the slave receiver routine receives bytes into
;* a buffer. The buffer pointer is loaded during the
;* I2C_INIT routine.
;*
;*
;*EMP=====

;*****
;*   I N T E R R U P T   C O D E   S E G M :   I I C
;*****
CSEG AT 023H

PUSH  ACC
PUSH  PSW
MOV   PSW,#8
AJMP  ADDR_RECOG

;*****
;*   C O D E   S E G M E N T
;*****
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

;*MPF:::I2C::I2C_SLAV.ASM:I2C_ADDR_RECOG=====*
;*
;* FUNCTION NAME:      I2C_ADDR_RECOG
;* PACKAGE NAME:      I2C
;* DESCRIPTION:
;* If an I2C interrupt occurs, and the STR bit is set,
;* I2C_ADDR_RECOG receives the incoming slave address.
;* If its own slave address is recognized, the slave
;* receiver or slave transmitter routine (depending on
;* the R/WN bit) is called
;*
;* INPUT:  --
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;=====*
ADDR_RECOG:
MOV     I2CON,#C_STRT
ACALL  I2C_RCV_ADDR
JB     I2C_ERR,EXIT_SLV_AD
ACALL  ADDR_COMPARE
EXIT_SLV_AD:
CLR     I2C_ERR
MOV     I2CON,#I2C_RELEASE
POP     PSW
POP     ACC
RETI

;*MPF:::I2C::I2C_SLAV.ASM:I2C_SLV_TRX=====*
;*
;* FUNCTION NAME:      I2C_SLV_TRX
;* PACKAGE NAME:      I2C
;* DESCRIPTION:
;* After the SLV_ADDR/W is received, the I2C_SLV_TRX
;* transmits a byte from the slave buffer. The pointer
;* to this buffer is loaded during the I2C_INIT function.
;* If an acknowledge is received, the pointer is
;* incremented and the next byte is transmitted. The
;* function is exit on reception of a NACK.
;*
;* Normally the slave routines are entered through an I2C
;* interrupt, but if the 8xC751 loses arbitration during
;* the slave address and it recognizes its own slave
;* address/W, the I2C_SLV_TRX function is entered at XXXX*
;*
;* PROTOCOL: <S><SLV_ADDR><W><D0><A><D1><A>...<A><Dn><N><P>
;*
;* REGISTER USAGE : Register bank 1, is used during the I2C
;* routines, it contains no static data, and is free
;* for the user outside the I2C routines
;*
;* INPUT:  --
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====*
I2C_SLV_TRX:
MOV     SLV_BUF_PTR_W,SLV_BUF_PTR
S_TRX:
MOV     A,@SLV_BUF_PTR_W
ACALL  I2C_TRX_BYTE
JB     I2C_ERR,EXIT_SLV_TRX
JB     NO_ACK,EXIT_SLV_TRX
INC     SLV_BUF_PTR_W
AJMP   S_TRX
EXIT_SLV_TRX:
RET

```

I2C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

;*MPF:::I2C::I2C_SLAV.ASM:I2C_SLV_RCV=====
;*
;* FUNCTION NAME:      I2C_SLV_RCV
;* PACKAGE NAME:      I2C
;* DESCRIPTION:
;*   After the SLV_ADDR/R is received, the I2C_SLV_RCV
;*   receives a byte into the slave buffer. The pointer
;*   to this buffer is loaded during the I2C_INIT function.
;*   After the byte is received, and acknowledge is send,
;*   the pointer is incremented and the next byte is
;*   received. The function is exit on if a start condition
;*   is detected.
;*
;*   Normally the slave routines are entered through and I2C*
;*   interrupt, but if the 8xC751 loses arbitration during *
;*   the slave address and it recognizes its own slave *
;*   address/R, the I2C_SLV_RCV function is entered at xx *
;*
;* PROTOCOL: <S><SLV_ADDR><R><D0><A><D1><A>..<A><Dn><A><P>
;*
;* REGISTER USAGE : Register bank 1, is used during the I2C *
;* routines, it contains no static data, and is free *
;* fr the user outside the I2C routines
;*
;* INPUT:  --
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP=====
I2C-SLV_RCV:
MOV     SLV_BUF_PTR_W,SLV_BUF_PTR
S_RCV:
ACALL  I2C_RCV_BYTE
JB     I2C_ERR,EXIT_SLV_RCV
MOV    @SLV_BUF_PTR_WQ,A      ;save byte
MOV    I2DAT,#0              ;send ACK
JNB    ATN,$
JNB    DRDY,EXIT_SLV_RCV
INC    SLV_BUF_PTR_W
AJMP   S_RCV
EXIT_SLV_RCV:
RET

;=====
;*   H I S T O R Y
;*=====
;*
;* 21-05-91   J.C. Pijnenburg original version
;*
;=====
END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

I2C_MAC.DEF

```

** DEFINE (I2C_INIT(Own_Slv_Addr, Slv_Buf_Addr, Retry))
(
  MOV OWN_SLV_ADDR, %%Own_Slv_Addr
  MOV SLV_BUF_PTR, %%Slv_Buf_Addr
  MOV I2C_MCB,
  ACALL _I2C_INIT
)

** DEFINE (I2C_TEST_DEVICE(Slv_Addr))
(
  MOV I2C_MCB, %%Slv_Addr
  ACALL _I2C_TEST_DEVICE
)

** DEFINE (I2C_WRITE(Slv_Addr, Count, Source_Ptr))
(
  MOV I2C_MCB, %%Slv_Addr
  MOV I2C_MCB+1, %%Count
  MOV I2C_MCB+2, %%Source_Ptr
  ACALL _I2C_WRITE
)

** DEFINE (I2C_WRITE_SUB(Slv_Addr, Count, Source_Ptr, Sub_Addr))
(
  MOV I2C_MCB, %%Slv_Addr
  MOV I2C_MCB+1, %%Count
  MOV I2C_MCB+2, %%Source_Ptr
  MOV I2C_MCB+3, %%Sub_Addr
  ACALL _I2C_WRITE_SUB
)

** DEFINE (I2C_WRITE_SUB_SWINC(Slv_Addr, Count, Source_Ptr, Sub_Addr))
(
  MOV I2C_MCB, %%Slv_Addr
  MOV I2C_MCB+1, %%Count
  MOV I2C_MCB+2, %%Source_Ptr
  MOV I2C_MCB+3, %%Sub_Addr
  ACALL _I2C_WRITE_SUB_SWINC
)

** DEFINE (I2C_WRITE_MEMORY(Slv_Addr, Count, Source_Ptr, Sub_Addr))
(
  MOV I2C_MCB, %%Slv_Addr
  MOV I2C_MCB+1, %%Count
  MOV I2C_MCB+2, %%Source_Ptr
  MOV I2C_MCB+3, %%Sub_Addr
  ACALL _I2C_WRITE_MEMORY
)

** DEFINE (I2C_WRITE_SUB_WRITE(Slv_Addr, Count_1, Source_Ptr_1, Sub_Addr, Count_2, Source_Ptr_2))
(
  MOV I2C_MCB, %%Slv_Addr
  MOV I2C_MCB+1, %%Count_1
  MOV I2C_MCB+2, %%Source_Ptr_1
  MOV I2C_MCB+3, %%Sub_Addr
  MOV I2C_MCB+4, %%Count_2
  MOV I2C_MCB+5, %%Source_Ptr_2
  ACALL _I2C_WRITE_SUB_WRITE
)

** DEFINE (I2C_WRITE_SUB_READ(Slv_Addr, Count_1, Source_Ptr, Sub_Addr, Count_2, Dest_Ptr))
(
  MOV I2C_MCB, %%Slv_Addr
  MOV I2C_MCB+1, %%Count_1
  MOV I2C_MCB+2, %%Source_Ptr
  MOV I2C_MCB+3, %%Sub_Addr
  MOV I2C_MCB+4, %%Count_2
  MOV I2C_MCB+5, %%Dest_Ptr
  ACALL _I2C_WRITE_SUB_READ
)

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

%* DEFINE (I2C_WRITE_COM_WRITE(Slv_Addr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count_1
MOV I2C_MCB+2,%%Source_Ptr_1
MOV I2C_MCB+3,%%Count_2
MOV I2C_MCB+4,%%Source_Ptr_2
ACALL _I2C_WRITE_COM_WRITE
)

%* DEFINE (I2C_WRITE_REP_WRITE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count_1
MOV I2C_MCB+2,%%Source_Ptr_1
MOV I2C_MCB+3,%%Sub_Addr
MOV I2C_MCB+4,%%Count_2
MOV I2C_MCB+5,%%Source_Ptr_2
ACALL _I2C_WRITE_REP_WRITE
)

%* DEFINE (I2C_WRITE_REP_READ(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count_1
MOV I2C_MCB+2,%%Source_Ptr
MOV I2C_MCB+3,%%Sub_Addr
MOV I2C_MCB+4,%%Count_2
MOV I2C_MCB+5,%%Dest_Ptr
ACALL _I2C_WRITE_REP_READ
)

%* DEFINE (I2C_READ(Slv_Addr,Count,Dest_Ptr))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count
MOV I2C_MCB+2,%%Dest_Ptr
ACALL _I2C_READ
)

%* DEFINE (I2C_READ_STATUS(Slv_Addr,Dest_Ptr))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Dest_Ptr
ACALL _I2C_READ_STATUS
)

%* DEFINE (I2C_READ_SUB(Slv_Addr,Count,Dest_Ptr,Sub_Addr))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count
MOV I2C_MCB+2,%%Dest_Ptr
MOV I2C_MCB+3,%%Sub_Addr
ACALL _I2C_READ_SUB
)

%* DEFINE (I2C_READ_REP_READ(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Dest_Ptr_2))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count_1
MOV I2C_MCB+2,%%Dest_Ptr_1
MOV I2C_MCB+3,%%Sub_Addr
MOV I2C_MCB+4,%%Count_2
MOV I2C_MCB+5,%%Dest_Ptr_2
ACALL _I2C_READ_REP_READ
)

%* DEFINE (I2C_READ_REP_WRITE(Slv_Addr,Count_1,Dest_Ptr,Sub_Addr,Count_2,Source_Ptr))
(
MOV I2C_MCB ,%%Slv_Addr
MOV I2C_MCB+1,%%Count_1
MOV I2C_MCB+2,%%Dest_Ptr
MOV I2C_MCB+3,%%Sub_Addr
MOV I2C_MCB+4,%%Count_2
MOV I2C_MCB+5,%%Source_Ptr
ACALL _I2C_READ_REP_WRITE
)

```


I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

I2C_PLM.H

```

/*=====*/
/*
/*      INCLUDE_FILE:   I2C_PLM.H
/*      PACKAGE:       I2C
/*
/*=====*/

/*=====*/
/*      GLOBAL FUNCTION PROTOTYPES
/*=====*/
I2C_INIT: PROCEDURE(Own_Slv_Addr,Slv_Buf_Addr,Retry) BIT EXTERNAL;
  DECLARE(Own_Slv_Addr,Slv_Buf_Addr,Retry) BYTE MAIN;
END I2C_INIT;

I2C_TEST_DEVICE: PROCEDURE(Slv_Addr) BIT EXTERNAL;
  DECLARE(Slv_Addr) BYTE MAIN;
END I2C_TEST_Device;

I2C_WRITE: PORCEDURE(Slv_Addr,Count,Source_Ptr) BIT EXTERNAL;
  DELCARE(Slv_Addr,Count,Source_Ptr) BYTE MAIN;
END I2C_WRITE;

I2C_WRITE_SUB; PROCEDURE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BYTE MAIN;
END I2C_WRITE_SUB;

I2C_WRITE_SUB_SWINC: PROCEDURE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BYTE MAIN;
END I2C_WRITE_SUB_SWINC;

I2C_WRITE_MEMORY: PROCEDURE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BYTE MAIN;
END I2C_WRITE_MEMORY;

I2C_WRITE_SUB_WRITE: PROCEDURE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BYTE MAIN;
END I2C_WRITE_SUB_WRITE;

I2C_WRITE_SUB_READ: PROCEDURE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count,Dest_Ptr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count,Dest_Ptr) BYTE MAIN;
END I2C_WRITE_SUB_READ;

I2C_WRITE_COM_WRITE: PROCEDURE(Slv_Addr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2) BYTE MAIN;
END I2C_WRITE_COM_WRITE;

I2C_WRITE_REP_WRITE: PROCEDURE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BYTE MAIN;
END I2C_WRITE_REP_WRITE;

I2C_WRITE_REP_READ: PROCEDURE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr) BYTE MAIN;
END I2C_WRITE_REP_READ;

I2C_READ: PROCEDURE(Slv_Addr,Count,Dest_Ptr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count,Dest_Ptr) BYTE MAIN;
END I2C_READ;

I2C_READ_STATUS: PROCEDURE(Slv_Addr,Dest_Ptr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Dest_Ptr) BYTE MAIN;
END I2C_READ_STATUS;

I2C_READ_SUB: PROCEDURE(Slv_Addr,Count,Dest_Ptr,Sub_Addr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count,Dest_Ptr,Sub_Addr) BYTE MAIN;
END I2C_READ_SUB;

I2C_READ_REP_READ: PROCEDURE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Dest_Ptr_2) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Dest_Ptr_1) BYTE MAIN;
END I2C_READ_REP_READ;

I2C_READ_REP_WRITE: PROCEDURE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Source_Ptr) BIT EXTERNAL;
  DECLARE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Cout_2,Source_Ptr) BYTE MAIN;
END I2C_READ_REP_WRITE;

```

I2C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

I2C_C.H

```

/*=====*/
/*
/*      INCLUDE_FILE:   I2C_C.H
/*      PACKAGE:       I2C
/*
/*=====*/

/*=====*/
/*      GLOBAL FUNCTION PROTOTYPES
/*=====*/

bit I2C_INIT(char Won_Slv_Addr, char *Slv_Buf_Ptr, char Retry);
bit I2C_TEST_Device(char Slv_Addr);
bit I2C_WRITE(char Slv_Addr, char Count, char *Source_Ptr);
bit I2C_WRITE_SUB(char Slv_Addr, char Count, char *Source_Ptr, char Sub_Addr);
bit I2C_WRITE_SUB_SWINC(char Slv_Addr, char Count, char *Source_Ptr, char Sub_Addr);
bit I2C_WRITE_MEMORY(char Slv_Addr, char Count, char *Source_Ptr, char Sub_Addr);
bit I2C_WRITE_SUB_WRITE(char Slv_Addr, char Count_1, char *Source_Ptr_1, char Sub_Addr, char Count_2, char *Source_Ptr_2);
bit I2C_WRITE_SUB_READ(char Slv_Addr, char Count_1, char *Source_Ptr, char Sub_Addr, char Count, char *Dest_Ptr);
bit I2C_WRITE_COM_WRITE(char Slv_Addr, char Count_1, char *Source_Ptr_1, char Count_2, char *Source_Ptr_2);
bit I2C_WRITE_REP_WRITE(char Slv_Addr, char Count_1, char *Source_Ptr_1, char Sub_Addr, char Count_2, char *Source_Ptr_2);
bit I2C_WRITE_REP_READ(char Slv_Addr, char Count_1, char *Source_Ptr, char Sub_Addr, char Count_2, char *Dest_Ptr);
bit I2C_READ(char Slv_Addr, char Count, char *Dest_Ptr);
bit I2C_READ_STATUS(char Slv_Addr, char *Dest_Ptr);
bit I2C_READ_SUB(char Slv_Addr, char Count, char *Dest_Ptr, char Sub_Addr);
bit I2C_READ_REP_READ(char Slv_Addr, char Count_1, char *Dest_Ptr_1, char Sub_Addr, char Count_2, char *Dest_Ptr_2);
bit I2C_READ_REP_WRITE(char Slv_Addr, char Count_1, char *Dest_Ptr_1, char Sub_Addr, char Count_2, char *Source_Ptr);

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

DEMO_ASM.ASM

```

$CASE
$TITLE(Assembly example program)
;*****
;*
;* SOURCE FILE : DEMO_ASM.ASM
;* PACKAGE : I2C
;*
;* Hours and minutes will be displayed on LCD display
;* Dot between hours and minutes will blink
;*
;*****
$DEBUG
$NOLIST

;*****
;* I N C L U D E S
;*
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_MAC.DEF)
$LIST

;*****
;* G L O B A L F U N C T I O N D E F I N I T I O N S *
;*****
EXTRN CODE(_I2C_INIT)
EXTRN CODE(_I2C_WRITE)
EXTRN CODE(_I2C_READ_SUB)

;*****
;* L O C A L D A T A D E F I N I T I O N S
;*
RAMVAR SEGMENT DATA ;Segment for variables
STACK SEGMENT DATA ;Segment for variables
USER SEGMENT CODE ;Segment for application program

;*****
;* L O C A L S Y M B O L D E F I N I T I O N S
;*
CLOCK_ADR EQU 0A2H ;Address of PCF8583
CL_SUB_ADR EQU 01H ;Sub address for reading time
LCD_ADR EQU 74H ;Address of PCF8577

;*****
;* D A T A S E G M E N T
;*
RSEG RAMVAR
TIME_BUFFER: DS 4 ;Buffer for I2C strings
LCD_BUFFER: DS 5

RSEG STACK
STACK_DATA: DS 10

;*****
;* C O D E S E G M E N T
;*
CSEG AT 00
AJMP APPL_MAIN

RSEG USER

APPL_MAIN:
MOV SP,#STACK_DATA-1
MOV DPTR,#LCD_TAB ;Pointer to segment table
MOV LCD_BUFFER,#00 ;Ctrl word for LCD driver
%I2C_INIT(22h,TIME_BUFFER,0) ;Init I2C interface
CLR A ;Prepare buffer
MOV TIME_BUFFER,A ;for clock int.
MOV TIME_BUFFER+1A
%I2C_WRITE(CLOCK_ADR,2,TIME_BUFFER) ;Initialize clock

REPEAT: %I2C_READ_SUB(CLOCK_ADR,4,TIME_BUFFER,CL_SUB_ADR)
;Read time

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

;-----*
;* Time has been read. Order: *
;*   hundreds of sec's, sec's, min's and hr's *
;-----*
MOV A,TIME_BUFFER+3           ;Mask of hour counter
ANL A,#3Fh
MOV TIME_BUFFER+3,A

ACALL CONVERT                 ;Convert time data to
                               ;LCD segment data

;-----*
;* Check if dot has to be switched on *
;-----*
ORL LCD_BUFFER+3,#01h

;-----*
;* If lsb of seconds is '0', then switch on dp *
;-----*
MOV A,TIME_BUFFER+1           ;Get seconds
RRC A
JC PROCEED
ORD LCD_BUFFER+1,#01         ;Switch on dp

;-----*
;* Display new time *
;-----*
PROCEED: %I2C_WRITE(LCD_ADR,5,LCD_BUFFER)
SJMP REPEAT                 ;Read new time

;-----*
;* CONVERT converts BCD data of time to segment data *
;-----*
CONVERT:MOV R0,#LCD_BUFFER+1  ;R0 is pointer
MOV A,TIME_BUFFER+3           ;Get hours
SWAP A                        ;Swap nibbles
ACALL LCD_DATA                ;Convrt 10's of hours
MOV A,TIME_BUFFER+3           ;Convert hours
ACALL LCD_DATA                ;Get minutes
SWAP A                        ;Convrt 10's of minut
MOV A,TIME_BUFFER+2           ;Convert minutes
ACALL LCD_DATA                ;Convert minutes
RET

;-----*
;* LCD_DATA gets data from segment table and *
;* stores it in LCD_BUFFER *
;-----*
LCD_DATA:
ANL A,#0FH                   ;Mask off LS-nibble
MOVC A,@A+DPTR               ;Get segment data
MOV @R0,A                    ;Save segment data
RET

;-----*
;* Conversion table for LCD *
;-----*
LCD_TAB:
DB 0FCH,60H,0DAH             ;'0','1','2'
DB 0F2H,66H,0B6H             ;'3','4','5'
DB 3EH,0E0H,0FEH             ;'6','7','8'
DB 0E6H                       ;'9'

;
END

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

DEMO_PLM.PLM

```

SOPTIMIZE(4)
SDEBUG
SCODE
/*=====*/
/*
/*      SOURCE FILE : DEMO_PLM.PLM      */
/*      PACKAGE      : I2C              */
/*
/* Hours and minutes will be displayed on LCD display */
/* Dot between hours and minutes will blink           */
/*
/*=====*/

Demo_plm: Do;
/*=====*/
/*      I N C L U D E S                  */
/*=====*/
$NOLIST
$INCLUDE(I2C_PLM.H)
$LIST

/*=====*/
/*      M A I N                          */
/*=====*/

Clock: Do;
/* Variable and constand declarations */

Declare LCD_TAB(*) Byte Constant (0FCh, 60H, 0DAH,
                                0F2H, 66H, 0B6H, 3EH, 0E0H, 0FEH, 0E6H);
Declare Time_Buffer(4) Byte Main;
Declare LCD_Buffer(5) Byte Main;
Declare Tab_Point Word Main;
Declare (LCD_Point, Time_Point) Byte Main;
Declare Segment Based LCD_Point Byte Main;
Declare Time Based Time_Point Byte Main;
Declare Tab_Value Based Tab_Point Byte Constant;

Declare Clock_Adr Literally '0A2h';
Declare LCD_Adr Literally '74h';
Declare Cl_Sub_Adr Literally '01h';

Call I2C_INIT(22h, .Time_Buffer, 0);
LCD_Buffer(0)=0; /* LCD control word */
Time_Buffer(0)=0;
Time_Buffer(1)=0;
Call I2C_WRITE(Clock_Adr, 2, .Time_Buffer);
/* Initialize clock */
Do While LCD_Buffer(0)=0; /* Program loop */
Call I2C_READ_SUB(Clock_Adr, 4, .Time_Buffer,
                 Cl_Sub_Adr); /* Get time */
LCD_Point=.LCD_Buffer(1); /* Init pointers */
Time_Point=.Time_Buffer(3);
Tab_Point=.LCD_Tab(0)+SHR(Time, 4); /* 10-HR's */
Segment=Tab_Value;
LCD_Point=LCD_Point+1;
Tab_Point=.LCD_Tab(0)+(Time AND 0FH); /* HR's */
Segment=Tab_Value;
Time_Point=Time_Point-1;
LCD_Point=LCD_Point+1;
Tab_Point=.LCD_Tab+SHR(Time, 4); /* 10-MIN's */
Segment=(Tab_Value OR 01H); /* dp */
LCD_Point=LCD_Point+1;
Tab_Point=.LCD_Tab+(Time AND 0FH); /* MIN's */
Segment=Tab_Value;
Time_Point=.Time_Buffer(1); /* Check sec's for
                               blinking */
LCD_Point=.LCD_Buffer+1;
If (Time AND 01H)>0 then
    Segment=(Segment OR 01H);
Call I2C_WRITE(LCD_Adr, 5, .LCD_Buffer);
/* Display time */
End;

End Clock;

End Demo_plm;

```

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

DEMO_C.C

```

/*=====*/
/*
/*      SOURCE FILE : DEMO_C.C
/*      PACKAGE      : I2C_DEMO
/*=====*/

/*MPP:::xxxxxx=====*/
/*
/* PACKAGE NAME: I2C_DEMO
/* DESCRIPTION:
/* This demo program reads the time from a PCF8583 clock*/
/* IC, and displays it to an LCD display (both available*/
/* at the I2C demoboard.
/*
/* Hours and minutes will be displayed on LCD display
/* Dot between hours and minutes will blink
/*
/*EMP=====*/

/*=====*/
/*  I N C L U D E S
/*=====*/
#include "I2C_C.H"

/*=====*/
/*  L O C A L  S Y M B O L  D E C L A R A T I O N S
/*=====*/
#define Clock_Adr      0xA2
#define LCD_Adr        0x74
#define Cl_Sub_Adr     0x01

/*=====*/
/*  L O C A L  D A T A  D E F I N I T I O N S
/*=====*/
rom char      LCD_Tab[]={0xFC,0x60,0xDA,0xF2,0x66,
                        0xB6,0x3E,0xE0,0xFE,0xE6};

/*=====*/
/*  M A I N
/*=====*/
void main()
{
    rom char      *Tab_Ptr;
    data char     Time_Buffer[4];
    data char     *Time_Ptr;
    data char     LCD_Buffer[5];
    data char     *LCD_Ptr;

    I2C_INIT90x22,&Time_Buffer,0);
    LCD_Buffer[0]=0; /* LCD control word*/
    Time_Buffer[0]=0;
    Time_Buffer[1]=0;
    I2C_WRITE(Clock_Adr,2,&Time_Buffer); /* Init clock*/

    while (1)      /* Program loop*/
    {
        I2C_READ_SUB(Clock_Adr,4,&Time_Buffer,Cl_Sub_Adr);
                                /* Get time*/
        LCD_Ptr = &LCD_Buffer[1]; /* Init pointers*/
        Time_Ptr = &Time_Buffer[3];
        Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /*10-HR's*/
        *(LCD_Ptr++)=*Tab_Ptr;
        Tab_Ptr = (LCD_Tab+(*Time_Ptr-- & 0x0F)); /* HR's*/
        *(LCD_Ptr++)=*Tab_Ptr;
        Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10-MIN's*/
        *LCD_Ptr++ = (*Tab_Ptr | 0x01); /* dp*/
        Tab_Ptr = (LCD_Tab+(*Time_Ptr & 0x0F)); /* MIN's*/
        *LCD_Ptr=*Tab_Ptr;
        Time_Ptr = &Time_Buffer[1]; /* Check sec's blinking*/
        LCD_Ptr = &LCD_Buffer[1];
        if ((*Time_Ptr & 0x01)>0)
            *LCD_Ptr = (*LCD_Ptr | 0x01);
        I2C_WRITE(LCD_Adr,5,&LCD_Buffer); /* Display time*/
    }
}

```

Programming the I²C interface

Author: Mitchell Kahn

Dr. Dobb's

JOURNAL

The Inter-Integrated Circuit Bus ("I²C Bus" for short) is a two-wire, synchronous, serial interface designed primarily for communication between intelligent IC devices. The I²C bus offers several advantages over "traditional" serial interfaces such as Microwire and RS-232. Among the advanced features of I²C are multimaster operation, automatic baud-rate adjustment, and "plug-and-play" network extensions.

Mention the I²C bus to a group of American engineers and you'll likely get hit with an abundance of blank stares. I say American engineers because until recently the I²C bus was primarily a European phenomenon. Within the last year, however, interest in I²C in the United States has risen dramatically. Embedded systems designers are realizing the cost, space, and power savings afforded by robust serial interchip protocols.

The idea of serial interconnect between integrated circuits is not new. Many semiconductor vendors offer devices designed to "talk" via serial links with other processors. Current examples include Microwire (National Semiconductor), SPI (Motorola), and most recently Echelon's Neuron chips. In all cases, the goal is the same: to reduce the wiring and pincount necessary for a parallel data bus. It simply does not make

Mitch is a senior strategic development engineer for Intel and can be contacted at 5000 W. Chandler Blvd., Chandler, AZ 85226 or at mkahn@sedona.intel.com.



economic sense to route a full-speed parallel bus to a slow peripheral.

Unfortunately for most serial-bus-capable devices, the choice of a bus protocol will dictate the CPU architecture. For example, only two CPU architectures implement an on-chip I²C port. If your choice of architecture precludes use of these architectures, then your only option is to implement the protocol in software.

The software implementation of the I²C protocol discussed in this article came about as a result of an implicit challenge during a staff meeting. One of our managers proposed that we hire a consultant to write a software I²C driver for the Intel 80C186EB embedded processor. Being somewhat new to the

group, I took exception (although not verbally!) to his suggestion. A weekend of intense hacking later, I presented the first prototype of the driver. My reward? I got to write a generic version of the driver for general distribution.

Design Trade-offs

Three distinct tasks are involved in implementing the I²C protocol: watching the bus, waiting for a specific amount of time, and driving the bus. This became apparent when I flowcharted 1 byte of a typical bus transaction; see Figure 1. The time delays associated with creating the bus waveforms would normally have been relegated to the 80C186EB's on-chip timers. I could not, however, assume that the end users of my code would be able to spare a timer for the software I²C port. I had to forego the elegance (and to some extent accuracy) of the on-chip timers for the sledgehammer approach of software timing loops. Luckily, the I²C protocol is extremely forgiving with regard to timing accuracy. The decision to use assembly instead of a high-level language stemmed directly from the need to control program-execution time. I had neither the time nor the inclination to hand-tune high-level code.

Having made the decision to use assembly language, I faced my next problem: Could I make the code portable? Intel offers a plethora of CPU and embedded-controller architectures. Would it be possible to make the code somewhat portable between disparate assembly languages? I found my answer in the use of macros.

Dr. Dobb's Journal, June 1992

Programming the I²C interface

I²C

All the basic building blocks of the I²C protocol (watching, waiting, and doing) can be compartmentalized into distinct macros. The algorithms that make up the I²C driver are written with these macros as the framework. You don't need to understand the intricacies of the I²C protocol to port these routines—you just need to know how to make your CPU watch, wait, and do.

For example, a 4.7- μ s delay is a common event during a transfer. The macro %Wait_4_7_uS implements just such a delay by using the 8086 LOOP instruction with a couple of NOPs for tuning; see Example 1(a). Total execution time is readily calculated from instruction timing tables. The same macro is ported to the i960 architecture in Example 1(b). Although I am a neophyte when it

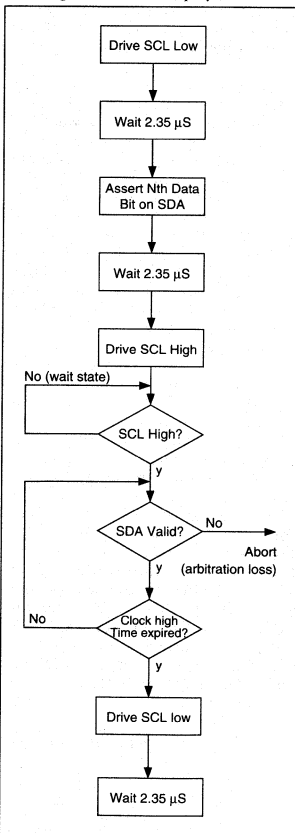


Figure 1: Flowchart of process for transmission of a single bit.

comes to i960 programming, I had no problems porting the core macros.

Hardware Dependencies

A few words about the target hardware are in order before I discuss the code. Any implementation of the I²C protocol requires two open-drain (or open-collector), bidirectional port pins for the Serial Clock (SCL) and Serial Data (SDA) lines. The code in this article was designed for the 80C186EB embedded processor, which has two open-drain ports on-chip. The two pins, P2.6 (SCL) and P2.7 (SDA), are part of a larger 8-bit port. Processors without open-drain I/O ports can easily implement I²C with the addition of an external open-collector latch.

Two special-function registers, P2PIN and P2LATCH, are used to read and write the state of the port pins. The 80C186EB allows the special-function registers to be located anywhere in either memory or I/O space. For this implementation, I chose to leave the registers in I/O space, even though this limited my choice of instructions. The 80186 architecture does not provide for read-modify-write instructions in I/O space (an AND to I/O, for example); it can only load and store (IN and OUT). So why did I limit myself? Again, I had to assume the lowest common denominator for our customers when designing my code.

Building the Framework

Early on in development, I decided to partition my code macros according to physical processes involved in the I²C

protocol. Code not directly involved in mimicking the actions of a hardware I²C port was not written as macros. For example, the code necessary to access the stack frame is not written as a macro, whereas the code needed to toggle the clock line is. This was done to isolate architecture-dependent code sequences from the more generic I²C functions. Macros were also not used for "gray areas" such as the shifting of serial data, which is both architecture dependent and physical in nature. The I²C functions that passed the litmus test fell into the three aforementioned categories of watching, waiting, and doing.

The "waiting" macros provide a fixed-minimum time delay. They are implemented using a simple LOOP \$ delay. The LOOP instruction decrements the CX register, then branches to the target (in this case itself) if the result is non-zero. The delay is $(n-1)*15+5$ clocks, where n is the starting value in the CX register. All the delays were calculated assuming a 16-MHz clock rate (62.5 nanoseconds per clock). The code still works at lower CPU speeds because the I²C protocol only specifies minimum timings. In fact, the delay macros are only "accurate enough," providing timings as close as I could get to the specified minimum without undue tuning.

The "watching" macros are "spin-on-bit" polling loops. These pieces of code wait for a transition on the appropriate I²C line to occur before allowing execution to continue. There are two polling macros for each of the two I²C signal lines; one for high-to-low transitions and one for low-to-high transitions. The

```

(a)
%*DEFINE(Wait_4_7_uS) (
    mov    cx, 5          ; 4 clocks
    loop  $              ; 4*15+5 = 65 clocks
    nop                    ; 3 clocks
    nop                    ; 3 clocks
    ; total = 75 clocks
    ; 75 * 62.5ns = 4.69uS (close enough)
)

```

```

(b)
define(Wait_4_7_uS, '
    lda    0x17, r4      # instruction may be issued in parallel
                        # so assume no clocks.
0b:      cmpdeco 0, r4   # compare and decrement counter in r4
                        # if !=0 branch back (predict taken)
                        # branch
                        #
                        # The cmpdeco and bne.t together take 3
                        # clocks in parallel minimum.
                        #
                        # 0x17 (25 decimal) * 3 = 75 clocks
                        # at 16MHz this is 4.69uS
')

```

Example 1: (a) 80C186 implementation of 4.7- μ s wait macro; (b) 80960CA implementation of 4.7- μ s wait macro.

Programming the I²C interface

I²C

polling of the SCL line that gives rise to an important feature of I²C: automatic, bit-by-bit baud-rate adjustment. Any device on the I²C bus may hold the clock line low in order to stall the bus for more time (a serial wait state). The other devices on the bus are then forced to poll the SCL line until the slow device releases control of the clock.

The `%Get_SDA_Bit` macro also falls under the category of "watching." Its function is simply to return the state of the SDA line without waiting for a transition. `%Get_SDA_Bit` is used primarily to pull the serial data off the bus when the clock is valid.

The "doing" macros control the state of the clock and data lines. As with the polling macros, there are four types—one for each transition of the SCL or SDA lines. The "doing" macros are named to reflect the physical operations they perform. For example, `%Drive_SCL_Low` always drives the SCL line to a low state. `%Release_SCL_High`, on the other hand, relinquishes control of the SCL line, which may then be pulled high or driven low by another device on the bus. A read-modify-write operation is used for the bit manipulation so that the other 6 bits of Port 2 are not affected by the I²C operations.

Getting on the Bus

Three procedures were created using the macro framework. I'll describe only the master transmit (Listing One, page

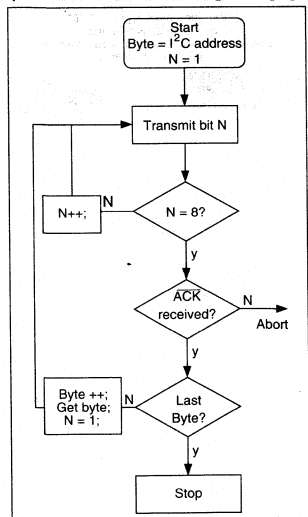


Figure 2: Flowchart for I²C transmit procedure.

106) and master receive functions (Listing Two, page 108), as they represent the needs of most I²C users. The slave procedure is long and intricate and will not be described here.

An I²C master transmission proceeds as follows:

1. The master polls the bus to see if it is in use.
2. The master generates a start condition on the bus.
3. The master broadcasts the slave address and expects an acknowledge (ACK) from the addressed slave.
4. The master transmits 0 or more bytes of data, expecting an ACK following each byte.
5. The master generates a stop condition and releases the bus.

The stack frame for the master transmit procedure, `I2CXA.A86`, includes a far pointer to the message for transmission, the byte count for the message, and the slave address. Far pointers and far procedure calls are used in all the procedures. No attempt was made to conform to a specific high-level language calling convention, although such a conversion would be trivial. The procedures save only the state of the modified segment registers.

The master transmit procedure performs error checking on the passed parameters before attempting to send the message. The maximum message length is set at 64 Kbytes by the segmentation of the 80186 memory space. This restriction could be removed by including code to handle segment boundaries. The transmit procedure also checks the direction bit in the slave address to ensure that a reception was not erroneously indicated. Errors are reported back to the calling procedure through the AX register. (The exact code is in Listing One.)

The first step in sending a message is getting on the I²C bus. The macro `%Check_For_Bus_Free` simply polls the bus to determine if any transactions are in progress. If so, the transmit procedure aborts with the appropriate error code. If the bus is free, a start condition is generated. The start condition is defined as a high-to-low transition of SDA with SCL high followed by a 4.7_μs pause. These waveforms are easily generated with the `%Drive_SDA_Low` and `%Wait_4_7_μs` macros.

All communication on the I²C bus between the stop and start conditions, including addressing and data, takes place as an 8-bit data value followed by an acknowledge bit. This led to the natural nested loop structure for the body of the procedure; see Figure 2.

The inner loop is responsible for transmitting the 8 bits of each data byte. Each transmitted bit generates the appropriate data (SDA) and clock (SCL) waveforms while checking for both serial wait states and potential bus collisions. A bus collision occurs when two masters attempt to gain control of the

Three distinct tasks are involved in implementing the I²C protocol: watching the bus, waiting for a specific amount of time, and driving the bus

bus simultaneously. The I²C protocol handles collisions with the simple rule: "He who transmits the first 0 on the SDA line wins the bus." To ensure that we (the master transmit procedure) own the bus, the SDA line is checked whenever transmitting a 1. If a 0 is present, then a collision has occurred (because another master is pulling the line low), and the transfer must be aborted.

Control is turned over to the outer loop after the 8 bits of data (or address) have been transmitted. The outer loop immediately checks for an acknowledge from the addressed slave. The transfer is aborted if an acknowledge is not received. At the end of the ACK bit the message length counter is decremented. Control is returned to the inner loop if more data remains, otherwise a stop condition is generated and the master transmit procedure terminates.

Registers are used for intermediate result storage throughout the body of the procedure. For example, the AH register is used to hold the current value (either address or data) being shifted onto the SDA line. This eliminates the need for local data storage within the procedure.

On the Receiving End

The steps involved in an I²C master receive transaction are almost identical to those in transmission:

1. The master polls the bus to see if it is in use.
2. The master generates a start condi-

Programming the I²C interface

I²C

- tion on the bus.
- The master broadcasts the slave address and expects an ACK from the addressed slave.
 - The master receives 0 or more bytes of data and sends an ACK to the slave after each byte. The master signals the last byte by not sending an ACK.
 - The master generates a stop condition and releases the bus.

A far pointer to the receive buffer is passed on the stack to the master receive procedure. The remainder of the parameters—slave address and message count—are identical between the two procedures. The received message length is fixed at 64 Kbytes, again because of segmentation. The error-checking, bus-availability sensing, and start-condition generation sections of the receive procedure are lifted verbatim from the transmit code.

The structure of the receive procedure differs slightly once the start con-

dition has been generated; see Figure 3. The slave address is transmitted using one iteration of the transmit procedure's outer loop. Control is passed to the receive loop once the slave acknowledges its address.

The receive loop structure is patterned after that of the transmit procedure. The inner loop controls the clocking of the SCL line and the shifting of the serial data off the SDA line into the CPU. Eight iterations of the inner loop are performed to receive each byte. The outer loop stores the received byte in the buffer, decrements the byte count, then sends an ACK to the slave. The last data byte is signalled by not sending an ACK.

Using the Procedures

Listing Three (page 110) shows a short program that uses both the master transmit and master receive procedures. The call to procedure I2C_XMIT displays the word "BUS-" on a four-character, seven-segment display controlled by the SAA1064 I²C compatible display driver. The time of day is read from the PCF8583 real-time clock by the call to procedure I2C_RECV.

Please note that interrupts must be disabled during the execution of both procedures. An interruption at an inopportune time (when the master is not in control of the clock) could cause the bus to hang. If you need to service interrupts periodically, then enable them only when the clock is driven low.

These procedures have been tested on a wide array of I²C devices ranging from serial EEPROMs to voice synthesizers. No compatibility problems have been seen to date.

Enhancing the Code

I've kicked around many ideas for enhancing the I²C procedures. You could,

for example, replace the timing loops with timed interrupts. That way, the CPU could perform useful work during the pauses. Along the same lines, the pauses could be scheduled using a real-time kernel, again improving CPU throughput. Finally, you could add a high-level language calling structure.

The use of timed interrupts adds an order of magnitude to the complexity of the code, but would be worth it for high-performance, real-time systems.

Conclusion

I²C is not the only game in town when it comes to serial protocols. Hopefully, some of the techniques presented here will carry over into the development of other "simulated" serial protocols, such as those targeted at the home-automation market. Who knows, maybe someday a snippet of my code may find its way into a truly intelligent dishwasher. I'll be waiting....

References

I²C Bus Specification, Philips Corporation (undated).

DDJ

Reprinted with permission of Dr. Dobb's Journal, 1992

Entire contents copyright © 1992 by M&T Publishing, Inc.

Unless otherwise noted on specific articles. All rights reserved.

ABP
American Business Press

The Audit Bureau

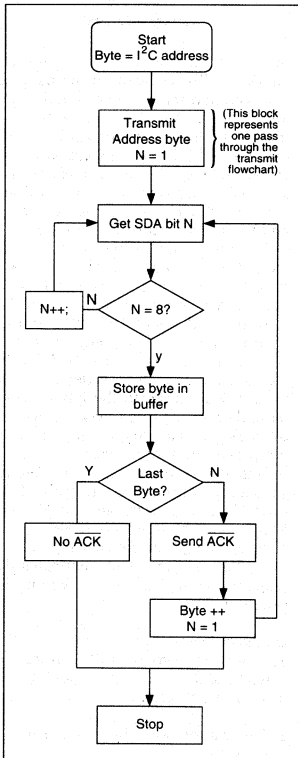


Figure 3: Flowchart for I²C receive procedure.

All the basic building blocks of the I²C protocol (watching, waiting, and doing) can be compartmentalized into distinct macros

Section 4

ACCESS.bus Technical Overview

Application Notes and
Development Tools for
80C51 Microcontrollers

CONTENTS

ACCESS.bus Technical Overview	275
Introduction	275
What is ACCESS.bus?	275
ACCESS.bus Hardware	275
ACCESS.bus Protocols	276
How ACCESS.bus Works	277
Electrical	277
Bus Transactions	277
Synchronization	279
Byte Framing and Acknowledgement	279
Addressing	279
Arbitration	279
Message Format	279
Control/Status Messages	279
Configuration	280
Device Identifiers	281
Device Capabilities Information	281
Application Device Types	282
Keyboard Devices	282
Locator Devices	282
Text Devices	282
Timing Rules	282
Transaction Timing Rules	282
Response Timeouts	282
Software Architecture and Development	282
Device Firmware Development	283
Host Software Architecture	283
Development Support	283
ACCESS.bus Industry Group	283
Philips Semiconductors Support	283
ACCESS.bus development kit	284
ACCESS.bus PC/AT controller board	286

ACCESS.bus™ technical overview

INTRODUCTION

ACCESS.bus™ (a BUS for connecting ACCESSory devices to a host system) is a peripheral interconnect system defined and developed by Digital Equipment Corporation and offered to the computer industry as an open standard.

This overview aims to introduce the prospective developer of ACCESS.bus systems or peripherals to the essential technical features of this interconnect. It is meant to be a general technical overview of the ACCESS.bus architecture. Under no circumstances should it be used as the basis for designing any device or system. Developers wishing to design a host system or peripheral device that implements ACCESS.bus should refer to the *ACCESS.bus Hardware and Protocol Specifications Version 2.0*, July 1992, available from the ACCESS.bus Industry Group, Digital's TRI/ADD Program, or Philips. Addresses and other information on support may be found in Section 6.

What is ACCESS.bus?

ACCESS.bus is a system for connecting a number of relatively low-speed I/O devices to a host computer, typically a desktop system, such as a workstation, personal computer, or terminal. Devices include both interactive

peripherals – keyboards, locators, hand-held scanners, bar code readers, and magnetic card readers – and non-interactive peripherals – printers, and in realtime control applications, signal transducers. Further, the ACCESS.bus protocol is general enough to accommodate a wide range of unusual peripheral types such as data gloves (see Figure 1).

ACCESS.bus has a bus topology architecture. That is, a single ACCESS.bus on a host can accommodate up to 125 peripheral devices. The total length of the cable connecting the devices on a common ACCESS.bus may be up to eight meters. The limiting factors are capacitance, which may not exceed 800pF, and the maximum voltage drop, which must allow maintenance of $+5V \pm 10\%$. Using an I²C bus extender that maximum distance may be lengthened. ACCESS.bus supports a maximum aggregate data throughput of approximately 80 Kbits/sec.

Digital has made the ACCESS.bus technology an open specification, enabling any vendor to implement it on host systems or in peripheral devices without fee or royalty.

ACCESS.BUS offers a number of advantages both to end users and to the developers of systems and peripheral

devices. A host computer needs only one hardware port to connect to a number of devices. The commonality in communication methods for a number of device types leads to economies in software and hardware development. As an open industry standard, ACCESS.bus will stimulate development of diverse peripheral devices, each usable with a number of different types of host systems.

ACCESS.bus incorporates more sophisticated technology and offers higher performance than any other bus-topology interconnect for desktop peripherals. Moreover, it is the first system of this kind to be offered as an open nonproprietary standard.

ACCESS.bus Hardware

At the hardware level, ACCESS.bus is based on the well-established Inter-Integrated Circuit (I²C) serial bus developed and patented by Philips. The serial bus architecture, in which a single data line carries one bit of information at a time, entails lower costs for cabling, connectors and controller circuitry than parallel bus architectures.

Standard low-cost I²C components, available from Philips, handle the logical complications of the bit-level handshaking. More details on these components are given in section 5.

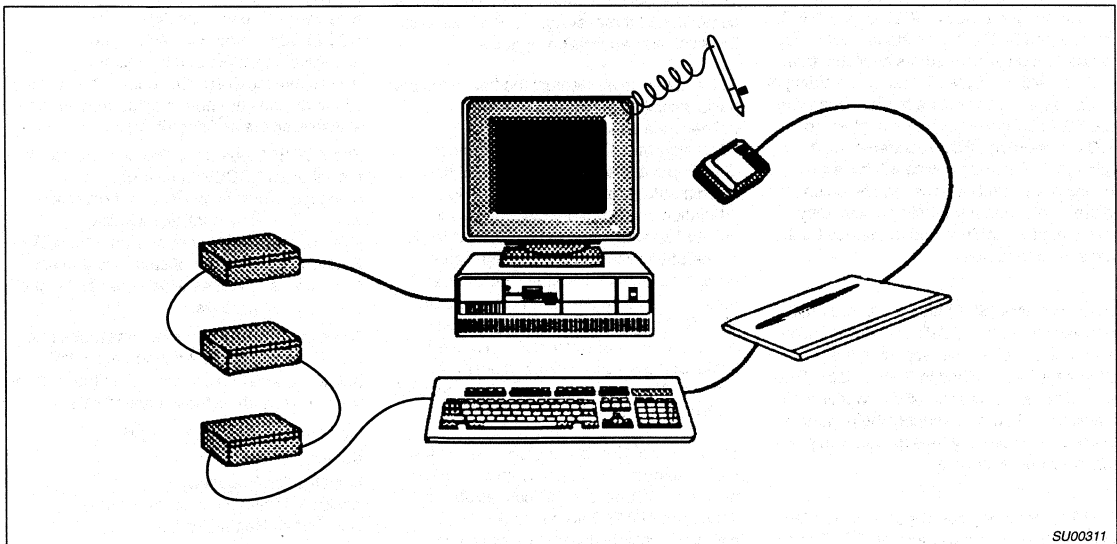


Figure 1. ACCESS.bus connects keyboards, locators, and text-type devices to a system.

SU00311

ACCESS.bus™ technical overview

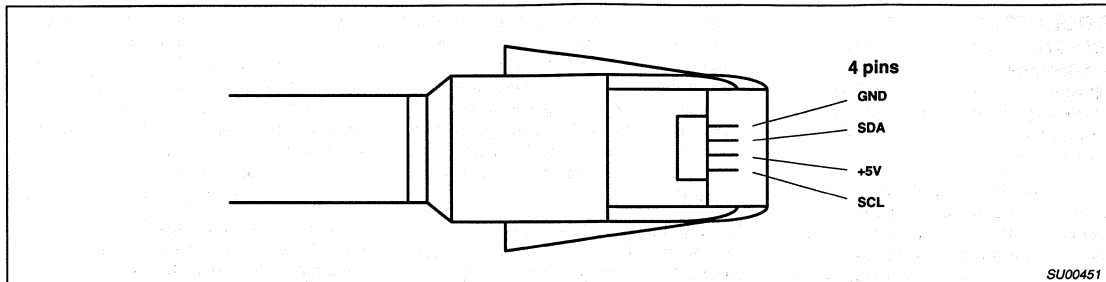


Figure 2. A shielded connector with four pins for connecting the ACCESS.bus cable to a system

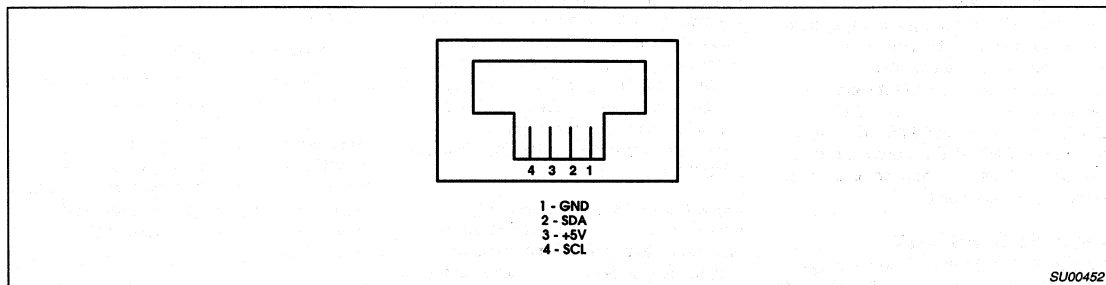


Figure 3. Female Connector Pin Identification (Not To Scale)

The physical medium for ACCESS.bus is a shielded cable containing four wires: serial data (SDA), serial clock (SCL), power (+5V), and ground (GND). It uses standard low-cost shielded modular connectors available from AMP and Molex (Figure 2 and 3). Shielding of the cables and connectors facilitates making ACCESS.bus-based systems conform to FCC radiation and ESD requirements. A typical ACCESS.bus device will have two connectors so that devices may be chained on the single bus; hand-held devices may have a captive cable joined to the bus trunk with a "T" connector.

The serial data (SDA) and serial clock (SCL) lines work together to define the information carried on the bus. That aspect of the technology is described in Section 2.3. The host computer drives the +5V power line with a minimum of 50mA to supply the peripheral devices. Devices may be supplied with power from an external source.

The I²C technology can support clock rates up to 100kHz. The maximum ACCESS.bus data transfer rate of approximately 80 Kbits/sec is derived from the top clock rate by subtracting the overhead imposed by the ACCESS.bus communication protocols for handshaking, addressing, and error control.

ACCESS.bus Protocols

The ACCESS.bus communication protocol is composed of three levels: I²C Protocol, Base Protocol, and Application Protocol.

At the lowest level, nearest the hardware, the basic discipline of the ACCESS.bus is defined as a subset of the Philips Inter-Integrated Circuit (I²C) bus protocol. The simple and efficient I²C Protocol defines a symmetric multi-master bus on which arbitration among contending masters is effected without losing data. I²C provides for cooperative synchronization of the serial clock for exchange of data between bus partners with different maximum clock rates. The I²C Protocol defines a bus transaction scheme with addressing, framing of bits into bytes, and byte-acknowledgement by the receiver. More detail on the I²C Protocol level is given in Section 2.

The next ACCESS.bus protocol level is the Base Protocol. This level, common to all types of ACCESS.bus devices, establishes the nature of ACCESS.bus as an asymmetrical interconnect between a host computer and a number of peripheral devices. The host plays a special role as a manager of the ACCESS.bus. Data Communication is always between host and peripheral device and never between two

peripherals. While the I²C Protocol provides for mastership by either the sender or the receiver of a bus transaction, in the ACCESS.bus protocol masters are exclusively senders and slaves are exclusively receivers. Of course, the host and all the devices are both master/senders and slave/receivers at different times.

The ACCESS.bus Base Protocol defines the format of an ACCESS.bus message envelope, which is an I²C bus transaction with additional semantics, including checksum reliability control. Further, the Base Protocol defines a set of seven control and status message types which are used in the configuration process.

The eight required interface messages that ACCESS.bus protocol defines are listed below. Parameters defined within the body of the message are listed in parenthesis.

Computer-to-device Messages:

Reset()
 Identification Request()
 Assign Address(ID string, new addr)
 Capabilities Request(offset)

Device-to-computer Messages:

Attention(status)
 Identification Reply(ID string)
 Capabilities Reply(offset, data frag)
 Interface Error()

ACCESS.bus™ technical overview

Two of the unique features of this configuration process are auto-addressing and hot plugging. Auto-addressing refers to the way that devices are assigned unique bus addresses in the configuration process, without the need for setting jumpers or switches on the devices. Hot plugging refers to the ability for attaching or detaching devices while the system is running, without the need for rebooting the host. The means by which the ACCESS.bus protocol provides these features is discussed in Section 2.

The highest level of the ACCESS.bus protocol, the Application Protocol, defines message semantics that are specific to particular functional types of devices. Different device types require different Application Protocols. Application Protocols have been defined so far for three device classes: keyboards, locators, and text devices. Each of these predefined classes is designed to be broad. The keyboard device protocol defines standard messages for reporting keystrokes and controlling keyboard peripherals. The protocol attempts to define the simplest set of functions from which common industry standard keyboard interfaces can be built. The locator device protocol defines a set of standard messages for reporting locator movement and key switch activation for mice, tablets, and other positioning devices. The protocol is designed to accommodate a range of basic locator devices such as a mouse or tablet. More complex devices can be modeled as a combination of basic devices or can provide their own device driver. The text device

protocol is intended to provide a simple way to transmit character or binary data to and from stream oriented devices such as a bar code reader, or modem. The sequential character stream model also serves as a common denominator for connecting RS-232 interface devices.

A major advantage in designing devices that conform to these general device-type semantics is that they may share device-specific software, both in the device-resident firmware and in the driver software needed in the host operating system to allow application programs to access the devices.

It is anticipated that further device-specific Application Protocols will be defined in the future, under the aegis of the ACCESS.bus Industry Group. Further, any device vendor may implement a special device protocol within the general message envelope defined by the Base Protocol.

Participation in all three of the protocol levels requires intelligence at the device level. The lower levels of this firmware are likely to be common to many devices. Higher levels of the firmware are expected to be more specific to the device and the application (Figure 4).

HOW ACCESS.bus WORKS

Electrical

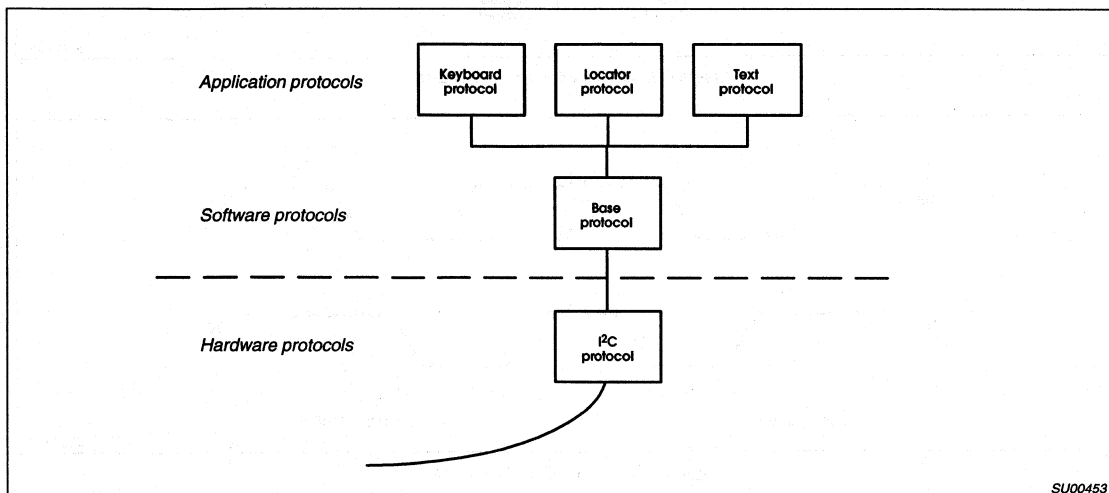
The host and devices are connected to both the serial data (SDA) and serial clock (SCL)

lines in an "wired-AND" logic configuration. The wired-AND may be implemented by connecting the data and clock output stages of each bus partner to the SDA and SCL lines respectively through open-collector or open-drain transistors. The standard I²C components include such output stages on-chip. The significance of the wired-AND logic is that any attached bus partner may force either of these lines to low (the ground level). When there is no output from any bus partner, the lines are held high by pull-up current sources in the host. Every bus partner can sense the level on both of these lines (Figure 5).

Bus Transactions

During a bus transaction, there is one clock pulse on SCL for each bit transferred on SDA. The SDA information is valid when SCL is high. During a transaction, the SDA must be stable between the rising and falling edges of the SCL pulse; SDA may change state only when SCL is low (Figure 6).

SDA transitions when the SCL is high are signals that delimit the bus transaction. When the ACCESS.bus is free, both SCL and SDA are high. A high-to-low SDA transition when SCL is high is a start condition; it signals the beginning of a bus transaction. A bus partner asserts mastership by pulling SDA low when the bus is free. A low-to-high SDA transition when SCL is high is a stop condition; it signals the end of a bus transaction. A master generates a stop condition when it relinquishes mastership (Figure 7).



SU00453

Figure 4. ACCESS.bus protocol hierarchy

ACCESS.bus™ technical overview

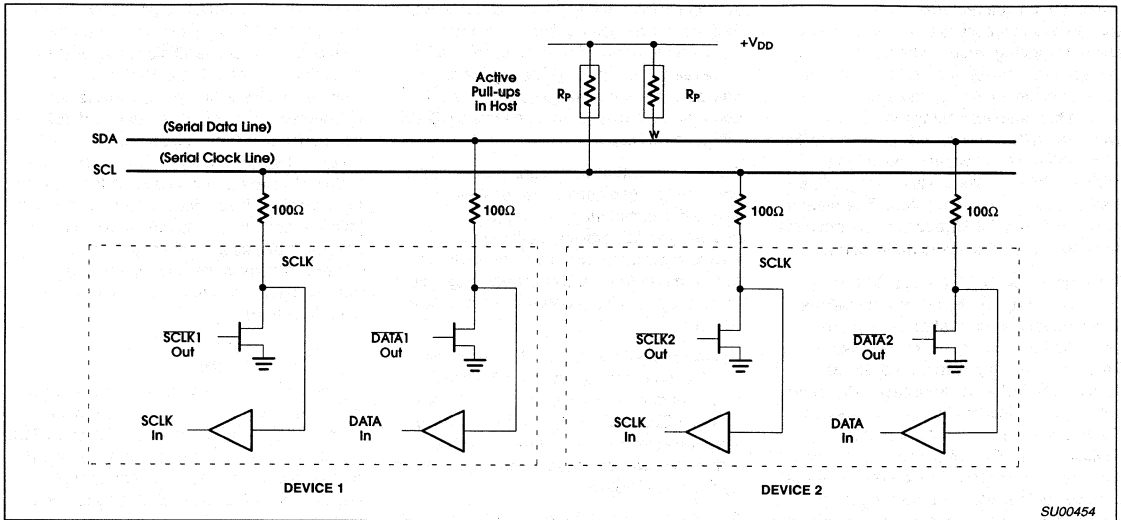


Figure 5. Connection of Devices to the I²C Bus

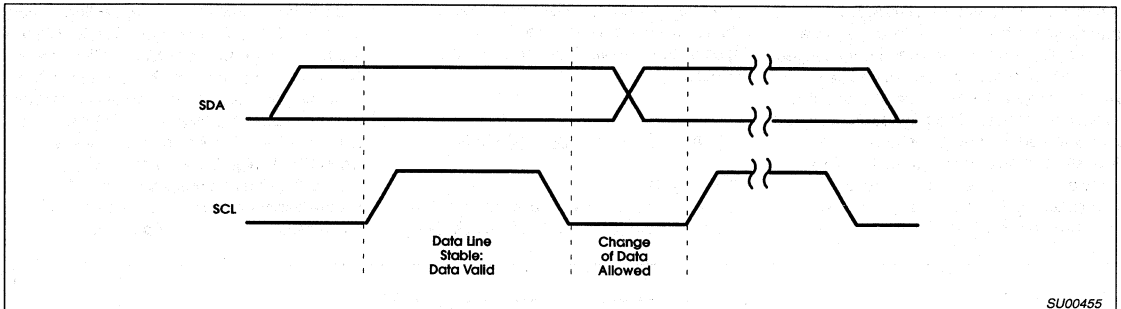


Figure 6. Bit Transfer on the I²C Bus

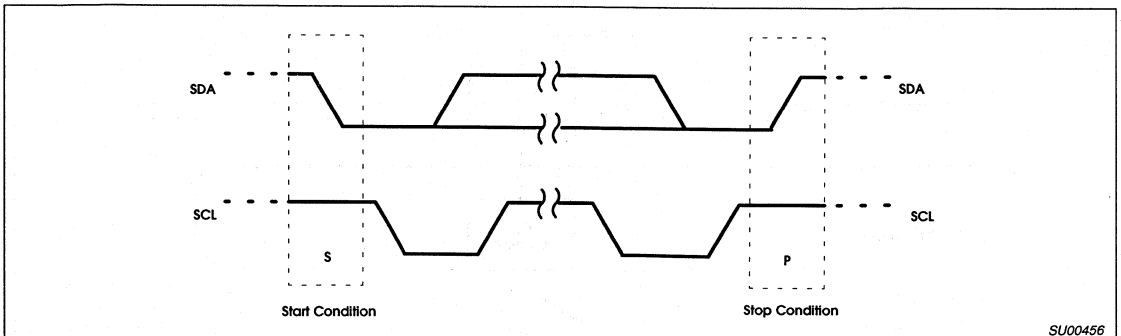


Figure 7. Start and Stop Conditions

ACCESS.bus™ technical overview

Synchronization

When a bus partner wishes to assert mastership of a free bus, it generates a start condition by pulling the SDA low. When SDA is low the new master begins the clock cycle, pulling the SCL low. All bus partners must be able to sense these events, and they respond by pulling all their SCL outputs low and beginning to count off their low periods. When each bus partner has reached the end of its low period, it lets its SCL output go high. Thus the SCL line will remain low for the duration of the longest low clock period among the bus partners.

When all the bus masters have reached the end of their low periods and let their SCL outputs go high, then the SCL goes high. All bus partners must be able to sense this event, and they begin counting their high period. The first bus master to reach the end of its high period pulls the SCL low again. In this way, all the bus partners simultaneously communicating on the bus are synchronized by a clock pulse whose low is as long as the longest of the low periods and whose high is as long as the shortest of the high periods. This synchronization persists until the master relinquishes the bus by generating a stop condition. The cooperative synchronization is a mechanism by which devices with slower clocks can regulate the operating rate of the bus. However, this mechanism, called "clock stretching", is not the normal means of data stream flow control. The ACCESS.bus protocol provides another mechanism for this purpose. (See Section 3.3.)

Byte Framing and Acknowledgement

During this synchronized exchange the master/transmitter puts data on the SDA, one bit for each clock pulse. Eight successive bits comprise a byte, the most significant bit going first. The ACCESS bus is a Big Endian System.

As the new master puts the first byte on the bus, all the other bus partners participate in the synchronization. The first byte of the transaction contains the address of the intended slave/receiver of the transaction. Each non-master can check the address bits as they appear and cease participating in the synchronization as soon as the address bit on SDA fails to match the corresponding bit of its own address. The address check may also be done at the end of the address transmission.

At the end of the first byte, the master/transmitter lets its data output go high for the next clock pulse and the slave/receiver whose address matches the

transmitted address, is obliged to acknowledge receipt of the byte by pulling the SDA low for this pulse. This 1-bit Ack continues after each byte of the bus transaction; the master lets its SDA output go high and the receiver must pull the SCL low. Failure of the receiver to acknowledge a byte is an exception condition, which requires the master to terminate the transaction.

Addressing

The slave/receiver of the bus transaction is determined by the address contained in the first byte. I²C uses the seven high-order bits of the first byte for addressing, and bit 0 to indicate whether the master is transmitting or receiving data. In ACCESS.bus, the master is always the transmitter, so bit 0 of the first byte of a transaction is always 0. Of the 128 7-bit addresses ACCESS.bus uses 15 addresses for general microcontrollers.

The host computer address is always 50h. In the configuration process to be described below, each peripheral device is assigned a unique address from the set of even numbers. 6Eh is used as a default address for devices before they have been assigned a unique address. A total of 125 addresses are available for devices on the bus.

Arbitration

What happens when two devices simultaneously assert mastership? While putting data on the SDA, each transmitting master is, of course, independently sensing the state of SDA. Whenever a contending master detects that the state of the SDA is different from the data value it is putting out during a clock high, the contending master backs off, and waits for the stop condition before trying again. Thus, two contending masters will both put data on the bus as long as they are putting out the same data. The first bit where they differ will cause the contender that put out a 1 to back off.

Thus, under normal expected operation contending masters trying to send to different bus addresses will resolve the contention by the end of the first byte of the bus transaction. In the ACCESS.bus Base Protocol, the second byte of a transaction is the address of the transmitting master. Thus, as long as bus addresses are unique, the mastership of the bus will be resolved by the end of the second byte of the transaction. However, if two devices have the same address and are trying to send identical messages to a common address, then they will both send the entire message in unison. This situation can happen only during the configuration process before devices have all been assigned unique addresses; it is discussed further in Section 2.9 below.

Note that this arbitration mechanism never causes lost data or wasted transmissions, since the addresses of the receiver and transmitter are necessary overhead, in any case, for any sensible bus protocol. Note that bus priorities during arbitration are fixed by the device addresses, first by the address of the receiver, and then, for messages addressed to a common receiver, by the address of the transmitter. Lower addresses have priority over higher addresses. Lockouts of devices with high addresses are prevented by a rule of the Base Protocol that requires partners to wait a minimum time after relinquishing mastership before asserting it again.

Message Format

An ACCESS.bus message comprises one I²C bus transaction. It consists of a string of bytes sent by a master/transmitter, each byte acknowledged by a one bit SCL-low Ack from the slave/receiver. The entire transaction is delimited by start and stop conditions generated by the master.

The first byte in the message is the receiver's unique address, as described above in Section 2.5. The second byte contains the transmitter's unique address.

The third byte of an ACCESS.bus message comprises two fields: Bits 2-7 provide a byte count for the body of the message. Thus, a message body can have 0 to 127 bytes. The message body is followed by a checksum byte, for error control. The checksum is the bitwise XOR of all the preceding bytes of the message.

The high order bit of the third byte is a Protocol Flag (P) to distinguish between data stream messages (P=0) and control/status messages (P=1). The data stream messages carry the application information being exchanged between the device and the host. The control/status messages are used to manage the ACCESS.bus protocol (Figure 8).

Control/Status Messages

The ACCESS.bus Base Protocol defines a number of control/status messages that pertain to the Interface Parts of devices. These control/status messages are used for the configuration process, in which devices are assigned unique bus addresses and connected with the appropriate drivers in the host. The configuration process is described in Section 2.9 to 2.11. In a control/status message, the message type is indicated by an operation code contained in the first byte of the message body. The various Interface Part control/status messages are shown in Table 1.

ACCESS.bus™ technical overview

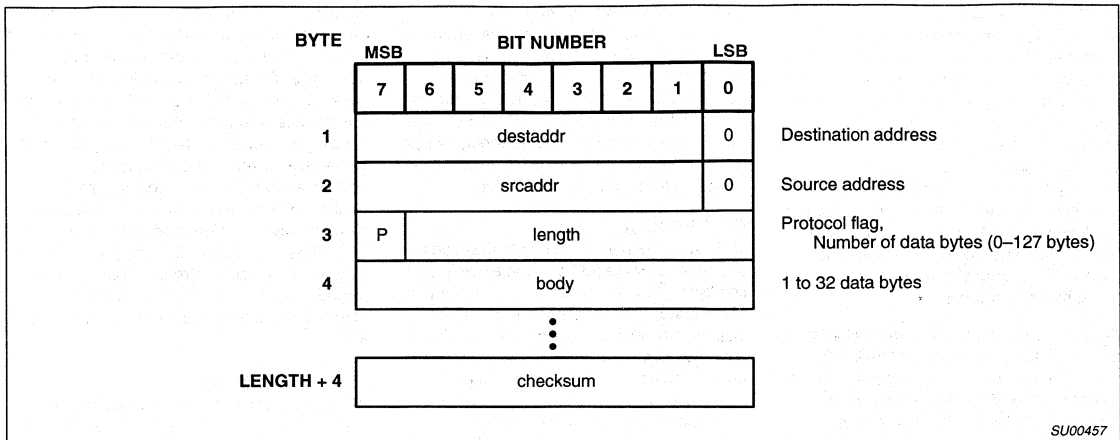


Figure 8. ACCESS.bus Big Endian bit ordering

Table 1. Interface Part Control/Status Messages

Computer-to-Device Messages Purpose

Reset()	Force device to power-up state and default I ² C address.
Identification Request()	Ask device for its "identification string."
Assign Address(ID string, new addr)	Ask device for its "identification string" to change its address to "new address."
Capabilities Request(offset)	Ask device to send the fragment of its capabilities information that starts at "offset."
Attention(status)	Inform computer that a device has finished its power-up/reset test and needs to be configured; "status" shall be the test result.
Identification Reply(ID string)	Reply to Identification Request with device's unique "identification string."
Capabilities Reply(offset, data frag)	Reply to Capabilities Request with "data fragment," a fragment of the device's capabilities string; the computer uses "offset" to reassemble fragments.
Interface Error()	Invalid checksum or premature end of message detected.

In addition, the Application Protocols define further control/status messages that are specific to particular device types. Some of these are discussed in Section 3 on the predefined device types.

AppHardSiga	Provision for a subdevice to generate an interrupt of the host system
AppTest	A message from the host to a peripheral device commanding it to test the Application Part subdevice
AppTestReply	A message by which a device replies to an AppTest command.

Configuration

The ACCESS.bus features auto-addressing and hot-plugging. These features are supported by the ACCESS.bus configuration process, which uses the seven types of control/status messages. Configuration consists of assigning unique bus addresses to the attached devices and connecting them with the appropriate drivers to provide host-resident application programs with access to the devices.

Configuration occurs when the device is powered-up or when it receives a Reset message. When the system is powered up, so are devices attached to the ACCESS.bus. Otherwise, devices are powered-up when they are hot-plugged into the bus. A device

may have a power source other than the +5V power line of the ACCESS.bus, but is must also be able to sense this line voltage and enter the power-up state when attached to the ACCESS.bus power source. When the system completes its boot up, the host sends a Reset message to all legal device addresses to put all devices in the power-up state.

Usually, when a device is powered up, it performs its specific self-testing. At the conclusion of self-test the device must assume the default address 6Eh and send an Attention message to announce its presence to the host. This message contains a single status byte to inform the host of the results of the power-up self-testing; zero indicates normal results and non-zero values indicate exception conditions that are specific to the device type.

On receiving an Attention message, the host sends an IDRequest message to the default address. Each device at this address replies with an IDReply message containing a unique 28-byte ID string described in the next section. The host is then able to assign unique ACCESS.bus addresses to each of the devices at the default address, by sending AssignAddress messages to the default address. Each AssignAddress message contains in its body the assigned address and the unique ID string of the corresponding device.

SU00457

ACCESS.bus™ technical overview

Device Identifiers

During configuration, before address assignment, each device can be identified by a 28-byte unique ID string, which may be partly or entirely encoded in the device ROM. The first 24 bytes of the ID string are understood as ASCII-encoded information characterizing the device type:

- protocol revision 1 byte ("A")
- module revision 7 bytes (e.g., "V1.0 ")
- vendor name 8 bytes (e.g., "DEC ")
- module name 8 bytes (e.g., "LK501 ")
- device number 32-bit signed integer

The first 24 bytes characterize the device's firmware and are encoded in the device ROM.

The remaining 4 bytes of the device ID string are understood as a 32-bit two's complement integer that uniquely identifies the device among devices of the same type. This integer may be provided as a unique serial number contained in the device ROM. Or, in the absence of such a serial number, interactive devices may use a random or arbitrarily determined number for this part of the ID string. As an aid to the host software, the Base Protocol specifies that, in the IDReply message, unique serial numbers be sent as positive integers and randomly generated numbers be sent as negative integers, in the two's complement sense.

In the random number case, it is possible that different devices of the same type may come up with the same 32-bit discriminator. In this case, the different devices will be assigned the same bus address, an undesirable situation. The ACCESS.bus specification suggests a guideline to help avoid identical random identifiers: use the number of cycles of the device's own clock between power up and the time the IDRequest message is received. The natural dispersion of the frequencies of these oscillators is likely to provide unique numbers.

The Base Protocol includes a provision to ensure against the unlikely circumstance that different interactive devices of the same type and without unique serial numbers will generate the same random number. Namely, each such device must send a Reset message to its own assigned address. This self-addressed Reset is sent only once between power-ups or external Resets, just before the device sends the first message instigated by a user action. Of course, the transmitting device will ignore the self-addressed Reset, but other devices

possibly at the same address will be reset and will go through configuration again. The Base Protocol specifies that a device using a random number in its ID string shall change that random number after receiving a Reset message. In this way, all interactive devices are guaranteed assignment of unique addresses before sending their first data-carrying messages.

If several non-interactive devices of the same type are to be attached to a single ACCESS.bus, then they must have hardwired unique serial numbers.

During normal operation, the host periodically checks the configuration by sending IDRequest messages to inactive devices. The host also sends IDRequest messages to all assigned addresses whenever it receives an Attention message from a device seeking configuration. The purpose of these IDRequest messages is to verify the current state of the ACCESS.bus – what devices are still connected and which devices are no longer present.

Device Capabilities Information

In order that a device be accessible to application programs running on the host, it must be connected to an appropriate software driver. Establishing this association is the last phase of configuration.

The appropriate driver will depend on the device type. There may be further parameters that characterize the device and which affect the choice of driver, or which at least must be furnished as arguments to the selected driver. Moreover, the application program may also need to be informed of these device parameters. The device Capabilities Information feature of the ACCESS.bus protocol allows a measure of device independence in the selection of drivers and provides for informing the host software of the device characteristics.

Device Capabilities Information is an explicit statement of a device's functional characteristics that are only implicit in the device type designation contained in the ID string, or that may even vary among individual devices of a given type. For example, the Capabilities Information about a keyboard might include the national alphabet used, or the Capabilities Information about a locator might include its resolution or units.

The Capabilities Information for each device is contained in a single human-readable ASCII-encoded text string stored on the device ROM. The ACCESS.bus Base Protocol defines a simple and compact grammar for building the capabilities string.

The semantics of the Capabilities Information is carried by keywords. The Base Protocol defines some keywords that can apply to all sorts of devices. Then each Application Protocol will define further keywords that are meaningful only for certain types of devices. To date, the ACCESS.bus Application Protocols define semantics for the Capabilities Information for generic keyboards, locators, and general text devices. The grammar allows for easy extension of the Capabilities Information specification.

An example of a simple mouse Capabilities string might be as follows:

```
(
  prot(locator)
  type(mouse)
  model(VSXXX)
  buttons(1(L)2(R)3(M)) dim(2) rel
  res(200 inch) range(-127 127)
  d0(dname(X))
  d1(dname(Y))
)
```

This would specify that the device uses the standard locator protocol; that it is a mouse, that it is model VSXXX; that it has three buttons designated left, middle and right corresponding with the respective bits in the keyswitch word; that it has two degrees of freedom, designated "X" and "Y", using inches as units with 200 counts per inch; that it reports relative values (displacement since last report) in the range -127 to 127 counts, and that its coordinate values correspond to X and Y.

After assigning a unique address to a device, the host sends it a CapRequest message to command it to send its Capabilities Information in a CapReply message. Of course, the device Capabilities string may well exceed the capacity of the message body of each CapRequest specifies where in the Capabilities Information string the fragment should start. The 'offset' value is repeated in the CapReply message, to be used as a check by the host in reassembling the Capabilities string. The offset is restricted to three values: send first (0), send again, and send next (offset from most recent string plus the number of bytes in the fragment).

The prot(), type(), and model() keywords must appear in that order and occur first in the Capabilities string. They must be within the first 128 bytes.

ACCESS.bus™ technical overview

APPLICATION DEVICE TYPES

The initial ACCESS.bus specification defines Application Protocols for three kinds of devices: keyboards, locators, and text devices. An important advantage of developing devices that conform to these defined protocols is the availability of pre-existing software to implement them, in particular, the drivers in the operating software of the host system through which application programs gain access to the devices.

Keyboard Devices

A generic keyboard consists of an array of key stations assigned numbers between 8 and 255. When any key station transitions between open and closed, the entire list of key stations currently closed or depressed is transmitted to the host.

In addition to reporting key stations, the generic keyboard device can support simple feedback mechanisms such as keyclicks, bells, and light-emitting diodes. These mechanisms are controlled explicitly from the host so that minimal keyboard state modeling is required. The keyboard mapping table can also be stored in the keyboard itself as part of the capabilities string.

Each key is assigned a unique 8-bit number (8-255). The first 8 codes are reserved for other keyboard functions. On each key transition, up or down, the keyboard will report the complete state of the key array as a list of zero to ten key stations that are currently down.

Example: user enters the modified keystroke Alt-Shift-A

Transition	Report
Alt down	Alt
Shift down	Alt Shift
'A' down	Alt Shift A
'A' up	Alt Shift
Shift up	Alt
Alt up	<empty list>

This reporting scheme is functionally complete in that the host can detect every key transition and it provides the full state of the keyboard on each report. No special resynchronization reports are needed.

More detailed information can be found in the ACCESS.bus Keyboard Device Protocol Specification.

Locator Devices

The *ACCESS.bus Locator Device Specification* provides for a device that has up to 15 degrees of freedom (with 16-bit precision) and up to 16 binary keys or buttons. Thus, in addition to such conventional pointer/locator devices as mouse, tablet, trackball. The ACCESS.bus locator protocol is suitable for valuator sets, such as dial boxes, and function key boxes with up to 16 function keys.

The locator capabilities information provides for specifying the number of switches and their designations (for example, "left", "right", "middle", etc.), whether the locator values are relative (like a mouse) or absolute (like a tablet or dial box), the resolution (counts per unit, and units), the dynamic range, and the names of the locator axes (for example, "x", "y", etc.).

The first 2-byte word of the event report message body contains a mask giving the state of the switches; the remaining words contain the value of each of the locator axes, either the absolute values or the change since the previous report in the case of relative devices. The locator report message is sent either on a regular sampling interval or on receipt of an AppPoll message from the host.

The locator-specific Application Protocol control/status messages are from the host to the device:

- AppPoll
Requests the device to report its state
- AppSamplingInterval
Sets the device sampling interval or instructs the device to report only when polled

More detailed information can be found in the ACCESS.bus Locator Device Protocol Specification.

Text Devices

The text device protocol is intended to provide a simple way to transmit character or binary data to and from stream oriented devices such as a bar code reader, or character display. The sequential character stream model also serves as a common denominator for connecting RS-232 interfaced devices.

A generic text device transmits a stream of 8-bit bytes from a character set. Simple control messages are defined to support flow control and to select communication parameters that might be used to interface with a modem. The capabilities string

contains information that identifies the specific character set and communication parameters used.

More detailed information can be found in the ACCESS.bus Text Device Protocol Specification.

TIMING RULES

To ensure good interactive response and to ensure that all devices will have access to the bus, the ACCESS.bus specifies rules on transaction timing. Further, specific timeouts are needed to avoid hanging up the interconnect when devices fail or are removed.

Transaction Timing Rules

The ACCESS.bus is designed primarily for interactive devices. A basic objective of the definition of the ACCESS.bus specification is that every interactive device should be able to update the host on its state at least once in every display video frame time. To help meet this criterion, the Base Protocol imposes some rules on the timing of a device's interaction with the bus.

Response Timeouts

In order that a dead or unplugged device will not hang up the system indefinitely, there must be time limits for responding to commands that require a response. The ACCESS.bus protocol specifies that devices shall complete the Reset command within 250ms. Further, a device shall respond to any command requiring a response within 40ms, or, in the case of commands that can be answered by several devices, within 40ms after the last device to respond.

SOFTWARE ARCHITECTURE AND DEVELOPMENT

An ACCESS.bus peripheral requires software at both ends of the bus transaction for managing all levels of the peripheral interaction: the I²C interface, the ACCESS.bus Base Protocol, and the ACCESS.bus Application Protocol. Further, the peripheral device requires software to support communication between the device microcontroller and the application-specific I/O transducer circuitry. Finally, the host system operating software must provide interfaces by which application programs can access both the ACCESS.bus devices and the ACCESS.bus itself. An important advantage of the ACCESS.bus approach is that the lower levels of the interaction are common to diverse device types, so they can be supported by the same or similar software modules.

ACCESS.bus™ technical overview

Device Firmware Development

The microcontroller in the device provides the intelligence for managing the device's participation in all the levels of the ACCESS.bus protocol. Use of the components with hardware I²C interface functionality, described in "MICROCONTROLLERS", can simplify the development of the lowest level of this software. Moreover, because they concern only the bus communication methods that are common to all sorts of peripheral devices, the I²C interface and the ACCESS.bus Base Protocol may well be implemented by reusing software previously developed for some of these components. And devices conforming to the semantics of the predefined standard Application Protocols may also benefit from the availability of some off-the-shelf code at the top level.

Of course, each device vendor will have to develop substantial firmware specific to his or her device. Philips and several third party vendors offer a range of tools to support firmware development for the standard components of the 80C51 family. These tools include cross assemblers and cross compilers for C and PL/M, in-circuit emulators with symbolic debugging and real-time trace support, and EPROM programming equipment. Generally, these software and hardware tools are for use with PC-compatibles as the development platforms.

Host Software Architecture

Vendors of host systems supporting ACCESS.bus will have to supply drivers and other kernel modules to provide access to the ACCESS.bus port, both for application program clients and for other system software, such as the interactive I/O handlers of the window system.

DEVELOPMENT SUPPORT

Both Digital and Philips Semiconductors offer technical support and assistance to developers of ACCESS.bus devices and host systems.

ACCESS.bus Industry Group

The ACCESS.bus Industry Group (ABIG) is an association of members interested in

promoting ACCESS.bus as an industry standard for the desktop connectivity of computer peripherals. As an association, ABIG is intended to maintain the ACCESS.bus specification as a simple, easy-to-implement, stable technology in the spirit of its design and contribute to the technical longevity of the ACCESS.bus architecture.

ABIG is an open industry group and anyone who has an interest in ACCESS.bus can be an ABIG member. An ABIG member is defined as a company including its divisions and subsidiaries, an organization, or a public or private institution.

There are two basic types of membership; General and Voting. General membership is open to everyone and Voting Membership is restricted to only those members who are actively developing a device or platform that incorporates the ACCESS.bus technology. ABIG is governed by an elected Steering committee of seven voting members.

ABIG Founding Members are those voting members who joined to form ABIG. They have the same voting privileges. The ABIG Founding Members are:

- AMP, Inc.
- Ceibo, Ltd.
- Computer Access Technology Corp.
- Digital Equipment Corp.
- Discrete Time Systems Corp.
- Honeywell Keyboard Division
- Input Technologies, Inc.
- ITAC Systems, Inc.
- Kensington Microware Limited
- Lexmark International, Inc.
- Logitech, Inc.
- Micro Computer Control Corp.
- Molex Inc.
- Mouse Systems Corp.
- New Idea Electronic Co., Ltd.
- Nexus Applied Research, Inc.
- Penny & Giles Computer Products Ltd.
- Philips Semiconductors
- Robert Clemens Research & Development
- Summagraphics Corp.
- Sun International, Inc.
- Welch Allyn, Inc.

For more details on ABIG, please contact ABIG directly at:

ACCESS.bus Industry Group
370 Altair Way, suite 215
Sunnyvale, California 94086
Telephone: 408-991-3517
FAX: 408-991-3773

Philips Semiconductors Support

An ACCESS.bus specification is available from Philips Semiconductors. This kit includes the complete specifications for the ACCESS.bus Base Protocol and pre-defined Application Protocols, as well as the Philips Data Handbook containing the detailed specification of the I²C bus, characteristics of the available integrated circuits which support it, application notes, and sample firmware code. The Data Handbooks also contain listing of development systems and third-party products supporting microcontroller firmware development, mentioned in the section entitled "Device Firmware Development".

Beside the 80C51-family microcontrollers, Philips and other manufacturers offer over 100 different components with built-in I²C support: memories, display controllers, data converters, clock/calendars, voice synthesizers, video processors, and others.

For technical questions on I²C call your local Philips Semiconductors sales office, or contact the Philips' Headquarters Application Group at [1]408-991-3518.



Purchase of Philips' I²C components conveys a license under the Philips' I²C patent to use the components in the I²C-system provided the system conforms to the I²C specifications defined by Philips.



ACCESS.bus Development Kit

A.b-DEV-KIT

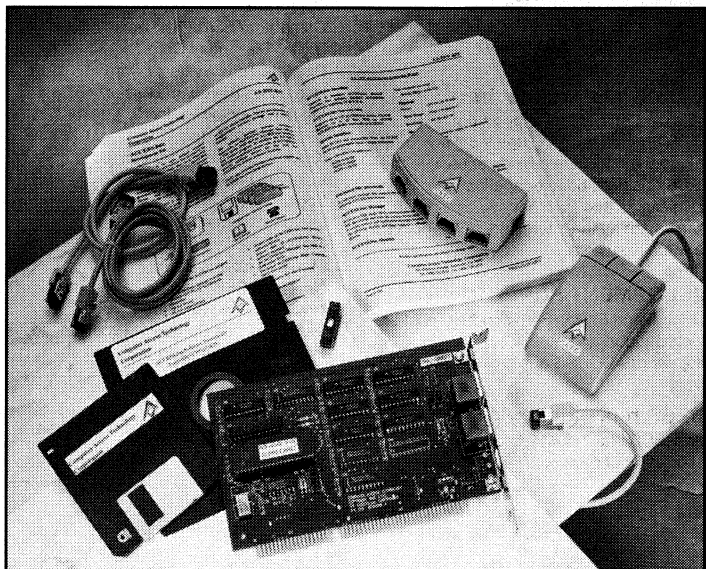
ACCESS.bus is an open industry standard providing a simple and uniform way to connect up to 125 devices to a single port on a computer. ACCESS.bus features 100,000 bits per second data rate, hardware arbitration, dynamic reconfiguration, a comprehensive capabilities grammar to support generic software device drivers, and off-the-shelf, low-cost I2C Microcontroller technology.

The A.b-DEV-KIT is an ACCESS.bus product development support package. It includes CATC's PC/AT A.b-125I ACCESS.bus controller board, an ACCESS.bus mouse, expansion box and cables and an 87C751 Microcontroller. The kit also includes a comprehensive software package, a user's manual and Hotline telephone support.

The software package includes on-board microcode, an ACCESS.bus Manager that runs as a TSR under DOS and an ACCESS.bus Monitor and Control program. In addition the A.b-DEV-KIT software includes source code of ACCESS.bus generic software drivers for the host and ACCESS.bus devices software modules.

Features

- Complies fully with the ACCESS.bus standard
- A comprehensive A.b user's manual
- Hotline telephone support
- ACCESS.bus hardware package including -
 - The A.b-125I PC/AT ACCESS.bus Controller
 - An A.b Mouse
 - An A.b expansion box
 - A.b cables (2 ft and 4 ft)
 - A Philips 87C751 Microcontroller
- A comprehensive software package including -
 - On-board ACCESS.bus Main Controller (MC) microcode
 - An ACCESS.bus Manager that runs as a TSR under DOS
 - An ACCESS.bus Monitor and Control program
 - Source code of an ACCESS.bus host generic software driver
 - Source code of a generic ACCESS.bus application layer software
 - Source code of ACCESS.bus devices software modules



The A.b-DEV-KIT includes ACCESS.bus accessories and comprehensive software

A.b-125I ACCESS.bus Controller Board

ACCESS.bus Interface

Controls a standard ACCESS.bus network. Provides two industry standard ACCESS.bus connectors, supplying 5V @ 0.75 A.

ACCESS.bus Network Size

Supports up to 125 ACCESS.bus devices. Physical distance up to 25 feet. With an external ACCESS.bus Buffer (optional) up to 250 feet.

System Interface

IBM PC/AT and compatibles. Uses the PC/AT 16-bit programmable input / output mechanism:

User selectable I/O addresses -

0x250 to 0x25F
0x260 to 0x26F
0x350 to 0x35F

User selectable interrupt -

IRQ10, IRQ11 or IRQ12

Physical

Power: +5V DC, 10 W max.

Temp. Range: +0 to +50 degree C

Board Size: 4.2" H x 6.5" W.

Warranty

90-day. Return to factory for repair or replacement at manufacturers option

A.b-DEV-KIT Software

On-board MC microcode

A real-time package that controls the operation of physical ACCESS.bus devices.

ACCESS.bus Manager

Runs as a TSR under DOS, communicates with the MC microcode and with the various device drivers. It routes control and application messages between the physical devices and their respective software drivers.

ACCESS.bus Monitor

A user-friendly, menu-driven program, displays user-selected ACCESS.bus messages and allows the user to control specific devices.

Source Code License

Source code of a generic ACCESS.bus software driver for the host. Source code of a generic ACCESS.bus application layer software. Source code of ACCESS.bus physical devices software modules.

Diagnostics

A comprehensive self test is performed on the board on power up. Diagnostics are run under control of the ACCESS.bus Monitor.

Product specifications are subject to change without notice

Computer Access Technology Corporation

949 Hillsboro Avenue, Sunnyvale, CA 94087

Tel: (408) 732 8910 Fax: (408) 730 1675



ACCESS.bus PC/AT Controller Board

A.b-125I

Description

ACCESS.bus is an open industry standard providing a simple and uniform way to connect up to 125 devices to a single computer port. ACCESS.bus features data rate of 100,000 bits per second, hardware arbitration, dynamic reconfiguration, a comprehensive capabilities grammar to support generic software device drivers, and off-the-shelf, low-cost I2C microcontroller technology.

The A.b-125I is a PC/AT adapter board that serves as an ACCESS.bus master. It allows the connection of multiple devices to a single port on the PC. The board can be used in Desktop connectivity as well as in Control and Instrumentation applications. The board is based on a Philips 8xC654 microcontroller with I2C interface.

The A.b-125I is offered with a comprehensive software package including on-board microcode and an ACCESS.bus Manager that runs as a TSR under DOS.

Features

- A highly integrated, half board design, uses a single 16-bit AT/ISA slot
- Full compliance with the ACCESS.bus standard
- On-board 8K bytes SRAM buffer memory
- A comprehensive software package including -
 - An on-board ACCESS.bus Main Controller (MC) microcode
 - An ACCESS.bus Manager that runs as a TSR under DOS



**The A.b-125 allows the connection of multiple devices
to a single port on the PC**

ACCESS.bus Interface

Controls a standard ACCESS.bus network. Provides two industry standard ACCESS.bus connectors, supplying 5V @ 0.75 A.

ACCESS.bus Network Size

Supports up to 125 ACCESS.bus devices. Physical distance up to 25 feet. With an external ACCESS.bus Buffer (optional) up to 250 feet.

Buffer Memory

8K x 8 bits (8K bytes) static RAM,

System Interface

IBM PC/AT and compatibles. Uses the PC/AT 16-bit programmable input / output mechanism:

User selectable I/O addresses -

- 0x250 to 0x25F
- 0x260 to 0x26F
- 0x350 to 0x35F

User selectable interrupt -

- IRQ10, IRQ11 or IRQ12

Software

A comprehensive software package is provided with the board. The software includes the on-board ACCESS.bus Main Controller (MC) microcode and the ACCESS.bus Manager that runs as a TSR under DOS.

CATC has additional ACCESS.bus software available including a Windows 3.1 version of the ACCESS.bus Manager, software device drivers, bus monitoring and control program and development tools. Call CATC for additional information.

Diagnostics

A comprehensive self test is performed on the board on power up.

Physical

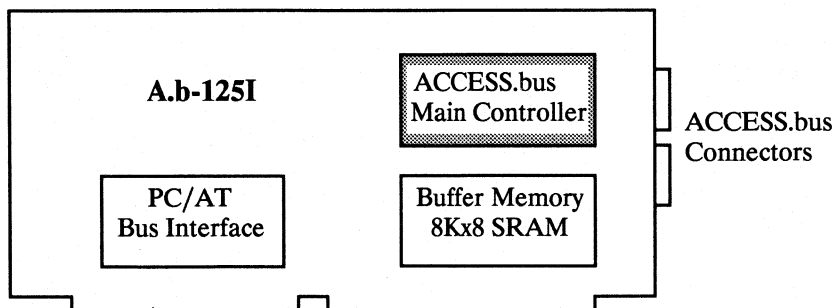
Power: +5V DC, 10 W max.

Temp. Range: +15 to +50 degree C

Board Size: 4.2" H x 6.5" W.

Warranty

90-day. Return to factory for repair or replacement at manufacturers option



Product specifications are subject to change without notice

Computer Access Technology Corporation

949 Hillsboro Avenue, Sunnyvale, CA 94087
Tel: (408) 732 8910 Fax: (408) 730 1675

Section 5

ACCESS.bus Application Notes & Articles

Application Notes and
Development Tools for
80C51 Microcontrollers

CONTENTS

AN445	ACCESS.bus mouse application code for the 8XC751 microcontroller	See Section 7
	Issues in desktop connectivity	291
	Finally, a plug-and-play solution	295
	Special Report: ACCESS.bus Specs And Products	297
	ACCESS.bus: A New Peripheral Bus	299
	Embedded control using ACCESS.bus	301
	A PC-to-ACCESS.bus interface card	309
	Taking a new bus	312
	Personal Digital Assistants: What's missing?	318
	The portable desktop: New connections for today's mobile user	320
	Who's hopping on the ACCESS.bus?	322
	ACCESS.bus revisited—ending the peripheral connection nightmare	324
	Seriously serial	327

Issues in desktop connectivity

Authors: Ata Khan and Greg Goodhue, Sunnyvale

INTRODUCTION

Desktop connectivity has become a major issue in system design. This paper identifies the criteria for evaluating a desktop bus or interconnect from data transfer rates and protocol flexibility to ease of implementation. It also compares the performance and cost implications of the typical desktop connectivity alternative.

DESKTOP CONNECTIVITY

Desktop connectivity is a method of connecting computer peripherals that are most often found on work surfaces, or desktops, to a host computer or workstation. Most often, desktop peripherals are low-speed Input/Output devices such as keyboards, mice, tablets, joysticks, and modems. This paper discusses devices which are usually dedicated to one computer, such as a PC, rather than high-speed peripherals such as laser printers and disk drives that are shared between users.

INTEREST IN DESKTOP CONNECTIVITY

Current interest in desktop connectivity comes from both users and manufacturers of PCs and workstations.

From the User's Side

Connecting low-speed I/O devices such as keyboards, mice, tablets, modems, and low-speed printers to a PC or a workstation has resulted in two problems for the user: a shortage of ports, and an excess of cabling.

Shortage of ports is a big problem, especially with computers that have few slots or none at all. A computer with two serial ports, a parallel port, and a keyboard port is soon overwhelmed by peripherals requiring external expanders or switches of some sort. In the worst case, the user may be unable to use all the available peripherals at one time. Newer notebook computers exacerbate this problem by having fewer ports than desktop machines.

An excess of cabling and connectors is one of the bane of connecting several desktop peripherals. By the time a user has figured out all the right cables, connectors, and gender changers, there is usually a small jungle on the desktop. From the user's point of view, there must be a better way.

From the Manufacturer's Side

Providing all the ports to which low-speed desktop I/O devices may be connected is expensive. Besides the actual hardware involved, valuable motherboard real estate is

used and, especially in compact machines such as notebooks or pen-based computers, it is tricky to provide a lot of external connectors in very limited space. Also, the number of ports provided is usually either too few or too many, usually the former.

Providing ports in hardware means connecting them to the CPU on the motherboard. Usually, this means each port must be provided with its own interrupt and dedicated address range. With systems, such as IBM type PCs, already being under severe constraints with regard to the number of I/O interrupts, adding sufficient ports for desktop use may become a problem.

The CPU is the de facto manager of all these low-speed desktop peripherals; this means that signals from these slow devices are now moving on the same channels that high-speed devices such as disks are connected. Allowing bicycles on the *Autobahn* is a good analogy to the current system.

Since each device has its own port, it also has its own device driver that "connects" the application software to the peripheral. This means that device drivers are hardware specific and different for each peripheral connected.

Reconfiguring devices without powering down a system is a major advantage for both manufacturers and users, as anybody who has ever powered off a networked UNIX workstation knows. If it were possible to add and remove peripherals dynamically, such as different keyboards or tablets, without reconfiguring or re-booting the system, this would be a major operational advantage.

CRITERIA FOR SELECTING A DESKTOP BUS

Ideal criteria for selecting a desktop bus are outlined below:

- Low-cost: Off-the-shelf, commodity type components a a minimum—ideally, just one—of these.
- Daisy-chained: By allowing components to connect into each other, both of the problems the user has with a shortage of ports and an excess of cables are solved.
- Dynamic reconfiguration: Hot-plugging and un-plugging of devices should be allowed without having to power down the system or re-boot it. This allows peripherals to be added, removed or defective ones swapped without long delays and inconvenience.
- Uniform interfaces: With daisy-chained devices, the hardware interface is uniform

by definition. This allows the same hardware layer drivers to be used for all peripherals. Software uniformity is also ideal but harder to implement and can, to some extent, be achieved through a layered protocol.

- Sufficient bandwidth: While devices such as keyboards, mice, and tablets usually do not present much of a problem, devices such as modems and printers may. System simulation should be carried out to see that the chosen bandwidth meets these needs without causing excessive delays, lost data, or other problems.
- Sufficient capacity: At the very minimum, six devices should be allowed (keyboard, mouse, tablet, modem, printer, misc.) and the cable specification should support this minimum.
- Inexpensive peripherals: Peripherals should be available that support this standard and are not significantly more expensive, if at all, than those available for other standards.
- Open standard: There should be no royalties, patent issues, or licensing fees associated with the use of a desktop connectivity standard.

CURRENT IMPLEMENTATIONS OF DESKTOP CONNECTIVITY

Three major platforms implement desktop connectivity standards: IBM and compatible PCs, workstations, and Macintoshes.

The PC platform and workstation platforms use dedicated I/O ports (RS-232 or other) for each peripheral being connected. Thus, for an IBM-style PC, each desktop peripheral has to have a port on the machine: one for the keyboard, one for the mouse, one for the tablet, etc. This gives rise to an onerous proliferation of cabling, connectors, and driver software where space is at a premium. In IBM-style notebook computers, there is usually a shortage of ports caused by limitations of space as well as power. The current solution is cumbersome, expensive to implement, and inelegant.

Workstations use basically the same approach to desktop connectivity. However, there is some saving grace here that restrictions on space, power, and cost are less severe than PCs.

The Macintosh™ solution, known as the Apple Desktop Bus™, is better than the PC and workstation solutions in the sense that it is more efficient in its use of space since peripherals can be daisy-chained and peripheral does not require a dedicated port.

Issues in desktop connectivity

ACCESS.bus™ : A PROPOSED DESKTOP CONNECTIVITY STANDARD

A desktop connectivity system for interconnecting low-speed peripherals was originally developed by Digital Equipment Corporation (Digital) in partnership with Philips Semiconductors and offered as an open standard. Called ACCESS.bus (a bus for connecting ACCESSory devices to a host system), the standard embodies most of the criteria of an ideal desktop connectivity standard.

ACCESS.bus (also referred to as A.b) is a daisy-chained bus (see Figure 1) which allows up to fourteen devices (there are provisions for these devices themselves controlling other devices for expansion.) The total length of the cable is allowed to be eight meters and the data throughput rate of this bus is approximately 80 kbits/sec.

ACCESS.bus is a fully open standard, without fee or royalty. The Advanced Computing Environment (ACE) initiative has designated A.b as an option in the Advanced RISC Computer (ARC) specification. Digital plans to organize a committee of user companies and A.b user groups to support the standard.

ACCESS.bus Structure

A.b is a layered protocol supporting a daisy-chained bus topology. There are three layers in the protocol (see Figure 2):

1. The Hardware (I²C) Protocol Layer:
This layer is based on the Inter-Integrated Circuit, or I²C, serial protocol developed by Philips. This protocol defines a scheme for performing bus transactions with addressing, framing of bits into bytes, and acknowledgement of each byte by the receiver.
2. The Base Protocol Layer:
This level is common to all A.b devices and builds on the Hardware layer to establish an asymmetric interconnect between a host computer and a number of peripheral devices. The A.b message envelope with control and status messages is defined here.
3. The Application Protocol Layer:
In this layer, devices are differentiated with message semantics that are specific to particular kinds and classes of devices.

In short, the mailman is I²C, the mail envelope is the Base Protocol, and the contents of the message are the Application Protocol.

The Hardware Protocol (I²C)

The Philips Inter-Integrated Circuit protocol, or I²C, is a 2-wire (clock and data) serial protocol which allows wire-AND connection of devices to the clock and data lines. The protocol allows devices to be either masters or slaves at any given bus transaction time (masters control the transaction and generate the clock signal).

I²C is a symmetric multimaster bus where several masters can contend for the bus and an arbitration scheme resolves bus mastership without loss of data or re-transmission. Different clock rates are allowed on the bus with a cooperative synchronization scheme for the serial clock; this allows bus transactions to be performed between bus devices with different clock rates and without requiring any clock locking schemes.

Device Connection: All devices are wire-AND'ed identically to the clock (SCL) and data (SDA) lines. Outputs must be open-drain or open-collector. Thus, any device may pull the bus low. When neither line is pulled low, the lines are held high by pull-up resistors. Every device must be able to sense the state of the line. In practice, this means a common ground level is required so that all devices see more or less the same input threshold.

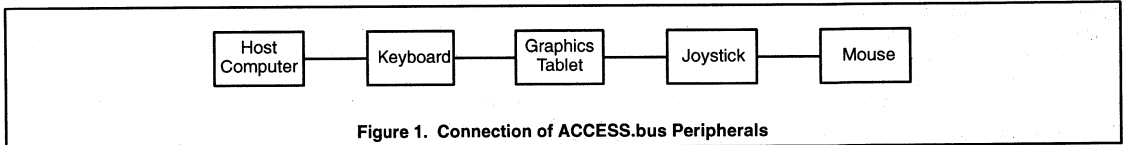


Figure 1. Connection of ACCESS.bus Peripherals

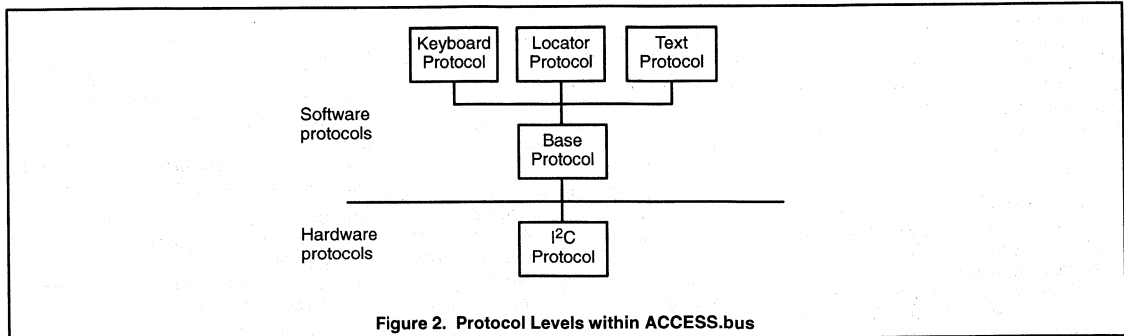


Figure 2. Protocol Levels within ACCESS.bus

Issues in desktop connectivity

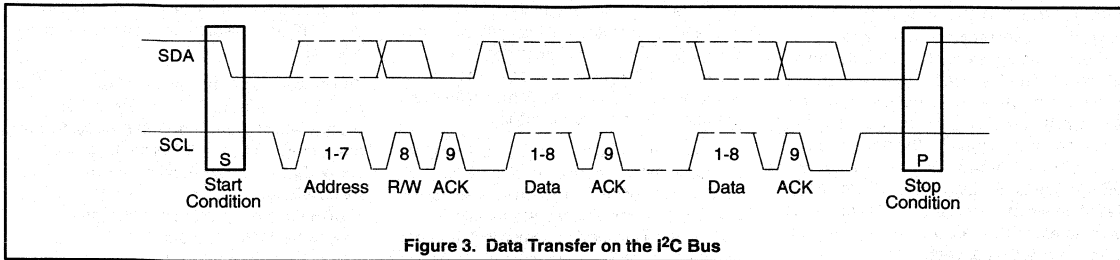


Figure 3. Data Transfer on the I²C Bus

Bus Transaction: The bus is idle when both SDA and SCL lines are high after a transaction has been completed or after power-up. To initiate a transaction, a master device pulls the SDA line low. This represents a Start condition on the bus and all devices are required to sense its occurrence. Once the Start condition is asserted, the master then takes the SCL line low, and starts putting data on the SDA line by driving it low or leaving it high and pulsing the SCL line high and then back low. Data (SDA) is not allowed to change while the clock (SCL) line is high (see Figure 3).

The master outputs the Address of the slave as the first byte transmitted (MSB first). After 8 bits have been transmitted, the slave, if one exists and recognizes its address, will pull the Data line low on the 9th clock bit time. This serves as an acknowledgement to the Master that the address has been received by the slave. A non-acknowledgment will cause the transfer to be aborted.

Bus Synchronization: I²C provides a mechanism of synchronization that allows devices with different clock rates to work seamlessly with no required phase relationships between device clocks.

When SCL is low, all devices (both masters and slaves) pull their own SCL outputs low and count out their low clock periods; when the low period expires, the devices release their SCL outputs. When SCL finally goes high, master devices count out their high clock periods until the master with the shortest clock period pulls SCL low again. Thus, the SCL low period is defined by the device with the longest low period and its high period by the device with the shortest high period.

This cooperative synchronization allows all devices to use a common clock and allows slower peripherals to regulate the speed of the bus.

Multimaster Arbitration: If two or more masters start transmitting on the bus, given that the clock is synchronized, each master then examines the state of SDA to see if its

state is the same as the value of the bit it transmitted. The instant a master sees a difference, it knows it has lost arbitration and terminates its attempted transaction. Thus, arbitration between masters sending out different bus address values will always be uniquely resolved within the address transmission time. If masters send exactly the same message with the same contents to the same address, then the message is sent, but in cases where this is not desirable, software interlocks can prevent this situation.

Since the arbitration is performed on a bit-by-bit basis, a form of priority is implied here where lower addresses (addresses with the first occurrence of a zero) have priority over addresses with ones in the same bit location (higher addresses).

The Base Protocol

The Base Protocol defines a number of control and status Messages that are common to all A.b peripherals. These are used for the configuration process in which peripherals are recognized and assigned unique address identifiers and then connected with appropriate device drivers to enable the application program to talk to them.

A Message has five parts to it:

- i. The first byte is the address of the destination or the receiver.
- ii. The second byte is the address of the source or the transmitter.
- iii. The third byte specifies whether the body of the message is control or data, if there are any sub-devices (0 to 3), and the length of the message body following in bytes.
- iv. This part is the Message body, from 1 to 32 bytes.
- v. A Checksum. This byte is the bit-wise XOR of all the preceding bytes in the message.

While the detailed syntax and structure of the messages will not be discussed here, there are seven basic Messages defined as follows:

Computer (Host) to Device:

Reset	Force device to Power-up state and default address
IdRequest	Ask device for its identification string (ID)
AssignAddress	Give device with recognized ID its unique bus address
CapRequest	Ask device to send Capability information

Device to Computer (Host):

Attention	Inform of device presence and result of Power-up test
IdReply	Send device ID string to host
CapReply	Send Capability information to host

The above features allow auto-addressing (eliminating hardwired addresses, jumpers, and switches) and hot-plugging. In general, operation is as follows:

At Power-up, the host device transmits a general Reset to all devices (assigned the same default address at power-up). The devices initialize themselves, perform a self-test and send the result back to the host by an Attention message. On receiving an Attention message, the host sends an IDRequest message to the Default Address, each device at this address sends a unique 28-byte ID string back to the host; the IDReply. On getting the IDReply, the host is then able to assign a unique A.b address to the device.

A Hot-plugged new device will send an Attention Message to the host seeking to be assigned an address. The host periodically checks the last logged configuration by sending IDRequest messages to all inactive devices. These actions allow the bus configuration to be dynamically altered.

In this layer, the particular peripheral's device driver has to be found, loaded and connected to the application program. This is done by determining the device type and Capability Information by the CapRequest/CapReply messages.

Issues in desktop connectivity

The Application Protocol

Application level protocols are device-specific and the message semantics are different for different defined device types as well as different sub-types. While device drivers are different at this level, the Hardware and Base layers are independent of the device type allowing much firmware to be shared. Even at the Application Protocol level, a common definition structure is used for similar device types to encourage a common approach to writing A.b device drivers.

So far, devices are classified into three broad types:

- i. Keyboards:
Up to 255-key devices are supported. Special function keys and annunciator support is built-in.
- ii. Locator devices:
Intended for mice, tablets, etc. This provides for devices with up to 15 degrees of freedom and up to 16 binary keys.
- iii. Text devices:
These are defined as data stream devices such as printers, modems, etc.

Details of the semantics are in the ACCESS.bus specification. Flow Control (XON/XOFF) is provided with a character count being specifiable.

ACCESS.bus IMPLEMENTATION

The A.b controller must support all layers of the protocol for both the system and peripheral ends of the bus, the I²C hardware, the A.b Base layer, and the A.b Application layer. This is most efficiently done by using a microcontroller with an I²C interface with enough on-chip memory to implement the A.b firmware, thus implementing the entire protocol in one, off-the-shelf, commodity priced component.

Philips manufactures a broad line of 80C51-based microcontrollers with built-in I²C interfaces and varying amounts of on-chip memory ranging from 2k bytes to 32k bytes of program memory. EPROM and OTP versions are available for system development as well as production. These microcontrollers provide all the intelligence

for managing device communications for all 3 layers of the protocol. Of course, since the Base and I²C layers are common, this firmware may be reused. Even at the Application layer level, the common message structure allows some reusability.

Since the 80C51 is an industry standard for microcontrollers, there is an abundance of third party support in the form of assemblers, compilers, development systems with In-Circuit-Emulation and symbolic debugging capability, EPROM programmers, etc. These tools are usually low-cost and are almost always PC-based.

The physical connection itself is made by using a 4-pin connector that Molex and AMP will offer as a standard catalog item. The 4 wires are Power, Ground, SDA, and SCL. The shielded cable used has roughly 70 pF/meter of capacitance and up to 8 meters can be used.

ACCESS.bus VIS-A-VIS THE IDEAL AND VERSUS AN EXISTING STANDARD

Comparing A.b to the list of ideal desktop connectivity attributes, we see that it fulfills them all except for the availability of low-cost off-the-shelf peripherals.

A.b uses low-cost standard components, supports dynamic reconfiguration, is daisy-chained, and allows fourteen devices (and more via sub-devices). With a data rate of 80kbits/second, it is ample for the class of low-speed desktop peripherals for which it is intended. It presents a uniform hardware and software interface allowing re-usability of most firmware and has built-in error checking at both byte and message levels.

ACCESS.bus will be implemented in forthcoming Digital workstations and, working with the ACE initiative, Digital and Philips will be striving to make it a standard for desktop connectivity. The issue of availability of peripherals presents a circular argument (which comes first) but, at this point, a keyboard, a mouse, and a graphic tablet are being developed for ACCESS.bus by major peripheral manufacturers. ACCESS.bus

connectors are being developed by Molex and AMP and will be a standard item in their offerings.

The closest existing alternative to ACCESS.bus is Apple Computer's ADB (Apple Desktop Bus, see Table 1). ADB is a daisy-chained bus, but has the following somewhat severe limitations versus ACCESS.bus:

- ADB does not support hot-plugging or dynamic reconfiguration.
- ADB has a maximum data rate of 10kbits/sec versus A.b's 80kbits/sec.
- ADB has a 3 device limit versus 14 for A.b (and A.b can support more via sub-devices).
- ADB is a closed, proprietary bus, whereas A.b is fully open.
- ADB is specified to work over a 5 meter length, while A.b can go 8 meters.

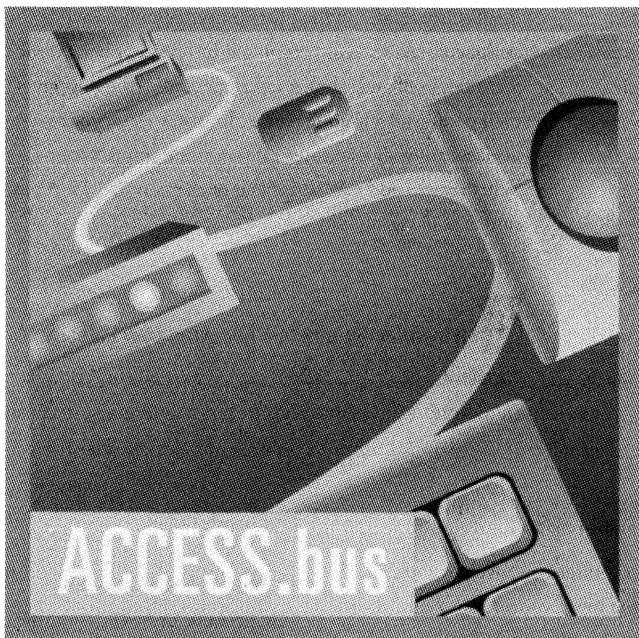
Table 1. ACCESS.bus vs. Apple Desktop Bus

FEATURE	APPLE DESKTOP BUS	ACCESS.bus
Hot plugging	Limited	Full support
Transfer rate	10kbit/sec	80kbit/sec
Maximum number of devices	3 devices	14 devices
Maximum length	5 meters	8 meters
Availability	Proprietary	Open to all

CONCLUSION

The issue of desktop connectivity for PCs and workstations has been neglected and solved piecemeal so far. ACCESS.bus represents the first coherent, well-defined, desktop connectivity standard which is solidly backed and, based on what it offers both users and manufacturers, should become established as a market standard as well.

Finally, a plug-and-play solution



With ACCESS.bus™, you can connect up to 125 peripherals to one PC or workstation port without having to reconfigure, quit your application, or reboot.

ACCESS.bus is a new industry-standard technology that provides a simple, low-cost way to connect keyboards, joysticks, tablets, high-speed modems, and other peripherals to a single port on your computer or workstation. Already supported by Microsoft DOS and Windows, Digital VMS, and Sun Solaris 2.X operating systems, ACCESS.bus is also backed by other industry leaders such as Compaq, Fujitsu, Honeywell, IBM, Intel, Key Tronic, Lexmark, Logitech, Motorola, National Semiconductor, and Philips.

Plug in your peripherals and keep going. That's it. No need to quit your application, load special software, reconfigure, or reboot—you don't even have to issue a command. In fact, because ACCESS.bus uses one port to consolidate all your peripherals, you don't even have to consider the complexities of multiple jacks, UARTs, slots, or options.

Reclaim a chunk of your office. By eliminating that unsightly cord snarl behind your computer, you'll gain more workspace. More room for other devices. More room to create.

Philips Semiconductors



PHILIPS

Imagine the possibilities. Whether you go down the hall, across the country, or around the world, you'll be able to plug your peripherals into any computer at hand. No need to hunt for compatible models. Everything will be interchanged effortlessly. And to that one computer or workstation, you'll be able to connect as many of the same types of peripherals as you need—or a broad range of them—into a single port. The potential in new applications is limitless.

In workgroup applications such as education, entertainment, and video conferencing, ACCESS.bus enables you to exchange ideas faster and more dynamically by plugging peripherals into a common monitor.

Because ACCESS.bus allows you to join in or drop out at will, computer games become multiple player games. By simply plugging into the system you can join forces or become a sudden new challenge from the opposition.

In simulation, you'll experience and learn from the decisions of your colleagues the moment the action takes place.

The possibilities are endless.



How is this technology possible? Based on ACCESS.bus microcontrollers, readily available from companies such as Philips, ACCESS.bus is supported by an industry-wide trade group. ACCESS.bus software intelligence resides in the host computer and in each peripheral. As soon as you plug it in, the peripheral device identifies itself and, almost instantaneously, the host gives each peripheral a unique name. In this way, all your devices can easily communicate over one computer line. Whether you add one peripheral or as many as 125, it's as simple as plugging it in.

Call for more information about the new industry standard. The *ACCESS.bus Industry Group*, or ABIG, is an industry-wide association formed to create and maintain the ACCESS.bus standard so tomorrow's technology can be in your hands today.

For more information about ACCESS.bus or how to become an ABIG member, call Philips at 1-800-447-1500, ext. 1080, or call ABIG at 408-991-3517 (fax 408-991-3773).

ACCESS.bus is a trademark of the ACCESS.bus Industry Group. All other product or service names mentioned herein are trademarks of their respective owners.
©Philips Electronics North America Corporation, 1993. Printed in U.S.A.

Enormous Gains for Developers

Computer vendors — Save precious space on the motherboard, especially in notebooks and other limited-space devices where the fewer external connectors required, the simpler your production and inventory. No more multiple jacks, slots, options, communications ports, or UARTs. ACCESS.bus uses only one port to consolidate the peripherals needed to run an application.

Device manufacturers — ACCESS.bus requires only one standard driver for different devices, so you save time and expense during driver development.

Systems integrators — Offer plug-and-play freedom. As technology gets more complex, you'll be able to offer simpler human interface solutions.

ACCESS.bus presents no limitation on communications ports, slots, or jacks.

Special Report: ACCESS.bus Specs And Products

EE Product News™

AN INTERTEC PUBLICATION

New Products For **Prototype Design**

The development of ACCESS.bus, a communications protocol for connecting multiple, low-speed I/O devices to a single computer port, and the formation last June of the ACCESS.bus Industry Group (ABIG) is spawning creation of a rapidly growing number of hardware and software products based on the ACCESS.bus' connectivity specs (see chart below). ACCESS.bus was jointly developed by Digital Equipment Corp. and Philips Semiconductors and is now owned and supported by ABIG, who is promoting the new bus as an industry standard. To date, up to 125 peripheral devices, including keyboards, mice, scanners, digitizers, and bar code readers have been made to operate independently on a single computer port.

At the hardware level, ACCESS.bus uses the I²C (Inter-Integrated Circuit) serial bus developed by Philips several years ago to simplify automotive electronics and other distributed control systems. This serial bus is designed to carry 1 bit of information at a time on a single data line. Today, a host of low-cost I²C components are readily available to handle the logical complications associated with bit-level handshaking.

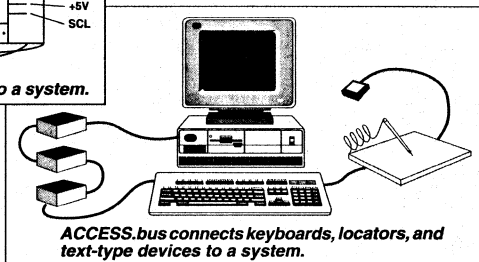
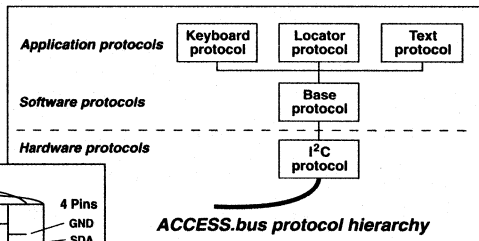
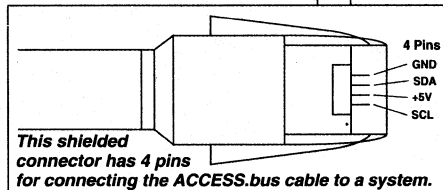
The physical medium for ACCESS.bus is a shielded cable with four wires for handling serial data (SDA), serial clock (SCL), power (5V), and ground (GND). The SDA and SCL lines work together to define information carried on the bus, and the host computer drives the 5V power line with a minimum of 50 mA to supply peripheral devices (peripherals can also be externally powered). A typical ACCESS.bus device has two connectors, permitting two or more peripherals to be daisy-chained together on the bus (handheld devices can have a captive cable joined to the bus trunk with a T-connector).

I²C technology supports clock rates up to 100 kHz and the maximum ACCESS.bus data transfer rate is approximately 80 kbits/s.

The ACCESS.bus communications protocol has three layers: I²C, Base and Applications. The I²C Protocol defines a symmetric, multi-master bus on which arbitration among contending masters is effected without losing data. I²C provides cooperative synchronization of the

serial clock for exchange of data between bus partners with different maximum clock rates, defining a bus transaction scheme with addressing, framing of bits into bytes, and byte acknowledgement by the receiver.

Base Protocol establishes an asymmetrical interconnect between host computer and peripheral devices. The host is the ACCESS.bus manager, and data communication is always between host and peripheral, never between two peripherals. While the I²C Protocol establishes mastership between the sender or receiver of a bus transaction, Base Protocol defines the format of an ACCESS.bus message envelope, which is an I²C bus transaction with



additional semantics, including checksum reliability control. Base Protocol also defines a set of seven control and status message types used in the configuration process.

The high-level Application Protocol defines message semantics specific to particular functional types of devices. To date, Application Protocol have been established for keyboards, locators and text devices and is intended to define the simplest set of functions from common, industry-standard interfaces. Further, device-specific Application Protocol models will be defined by the ACCESS.bus

Special Report: ACCESS.bus Specs And Products

Industry Group; and, of course, any vendor can implement a special device protocol within the general message envelope defined by the Base Protocol.

Electrically, host and peripheral devices are connected to serial data (SDA) and serial clock (SCL) lines in a wired-AND logic configuration, which can be implemented by connecting data and clock output stages of each bus partner to the SDA and SCL lines, respectively, through open-collector or open-drain transistors. Standard I²C components include these output stages on-chip. Significance of the wired-AND logic is that any attached bus partner can force either of these lines to LOW (GND); and when there is no output from any bus partner, lines are held HIGH by pull-up current sources in the host. Every bus partner can sense the level on both of these lines.

ACCESS.bus can be adapted to any platform and presently requires use of a controller board in the computer, but within 12 months, computer motherboards containing the necessary ACCESS.bus circuitry are expected to begin to appear. Software drivers are also available for DOS and Windows.

For additional information on ACCESS.bus v2.0 specs and on membership to ABIG, contact:

ACCESS.bus Industry Group, 415-112 N. Mary Ave., Sunnyvale, CA 94086, (408) 991-3517, FAX (408) 991-3773.

ACCESS.bus: A New Peripheral Bus

Author: Michael Burton

Midnight™ Engineering

Would you like to be able to plug a keyboard, two mice, a trackball, a modem and a printer into a single port on your PC? And then, while the computer is still powered up, unplug one of the mice and plug in a bar code reader? You can do all of this and more with ACCESS.bus.

Digital Equipment Corporation and Philips have joined forces to propose this new open desktop connectivity standard. The ACCESS.bus Industry Group (ABIG) has been formed to regulate and promote the new bus. ABIG members include DEC, Honeywell, Logitech, Philips and Sun Microsystems, to name just a few.

There are many advantages to the ACCESS.bus standard. It is low cost, dynamically reconfigurable, relatively inexpensive and the interface is uniform for all devices. ACCESS.bus is also an open standard, unlike Apple Computer's comparable Apple Data Bus (ADB).

Bus Description

ACCESS.bus allows multiple peripheral devices to be simultaneously supported on a single computer port in a daisy chain fashion, somewhat like a SCSI daisy chain. These peripherals may operate simultaneously at transfer rates of up to 125,000 baud, with a maximum data throughput rate of approximately 80,000 baud. This is ideal for low speed peripherals such as keyboards, modems, trackballs and mice.

A maximum cable length of 8 meters is permitted. A four pin, shielded rectangular connector is used on ACCESS.bus cables. The maximum amount of power available is 1 amp at 5 volts. The bus can support up to 125 peripheral devices, but the normal practical limit is 14. By way of comparison, the Apple Data Bus supports 5 meters of cable and 3 devices with a data rate of 10,000 baud.

ACCESS.bus is a layered protocol. There are three layers: the hardware protocol layer, the Base protocol layer and the Application

protocol layer. Using a postal analogy, the hardware protocol layer is the mail carrier, the Base protocol layer is the envelope and the Application protocol layer is the contents of the envelope.

The hardware protocol layer is based on the I²C (Inter-Integrated Circuit) serial protocol, which is directly supported by the Philips 8051 family of microprocessors (80CL410, 80C552, 80C652, 80C528, 87C654 and 87C751). This protocol defines a scheme for performing bus transactions, including message addressing, the framing of bits into bytes and the acknowledgment of each byte by the receiver.

The Base protocol layer is a software protocol that is common to all ACCESS.bus devices and that builds on the hardware protocol layer to establish the connection between the computer and a number of peripheral devices. The Base protocol layer specifies device power-up, identification, addressing and the message envelope for device-specific data and control information.

The Application protocol layer is a software protocol that differentiates message contents for specific kinds and classes of devices.

Hardware Protocol Layer

The hardware I²C protocol layer is a 2-wire (clock and data) serial protocol that allows wire-AND connection of peripheral devices to the clock and data lines. Any device may be either a master (controlling the transaction and generating the clock) or a slave for any given bus transaction. Several masters can contend for the bus and an arbitration scheme resolves bus mastership without data loss or retransmission. The clock rates for various peripherals may vary widely, since the bus has a cooperative serial clock synchronization scheme.

Base Protocol Layer

The Base protocol layer contains definitions for a number of control and status messages that are common to all ACCESS.bus peripherals. The messages are used for the

configuration process, where peripherals are recognized, assigned unique address identifiers and are then connected with appropriate device drivers to enable an application program to talk to them. A message has five parts to it:

1. The first byte is the address of the destination or receiver.
2. The second byte is the address of the source or transmitter.
3. The third byte specifies whether the body of the message is control or data, if there are any sub-devices (0 to 3) and the length of the message body.
4. This part is the message body, which can be from 0 to 127 bytes in length.
5. This last byte is the checksum, a bit-wise exclusive-or of all the preceding bytes in the message.

There are eight base messages, shown below.

Computer to Device Messages

1. **Reset**—Force the device to its power-up state and to its default I²C address.
2. **IdRequest**—Ask the device for its identification string.
3. **AssignAddress**—Tell the device with a matching identification string to change its address to a new address.
4. **CapRequest**—Ask the device to send its capabilities information.

Device to Computer Messages

1. **Attention**—Inform the computer that the device has finished its power-up/reset tests and that it needs to be configured.
2. **IdReply**—Reply to an IdRequest with the device's unique identification string.
3. **CapReply**—Reply to a CapRequest with a fragment of the device's capabilities string.
4. **IfError**—Invalid checksum or premature end of message detected.

ACCESS.bus: A New Peripheral Bus

When a peripheral device powers up or resets, its initial device address is always 6Eh. The Base messages are used to reset this device address to a unique address between 02h and 7Eh (125 assignable addresses).

Application Protocol Layer

Application protocol layer messages are specific to the peripheral device and the message layouts are different for each device type, as well as for each device sub-type. This means that the device drivers are different at this level, but since the hardware and Base protocols are device-independent, much of the firmware support code can be shared by different devices. Even at the Applications protocol level, a common message structure is used for similar device types, so that a common approach can be used in writing ACCESS.bus device drivers. To date, peripheral devices for ACCESS.bus have been classed into three broad categories:

Keyboards—May have as many as 255 keys. Special function keys and annunciators are supported.

Locator devices—Includes pointing devices

such as mice, trackballs, graphic tablets, etc. Provides for devices with up to 15 degrees of freedom and up to 16 binary keys.

Text devices—Devices that support data streams (e.g., modems, printers, etc.).

What ACCESS.bus Means to You

ACCESS.bus is a coherent, well-defined desktop connectivity standard. It is solidly backed by DEC, Philips and others in the industry and it offers much to both users and peripheral manufacturers. Since it is so new, there are plenty of opportunities for entrepreneurs to implement ACCESS.bus devices with fewer direct challenges from the big guys. Peripheral devices that already use an 8051 microprocessor only need to change their I/O firmware, so the impact of designing for a new bus can be minimized. Watch out, though—Microsoft and others are starting to pay attention to ACCESS.bus, so the window of opportunity is getting smaller.

ACCESS.bus Access

The ACCESS.bus Industry Group will provide free information, including the ACCESS.bus Specification, to anyone who asks for it. It

costs nothing to join ABIG (as a company) if you plan to develop an ACCESS.bus peripheral. ABIG's address is:

ACCESS.bus Industry Group
370 Altair Way Suite 215
Sunnyvale, CA 94086
408-991-3517 or FAX: 408-991-3773

At least one company provides and ACCESS.bus Development Kit, at a cost of \$1500. The kit includes a controller board, a Logitech ACCESS.bus mouse, an expansion box, two cables, a Philips 87C751 microprocessor, a comprehensive software package and documentation for everything. Their address is:

Computer Access Technology Corporation
3375 Scott Blvd. #410
Santa Clara, California 95054
800-909-CATC (2282)
408-727-6600
FAX: 727-6622

Michael Burton is a Senior Software Engineer, at Key Tronic Corporation, Spokane, WA.



Embedded control using ACCESS.bus

CIRCUIT CELLAR **I N K**®

THE COMPUTER APPLICATIONS JOURNAL

June 1993 — Issue #35

COMMUNICATIONS

ACCESS.bus Design Tips

High-speed Modem Basics

Designing With IR LEDs

Embedded Interrupts on the '386SX

Component Selection Issues



Embedded control using ACCESS.bus

Embedded Control Using ACCESS.bus

Macintosh users have long been used to plugging multiple peripherals together with a single kind of cabling system using Apple's Desktop Bus. Now, ACCESS.bus promises to clean up the cable clutter for PCs, too.

FEATURE ARTICLE

David Wyland

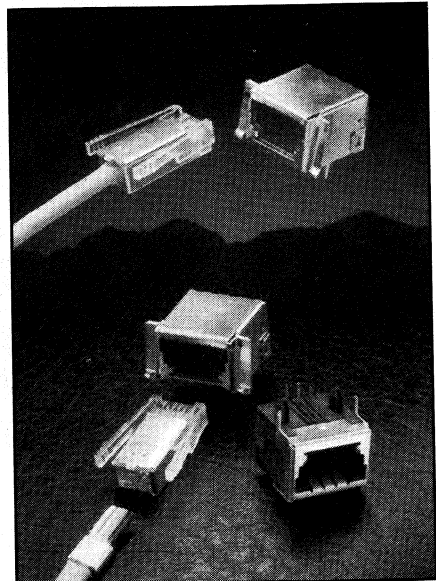
Wouldn't you love to have a perfect serial interface for general-purpose instrumentation and control that works every time you plug it together? One with all the aggravations associated with RS-232 eliminated? Could you appreciate an interface ten times as fast as 9600 bps, but automatically slows down if it needs to? Could you find some use for the real estate saved by using TTL signal levels instead of ± 12 volts (so you don't need space or power for level converters)? Would you like to simplify your system interconnects by using a multidrop bus connection? Would you like a cabling solution where you can have one kind of cable for all situations? Would you object to never wiring a null modem again, or never again pondering if CTS/RTS/DTR/DSR are wired to the wrong pins? Would you like this interface to be an openly defined industry standard, supported by lots of devices and by several large companies?

This wonderful, utopian bus actually exists! The ACCESS.bus meets the standards I outlined in my "wish list." It is a serial bus, but there are no "settings" to concern yourself with. The source (Master) sends data to the destination (Slave) in 8-bit bytes with an acknowledge at the end

of each byte. The ACCESS.bus transfers data at up to 100 kbps (400 kbps in the future). It features automatic slowdown by either the sending or receiving device as required. It uses TTL signal levels (0 and +5 volts) for data transmission. It is a multidrop bus using modified modular-phone cable with a simple 4-wire connection. ACCESS.bus devices can also be *hot plugged*, meaning that it lets you safely add or remove devices from the ACCESS.bus while it is running. It doesn't even have addresses to set or DIP switches to fiddle with! The CPU automatically assigns addresses at reset or when a device is powered up.

Best of all, the ACCESS.bus is on its way to being an industry standard for PCs. The ACCESS.bus is a result of the work by DEC and Philips/Signetics to create a desktop bus for the PC. The goal of the desktop bus is to simplify the cabling to keyboards, mice, graphical tablets, lightpens, and so forth.

ACCESS.bus is a software overlay on the ubiquitous PC bus standard, so it can easily leverage off the momentum already established for that standard. The PC bus has over 150 chips available that support it. There



Embedded control using ACCESS.bus

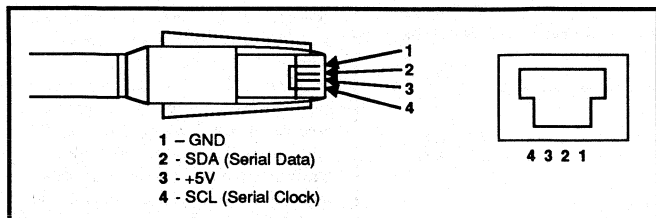


Figure 1—Based on PC, ACCESS.bus uses a simple four-wire interface that provides not only a data channel, but also power to peripherals. The proposed modular connector locks in place and eliminates orientation confusion.

are at least eight 8051 microcontroller chip derivatives available with PC bus UARTs on them. There is also an PC UART—the PCD8584—that you can lash to the processor of your choice.

ACCESS.BUS BASICS

The ACCESS.bus is a serial bus that uses a four-wire interconnection standard. The cable called out in the standard is a modified modular-phone cable (see Figure 1) and is shielded to minimize REI. The cable contains two signals—serial clock and serial data. The other two wires are power and ground. The power line supplies +5 volts at up to 1 amp for powering small devices such as mice, keyboards, and so forth, directly from the cable.

The ACCESS.bus is a half-duplex bus and uses a multidrop protocol where each device on the bus has a unique address. Up to 124 devices can share the bus, with addresses 00h, 50h, and 6Eh reserved. The maximum bus capacitance of 400 pF restricts the number of devices and constrains the cable length to 8 meters. This capacitance limit ensures a rise time of less than 1 μs when termination resistances of less than 2.5k ohms are used. You connect devices to the bus in parallel.

The Serial Data line (SDA) carries data transmissions which are clocked into the receiving device by the Serial Clock line (SCL). If the receiving device needs more time, it holds down the SCL line until it is ready for more data. The specification limits this hold time to 2 ms. The sending device initiates a message by changing the

SDA line from high to low while SCL is high, and terminates the message by taking the SDA line low to high while SCL is high.

Since any device can send a message to the CPU at any time, collisions are possible. This happens when multiple devices see the bus in a "not busy" state and start sending a message simultaneously. The devices do collision detection to sense such events. Each one checks to see that the data on the bus is the same as the data it is putting on the bus. In case of a collision, the two devices will eventually try to send different data bits. Since the bus is open drain, the one sending a low level wins, so the device

trying to send a high level detects a bus error, stops, and retries its transmission later. Plugging a new device on the bus might corrupt a message in progress, but the same collision detection mechanism will also sense such a corruption. Collision detection coupled with the open-drain nature of the bus are the key features of the standard that allow hot plugging.

The ACCESS.bus moves data in 8-bit bytes similar to RS-232. However, byte transfers over the bus use the PC bus protocol. PC defines byte transfers as follows: The transmitter sends the most-significant bit first, followed by an acknowledge bit supplied by the receiver (see Figure 2). Messages begin with a Start condition and end with a Stop condition. This differs from RS-232 that has Start and Stop bits for each byte. The ACCESS.bus provides a standard that organizes groups of bytes into messages, defines how to assign addresses to slave devices, and defines all messages as writes—from CPU to slave or from slave to CPU.

The ACCESS.bus messages vary in length from 1 to 127 bytes. Figure 3 shows the message protocol. Each message consists of a destination

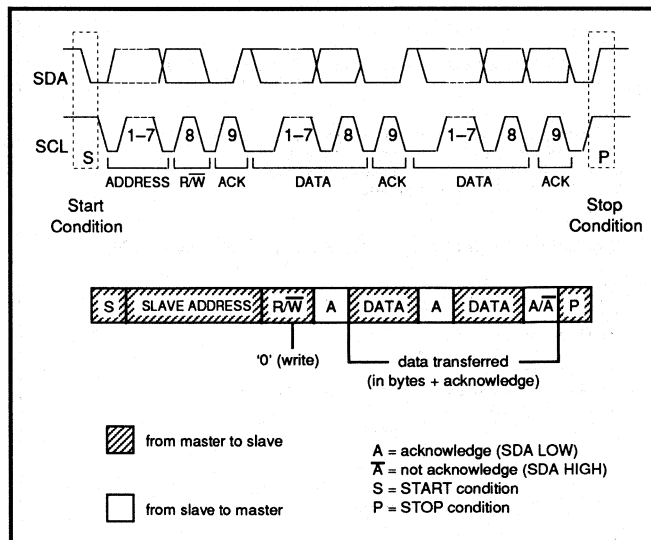


Figure 2—In PC, serial data (SDA) is sampled when the serial clock (SCL) is high. The basic packet of information consists of a destination address, a R/W flag, data, and start/stop framing. A simple ACK/NACK status is sent back by the destination device.

Embedded control using ACCESS.bus

Listing 1—The main 80C852 CPU message transfer code handles the master side of the ACCESS.bus interface.

```

:Subroutine to send a message from CPU to I/O device

SEND:
Enter with slave address, length, pointer to bytes of data
Set to Master mode, send Start bit by writing 68h to Control reg.
Check for valid start
Send I/O address using SBYTE
Send CPU address = 6Eh using SBYTE
Send length using SBYTE
Send data bytes using SBYTE
Calc.checksum = XOR of all bytes from slave address through last
data byte
Send checksum using SBYTE
Send stop bit
Clear from Master mode, set to slave mode (default):
Set Assert Acknowledge by writing 04h to Control register
Exit

:Subroutine to send one byte

SBYTE:
Write byte to data register
Wait for acknowledge from Status reg.: can be interrupt response
Exit

:Error routines

ARB:
On Send arbitration error, send stop bit, clear from master mode
and restart at SEND.

NAK:
On Send not acknowledge, send stop bit, clear from master mode
and restart at SEND.

TIMO:
On Timeout, exit with error code

:Subroutine to read a byte from I/O device to CPU

READ:
Set slave address
Set message length = 81h = 1 byte with command bit = 1
Set command byte = 10h (Read Request)
Call SEND to send Read Request message
Call RECV to receive message
Exit

:Subroutine to receive a message from I/O device to CPU

RECV:
Note: CPU in slave mode as default
Receive CPU Address = 6Eh
Receive master (sender) address
Receive message length
Receive data bytes
Receive checksum but don't acknowledge yet
Verify checksum
Send acknowledge if checksum OK, not if not OK
Receive STOP condition
Exit

:Subroutine to receive one byte

RBYTE:
Wait for Interrupt
Exit: Return byte on interrupt, set interrupt

```

address, a source address, a byte count, a control/data flag bit, the message with 1–127 bytes of data, and a checksum. The PC UART transfers each byte automatically, and each byte receives an acknowledge from the destination device. The PC UART hardware handles message initiation, byte acknowledge, speed control, and message termination.

If the Control/Data flag in the byte count field is a 1, the message is a command. The operation code is the first byte immediately after the byte count. Opcodes in the range of 00h through 7Fh are available for general use. For instance, I use opcode 10h as a Read Request command to an I/O device. The ACCESS.bus reserves opcodes from 80h through FFh for control functions. These control functions and their reserved codes include: Reset (F0h), Identification Request (F1h), Assign Address (F2h), Attention (E0h), Identification Reply (E1h), and Interface Error (E3h).

AN ACCESS BUS SYSTEM

ACCESS.bus systems are simple to design and work with. You have only two signal wires and one power wire to deal with. The PC bus is an open-drain pull-down bus with a single pair of pull-up termination resistors, which are typically installed at the CPU end. You determine the resistor value from the maximum-rated drive current (3 mA) for the PC drivers. The resistor value also determines the data transmission speed since the resistor current charges the line capacitance. A 2k-ohm resistor works well, as shown in Figure 4. An optional 100-ohm resistor in series with each driver helps kill noise. The +5 V is provided through a fuse or a current-limited regulator. A current-limited regulator has the advantage of automatic recovery with no fuse to replace after a failure. The choice of resistor size and fuse method is about all the hardware design you must do.

A good way to explore the ACCESS.bus is to use it in a system. I'll make a simple system with one CPU and three I/O ports. Each port has 8 bits of digital input and 8 bits of digital output. Figure 4 shows a block

Embedded control using ACCESS.bus

diagram of my system. I use four microcontroller chips: an 83C652 as the primary CPU and three 87C751s as I/O device controllers. These chips are variants of 8051 microcontroller chips with PC UARTs. I use the microcontrollers to implement the ACCESS.bus communication protocol over their PC connections. The 83C652 is a 40-pin device with external EPROM and RAM. These features make it useful as the central CPU. The 87C751s are 20-pin, 300-mil components with internal EPROMs, the PC interface, and two 8-bit bidirectional I/O ports. These devices will serve as the I/O device controllers in my prototype.

The three 87C751s each provide peripheral device control. The PC interface uses two of the three bits on Port 0; Port 1 receives the 8 bits of data input; and Port 3 supplies the 8 bits of data output. Note that both ports are bidirectional. If you need more data I/O, you can use Port 1 as a bidirectional 8-bit data bus, and use Port 3 as an 8-bit address bus. This allows up to 256 bytes of data I/O from this single device.

As you can see, this is a simple system in terms of hardware design. Unlike RS-232, there are no level converters, no baud rate configuration switches, and the connector pinout is simple and fixed for all devices. The bus connection method simplifies cabling and means there is only one PC UART at the host end rather than one for each I/O device. This also means you can add I/O devices to the system without adding hardware to the CPU.

The bus connection method is possible because messages can be more than one byte long. In ACCESS.bus, there is a multibyte message between Start and Stop codes that contains the addresses of the source and destination of the message. This allows several devices to share the same bus.

The hardware design of our ACCESS.bus system is simple: the software makes it work. The software converts data into ACCESS.bus messages for transfers between the CPU and I/O devices, and sets up the addresses of the devices when they power up.

Listing 2—The message transfer code for the 87C751 I/O Controller handles the remote data acquisition and control.

```

;Subroutine to send a message from I/O device to CPU
SEND:
  Enter with slave address, length, pointer to bytes of data
  Set Master mode: Write 50h to Config = req bus master. 100 kbps
  Send I/O address using SADDR
  Send CPU address = 6Eh using SBYTE
  Send length using SBYTE
  Send data bytes using SBYTE
  Calc. checksum = XOR of bytes from slave addr through last byte
  Send checksum using SBYTE
  Send stop bit
  Set to Slave mode: Write 90h to Config. Reg. (default mode)
  Exit

;Subroutine to request bus mastership and send address
SADDR:
  Wait for ATN bit in Control Register, go to SAERR if error
  Go to send byte routine

SBYTE:
  Set bit counter to 8
  Write MSB to Data Register
  Rotate left for next bit
  Wait for bit sent: wait for ATN in Control Register
  Decrement bit count and loop back to SBYTE+1 if not zero
  Set to receive mode to receive ack: Send A0h to Control Reg
  Wait for ATN
  Exit; Return acknowledge status

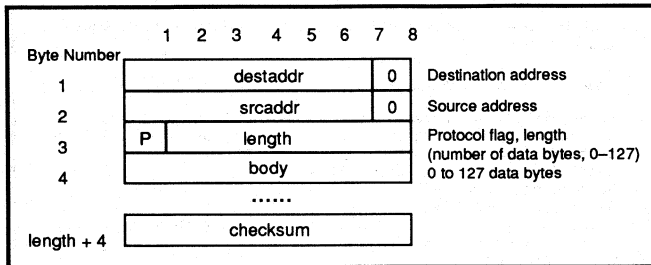
;Error routines
SAERR:
  On Send arbitration error, send stop bit, clear from master
  mode and restart at SEND.
NAK:
  On Send not acknowledge, send stop bit, clear from master mode
  and restart at SEND.
TIMO:
  On Timeout, exit with error code

;Subroutine to receive a message from CPU to I/O Device
RECV:
  Enter in Slave mode (This is the default mode)
  Receive slave address: call RDACK
  Receive CPU Address = 6Eh
  Receive message length
  Receive data bytes
  Receive checksum but don't acknowledge yet
  Verify checksum
  Send ack if checksum OK and slave addr compares; otherwise not
  Receive STOP condition
  Check for Read Req. command = command with Operation code 10h
  If Read Req.t, get byte of input data from Port 1 and call SEND
  Exit

;Subroutine to receive one byte
RBYTE:
  Set bit counter to 7, clear accumulator
  Wait for bit
  Get bit, clear ATN
  Rotate to LSB
  Decrement bit count and loop back to RBYTE+1
  Wait for last bit
  Get bit, don't clear ATN
  Rotate to LSB
  Send acknowledge
  Wait for ATN
  Check for errors
  Exit

```

Embedded control using ACCESS.bus



ACCESS.BUS PROGRAMMING

There are two areas to the software design for an ACCESS.bus system: the set of routines that send data to and receive data from the remote devices, and the initialization code that assigns the soft addresses to the devices when the system powers up. Initialization sequences are also required when a device is plugged onto a bus that is already running. Let's look at the operating routines first. The CPU sends messages to an I/O device, and I/O devices send messages to the CPU. The message protocol is the same in both of these cases. Any I/O device can send a message to the CPU at any time. For example, a keyboard sends key data to the CPU whenever you press a key. The sending device is always the master and the receiving device is always the slave.

My 4-MPU system can write bytes from the CPU to the output port of a selected I/O device, and it can read bytes from the input port of a selected I/O device. To write a byte from the CPU to another device, the CPU must send a message with the appropriate address. In this case, the CPU sends a data message to the desired I/O device address. The I/O device receives this data and writes it to its output port.

Figure 4—Many 87C751-based I/O ports may be added to the 83C652-based ACCESS.bus system with just a single pair of wires.

For the CPU to read a byte, it sends a command message called Read Request to the I/O device, and it responds with a data message containing the data. The Read Request command is a single-byte command message which is the opcode. I use a user-definable opcode (10h) for the Read Request.

CPU PROGRAM

The PC controls in the 83C652 and the 87C751 are different. The 83C652 contains a full PC UART. The 87C751 has a less-capable interface, so the program does the serialize, deserialize, and timing-control functions.

Listing 1 shows the 83C652 routines for message transfer. The CPU writes a byte to the output port of an I/O device by sending a Read Request command (10h) to the I/O device. In turn, the I/O device responds by sending a one-byte message with data to the CPU.

The 83C652 PC interface has four 8-bit registers: Control, Status, Slave Address, and Data. The Control register controls PC communication. It enables the PC UART, sets the maximum bit rate, sends Start and Stop states, sends byte acknowledge messages, and enables an 8051 inter-

Figure 3—The standard ACCESS.bus message packet is based on PC, but includes additional information. The packet consists of a destination address, source address, length, data bytes, and a simple checksum.

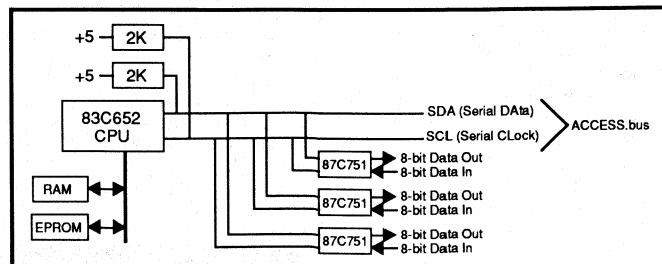
rupt for status changes. The Status register provides a 5-bit status code indicating function completion or error. The Slave Address register provides the response address for messages that are sent to the CPU. The protocol sets the CPU Slave address to 50h. The Data register sends and receives 8-bit bytes of data.

I/O DEVICE PROGRAM

Listing 2 shows the message transfer program for the 87C751. When the CPU writes a byte to the I/O device, it sends a one-byte data message. The I/O device program receives this message from the CPU and writes it to Port 3. When the I/O device receives the Read Request from the CPU, it reads the data on the input port and sends a message to the CPU.

The 87C751 PC interface has four registers: Configuration, Control, Status, and Data. The Configuration register defines whether the PC interface is in Master or Slave mode. It also holds the timer value for the maximum baud rate. The Control register sends the Start and Stop conditions to the bus and indicates attention and data ready conditions. The Status register indicates the status of Start and Stop conditions. The Data register provides one bit of storage.

The 87C751 program provides data serialize and deserialize functions, slave address recognition, and individual bit timing. These routines complicate the program but reduces the cost of the silicon in the system.



Embedded control using ACCESS.bus

It is also practical since the 87C751 is used as a simple I/O controller and the program hasn't much to do except tend the PC interface and pass the data to the digital I/O port the '751 serves.

SOFTENING UP THE ADDRESSES

An address is assigned to each I/O device by the CPU. The I/O device stores the address and uses it when communicating with the CPU. When the CPU sends a message to the I/O device, this address is part of the message. If the destination address matches the address stored in its register, the I/O device accepts the message; otherwise it ignores the message. When the I/O device sends a message to the CPU, the message includes the source address so the CPU can tell who sent it. The I/O device address is also called the *slave address*. The CPU has a fixed slave address of 50H. The CPU assigns each device on the bus a unique I/O address as part of the initialization sequence.

The CPU also assigns an I/O address to each device plugged into the bus while the bus is running. The device plugged into the bus notifies the host of its existence, and the host assigns it a unique address.

Each device sends an Attention command to the CPU asking for an address assignment when the device powers up or is reset. The Attention routine in the CPU receives the command and sends an Identification Request command to the default I/O device slave address, 6Eh. The device responds with an Identification Reply command. This consists of a 29-byte ID string including device type, model, and so forth, plus a unique 32-bit number, typically a random number. The random number lets the CPU distinguish between identical devices.

The CPU records this data, picks the next available soft address and sends it with an Assign Address command to the device. The CPU sends a copy of the identification string with the new device address to ensure that the correct device receives it. As a final precaution, the I/O device sends a Reset command to its own address in case another identical device had an identical 32-bit random

Listing 3—Each device on the ACCESS.bus is assigned an address when it's connected or when the bus is reset. The CPU initiates the process.

```

: Setup Code at Reset

RESET:
Write C9h to Control Register to enable I2C, interrupt, 100 Kbaud
Write 6Eh to Slave Address register
Loop to send a Reset command to each I/O device address, 01h
through FEh
Exit to main program loop

: Interrupt routine to service Attention Command from slave at 6Eh
to CPU at 50h

ATTN:
Send Identification Request command to 6Eh (default slave address)
Wait for Identification Reply. If no response in 40 ms, exit
Put identification string in device table. Select next available
slave address.
Send Assign Address command with ID string and new slave address
Exit

```

number and was assigned the same slave address. In this case, the arbitration ensures that only one device sends a message at a time. One device sends its reset command first, and the other device receives it before it can issue its own reset command.

The sequence for soft address initialization of the CPU is summarized below and is shown in pseudocode in Listing 3. The initialization of each I/O device is the complement of this, and is shown in Listing 4.

- CPU broadcasts a reset command and all devices revert to default address: 6Eh.
- Each device sends an Attention message, informing the CPU of its existence.

- CPU sends an ID Request command to the default slave device address: 6Eh.
- All devices try to respond with their ID data containing a unique 32-bit number.
- PC bus arbitration causes the messages to be received one at a time.
- CPU records the ID data and random ID number for each device.
- CPU sends an Assign Address command with its new slave address to each device.
- Each device sends a reset to its assigned address to solve any duplicate device problems.

The Attention routine in the host

Listing 4—ACCESS.bus Pseudocode for 87C751 I/O Controller for Soft Address Initialization

```

RESET:
Set to Slave mode: Write 90h to Config. Reg. (default mode)
Write 00h to Control Register
Set 6Eh as default slave address
Send Attention command to CPU at 50h
Wait for Identification Request command from CPU
Send Identification Reply command to CPU with 32-bit random ID #
Wait for Assign Address command.
Check ID Reply against ID string. If match, accept new slave addr.
Send Reset command to new slave address.
If arbitration error, wait for N+50 usec, where N is 8 LSB's of
32-bit random number
If reset received at new slave address while waiting, start over
at RESET.
Exit to main program loop


```

Embedded control using ACCESS.bus

that does the soft address assignment automatically is the same one that handles hot plugging. When you plug a new device into the bus, it powers up in a reset state and issues an Attention command. The CPU responds with an Identification Request, and slave address assignment proceeds like at system power up.

One of the side benefits of automatic address assignment is the CPU generates a record of all devices currently active on the bus. Each time you add a new device to the bus, the CPU automatically updates this table. The CPU does not automatically update the table when you remove a device, however. If you want this feature, you must add an additional background scan routine.

WHERE CAN I GET MORE INFORMATION?

Information on the ACCESS.bus is available from the ACCESS.bus Industry Group. They will supply you with the latest detailed specification of the bus. Another good source for both ACCESS.bus and PC bus information is the Philips Semiconductor Data Handbook for 80C51-based 8-bit Microcontrollers. It contains information on both buses as well as the PC chips that can be used with them. Computer Access Technology provides boards and software for ACCESS.bus. You can quickly set up a system using a PC as a host with their boards. 

David Wyland specializes in Scheduled Inventions, combining his background in analog and digital system architecture to run his consulting group. Contact him at The Wyland Group, 15213 Bowden Ct., Morgan Hill, CA 95037, (408) 778-3860.

CONTACT

ACCESS.bus Industry Group
415-112 N. Mary Ave., Ste. 265
Sunnyvale, CA 94086
(408) 991-3517

Philips Semiconductors
811 E. Arques Ave.
P.O. Box 3409
Sunnyvale, CA 94088-3409
(408) 991-3518

Computer Access Technology Corp.
3375 Scott Blvd., Suite 410
Santa Clara, CA 95054
(800) 909-2282

© Circuit Cellar INK, The Computer Applications Journal.
Reprinted with permission.

A PC-to-ACCESS.bus interface card

A PC-to-ACCESS.bus Interface Card

by Robert Clemens and Tom Stockbrand

The card and software described here are for a first-generation interface that we supplied to DEC and Signetics as a means of quickly getting an ACCESS.bus interface in a PC. ACCESS.bus is a handy tool for solving the problem of not enough hardware ports. It also allows for adding or removing peripherals while the system is running (hot plugging). For example, you may want to navigate through Windows using a mouse while in one application, but use a digitizing tablet or track ball in others. You can have them all available to plug in any combination and still have serial ports available.

BACKGROUND

A few years ago, Ken Olsen at DEC asked our group to solve the problem of hooking a bunch of desktop peripherals together without the rat's nest of wiring that is usually needed for this task. Our A/D group had

adopted the Philips I²C bus as the means to build our display system prototypes. We were exploring the idea of extending it to the world outside a PC board. Since it is a fairly fast (100 kbps) serial bus that can be daisy chained and hot plugged, it was very attractive. It turned out that the workstation group had been thinking of the same thing, so the project was born.

Our group in Albuquerque worked on the hardware, the Display Systems Group back in Westford, Mass., did the software architecture, and Robert did the software design and construction. We soon joined forces with Signetics (now Philips) and they have taken over the work (along with the ACCESS.bus Industry Group) of getting the ACCESS.bus supported and encouraged.

The hardware problem was that the specs for the I²C bus required a 1- μ s maximum rise time. This limits the capacitive load to 300 pF with the specified 3k-ohm pull-ups. This is not very much head room if any significant amount of cable is to be driven. The hardware solution that we hit upon was to drive the bus with current sources rather than with resistors. That way the rise time specification could be met with a load as high as 800 pF. Peter Sichel and the design team at DEC worked

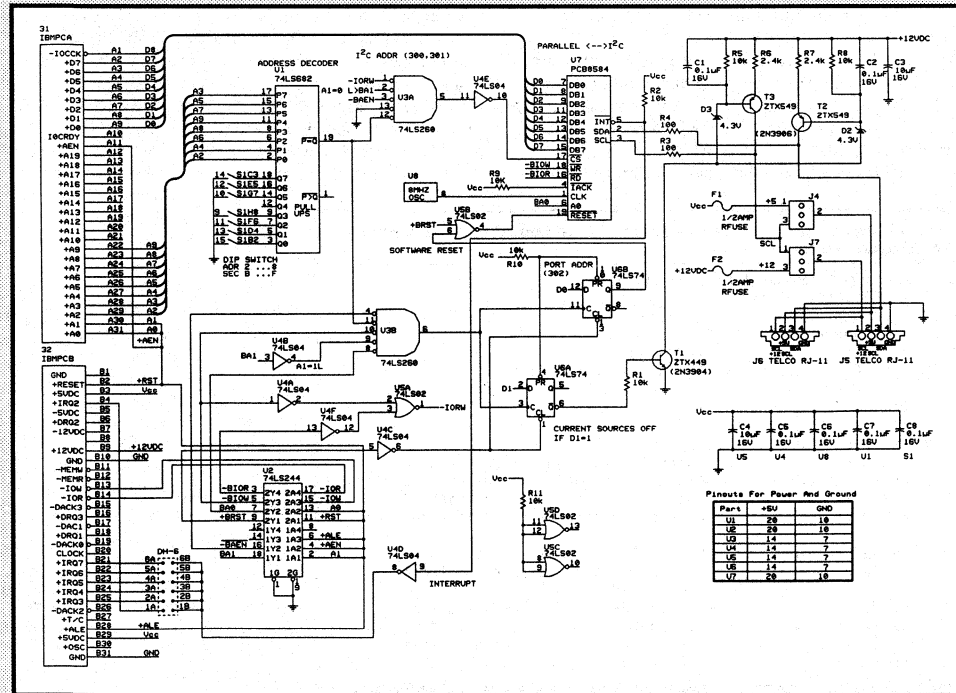


Figure 1—The core of the PC-to-ACCESS.bus interface board is a Philips PCD8584 PC serial-to-parallel converter chip.

A PC-to-ACCESS.bus interface card

out a fairly spare software protocol, and other details were worked out such as standardizing on a plug and a low-capacitance cable.

At this time, the ability to put two formal hosts on the bus isn't in the spec. It *is* possible, though [more information on this is available from Robert]. In fact, we think providing a means of hooking two or more CPUs together is one key to success for the concept. Therefore, there is a requirement to be able to shut off the current sources in all but one of them so as to limit the current that has to be sunk by the PC hardware. The board described here has current limiting using PTC resistors, but no direction limiting. A revised version should provide current direction limiting or just disable the local power supplied to the bus when the current sources are turned off.

HARDWARE DESCRIPTION

Figure 1 contains the schematic of the PC-to-ACCESS.bus interface card. The core chip on the board is a serial-to-parallel converter for the PC bus made by Philips called the PCD8584. It is very similar to a UART but provides for the particular sophistication needed by the PC protocol. It is driven with a standard PC address decode structure. There are current sources on the card for powering the bus and a means to turn them off by setting a port bit and also to do a software reset.

In this first-generation board, we limited the current that would run down the power supply wire by means of PTC resistors. They work, but are slow and expensive. A better way is to provide the +5 V with a regulator chip that is current limited to 0.5 A, adding fuses "just in case." The latest ACCESS.bus specification does not

require the +12-V option as provided in this design. The detailed specs are outlined later in this article.

SOFTWARE DESCRIPTION

I use a base address of 300h as an example in this discussion. An IORead from address 300h gets the status of the PC interface. An IOWrite to address 300h issues a command. A Read from 301h gets the PC data into the PC and a Write to 301h puts data out to the PC bus. An IOWrite to address 302h loads the port bits. Setting the LSB (bit zero of address 302h) to 1 causes a software reset (it is cleared with a hardware reset and must be cleared again after a software reset). Setting bit one of port 302h turns the current sources OFF.

The duties of a driver or application that wants to talk to an ACCESS.bus network are the address configuration and management of devices during power-up of the host, recognition and configuration of new devices as they are plugged in, and management of messages to and from ACCESS.bus devices and PC-based applications. Management of the devices is the same in implementations for all platforms. The handling of messages to and from host-based applications are platform specific.

As mentioned in main article, in order to maintain a basic working ACCESS.bus system, a table of devices must be built in software that keeps track of a device's current running status, its PC address, ID string, capabilities string, and a current pointer into the capabilities string. Also, a global variable for bus status must be maintained because a forced reset of the bus causes certain time-dependent actions to occur, and it is possible for a misbehaved device to create a bus error that can affect the state of the other devices on the bus.

Listing 1 shows some basic program structures and condition states that are useful when writing the software.

The last components needed are the routines that parse the message stream, install the appropriate hardware I/O and interrupt routing, and a link into the platform's timebase to drive time-dependent activity.

A Turbo Pascal source file is available on the Circuit Cellar BBS that works with this hardware design to implement a simple ACCESS.bus system for an IBM-compatible PC. This system allows viewing messages from devices, sending a message to a device, and starting/managing a reset sequence.

Listing 1—There are several data structures and condition code definitions that are helpful when writing ACCESS.bus support code.

```
Recommended device data structure:
  AB_DEVICE = record
    status      : integer;
    address     : integer;
    ID          : string[30];
    Capabilities : string[MAXCAPABILITIESLEN];
    CapOffset   : word;
  end;

Recommend global bus conditions:
  BusReset      Software is starting a reset sequence
  BusAssignAddress Software is assigning addresses to devices
  BusConfigured Software can begin normal operation
  BusError      Error has occurred, attempt to reset bus

Recommended device conditions for each device are:
  DevReset      Device is being reset or no device is
                 at this address
  DevWait       Device is busy, do not send message
  DevConfirm    Confirm device is still connected
  DevConfigured Device is ready for normal operation
  DevError      Device has a problem
```


A PC-to-ACCESS.bus interface card

DETAILED HARDWARE INSTRUCTIONS

JUMPERS

There are three headers on the board, each with a single shorting jumper. One is to select one of the six Interrupt Request Levels (IRQ 2-7). The other pair is to set the unit to provide either 12 volts or 5 volts to the RJ-11-style output connectors. There is also one DIP switch to be used in selecting the address group to which the card will respond.

IRQ

The IRQ header has six pairs of pins. The upper row of pins is connected to the output of the -INT line from the PCB 8584 serial-to-parallel converter chip through an inverter. The lower row of pins is connected to IRQ 2-7 in order across the plug from left to right. In order to use the Interface card, one IRQ should be selected by moving the jumper clip to the appropriate horizontal position and using it to jump the upper to the lower pin at that point.

POWER TO THE LOADS

Just behind the output sockets on the card are a pair of three-pin headers. A jumper on the upper one selects whether +5 V or clock (SCL) is applied to pin 2 of the output socket. A shorting block on the lower header determines whether +12 V or SCL is applied to pin one (the bottom pin) of the output sockets. The output sockets are wired in parallel. The two voltages are supplied through positive temperature coefficient resistors, used as resetting fuses, which have a resistance of 0.5 ohm and will allow 0.5 A to pass before starting to heat up and limiting the current that can be supplied externally.

For 5-V operation, the jumper clip on each header should short the center pin to the upper pin. For 12-V operation, they should each short the center to the lower pin on the header. If the jumpers are in the wrong position (either connecting the power to the wrong line or supplying both pins 1 and 2 with power, or neither) no harm will be done, but the system will not work.

ADDRESS SELECTION

The only DIP switch on the board is for selecting the port address for the interface card, modulo 4. The switch position labeled "1" is unused. The seven switches marked 2-8 enable the corresponding lines for address comparison with the corresponding address lines on the PC bus. For example, if the base address for the card is to be 300 (the usual case) then all switches should be closed, grounding their respective inputs to the address comparator, except number 8 which will then be pulled high internally. Internally, the address comparator's "9" input is held high at all times. This is because address bit 9 must be a one for all port accesses on a PC. Since

address bits 0-1 are not fed to the comparator, its output will be true for addresses 300 through 303. The low address [e.g., 300] is for reading or writing data to the ACCESS.bus. The next one [e.g., 301] reads or writes the command/status register and the third [e.g., 302] writes from PC to the two data port register bits for doing a software reset (D0) or turning off the current sources (D1).

The lowest address in the available space is 200 [all switches closed or "On"] since this corresponds to bit 9=1 alone. The highest address is obtained with all switches open yielding 3FCh as the base address. This address range can be shown as 01000000xx to 11111111xx in binary, where xx are the low-order bits used to select the port addresses 0-3 as described above.

JUST THE BEGINNING

As ACCESS.bus catches on and makes your desktop a lot less cluttered, a board such as the one we describe here can be your ticket to exploring the possibilities ACCESS.bus opens up. Have fun. ☐

Robert Clemens has done extensive programming for MIDI interfaces including sound editors for the PC. He also does embedded application programming in 8051-type microcontrollers using I²C and ACCESS.bus protocols. He is available at MediaMultiTech, P.O. Box 426, Durango, CO 81302, (303) 247-4726.

Tom Stockebrand spent 28 years working for DEC, mostly in the capacity of product design engineering and management for peripherals such as tape drives, communications interfaces, and displays. He did the original DECtape design, and his group did the original DEC VT50 series of terminals. In the 1980s, he ran an A/D group developing very high resolution displays at DEC's Albuquerque facility. He now has a small engineering consultancy called LGK Corp. which tries to provide a one-stop garage shop for getting breadboards and prototypes designed and built.

Taking a new bus

ANDREW SEYBOLD'S **OUTLOOK** ON PROFESSIONAL COMPUTING

Volume 10, Number 12

ISSN: 0895-3821

July 1993

An Outlook Reprint

This article is reproduced here exactly as it originally appeared. It has not been abridged, revised, or updated since its first publication. *The Outlook* has prepared this reprint at the request of Philips Semiconductors.

Andrew Seybold's Outlook on Professional Computing reserves the right to alter formatting as space requirements dictate, but no changes have been made in the textual material reprinted.



ACCESS.bus is a serial bus that supports "hot" attachment of up to 125 devices to a PC via a daisy-chained type of connection. Each device has its own address and is "recognized" by the system.

Technology

Taking a New Bus

Since IBM introduced the PC in August of 1981, there have been continuous advances in the state of the PC. The first Intel-based machines came complete with 64K of RAM (expandable to 256K), a choice of monochrome or Color Graphics Array (CGA) monitors, and sported a cassette-player interface.

Over the years, we have seen video choices expand to include EGA, MGA, VGA, Super VGA, and beyond. Cassette players gave way to 10 MB hard disks, and then 20, 40, 60, 80 and now 200 MB drives. Hard disk interfaces have evolved from SASI to IDE and SCSI. And hard disk drives are now available with 500 or more megabytes of storage.

Memory configurations have increased to 4, 8, 16 and more megabytes of RAM. CPUs have gone from 5 MHz, 8-bit processors to 66 MHz, 64-bit Pentium devices. Computer motherboards now include serial ports, parallel ports, video, memory caches, and other "standard" features not even contemplated only a few short years ago.

(continued on page 3)

© 1993 by Pinecrest Press, Inc., P.O. Box 917, Brookdale, CA 95007, Telephone (408) 338-7701
Reproduction in whole or in part without express written permission is prohibited.

Taking a new bus

Vol. 10, No. 12

Andrew Seybold's Outlook on Professional Computing

3

Keyboards

Keyboards also went through a major change—from XT to AT-compatible units with added functionality. Then we began adding pointing devices such as mice and trackballs—first by making use of existing serial ports and, more recently, with the addition of the PS/2 mouse port.

But the computer itself has changed more than the keyboard and mouse interfaces. Certainly, there have been changes in the types of keyboards, and a number of different types of pointing devices are available. But the basic connection to the PC has not changed since the introduction of the IBM-AT systems in the mid-1980s.

Why should we change something that works okay, is inexpensive to produce, and has become almost as standard as an AC power plug? Sure, there are differences in types of keyboards, and you can't simply plug any pointing device into just any type of computer. For the most part, however, this one area of the computer has been very stable. There's no reason to muck around with it, right? Wrong!

Apple's ADB Bus

Meanwhile, Apple introduced the Macintosh computer and, over the years, developed the Apple Device Bus (ADB) that provides a pair of connections on the back of the computer. Users have choices as to how they connect their keyboard and mouse or trackball. Since the ADB is designed to handle multiple devices (up to 3 recommended according to Apple), the keyboard can be plugged into the computer and the mouse plug can be inserted into the second ADB socket on the computer, or it can be plugged into the keyboard.

To connect devices using the Apple Bus, the computer must be turned off. With IBM-compatible machines, once the system has been booted, if a keyboard or mouse is disconnected, the system must be re-booted when the device is plugged back in.

Enter Digital Equipment Corporation

A number of years ago, engineers at Digital Equipment Corporation decided to work on a universal keyboard and pointing device architecture for its workstations. The result of this design effort was a new bus architecture that makes use of the I²C (Inter-Integrated Circuit, invented by Philips Semiconductor) serial bus to provide an efficient, cost-effective method of connecting devices.

DEC incorporated this new bus architecture in the Maxine, the DEC Personal Workstation 5000. DEC then decided to give it to the industry and an association to promote. Thus, the ACCESS.bus Industry Group was formed in 1992. This association has more than 50 members, representing vendors from all parts of the computer industry. Among other activities, this group has a technical workgroup preparing a high-speed ACCESS.bus specification to extend the speed of the bus to between 500K and 1 MBit/second as a first step.

Keyboard manufacturers such as Honeywell, Keytronics, and Lexmark, Logitech, a manufacturer of mice and trackballs, as well as many major computer vendors, are either members of the association or have attended meetings and indicated a willingness to take part in this association.

What is ACCESS.bus?

The technical description of ACCESS.bus is that it provides a simple, uniform way to link a local computer (terminal, PC, or workstation) to a number of low-

Important: This page contains the results of proprietary research. Reproduction in whole or in part is prohibited. For reprints call (408) 338-7701.

Taking a new bus

speed I/O devices such as a keyboard, mouse, tablet, or a 3-D tracker. In more direct terms, it is a serial bus that permits attachment of up to 125 devices to a PC via a daisy-chained type of connection. And it does so in such a way that each device is "recognized" by the system and each device can be connected and disconnected while the system is running.

A Simple Demonstration

Our first exposure to ACCESS.bus was a demonstration of a game running on a standard PC. The keyboard and mouse looked like standard devices. Upon closer examination, however, we saw they were connected to the system by telephone-type plugs, and the keyboard had not one, but two receptacles.

The game was a Pong-type, where a "paddle" is controlled by the mouse. The player is supposed to hit the bouncing ball with the paddle, bouncing it against a wall. The ball returns to be hit again. After we watched this elementary game for a few minutes, a second mouse was plugged in, and the game immediately split into a two-player game. Each player could control his own paddle with his own mouse, and the game became interactive.

This interactivity was prompted by one action, and one action only—the insertion of the second mouse plug into an ACCESS.bus connector. A third mouse was connected, and a third player was automatically added to the game. Finally, a fourth device was added (this time a trackball), and we had four players all chasing the same ball, trying to hit it to the other players.

This is perhaps a simplistic example of ACCESS.bus, but it does demonstrate the power of this new technology. Adding any type of device causes the control program to recognize what type of device it is, and what function it will perform in the system. Want to have two keyboards and two mice on the same system, both with the same capabilities? No problem!

Using ACCESS.bus

I left the demonstration with a demo kit that contained two mice, a trackball, a keyboard, a standard Industry Standard Architecture (ISA) card, and some software. The ultimate goal of the ACCESS.bus folks is to have the chip set built directly onto the systems board. But in the interim, there is an ISA card that contains the chip set and into which ACCESS.bus connectors are plugged. This card can be installed in any standard Intel-based PC.

The existing keyboard connector on the PC is jumpered to the ACCESS.bus board and the first ACCESS.bus connector is plugged into the card. Once the software is installed, the system works just as you would normally expect, with a standard PC keyboard and mouse. Our installation was accomplished by connecting a cable from the PC card to the keyboard, and from the keyboard to a multi-port connector. From this connector, we plugged in a mouse and fired up the system.

It worked just as our original keyboard and mouse did, and the standard mouse functions normally. Next, we plugged in the trackball, since our preferred pointing device is a trackball. Now we had one keyboard, a mouse, and a trackball connected to the system. Without having to do anything but plug them in, both the mouse and the trackball functioned perfectly.

Since we had an extra mouse, we connected it, too. This is the set-up we've been using for the past few weeks. We have a mouse to the right of the keyboard, one to the left (I'm left-handed), and a trackball, also mounted to the right. No matter which pointing device we reach for, it works.

Taking a new bus

Other Devices

Each device on the ACCESS.bus has its own address and each device is "recognized" by the system. If we plug in a keyboard, the system knows it is a keyboard. It recognizes a mouse, a trackball, and any other type of device it knows about.

The possibilities are endless. In its present form (release 2.1), ACCESS.bus also recognizes tablets, a 3-D tracker, and handheld scanners. This technology has the potential to provide a universal bus for many other types of input devices, and could be used for security devices, as well. Since the connectors are inexpensive telephone-type plugs, and since existing hardware includes cabling to permit devices to be plugged into a multiple connector box, each user of a system could have a unique device with a specific built-in ID.

Applications

Such capabilities could be used to enable features and functions, permit access to different types of services, and even to disable the system completely if a particular device was disconnected. Further, since this bus supports "hot" connections, the PC would not have to be re-booted if a new device or security module was added.

If you let your mind wander a bit, you can envision the variety of devices that could be connected to share this bus. With 125-device capability, it would not be too farfetched to think modems, scanners, and various types of printers would share this bus—with a single connection to the PC.

Think of the number of slots (or lack thereof) that could be designed into PCs to make use of the ACCESS.bus. A single connector on the motherboard would enable the use of, and support for, up to 125 devices—each of which is simply plugged into an empty socket on the ACCESS.bus cable. Add a new device-specific driver, and off you go!

SCSI busses work on the same premise, except that the Small Computer Systems Interface (SCSI) bus is designed for fast data exchange and can only support a total of 8 devices. Why use the SCSI bus for all the lower-speed devices you would like to add?

Technical Information

The current ACCESS.bus specification is release version 2.1. Hardware specifications have been formalized and can be easily implemented by any vendor that wants to make use of this architecture.

The Demo Kit

A demonstration kit of ACCESS.bus hardware and software is currently available from Computer Access Technology. As mentioned above, this kit contains a standard ISA quarter-length card, cabling, a keyboard, two mice, and a trackball. The keyboard is made by Honeywell (its keyboard division has been sold to Keytronics, which is continuing to support ACCESS.bus).

Installing the system is easy. The ISA card mounts in the PC, a jumper is provided to connect the PC keyboard port to the card, and the ACCESS.bus cabling is run from the card. In our configuration, we ran a single ACCESS.bus cable from our floor-mounted DEC MT Tower system up to our desktop. We then connected it to the keyboard, and used the other connector on the keyboard to connect to a 4-way junction box.

At the junction box, we connected both mice and the trackball. We installed the software and were off and running. Having more than a single pointing

Taking a new bus

6

Andrew Seybold's Outlook on Professional Computing

July 1993

device available is nice. There are times when we really like the trackball, but there are also situations where we prefer a mouse.

Cabling and Connectors

ACCESS.bus uses four-pin, shielded MOLEX SEMCONN or AMP SDL, modular-type connectors (these resemble the standard RJ-11 telephone clip-in plugs, but are a little larger). The entire bus only requires four wires—ground, +5 Volts, Serial Data (SDA), and Serial Clock (SCL). The cables themselves are shielded, and devices can be located up to eight meters (about 25 feet) from the computer.

In addition, provisions have been made for ACCESS.bus devices to be powered externally, if required. RF filtering and power shielding are also specified.

The Devices

Every device on the bus must be a microcontroller with I²C interface and behave as either a master transmitter or a slave receiver, exclusively, as defined in the I²C specification.

Initially, all devices respond to a default power-up address. During the configuration process, the computer assigns a unique address to every device on the bus. ACCESS.bus supports multiple like-devices without switches or jumpers.

The Message Format

A message transmits information between a device and the computer or between the computer and one or more devices. These messages are a minimum of four bytes in length and are one of two types—a Device Data Stream or Control/Status Stream. The difference is indicated by the protocol flag. The maximum theoretical length of a message is 131 bytes (127 data types and four bytes for overhead).

Using this message format, each device provides the proper type of information to the computer, and each device is totally independent of others. Hot connections are permitted, and software can be used to recognize new devices as they are added to the string of connected devices.

ACCESS.bus Implications

The ACCESS.bus Industry Group, the organization promoting this new bus architecture, has been working with many companies in the hope that ACCESS.bus will become the new *de facto* standard within the PC industry.

It has gathered support from almost every sector of the industry, including both hardware and software vendors. The hope is that one of the major companies experimenting with the bus will stand up and announce to the world that they are going to be providing ACCESS.bus as a standard part of their hardware platforms in the future.

The organization has done its homework extremely well. We believe this bus architecture should be implemented within the PC industry. Once the first vendor stands up and announces its intention to make use of ACCESS.bus, all the other vendors will include this capability into their products, as well.

Standards

Unlike some of the other standards that have been brought to the PC market by a specific vendor, ACCESS.bus has been introduced by an organization of many different vendors. Once accepted by the industry, it will find its way into

Taking a new bus

Vol. 10, No. 12

Andrew Seybold's Outlook on Professional Computing

7

next-generation products in short order. Companies are developing mice, trackballs, tablets, keyboards, and other devices that will be ACCESS.bus-compatible and will add functions to the PC.

Existing PCs can be retrofitted with the ACCESS.bus via the ISA card. Thus, this is not a case of a new technology making existing equipment obsolete. We would like to see Microsoft build ACCESS.bus recognition into its operating systems, or at least make it available as an add-in option. Major hardware vendors investigating the use of ACCESS.bus will then step up to the microphone and endorse both the concept and the accomplishments of the ACCESS.bus Industry Group.

Conclusions

We have looked at the ACCESS.bus technology, investigated the vendors pursuing its implementation in mainstream computing, and can find no reason for not implementing it. Once ACCESS.bus becomes accepted and is included as part of all PCs being built, it will open new worlds of connectivity.

Consider that ACCESS.bus can be used in servers, desktops, portables, and even handhelds. It could be built into PCMCIA cards to provide ACCESS.bus connectivity to handhelds. Users could carry their own ACCESS.bus ID module with them to activate access to data, or to permit the computer to reconfigure itself.

In the education market, ACCESS.bus provides multiple device access so that a single PC can accommodate multiple users. Adding a second joystick would change an educational game from a computer-vs.-player mode to a player-vs.-player mode, and additional security could be included.

In the business market, the adoption of ACCESS.bus means that many different types of peripherals can be easily added to virtually any computer. Adding devices would not require opening the machine and installing a new card.

In the game market, vendors could easily adapt their products to recognize how many players were on-line and modify the game accordingly.

In short, ACCESS.bus is a technology whose time has truly come. It is stable, inexpensive, easy to implement, and extends the functionality of the computer. We believe it should be embraced by the entire PC industry.

Come on folks! We know it's out there and we want it!

—Andrew M. Seybold

For further information, contact:

Philips Semiconductors
811 E. Arques Avenue
P.O. Box 3409
Sunnyvale, CA 94088-3409
408-991-3518

ACCESS.bus Industry Group
415-112 N. Mary Avenue
Suite 265
Sunnyvale, CA 94086
408-991-3517

Computer Access Technology Corp.
3375 Scott Blvd., Suite 410
Santa Clara, CA 95054
408-727-6600
800-909-2282

Personal Digital Assistants: What's missing?

ANDREW SEYBOLD'S **OUTLOOK** ON MOBILE COMPUTING

Volume 1, Number 10

ISSN: 1066-8845

October 1993

An Outlook Reprint—

The following article is reproduced here as it originally appeared. Formatting has been altered, but no changes have been made in the textual material. *The Outlook* has prepared this reprint at the request of Philips Semiconductors.

Personal Digital Assistants

What's Missing?

Well, we now have two PDAs on the market (four if you count the two EO devices). While Apple is touting that it has sold more than 50,000 of its Newton MessagePads, we have yet to find anyone who has been able to successfully use one to communicate to or from desktop computers or to access an e-mail system. This is also true of the Tandy Z-PDA (Zoomer).

There are several reasons for this. First, a pen-based system is not conducive to interacting with a text-based (ASCII) mail system since each and every character has to be recognized and converted from electronic ink to ASCII before it can be sent. Second, the wired communications options provided with today's PDAs are not easy to implement, nor is there a viable wireless link.

While pondering these issues and trying to determine how one would go about providing a handheld PDA that would enable users to access e-mail services, a possible solution occurred to us. We have passed this idea on to several potential PDA vendors. It is a simple solution and one that could be implemented quickly.

Wireless and ACCESS.bus

Our cover story in the July 1993 *Outlook on Professional Computing* (Vol. 10, No. 12) was about a new bus architecture called ACCESS.bus. We described this new bus as follows: "The technical description of ACCESS.bus is that it provides a simple, uniform way to link a local computer (terminal, PC, or workstation) to a number of low speed I/O devices such as a keyboard, mouse, tablet, or a 3-D tracker. In more direct terms, it is a serial bus that permits attachment of up to 125 devices to a PC via a daisy-chained type of connection. And it does so in such a way that each device is "recognized" by the system and each device can be connected and disconnected while the system is running."

The on-board portion of ACCESS.bus is a custom chip and a single telephone-type connector—the amount of real estate it would take up and the cost of implementation would be of little concern to design teams. Once outfitted with the ACCESS.bus I/O system, a PDA could be connected to multiple devices via a single connector. And since the system supports "hot" connect/disconnect, and the software is smart enough to recognize that one or more devices have been attached or removed, it could be incorporated to provide any number of fully automatic functions.

Personal Digital Assistants: What's missing?

For example, suppose a PDA was equipped with the ACCESS.bus connector and was normally used as a pen-based handheld device. The user could also have a small keyboard that, when attached to the PDA, would automatically put the PDA into a keyboard input mode. We can take this one step further and build an external keyboard on top of a wireless modem (housed in the same form factor), and then connect this combination through the ACCESS.bus. You could take this still further and have the software recognize that this device combination had been attached, and thus automatically launch the communications program.

The Next Step

Now that we have ACCESS.bus in our PDA, it will recognize the device that is plugged in and make the proper connection for the device. Such a device could be a wireless modem, a wired modem, a combination of both, or any other device we might want to use with our PDA. What about a direct link via ACCESS.bus to your desktop computer? It is our understanding that in the near future many desktop PC companies will offer systems with an integrated ACCESS.bus system. In the meantime, ACCESS.bus is available as an ISA card that can be added to any IBM-compatible machine (the ACCESS.bus Station by CATC).

A real advantage of using ACCESS.bus is that a single connector located on a PDA can be your ACCESS to any number of devices (pardon the pun). All a user needs to do is to plug in a device—the system knows what the device is and how to work with it.

Up for Grabs

We believe that PDAs will need to have keyboards for at least the next year or two. We also believe that the keyboard does not necessarily have to be an integral part of each PDA—it can be an external unit that is connected only when access to e-mail and remote information is required. We also believe that ACCESS.bus is an architecture ideally suited to provide not only the keyboard connection, but also connections for modems, direct to PCs, and to any number of other devices.

We have not changed our minds about the importance of PCMCIA slots on such machines, but most of them will only have a single PC card slot. An ACCESS.bus connector, in addition to the PCMCIA slot, would certainly add value to the PDAs.

The next several generations of PDAs are on the drawing boards now. There is still time to take advantage of this new technology. Maybe some day people will be willing to walk around with PDAs that only have a pen as an input device, but for now we are still betting that most mobile users have computers and readily recognize the advantages of being able to access their e-mail and other information when they are not at their desks. The combination of ACCESS.bus, a keyboard, and a wireless modem seems like a natural fit to us. If you want to find out more about ACCESS.bus, you can contact the Industry Group at:

ACCESS.bus Industry Group
370 Altair Way, Suite 215
Sunnyvale, CA 94086
408-991-3517
fax 408-991-3773

—Andrew M. Scybold

Important: This page contains the results of proprietary research. Reproduction in whole or in part is prohibited. For reprints call (408) 338-7701.

The portable desktop: New connections for today's mobile user

Reprinted from PC Week • February 14, 1994

PCWEEK



The
**Portable
Desktop**
*New
connections
for today's
mobile user*

By MICHAEL R. ZIMMERMAN

Mobile computing will become more practical this year as several budding standards make their way from spec sheets to products.

ISA PNP (Plug and Play), Access.bus, and PCMCIA's ExCA (Exchangeable Card Architecture) are three standards that will make such things as docking a notebook, connecting multiple peripherals, and "hot swapping" PCMCIA cards nobraiers.

"These integral pieces will make mobile computing more pleasurable," said Andrew Seybold, publisher of Outlook On Mobile Computing, a newsletter based in Santa Clara, Calif. "It's happening now."

Access.bus

Perhaps the most unfamiliar player in the trio of standards is Access.bus, which can support scores of peripheral devices through a single SDL connector. Users

can run an Access.bus-compliant notebook's SDL connector, which is slightly wider than a typical RJ-11 jack, through numerous daisy-chained multiport boxes to support as many as 125 peripheral devices.

Parts of Access.bus have been used in computers and consumer electronics for years. The standard, as it applies to PCs, incorporates two intelligent logic chips and software drivers that support hot swapping—the ability to add or detach devices without having to first turn off or reconfigure the PC. Access.bus detects which devices have been added or detached and automatically reconfigures the system.

A handful of companies, such as Santa Clara-based Computer Access Technology Corp., currently manufacture Access.bus-compliant adapter cards and drivers for desktop PCs. But Eldad Granot, director of marketing and a member

of the Access.bus Industry Group, in Santa Clara, said several large notebook makers also are gearing up to release portable systems equipped with Access.bus technology built onto the motherboard.

In addition, Microsoft Corp. is working with OEMs to announce its support for Access.bus at the WinHEC show in San Francisco this month, said Keith Kegley, product manager for new product development in Microsoft's hardware group. The Redmond, Wash., software giant may embed Access.bus drivers on the CD ROM for its Chicago operating system, he said. Although the drivers would not be part of the installation setup, users would be able to load the drivers when needed.

Most major PC makers, such as IBM, Compaq Computer Corp., and Dell Computer Corp., are currently evaluating the technology and may stand up with

The portable desktop: New connections for today's mobile user

Microsoft in an endorsement this month, sources said.

Plug and Play

The PNP spec is more popular because of influential supporters such as Microsoft and Compaq. Like Access.bus, it benefits both desktop and notebook PCs, but it enhances each type of system differently.

When fully implemented in a desktop, for example, users can add and detach add-in cards without having to adjust jumper switches or load special drivers.

Notebook users will benefit from being able to dock a PNP-compliant notebook into a compliant docking station without having to shut down or reconfigure the notebook first. This is called "hot docking."

"[Hot docking] will be a nice feature because you won't have to interrupt what you're doing to connect to the network," said Scott Kelly, technical-support engineer for product development at Itron Inc., a software developer in Spokane, Wash. Kelly is purchasing docking stations with every new notebook.

"As laptops become the primary machines at more and more companies, hot docking will be a big deal," added Rod Corder, vice president of engineering at New Media Corp., a PCMCIA card developer in Irvine, Calif. PNP was developed by Compaq, Intel Corp., and Phoenix Technologies Ltd., and more than 30 PC and card vendors are now building PNP-compliant products.

But users will not see true PNP products until late this year when operating systems, including Microsoft's Chicago, are available with the technology built in, said developers. NEC Technologies Inc. recently demonstrated hot-docking capabilities with its Versa notebook and docking station, but that was only possible because of a beta version of Chicago that the machine was running.

Until then, users will be required to load drivers for PC cards and notebooks upon installation. Intel offers hardware and software kits to developers that eliminate the need for users to have to set jumper switches.

Despite efforts to enhance the docking station/notebook relationship, some users will remain true to portable adapters and PCMCIA cards for network connections.

"The docking stations are fine but usually too expensive," said Chris Wendt, network administrator at EduServe Technologies Inc., a student-loan servicer in St. Paul, Minn. "But buy a pocket adapter and [users] can take it with them to a remote site, sit at a desk, and plug into the network."

PCMCIA ExCA

Another specification that's used in products today is ExCA, a subset of the general PCMCIA specification for card and socket services. ExCA applies to systems, rather than adapters, ensuring compatibility of PCMCIA sockets between compliant systems, according to PCMCIA officials. NEC's Versa 40E and 50E and Compaq's Concerto convertible notebooks, for example, are ExCA-compliant.

Work also continues in related areas of the PCMCIA group that will boost support of card and socket services. In fact BIOS developer SystemSoft Corp., of Natick, Mass., hopes to bring PNP and card and socket services closer together—a prospect that could mean even greater functionality for notebooks.

"We're trying to make sure in our PNP BIOS efforts that the PCMCIA bus is treated properly and integrated into the PNP BIOS," said Paul Sereiko, director of product marketing at SystemSoft. "As the year progresses, and as PNP BIOS and PCMCIA become more integrated, there will be more robust offerings."

Emerging standards

Access.bus

Main benefit: Ability to daisy-chain and "hot swap" up to 125 peripherals.

Supporting vendors

- ▶ Computer Access Technology Corp.
Access.bus Station PC AT add-in card
- ▶ Key Tronic Corp.
WN101 keyboard and LifeTime mouse
- ▶ Logitech Inc.
OEM mouse

ISA Plug and Play

Main benefit: Ability to "hot dock" compliant notebooks into compliant docking stations.

Supporting vendors

- ▶ Compaq Computer Corp.
Concerto notebook
- ▶ IBM
ThinkPad notebook
- ▶ NEC Technologies Inc.
Versa 40E and 50E notebooks

PCMCIA ExCA

Main benefit: Assures PCMCIA socket compatibility between compliant systems.

Supporting vendors

- ▶ Compaq Computer Corp.
Concerto notebook
- ▶ NEC Technologies Inc.
Versa 40E and 50E notebooks

This list is representative, not all-inclusive.

COPYRIGHT © 1993 ZIFF-DAVIS PUBLISHING COMPANY, L.P. ALL RIGHTS RESERVED.

For more information, contact:

ACCESS.bus Industry Group
370 Altair Way, Suite 215
Sunnyvale, CA 94086
Tel: (408) 991-3517
Fax: (408) 991-3773

Computer Access Technology
Corporation (CATC)
3375 Scott Blvd., Suite 410
Santa Clara, CA 95054
Tel: (408) 727-6600
Fax: (408) 727-6622

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Tel: (206) 936-5940
Fax: (206) 936-7329

Who's hopping on the ACCESS.bus?

REPRINTED FROM WINDOWS SOURCES ■ APRIL 1994

THE MAGAZINE FOR WINDOWS EXPERTS

Windows

SOURCES

Update

SYSTEMS ROUNDTABLE

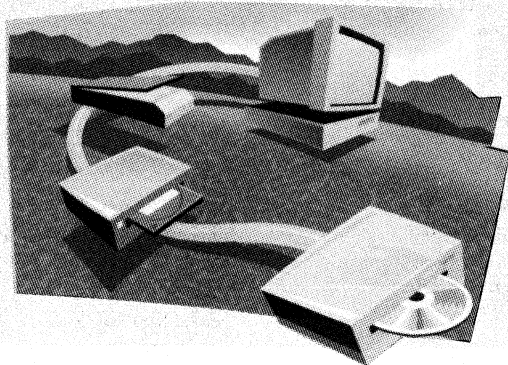
Who's Hopping on the ACCESS.bus?

ACCESS.bus is an emerging technology that will (theoretically) support up to 125 low-speed serial I/O devices (mice, printers, scanners) daisy-chained from a single port. Similar in operation to the Apple Desktop Bus, the ACCESS.bus has the potential to eliminate the plethora of ports that clutter the back of PCs. ACCESS.bus was developed by Digital Equipment Corp. and Philips Semiconductors and is now controlled by the independent ACCESS.bus Industry Group. It shows signs of becoming an industry standard, and we asked key systems and peripherals vendors if, and where, ACCESS.bus fits into their plans. Some believe future monitors will be populated by ACCESS.bus ports for peripherals, while others think it has potential to shrink systems by reducing serial port slots. AST Research Corp., Acer America Corp., and Gateway 2000 declined to comment on ACCESS.bus technology, but several other vendors offered their opinions.

Dell Computer Corp. won't be selling ACCESS.bus systems until the beginning of 1995 at the earliest. Distinguished engineer Jay Bell says the four-wire, telephone-style connector for the ACCESS.bus won't replace the serial bus: Future systems will have a serial port and the ACCESS.bus. He predicts that ACCESS.bus

will take over but not in the first year. "I think it will take a couple of years for the peripherals to migrate over," says Bell.

AT&T Corp. is a definite no-go for ACCESS.bus. Systems architect Tom Heil says that after analyzing ACCESS.bus more than a year ago: "We didn't see a need to partition a desktop environment. None



Don Backer

Who's hopping on the ACCESS.bus?

Update

of our desktop systems would be partitioned that way."

NEC Technologies' principal engineer Tom Schmidt foresees a day when monitors will conform to the proposed VESA Display Data Channel specification and support ACCESS.bus. "I think ACCESS.bus will become increasingly important because it allows a common channel for more than one device," says Schmidt. "The real attraction is that you can shrink the [PC] boxes by limiting the number of connections."

Compaq Computer Corp.'s director of marketing Andrew Watson wouldn't comment on Compaq's ACCESS.bus plans, but said that "one of the problems with PCs is too many ports, and ACCESS.bus could solve that." He said that Compaq "will consider anything that will allow us to reduce the number of ports."

IBM Personal Computer Co.'s David Bradley, manager of personal computing, is analyzing ACCESS.bus technology and says you'll see it implemented in IBM's PC divisions when compatibility and cost issues are resolved. He says that the ACCESS.bus peripherals "must be able to mimic current keyboard scan codes that older applications recognize, and it must not cost a great deal to implement."

Logitech has an ACCESS .bus mouse ready to ship, but hasn't sold one yet. "We don't have users asking for the bus, but we have one ready to be shipped to OEMs if they ask for it," says Denis Pavillard, Logitech's pointing device software manager. "We need a market to push it." He expects that Microsoft will be a trigger for ACCESS .bus technology. Microsoft will support ACCESS.bus in Chicago (Windows 4.0) and the Plug and Play specification.

Lexmark International, the printer manufacturer, showed a prototype ACCESS .bus keyboard at fall Comdex. Advisory engineer Jim Moses says, "It's a chicken or egg thing. You have to develop things to create a market. ACCESS will take a big step forward if computer companies integrate it into the motherboard and provide BIOS hooks."

Digital Equipment Corp. pioneered ACCESS.bus in its DECstation 5000 Unix workstation but the company is waiting for PC vendors to introduce ACCESS .bus systems before committing to the PC platform. Paul E. Nelson, DEC's senior engineer for input devices says, "I'd put it in, but management wants a market first." Nelson's ideal ACCESS .bus implementation: an ACCESS.bus keyboard with its own separate power supply, to interface with daisy-chained peripherals.

—John McDonough

COPYRIGHT © 1994 ZIFF-DAVIS PUBLISHING COMPANY L.P. ■ ALL RIGHTS RESERVED

For more information, contact:

ACCESS.bus Industry Group
370 Altair Way, Suite 215
Sunnyvale, CA 94086
(408) 991-3517

Computer Access Technology
Corporation (CATC)
3375 Scott Boulevard, Suite 410
Santa Clara, CA 95054
(408) 727-6600

Philips Semiconductors
811 E. Arques Avenue
P.O. Box 3409
Sunnyvale, CA 94088-3409
(408) 991-3518

ACCESS.bus revisited— ending the peripheral connection nightmare

COMPUTER DESIGN'S

OEM INTEGRATION

KEVIN GARDNER
PHILIPS SEMICONDUCTORS

ACCESS.bus revisited—ending the peripheral connection nightmare

Ideally, if you operate one of today's powerful nomadic PCs on the road, you should be able to plug into some kind of power supply pod or docking station upon return to your desktop. A portable would then assume the role of a standard desktop computer, and you'd enjoy a full-sized keyboard, high-performance display, mouse and other accessories. The approach would provide performance equivalent to a standard desktop



machine, but without the problems of disk swapping, file synchronization and duplication of expensive hardware.

But what's the best way to accomplish this? On most portables, there isn't much room for connectors stout enough to reliably handle repeated insertions and withdrawals. What's more, on most desks there isn't much room for additional hardware and enclosures, and many nontechnical users don't want their desks entangled in a web of separable interconnects and bulky cables needed to connect peripherals.

Simply serial

Enter the updated ACCESS.bus, a network-on-a-desktop that conquers the limitations of peripheral connections. A single ACCESS.bus interface on a portable PC serially communicates with many peripherals over a single cable.

Although initially introduced a few years ago as a method of supporting about a dozen peripherals, today's ACCESS.bus can accommodate up to 125 peripherals. These can include full-sized keyboards, mice, bar-code readers, low-speed printers and even monitors and multi-dimensional pointing devices such as virtual reality gloves.

Kevin Gardner is microcontroller product marketing manager, Philips Semiconductors (Sunnyvale, CA)

Everyone can now take advantage of a single low-cost TTL serial interface operating at raw data rates to 100 kbits/s and higher.

Significantly, an ACCESS.bus peripheral can be connected quickly and easily, without special procedures. No driver installation or device configuration is necessary. In fact, if you change a peripheral after you've powered-up, you don't need to reboot. You just plug in your peripheral and play. It's that simple.

One size fits all

Today, ACCESS.bus, originally developed by Digital Equipment Corp and Philips Semiconductors, is offered as an open industry standard—free of fees or royalties. Indeed, an independent ACCESS.bus Industry Group (ABIG) now owns the bus specification and controls development of the protocols. ABIG currently boasts 55 members, including Intel, DEC, Fujitsu, Honeywell, IBM, Key Tronic, Lexmark, Logitech, Microsoft, Philips Semiconductors and Sun Microsystems.

ACCESS.bus is also supported by Microsoft in both MS-DOS and Windows, and by DEC VMS and Sun Solaris 2.x OSs. Several ABIG members also offer ACCESS.bus software-development tools, including source code modules, cross assemblers, in-circuit emulators and EPROM programmers. Tailored compilers also assist C and PL/M codesmiths.

Electrically, ACCESS.bus is based on the popular I²C (inter-integrated circuit)

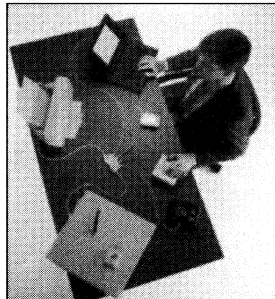
bus—an inexpensive two-wire serial implementation widely used to transfer data between 8051-family 8-bit microcontrollers in close proximity on circuit boards or within a system. These days, I²C ports are included on many other microcontrollers and memories. Some UARTs, LCDs, I/O expanders, clock/calendars, and video and audio devices made by a variety of chip makers and peripheral manufacturers also include I²C implementations.

One of the strongest features of ACCESS.bus is its simplicity. Only four wires are required: a serial data line, a clock, power (+5 V) and ground. Cables and connectors are similar to RJ-style modular telephone cords, with the addition of shielding to meet FCC EMI/RFI guidelines. In operation, ACCESS.bus peripherals are typically daisy-chained on cable runs that extend up to 25 ft. For some applications, this distance can actually be extended to 250 ft.

Hot swapping

A host computer configures all devices on the bus by assigning a unique address to each attached device at power-up, or when a device is connected. Significantly, such auto-addressing lets devices be added or removed without rebooting or powering-down a system. The host

interface configures each new device as it's plugged in, also polling the ACCESS.bus periodically to determine if any devices have been disconnected. On the host side, a typical ACCESS.bus interface is implemented with one microcontroller, two current source pull-ups, and an interface from the con-



A single, daisy-chained serial ACCESS.bus structure can support a variety of full-sized peripherals—especially keyboards and monitors. This typical configuration converts a powerful portable PC into a useful desktop adjunct.

ACCESS.bus revisited— ending the peripheral connection nightmare

SPOTLIGHT ON PORTABILITY

troller to the host CPU. This compact configuration makes it easy to include the ACCESS.bus interface on virtually any portable computer's motherboard.

Unadorned device drivers

On each peripheral, a single ACCESS.bus microcontroller is also needed. As each device includes its own device-specific configuration information, however, it's easy to reconfigure a generic device driver so you don't have to code custom drivers. A generic pointing device driver, for example, could operate with a mouse, trackball, joystick, graphics tablet or data gloves.

ACCESS.bus also supplies up to 1 A of current to power devices on the bus, although peripherals can also obtain power externally if desired. Each device connects to the bus' data and clock lines via open-collector or open-drain transistors, which are part of the ACCESS.bus hardware implemented in the controllers. In operation, all ACCESS.bus devices (host and any peripherals) are deemed bus masters when sending messages, and slaves when receiving messages; all messages on the bus are from one device to another, communicating in a half-duplex collision detection mode. If the

host wishes to receive position data from a pointing device such as a mouse, it sends a message to the peripheral asking for that information. The peripheral responds with a separate message containing the requested information.

The payoff

ACCESS.bus delivers a simple means for connecting nearly unlimited numbers of peripherals without rebooting. Such simplicity has multiple payoffs:

- OEMs only have to provide one ACCESS.bus port per portable computer. Unlike conventional multi-port approaches, no multiple UARTs are needed;
- Plug-and-play peripherals can readily be connected to virtually any and all platforms, regardless of whether they are DOS-based PCs, Macintoshes or workstations;
- Developers can create software that handles multiple cursors. This opens totally new application possibilities. For example, computer games could support multiple players that can join in or drop out at will;
- Peripheral makers can get out of the specialized software driver business and concentrate on their core business, thereby saving development

costs. They can also avoid authoring lengthy installation procedures and manuals;

- Distributors and resellers can reduce inventories; only one version of each peripheral is needed. They'll also save time and manpower by not having to explain installation to oft-confused end-users;
- Organizations will save on duplicate peripherals for people working on multiple systems. You could, for example, unplug a favorite mouse, trackball, digitizer or scanner, and plug it into another computer. ■

For more information about the technologies, products or companies mentioned in this article, call or circle the appropriate number on the Reader Inquiry Card.

ACCESS.bus Industry Group
Sunnyvale, CA
(408) 991-3517

Philips Semiconductors
Sunnyvale, CA
(408) 991-3518

Seriously serial

BYTE

THE MAGAZINE OF TECHNOLOGY INTEGRATION

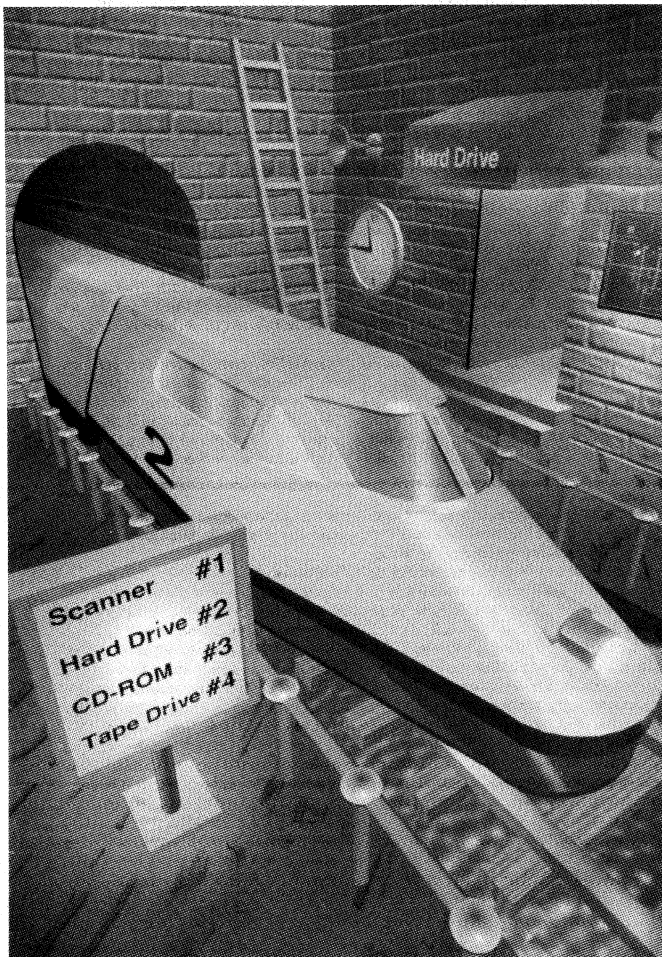
Reprinted with permission, from the August 1994 issue of BYTE Magazine.
© 1994 by McGraw-Hill, Inc., New York, NY. All rights reserved.

State of the Art

SERIOUSLY SERIAL

The venerable RS-232 standard may be showing its age, but Access.bus and FireWire demonstrate that there is a lot of life left in serial technology

MARK CLARKSON



Those new gigabyte hard drives, flat-screen true-color monitors, quad-spin CD-ROM drives, and V.Fast modems are all very nice, and they add a lot of functionality to your computer system; however, the PC still has a long way to go in terms of letting you connect these devices quickly and easily to your computer. If you want proof, you need look no further than the rat's nest of cables that's coiled behind my computer—and probably behind yours, too. There must be a better way to hook all those fancy peripherals up to our computers.

Yes, Serial

It appears to many in the industry that most of the next generation of peripheral connections will be serial, transferring data 1 bit at a time rather than 8 or more bits in parallel. Why? Parallel communications require more wires to carry the signals. More wires means fatter, more expensive cables and wider, more expensive connectors with more pins to bend and break. Also, the more wires bundled together into a cable, the more electrical interference there is between signal wires, and the more thoroughly they have to be shielded. The more signals you deal with at the same time, the harder it is to keep them all in sync. And the faster and farther you try to push that signal, the worse the problems become.

Add to this the fact that the size of computers continues to shrink. Some fit into a shirt pocket; they have little room for connectors today and will have less tomorrow. Yet even as computers become smaller in size, their power increases, and more than ever, people need to be able to hook them up in the real world. What they want then are small connectors and darned few of them. Serial links, with intrinsically fewer wires, require smaller connectors.

continued

HEIK DAMON © 1994

AUGUST 1994 **BYTE**

Seriously serial

State of the Art Seriously Serial

Pumping Up the Parallel Port

An alternative to high-speed serial is sitting right in your PC

The Centronics-style parallel port, says Larry Stein, president of Farpoint Systems (Jersey City, NJ), "became an industry standard the day IBM introduced the personal computer with a parallel port on it." At that time, a screamingly fast peripheral was a 240-character-per-second dot-matrix printer, which received its data in ASCII. Today, you're more likely to be hooked up to an 8-page-per-minute laser printer, pumping down megabyte-size color graphics files. Yet if you own a newer PC, the parallel port on the back of it is identical in performance to the 1981 model whose speed tops out at around 150 KBps. Maintaining even that rate is processor-intensive—your CPU has to oversee moving data to and from the port, including all the handshaking between the computer and peripheral.

The 150-KBps parallel port is woefully slow for many new printers, which can accept data at much higher speeds, as well as for other peripherals such as tape drives. It is inadequate for even the slowest of single-spin CD-ROM drives. It's obvious that the parallel port needs major pumping up if it's going to survive on the multimedia desktop.

The IEEE 1284 standard defines a newer, faster, and better parallel port with some major muscle in it. The good news is that 1284 ports are backward-compatible with existing parallel ports; the Hewlett-Packard DeskJet will plug right in, using existing cable and connector. Even better news is that, for only the cost of a new cable, a 1284 parallel port can inject new life into your old printers.

Current parallel connections are notoriously flaky at distances of more than 10 feet. "With 1284," says Stein, who is also chairman of the 1284 committee, "we wanted to go a minimum of 10 meters. It's important to realize that, in some cases, 1284 has the parallel port going 100 to 200 times faster than it was meant to originally. You can't do that using S2 cables, so the 1284 standard defines the cable as well. Now we can guarantee that when a user buys a 1284 port, a 1284-capable product, and a 1284 cable, it'll work at 2 feet, 10 feet, or 30 feet."

IEEE 1284 actually defines four different modes for the new parallel port: nibble, byte, ECP (Enhanced Capabilities Port), and EPP (Enhanced Parallel

Port). All modes have at least some bidirectional ability, allowing the printer to talk back to your computer. Data passing from the peripheral to the host is called *back-channel communication*.

The first two modes, nibble and byte, provide for relatively slow back-channel communication, 4 and 8 bits at a time, respectively. ECP, intended mainly for host-to-printer connections, can achieve data rates of up to 4 MBps in both directions. The maximum speed depends on the peripheral and host computer.

EPP allows you to attach devices such as CD-ROM and hard drives, which would normally plug into the internal bus, to the parallel port. In addition to high data speeds, EPP allows the system to regard the parallel port as an extension of the system bus. Although not as sophisticated as FireWire, EPP lets you hang multiple peripherals off a single port.

All 1284-compliant devices can identify themselves and their capabilities to the host computer, letting the system know whether to speak in EPP, ECP, nibbles, or plain old Centronics. Microsoft has announced support for ECP-

Get on the Serial Bus

Ideally, people want to hook together lots of peripherals—and many different kinds of peripherals—with as little muss and fuss as possible. The way to do that, says Apple research scientist David James, is by using a bus: "People already know how to transfer data on buses, to and from a large number of devices. There's really no problem mapping a keyboard, a network, a disk, a storage device, or other things to a bus because everybody's been doing that for years."

"But until the last few years," James continues, "people had always thought that buses had to be constrained to a backplane—inside a chassis." A serial connection, however, extends the idea of a bus outside of the computer, to the desktop and beyond.

Computers and peripherals simply read from and write to addresses on the serial bus. Those addresses represent other peripherals. These reads and writes can be interleaved, allowing multiple devices to communicate across the bus at the same time, so you can string a whole slew of peripherals off of a single port on a PC. James also points out that these transfers aren't restricted to those between the host computer and a peripheral: "They can go from one disk to another or from disk to printer, autonomously. The computer would tell the disk to start dumping data to some device, and then the disk would continue on its own, while your processor does something else."

Sound good? A number of people think so. Two new serial-bus standards, Access.bus and P1394 (or FireWire), are hot

candidates to become the primary peripheral ports on your desktop computer.

Access.bus

Access.bus is a new standard intended to connect relatively low-speed devices such as keyboards, mice, modems, and printers. Originally conceived by DEC and Philips/Signetics and similar to the Apple Macintosh desktop bus, Access.bus is now being developed under the auspices of the ABIG (Access.bus Industry Group).

Access.bus runs on a thin four-wire cable that resembles the one that currently connects the keyboard or mouse to your computer. Each end sports a single small connector that's a little bigger than an RJ-11 modular phone jack. Most Access.bus devices offer two sockets—in and out—to let devices be easily chained together, similar to

Seriously serial

compliant devices within Chicago, the next version of Windows. Chicago will use the ID string returned by ECP-compliant peripherals to automatically install the proper drivers.

An EPP can even drive your existing printers much faster. Copy some graphics files to your plain-Jane, non-1284 printer today, comments Stein, and you'll get throughput ranging from 6 to 14 Kbps. With an EPP port and a new 1284 cable, however, you could get as much as 500 Kbps. "By the end of the year," Stein says, "you should see ECP printers capable of 400 Kbps to 2 MBps."

And the 1284 port has one thing going for it that none of the proposed serial standards do: an installed base of millions on millions of parallel-port-equipped devices, all of which will plug right in.

But there's still a serious question about how well these new parallel ports will work when you add in all the extra baggage needed to allow them to compete with the new serial buses. To me, this technology stinks of death. The more I read about it and how much work goes into tweaking parallel ports to run 100 times faster than what was intended—with special cables, strict capacitance rules, and so on—the more convinced I am that the serial-port solutions are going to take over pretty quick.

the way SCSI devices are interconnected.

Access.bus has a significant advantage over standard RS-232 serial connections. Today, for each and every serial peripheral, you need a separate port, interrupt, system address, and perhaps more. In theory, Access.bus will let you run up to 125 peripherals—over a total cable distance of 8 meters—from a single jack in the back of your computer. "The beauty of Access.bus," says David Rogers, manager of new business development for Computer Access Technology (Santa Clara, CA), "is that I'm using only one hex address, whether I've got one device on that port or 125."

Hot Plugs, Cold Boots

Every Access.bus peripheral, from laser printer to lowly mouse, is intelligent. Each contains a microcontroller that can identi-

fy the device to the bus and pass data along to the next peripheral in the chain.

According to Rogers, Access.bus intelligence will improve your system's performance. "The messages passing over the Access.bus are off-loaded from your computer and CPU. The Access.bus host and the peripherals in the line are handling the passage of the messages, the bus arbitration, and so on. The system will actually run faster, because you don't have so many interrupts to the CPU." Where an RS-232 serial port interrupts the processor with every bit of information received, data moves along the Access.bus in the form of messages that are up to 127 bytes long, and each message generates only a single interrupt. This translates into far better system performance.

But there's more. Access.bus also supports *hot plugging*. This means that you can disconnect peripherals and plug them in without having to power the computer down or reconfigure the system. There are no jumpers to set, no DIP switches to throw, and no IRQs (interrupt requests) to reconfigure. Everything is automatic.

When the system is first powered up, the Access.bus master inside your computer sends a message to every device on the bus. Each device responds with its ID number and a string that identifies what type of device it is—for example, a locator, keyboard, or text device—and gives any special capabilities and characteristics.

The bus master assigns each peripheral an address on the serial bus and maintains a table of attached devices and their addresses. The individual peripherals watch the bus for reads or writes to their particular addresses and move data onto and off the bus accordingly.

And whenever you pull a peripheral off the bus or plug in a new one, the Access.bus master notices this and dynamically reconfigures the bus, requiring no user input at all. "For example," says Rogers, "we have a demo blackjack game for Windows, which uses Access.bus drivers for multiple mouse input. I can hang six mice on my PC. Each has its own address, and each can only manipulate the chips and cards associated with it," he continues. "As I add or subtract players from the game, by adding and subtracting mice, the Access.bus automatically reconfigures the system accordingly."

Open Access

Access.bus is an open standard that promoters hope will find a place on a variety

of platforms, including PCs, Macs, and workstations. And the industry *needs* a new standard, especially in the fast-expanding arena of notebooks, subnotebooks, and PDAs (personal digital assistants)—smaller products with smaller connection space, where port real estate is at a real premium. Currently, users of such small computers who want to connect multiple peripherals must often suffer with proprietary interfaces and ungainly docking stations.

According to Rogers, "If a notebook manufacturer can add a small phone-jack-type port without having to add a proprietary connector, then it has opened up its product line to additional third-party solutions."

Another limitation involves the number of access ports available. "Even on the desktop," Rogers notes, "where port economy is not as important, I'm still limited to four comm ports. If I want to go to a multiple RS-232 connection, I'm paying a premium, and I still have to worry about lower-level interrupts and DMA calls and older software that's unable to find the additional ports."

A new specification for Super VGA monitors, called DDC2, calls for Access.bus to be incorporated into the monitor-to-PC connection. This will let you manipulate the front-panel controls—video mode, tint, brightness, and color—through software. In addition, manufacturers will be able to put Access.bus receptacles on the monitor, allowing you to plug your keyboard or mouse into your monitor, which will be more convenient than the standard back-of-the-computer location.

How far away is Access.bus? Very close indeed, says Rogers. "Today, I can buy a host adapter, keyboards, trackballs, joysticks, mice, modems, and RS-232-to-Access.bus converters for older 232-based products. And I can operate under Windows 3.1, Windows NT, Solaris 2.3, and DOS. And if Chicago was shipping today, I could run under Chicago, too, because those drivers already exist."

How Fast Is Relatively Slow?

For all its advantages, Access.bus won't serve for every type of peripheral because it doesn't have enough bandwidth. Access.bus runs at speeds of up to 125 Kbps, which is not fast enough for multimedia applications involving high-quality audio and video or for hooking up hard drives. It is fast enough to run any device (e.g., mice, trackballs, joysticks, printers, keyboards,

Seriously serial

State of the Art Seriously Serial

“The marketplace clearly wants a connection to carry high-performance video and audio. It wants a connection that provides much higher bandwidth than has been required in the past and that connects to both PCs and consumer products. For the first time, we have a [P1394, or FireWire] cable that is being designed into both the computer and consumer worlds.”

Bryan Bell
Texas Instruments



and so on) that you might have attached to a serial port.

“With Access.bus today,” says Rogers, “I could have a 19.2-Kbps modem transmission in progress, be using my keyboard or my mouse, and have a bar-code scanner operating—all at the same time, on the same bus.”

FireWire

If Access.bus isn't fast enough for you, then maybe it's time to step on up to FireWire. This new high-speed desktop serial bus, based on the ANSI draft standard P1394, is being developed jointly by Apple and Texas Instruments.

FireWire also uses a flexible cable plus a nice, small connector inspired by the one used in the Nintendo Gameboy. You can reach behind your machine, without looking, and plug it in.

FireWire offers many of the same advantages as Access.bus. You can daisy chain peripherals on a FireWire bus, hanging up to 63 devices off a single port. Up to 1022 FireWire buses can be bridged together, which should provide enough peripherals for anybody. As with Access.bus, FireWire provides hot plugging and automatic configuration. There's no need to set DIP switches or pull jumpers; you just plug in your cables and, as long as everything's connected correctly, everything

works. (See the figure “P1394 Serial-Bus Physical Topology” on page 122.)

But FireWire takes this concept to a whole new level. The goal is for each and every FireWire-compatible device on the planet to have its own unique 64-bit ID number. If you plug in a mouse (e.g., a Logitech three-button mouse), FireWire can identify it. It not only knows it's a Logitech three-button mouse but also exactly which Logitech three-button mouse it is. If two identical mice are connected to the system, FireWire can tell which is which.

For example, says Apple's James, “Say you had a disk drive with the unique identifier ABCD at location one. Then you move that disk, and the system finds that ABCD is now at location five. That's all right, because it just adjusts the operating-system tables accordingly. The unique identifier makes it very easy to find out where a peripheral has moved to.”

How Far, How Fast?

Even if you use Access.bus, says James, “you still need another, faster bus for your disk. Well, why shouldn't you just use that faster bus for everything? The clear win is not in adding another connector to the computer but in eliminating one.”

Although it's significantly faster than Access.bus, FireWire is still strictly limited in the bandwidth it can deliver and the distance it can push a signal. It was, after all, designed for the desktop, and reasonable compromises had to be made to meet FireWire's low-cost objectives. According to James, “You certainly wouldn't want to run FireWire [over a cable length of] 50 meters.”

FireWire is no slouch. It operates at speeds ranging from 100 Mbps to 400 Mbps, which—protocols and overhead aside—should translate into 5 to 20 MBps of data actually humping across the wire from point A to point B.

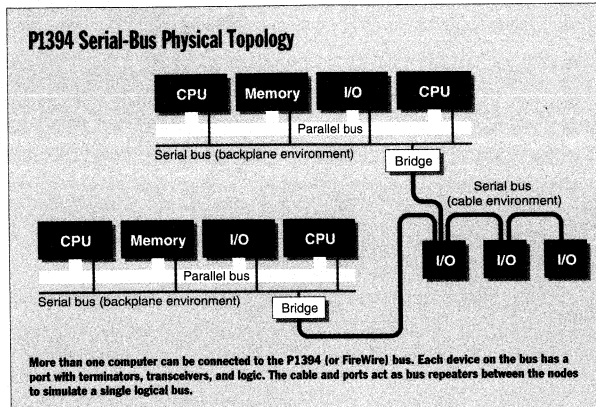
To drive data reliably at such high speeds, FireWire uses a technique called *differential signaling*. The cable contains two data lines (I'll call them A and B), and it uses both together to signal 1 bit of data. For a logical 1, A is high and B is low; for a logical 0, A is low and B is high. A FireWire cable also distributes power of from 8 to 40 VDC, at up to 1.5 amps.

continued

ILLUSTRATION: DAVID LESH © 1994

Seriously serial

State of the Art Seriously Serial



FireWire's speed is not only fast enough for normal serial communications between mice, modems, and such; it's also fast enough to support real-time video and high-fidelity audio. FireWire supporters want to see it break out from the desktop and into the consumer electronics arena. They predict that FireWire ports will appear on camcorders, VCRs, and CD players. And if you can hook up camcorders and VCRs, why not just hook your camcorder to your VCR *through* your computer? Just think how easy that would make it to get video into your FireWire-equipped, multimedia computer.

"The marketplace clearly wants a connection to carry high-performance video and audio," says Bryan Bell, manager of computers and computer peripherals at Texas Instruments. It wants a connection that provides "much higher bandwidth than has been required in the past and that connects to both PCs and to consumer products. For the first time, we have a cable that is being designed into both the computer and the consumer worlds. It's really revolutionary."

Will They Fly?

Which of these serial standards is likely to catch on: Access.bus or FireWire? Maybe both? Or something like serial SCSI-3? Access.bus is basically a replacement for aging RS-232 technology, while FireWire has its sights set on new multimedia applications and consumer electronics. In addition, higher bandwidth comes at a price. Where Access.bus compliance might add 10 or 25 cents to the

cost of a peripheral, FireWire will likely add about \$1 to \$10.

Some pundits doubt Access.bus's usefulness as a printer connection. The communications needs of printers far exceed those of simple keyboards, mice, and modems. At 125 Kbps, Access.bus runs at only one-quarter the speed of today's Centronics-style parallel ports. PostScript documents that run 250 KB per page are not uncommon, even today, and that's just text with fancy fonts. Increasingly, printed documents are incorporating complex, computer-generated charts, drawings, and bit maps. As video and multimedia emerge as mainstream applications, we'll be seeing captured video stills, which will inflate document sizes even more. Adding color to this mix makes the problems more acute.

At present, neither of these two standards has any installed base to speak of. "It's a Catch-22," says Apple's James. "It's hard to justify putting connectors on a motherboard before peripherals exist that connect to it." Still, the advantages of size, speed, and standardization are too great to be ignored. A trickle of serial-bus products is already appearing and should turn into a torrent by the middle of next year.

"The idea of the serial bus," says James, "has allowed the bus to creep out of the box and onto the desktop. The interesting question then is how far will it creep? Will it just be to the desktop or will it eventually cover the whole building?" ■

Mark Clarkson is a freelance science writer living in Wichita, Kansas. He can be reached on the Internet or BIX at mclarkson@bix.com.

Section 6

Control Area Network (CAN) Bus

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

Control Area Network (CAN) overview	335
82C150 CAN serial linked I/O device (SLIO) with digital and analog port functions	336
82C200 Stand-alone CAN-controller	365
PCA82C250 CAN controller interface	401

Control Area Network (CAN) overview

CAN OVERVIEW

The Control Area Network (CAN) is a multiplexed wiring protocol developed by Bosch for use in automotive applications. As products supporting CAN have been made available by Philips Semiconductors and other semiconductor manufacturers, the protocol has become used in other industries including: industrial automation, machine tools, medical equipment, and building environmental control, to name a few. The CAN protocol is attractive for use in a wide

range of applications because it has powerful error detection capabilities and features differential drive, and can be used comfortably in critical high noise environments. CAN is also very flexible in terms of the transmission media and the connection scheme, and is generally easy to adapt to most applications.

Philips offers a wide range of parts that support the CAN protocol, including

stand-alone parts as well as microcontrollers with integrated CAN interfaces. Datasheets for the 82C200 (Stand-alone CAN controller), 82C150 (CAN serial Linked I/O device), and 82C250 (CAN transceiver) are included in this section of this databook. Datasheets for the microcontrollers that have an integrated CAN interface (8XC592 and 8XC598) are included with Philips' 80C51 family products in Section 3 of this databook.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

1 FEATURES

- Single-chip I/O device with CAN protocol controller
- Meets CAN protocol specification version 2.0 A and B (passive) with restricted bit timing
- Fully integrated clock oscillator (no crystal required)
- 16 configurable digital or analog I/O port pins
- Each of the port pins individually configurable via CAN-bus: port direction, port mode and event capture facilities for inputs (event driven or polling)
- Up to sixteen digital inputs:
automatic transmission of a CAN message on a change on inputs individually selectable
- Up to sixteen 3-state outputs
- Up to two quasi-analog outputs with 10-bit accuracy
- 10-bit analog-to-digital converter with up to six multiplexed analog input channels
- Two general purpose comparators
- Bit rate from 20 kbit/s up to 125 kbit/s using internal oscillator
- Automatic bit rate detection and calibration
- Up to sixteen P82C150 nodes for one CAN-bus system
- Four identifier bits programmable
- SLIO functions controlled by a single intelligent node ("host")
- Sleep-mode with wake-up via CAN-bus
- Differential CAN-bus input comparator and CAN-bus output driver
- 5V \pm 4% supply voltage range
- Operating temperature range from -40 to +125°C

2 GENERAL DESCRIPTION

The P82C150 is a single-chip 16-bit I/O device including a Controller Area Network (CAN) protocol controller with automatic bit rate detection and calibration. It features 16 configurable I/O port pins with programmable direction, digital and analog modes.

The P82C150 provides a configurable event capture facility supporting automatic transmission caused by a change on the port input pins.

The clock oscillator requires no external components, thus, the cost of the CAN link is reduced significantly.

The P82C150 is a very cost-effective way to increase the I/O capability of a microcontroller based CAN node as well as to reduce the amount and complexity of wiring. Advanced safety is provided by the CAN protocol.

Applications:

- Body electronics and instrumentation in automotive applications
- Sensor/actuator interface in automotive and general industrial applications
- Extension of I/O capabilities of microcontroller based CAN nodes

3 ORDERING INFORMATION

TYPE NUMBER	PACKAGE			TEMPERATURE RANGE (°C)
	NAME	DESCRIPTION	VERSION	
P82C150 AHT	SO28	Plastic Small Outline; 28 leads; body width 7.5 mm	SOT136-1	-40 to +125

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

4 BLOCK AND FUNCTIONAL DIAGRAMS

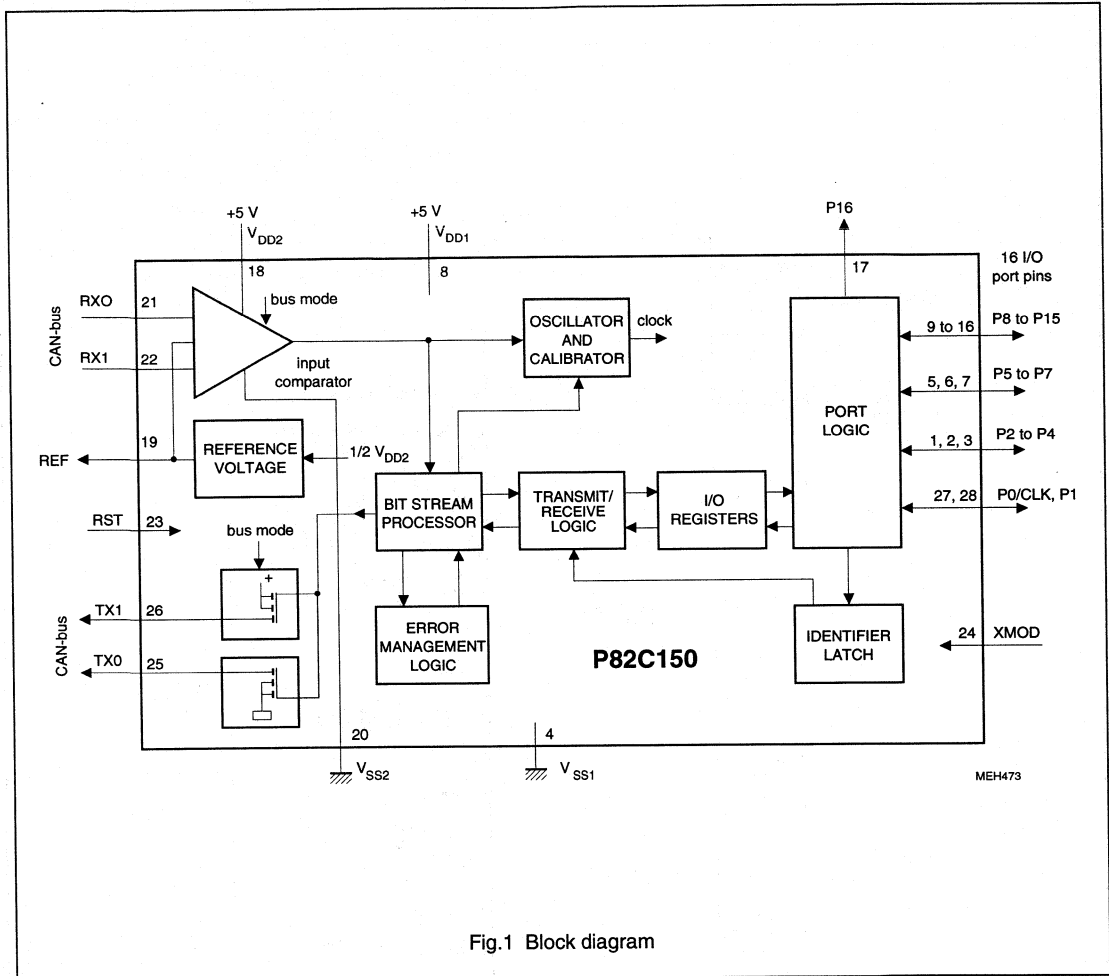


Fig.1 Block diagram

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

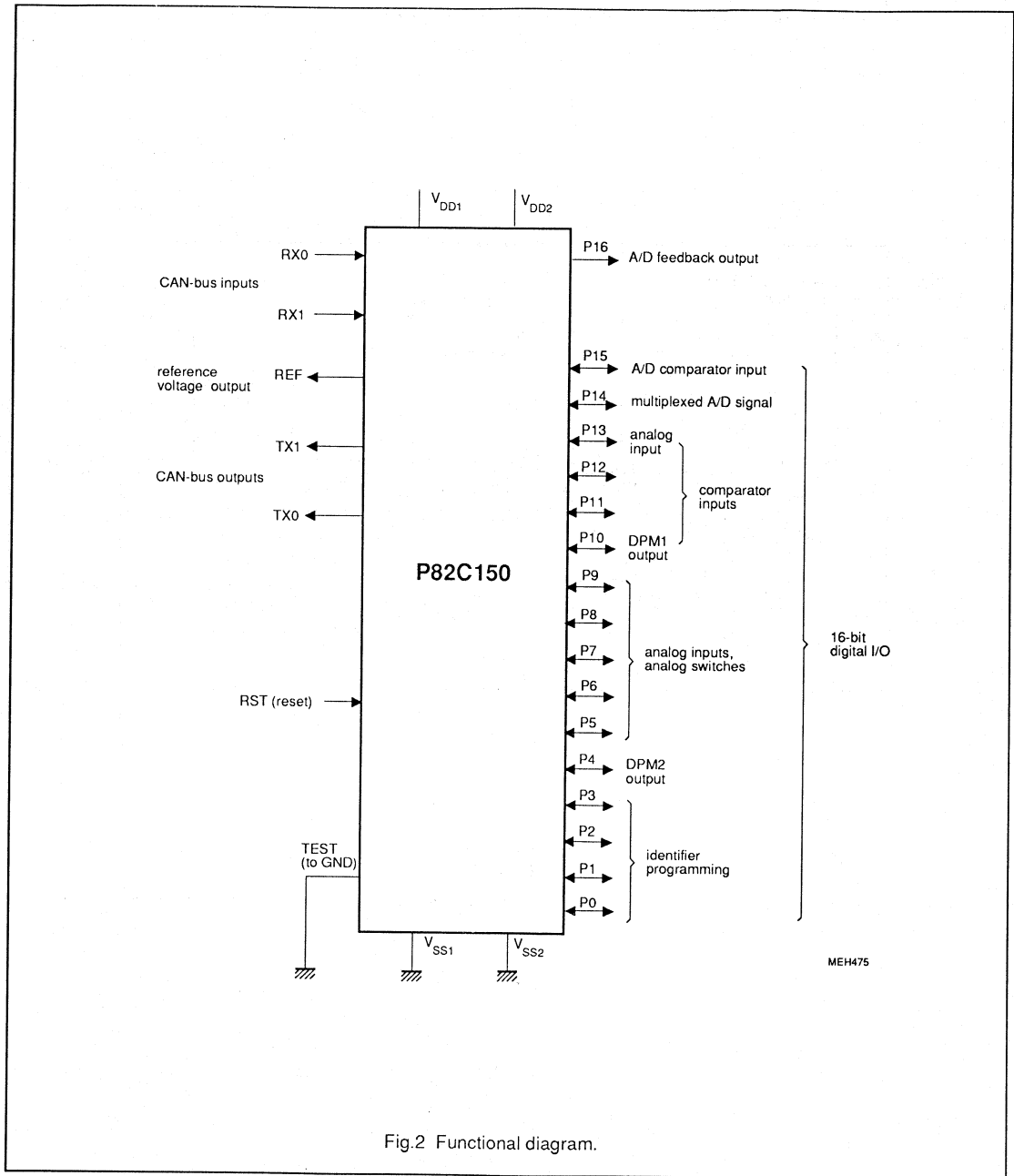


Fig.2 Functional diagram.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

5 PINNING

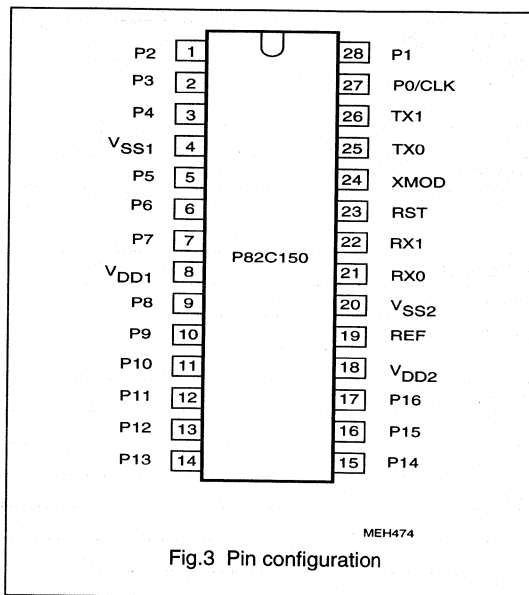
SYMBOL	PIN	DESCRIPTION
P2	1	I/O ports P2 to P3 ; Identifier programming input
P3	2	
P4	3	I/O port 4 ; DPM2 output
V _{SS1}	4	Logic ground (0V; logic circuits and CAN-bus driver)
P5	5	I/O ports P5 to P6 ; analog input
P6	6	
P7	7	I/O port 7 ; analog input or A/D comparator 1 output
V _{DD1}	8	+5V supply voltage (logic circuits and CAN-bus driver)
P8	9	I/O port 8 ; analog input or comparator 3 output
P9	10	I/O port 9 ; analog input or comparator 2 output
P10	11	I/O port 10 ; comparator 3 inverting input or DPM1 output
P11	12	I/O port 11 ; comparator 3 non-inverting input
P12	13	I/O port 12 ; comparator 2 inverting input
P13	14	I/O port 13 ; comparator 2 non-inverting input
P14	15	I/O port 14 ; multiplexed analog signal
P15	16	I/O port 15 ; A/D comparator input
P16	17	Feedback output of A/D converter
V _{DD2}	18	+5V supply voltage (CAN input, oscillator, reference)
REF	19	Reference voltage output (1/2 V _{DD2})
V _{SS2}	20	Analog ground (0V; CAN input, oscillator, reference)
RX0	21	CAN-bus input
RX1	22	
RST	23	External reset input (active-HIGH) for internal oscillator mode; pulled to +5V for external oscillator mode (see Section 10.3)
XMOD	24	Connected to GND for internal oscillator mode ; external reset input (active-LOW) for external oscillator mode (see Section 10.3)
TX0	25	Open-drain CAN-bus output : dominant = LOW; recessive = floating
TX1	26	Open-drain CAN-bus output : dominant = HIGH; recessive or at bus mode 2 floating
P0/CLK	27	I/O port P0 , Identifier programming input in internal oscillator mode; clock input in external oscillator mode (see Section 10.3)
P1	28	I/O port P1 ; identifier programming input

Note

1. In this documentation the port pins are referred to by their symbols, not by their pin number. For example P15 means I/O port 15 at pin 16

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150



6 FUNCTIONAL DESCRIPTION

6.1 I/O functions

The P82C150 provides 16 port pins (P15 to P0) which are individually configurable via CAN-bus. Besides the digital I/O functions some of these port pins provide analog I/O functions.

Digital input functions:

Input levels HIGH and LOW on the port pins (P15 to P0) can be read in two ways by the host node:

- **Polling:** a Remote Frame is sent to the P82C150 to be answered by a Data Frame containing the Data Input Register contents.
- **Event capture:** in case of edge-triggered mode, the P82C150 sends the same Data Frame caused by the event of a rising and/or falling edge on the corresponding port pins (see Table 2).

Digital output functions:

The Data Output Register is set via a CAN message. Its content is only output when the corresponding bits of the Output Enable Register are set to "1".

Analog input/output functions:

- Up to six multiplexed analog input signals for A/D conversion or general purpose

- Up to two quasi-analog output channels (DPM)
 - Two input comparators, for example for window comparator applications
 - A separate A/D input comparator with feedback output
- A/D converted digital results are obtained by reading the A/D register. Analog functions of each port pin are individually controlled by the Analog Configuration Register.

Writing the I/O registers is done serially via CAN-bus by Data Frames. The first data byte contains the register address, and the second and third data bytes represent the register contents. If a read-only register is addressed, the contents of the second and third data bytes are ignored.

It is recommended to set unused port pins to HIGH (100 kΩ resistor to V_{DD}).

6.1.1 DATA INPUT REGISTER (ADDRESS 0; READ ONLY)

This register contains the states of port pins P15 to P0 which are transmitted on request, or automatically by change of one of the input levels, provided that the respective input is configured to event capture mode (see Table 2). When an edge is detected the port state is loaded into the transmit buffer after the Control Field of the triggered message is sent. Thereby a delay for input settling is provided. If between edge detection and transmission of the data input register another input signal change at the input port occurs, the corresponding data input register bit is overwritten by the current input port value. Additionally the register content is sent automatically after wake up or bus mode change, once the bit time has been calibrated (part of the "sign-on" message).

6.1.2 POSITIVE EDGE REGISTER (ADDRESS 1; WRITE ONLY)

This register contains configuration information per port pin for the event capture facility. The corresponding PE-bit (see Table 2) has to be set to "1" to enable capturing of the falling edge.

6.1.3 NEGATIVE EDGE REGISTER (ADDRESS 2; WRITE ONLY)

This register contains configuration information per port pin for the event capture facility. The corresponding NE-bit (see Table 2) has to be set to "1" to enable capturing of the falling edge.

The combination of PE and NE functions is possible.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

Table 1 I/O Register map.

MSB	LSB															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDRESS 0: DATA INPUT																
DI15	DI14	DI13	DI12	DI11	DI10	DI9	DI8	DI7	DI6	DI5	DI4	DI3	DI2	DI1	DI0	
ADDRESS 1: POSITIVE EDGE																
PE15	PE14	PE13	PE12	PE11	PE10	PE9	PE8	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0	
ADDRESS 2: NEGATIVE EDGE																
NE15	NE14	NE13	NE12	NE11	NE10	NE9	NE8	NE7	NE6	NE5	NE4	NE3	NE2	NE1	NE0	
ADDRESS 3: DATA OUTPUT																
DO15	DO14	DO13	DO12	DO11	DO10	DO9	DO8	DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0	
ADDRESS 4: OUTPUT ENABLE																
OE15	OE14	OE13	OE12	OE11	OE10	OE9	OE8	OE7	OE6	OE5	OE4	OE3	OE2	OE1	OE0	
ADDRESS 5: ANALOG CONFIGURATION																
ADC	OC3	OC2	OC1	0	M3	M2	M1	SW3	SW2	SW1	0	0	0	0	0	0
ADDRESS 6: DPM1																
DP9	DP8	DP7	DP6	DP5	DP4	DP3	DP2	DP1	DP0	0	0	0	0	0	0	0
ADDRESS 7: DPM2																
DQ9	DQ8	DQ7	DQ6	DQ5	DQ4	DQ3	DQ2	DQ1	DQ0	0	0	0	0	0	0	0
ADDRESS 8: A/D																
AD9	AD8	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	0	0	0	0	0	0	0

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

Table 2 Programming of the I/O registers to event capture on edge or to digital output (X = don't care).

REGISTER CONTENTS OF PARTICULAR PORT PIN		DIGITAL INPUT FUNCTION				DIGITAL OUTPUT FUNCTION
		Polling	Event Capture on Edge			
			RISING	FALLING	RISING AND FALLING	
Positive edge	PE	X	1	0	1	X
Negative edge	NE	X	0	1	1	X
Output enable	OE	X	X	X	X	1

6.1.4 DATA OUTPUT REGISTER (ADDRESS 3; WRITE ONLY)

This register contains the output data for the port pins. The output drivers are bitwise enabled by OE (see Section 6.1.5). New data for the output port register are processed and written to the output ports directly after the corresponding CAN message to the P82C150 is successfully checked and becomes valid.

6.1.5 OUTPUT ENABLE REGISTER (ADDRESS 4; WRITE ONLY)

This register controls the output drivers of the port pins. The corresponding Output Enable Register bit has to be set to "1" to enable an output driver. If set to "0", the corresponding output driver is disabled (floating; see Fig.4).

6.1.6 ANALOG CONFIGURATION REGISTER (ADDRESS 5, READ/WRITE)

This register contains the bits ADC, OC3 to OC1, M3 to M1 and SW3 to SW1 (see Fig.5).

ADC bit (A/D conversion start bit; write-only bit):

The P82C150 starts an A/D conversion cycle at ADC = 1 ended with the transmission of a message containing the result. After that, the ADC bit is reset automatically.

OC3 to OC1 bits (comparator output data; read-only bits):

The bits OC3 to OC1 represent the logical output level of the analog comparators at input port pins P10, P11, P12, P13 and P15. The P82C150 sends back the logical output value of these comparators after having received a Data Frame (see Section 6.2.3) addressing the Analog Configuration Register. The comparator outputs can be monitored at the output port pins P8, P9 and P7.

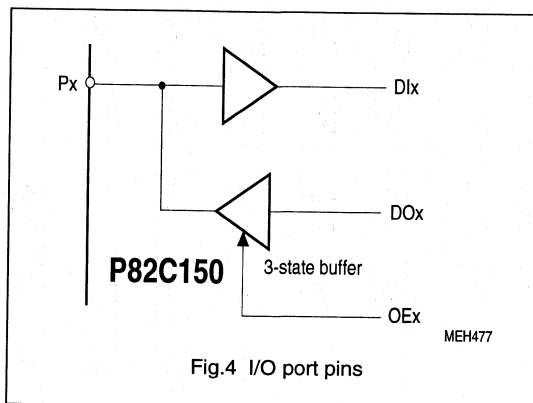


Fig.4 I/O port pins

M3 to M1 bits (multiplexer control bits; write-only bits):

The logical value of the comparators is monitored on port pins P8, P9 and P7 (see Fig.4) by setting M3 to M1 to "1", provided that these pins are configured as outputs (OE = 1). Additionally the register content is sent automatically when the corresponding port bits in the Positive Edge Register and/or Negative Edge Register and the corresponding bits in the Output Enable Register are set (see Fig.4).

SW3 to SW1 (analog switch control bits; write-only bits):

One of the analog switches S1 to S6 can be closed by setting the switch bits to the corresponding value (see Fig.5).

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

SW3	SW2	SW1	
0	0	0	no switch closed (S0)
0	0	1	S1 closed
0	1	0	S2 closed
0	1	1	S3 closed
1	0	0	S4 closed
1	0	1	S5 closed
1	1	0	S6 closed
1	1	1	reserved

Evidently if P14 is driven, it may not be connected to any other driven pin via the internal analog switches (avoid short-circuit!).

6.1.7 DPM1 REGISTER (ADDRESS 6; WRITE ONLY)

A quasi-analog output signal on port pin P10 is generated by distributed pulse modulation (DPM; see Fig.8) if the Output Enable bit is set (OE10 = 1). The DPM1 output signal is inverted by setting DO10 = 1. The number of output pulses during a DPM period is given by the DPM1 Register value. These pulses have $4 \times t_{CLK}$ length and are distributed over the DPM period (see Fig.8). An analog voltage is provided after smoothing the output signal by an external RC combination.

6.1.8 DPM2 REGISTER (ADDRESS 7; WRITE ONLY)

The function of the DPM2 output (P4) and the DPM2 Register correspond to the definition of DPM1.

6.1.9 A/D REGISTER (ADDRESS 8; READ ONLY)

This register contains the result of the A/D converted level of that I/O pin which was selected by the SW bits. The conversion is started by ADC-bit set to "1" (see Section 6.1.6), or by transmitting a Data Frame addressing the A/D Register.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

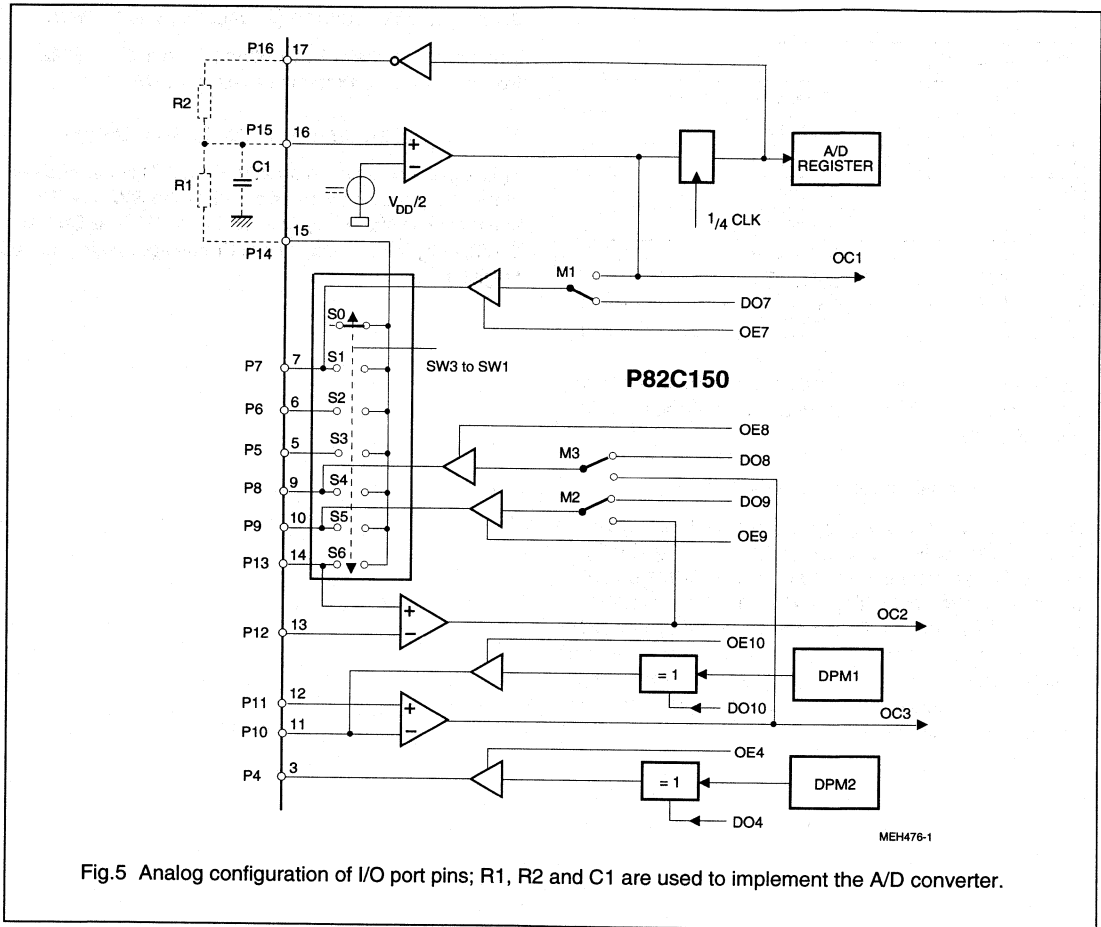


Fig.5 Analog configuration of I/O port pins; R1, R2 and C1 are used to implement the A/D converter.

6.2 CAN functions

The P82C150 meets the CAN protocol specification version 2.0 A and B (passive) with restricted bit timing because of the on-chip RC-oscillator and the automatic bitrate detection.

In a system with P82C150 nodes there must be at least one conventional crystal-driven CAN controller (host node) which is compatible to the CAN specification V1.2 or later to control P82C150 nodes. Host nodes compatible to CAN specification V1.1 can also be used provided that the P82C150 nodes are powered by a high-accuracy power supply or they are in external oscillator mode (refer to section 10.3).

Each time a P82C150 node receives a Data Frame, it initiates the transmission of a Data Frame containing four bits status information, the register address (previously received) and the current contents of the addressed register (exception: see Section 6.2.3). This enables the host node to verify that the addressed register has correctly been written in case of writable registers, and to read the contents in case of readable registers.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

Table 3 Message types and format.

FRAME	TRANSMISSION BY 82C150	RECEPTION AT 82C150
Data Frame	yes (DLC = 3; DIR = 1)	yes (DLC = 3; DIR = 0; calibration message with DLC = 2 to 8 allowed, see Section 6.2.10)
Remote Frame	no	yes (DLC = 3; DIR = 1)
Error Frame	yes	yes
Overload Frame	yes (only as a response)	yes

Note

1. DLC = Data Length Code; DIR = LSB of Identifier (see Section 6.2.1).

6.2.1 CAN IDENTIFIER

Data and Remote Frames to be processed by the P82C150 are of Standard Format with 11 Identifier bits

ID.10 to ID.0. Frames with extended Identifier (CAN specification version 2.0 B) are ignored.

1 = recessive; 0 = dominant

ID.10		IDENTIFIER								ID.0		
0	1	P3	1	0	P2	P1	P0	1	0	DIR	RTR	
0	1	P3	1	0	P2	P1	P0	1	0	DIR	RTR	

P3 to P0

Programmable identifier bits read from port pins P3 to P0 during reset.

DIR

“1” for transmission of Data Frames to the host. It must be set to “1” in Remote Frames and to “0” in Data Frames received from the host.

RTR

Remote Transmission Request bit.

The input levels on P3 to P0, for example set by resistors to V_{SS} to V_{DD} , are latched in the Identifier latch with the falling edge of the RST input signal. They represent the variable part of the Identifier, while the remaining bits are fixed (mask-programmed), P3 to P0 can be used as I/O ports after reset.

The way of identifier programming is based on two facts:

- Each P82C150 operates with only two Identifiers at which the higher priority Identifier is used for Data Frame reception (there is an extra Identifier for calibration purposes).
- There can be maximum sixteen P82C150 circuits in one network.

6.2.2 TRANSMISSION OF DATA FRAMES

Data Frames transmitted by the P82C150 contain three data bytes (see Fig.6). The first data byte contains the status information and the register address A3 to A0, the other two data bytes contain the content of the addressed I/O Register.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

Table 4 Data Frame Byte 1

STATUS				REGISTER ADDRESS			
RSTD	EW	BM1	BM0	A3	A2	A1	A0

Status	
RSTD	It is "1" in the first message (" <i>sign-on</i> " message) after the successful detection of the bit rate (bit time calibrated).
EW	"1", if the error warning limit (32) is reached. In the " <i>sign-on</i> " message RW is always "1".
BM1	Bus mode status bits.
BM0	
Register address	
A3	Register address bits.
A2	
A1	
A0	

The EW status bit is set when the Receive Error Counter or the Transmit Error Counter have exceeded the Error Warning Limit of 32, also temporarily, since the last successful transmission of a message.

After each successful message transmission, the P82C150 delays the transmission of a possibly further pending message for three bit times. The reason is to give other CAN controllers - with a lower identifier priority - the possibility to transmit a message in case of faulty contact at one of the edge-triggered port pins.

6.2.3 RECEPTION OF DATA FRAMES AND REMOTE FRAMES

Received Data Frames have the same format as transmitted ones, only the DIR-bit (ID.0) in the Arbitration Field is different. The status bits RSTD, EW, BM1 and BM0 are ignored during reception.

The P82C150 confirms each reception of a Data Frame by transmitting a Data Frame containing the (new) contents of the addressed I/O Register.

Exceptions from the rule:

1. Analog Configuration Register:

If a P82C150 receives a Data Frame addressing the Analog Configuration Register and the ADC bit is set to "1", it will respond with two messages. The first message returns the contents of the Analog

Configuration Register. The control instructions are executed (e.g. next analog input channel selected), and an A/D conversion cycle is started after a set-up time. After finishing the A/D conversion cycle, the second message is transmitted containing the result (A/D Register).

2. A/D Register:

On receiving a Data Frame addressing the A/D Register, the P82C150 starts an A/D conversion cycle. It automatically returns the result of the conversion (A/D Register) by transmitting a respective Data Frame after finishing the A/D conversion cycle.

3. At normal operation, the calibration messages are confirmed by returning a dominant bit in the acknowledge slot. There is no particular confirmation message returned by the P82C150. Only after entering the calibrated state (start-up), a Data Frame ("*sign-on*" message) containing the Data Input Register contents is transmitted indicating to the host node, that the P82C150 is now ready for transmission.

Remote Frame:

Received Remote Frames must have the Data Length Code DLC = 3 (Remote Frames with DLC ≠ 3 are ignored). It is answered by a Data Frame containing the contents of the Data Input Register.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

6.2.4 CAN-BUS MODES

The P82C150 can pass through four CAN-bus modes under certain conditions (see Fig.7). In the bus modes 0 to 2 (see Table 5) the P82C150 is operating with different input comparator configurations. Bus mode 3 is the power reduced Sleep Mode.

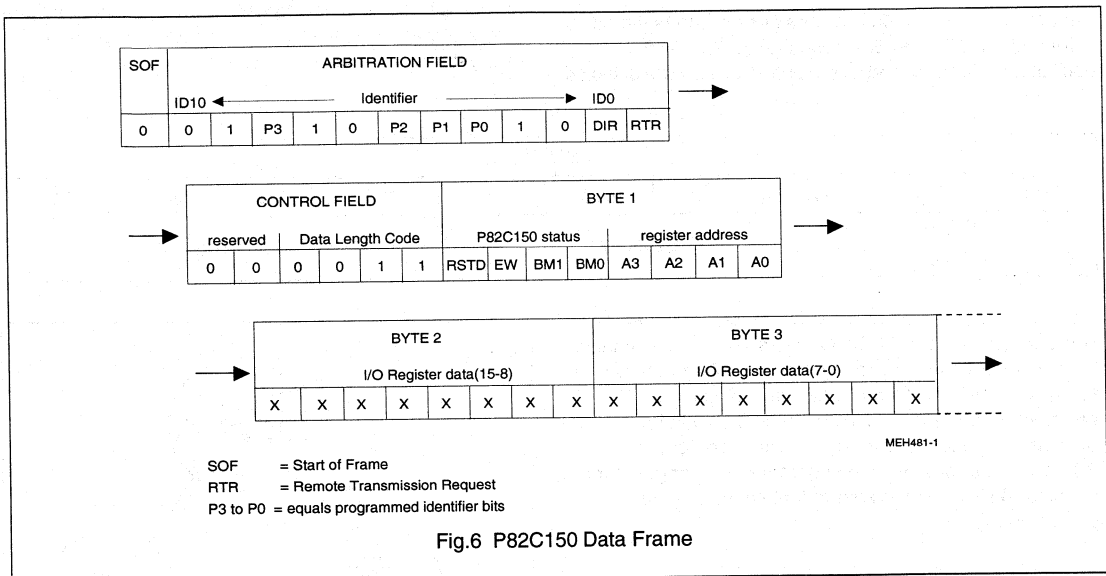
The bus modes support:

- Communication on two balanced wires (differential system)
- Communication on one wire in a two-wire differential system

- Sleep Mode with wake-up via either a dominant signal on RX0 or RX1 input
- Connection of a second transmission medium (redundancy)

There are two possibilities for condition 1 to switch to the next mode (see Fig.7):

- Overflow of the bit counter when 8192 is reached since the last calibration message
- Overflow of the Transmit Error Counter (>255; bus-off limit reached).



SOF = Start of Frame
 RTR = Remote Transmission Request
 P3 to P0 = equals programmed identifier bits

Table 5 Can-bus modes.

BUS MODE	BITS		RECEPTION LEVEL		TRANSMISSION	
	BM1	BM0	Recessive	Dominant	TX1	TX0
0 = Differential	0	0	RX0 > RX1	RX0 < RX1	enabled	enabled
1 = One-wire RX1	0	1	RX1 < REF	RX1 > REF	enabled	enabled
2 = One-wire RX0	1	0	RX0 > REF	RX0 < REF	disabled	enabled
3 = Sleep	1	1	RX0 > REF and RX1 < REF	RX0 < REF or RX1 > REF	disabled	disabled

Note

1. Output TX1 is disabled in bus mode 2 to tolerate short-circuit between the CAN-bus wires CAN_H and CAN_L.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

When the bus mode changes, all I/O Registers are cleared and outputs become floating (OE bits cleared). That means the I/O ports return to a fail-safe state whenever the P82C150 loses connection to its host controller. This is a kind of network watchdog function. The status bits are set to the following values after a bus mode change:

- RSTD = 1
- EW = 0
- $BM_{new} = BM_{old} + 1$

The programmed Identifier bits remain unchanged.

After reset the P82C150 changes directly into bus mode 3 (Sleep Mode). During Sleep Mode, the internal RX oscillator is stopped, and all the output drivers are disabled

(I/O Register contents cleared). A P82C150 in Sleep Mode can be waken up via CAN-bus lines (dominant level on RX0 or RX1) or by a reset condition.

6.2.5 BIT TIMING

The Nominal Bit Time of the P82C150 is subdivided into 10 Time Quanta. The Synchronization Time Segment (SYNC_SEG) and the Propagation Time Segment (PROP_SEG) are each one Time Quantum long. The Phase Buffer Segment 1 (PHASE_SEG1) and the Phase Buffer Segment 2 (PHASE_SEG2) are each four Time Quanta long. The Resynchronization Jump Width (SJW) is four Time Quanta long. The sample point is located at the end of the Phase Buffer Segment 1. The Nominal Bit Time is internally adjusted to that bit timing which is provided by the crystal driven host (calibration message).

1 BIT TIME									
BT1	BT2	BT3	BT4	BT5	BT6	BT7	BT8	BT9	BT10
SYNC SEG	PROP SEG	PHASE-SEG1				PHASE-SEG2			

The usable bus length at a given bit rate is reduced in comparison to other CAN controllers with programmable bit timing because the Propagation Time Segment is fixed to $\frac{1}{10}$ length of the Nominal Bit Time. The bit segmentation of the crystal driven host should be programmed like the fixed bit segmentation of the P82C150, e.g. one bit time segment is $\frac{1}{10}$ length of the Nominal Bit Time (refer also to table 8 for bit time programming).

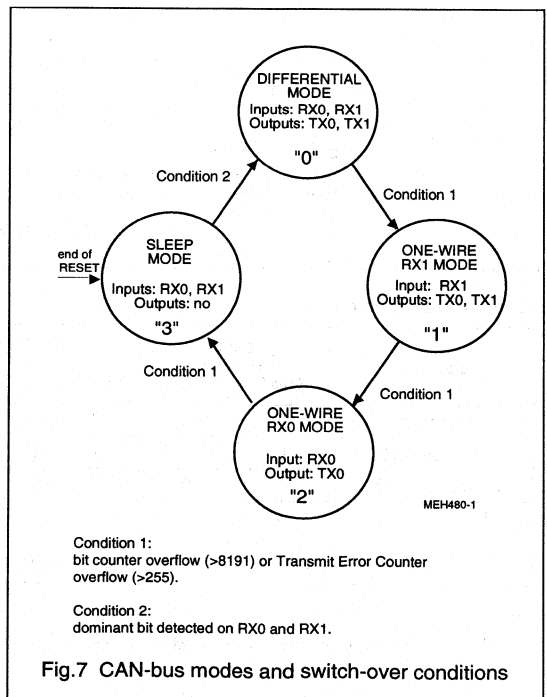


Fig.7 CAN-bus modes and switch-over conditions

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

6.2.6 CAN-BUS TRANSCEIVER

The transceiver of the P82C150 consists of the configurable input comparator and of complementary open-drain driver outputs. The reference voltage REF is an additional output.

CAN-bus input comparator (RX0, RX1):

The input comparator monitors the transient voltage on RX1 and RX0.

The result of the input comparator is "1" if the voltage levels of the CAN-bus lines are regarded as recessive, and "0" if they are regarded as dominant.

The recessive state and the dominant state are not equivalent and may not be mixed-up.

The input comparator is configurable depending on the four CAN-bus modes (see Table 5), supporting battery-powered applications (Sleep Mode) and tolerance against bus wiring failures.

CAN-bus output drivers (TX0, TX1):

The output driver function is shown in Table 6, The output driver TX1 is disabled in bus mode 2 to tolerate a short-circuit between the CAN-bus lines in a two-wire differential CAN physical layer.

Table 6 CAN-bus driver output function.

CAN OUTPUT	RECESSIVE	DOMINANT (MODES 0 AND 1)	DOMINANT (MODE 2)	RESET STATE, BUS-OFF AND SLEEP MODE (MODE 3)
TX0	floating	LOW	LOW	floating
TX1	floating	HIGH	floating	floating

6.2.7 TRANSMIT AND RECEIVE LOGIC

The transmit and receive logic stores the destuffed bit stream which was received or is about to be transmitted. The incoming Identifier is compared with that of the P82C150. The content of the message is transferred to the port logic in case of matching.

At transmission, the message about to be sent is put together: the Identifier, the status information, the register address and the content of the addressed register from the port logic.

6.2.8 BIT STREAM PROCESSOR AND ERROR MANAGEMENT LOGIC

The Bit Stream Processor (BSP) is a sequencer to control the data stream between the transmit/receive logic (parallel data) and the on-chip CAN transceiver (serial data). Reception/transmission, bit stuffing/destuffing, arbitration and error detection, according to CAN protocol specification version 2.0 A and B (passive), are performed. Further, automatic re-transmission of corrupted messages is handled by means of continuously comparing the output bit stream with the input bit stream. Moreover, the Bit Stream Processor provides control information to calibrate the internal bit time.

The Error Management Logic is responsible for the complete CAN-inherent error management.

6.2.9 OSCILLATOR AND CALIBRATION

The P82C150 contains an on-chip RC-oscillator. The bit time is automatically calibrated by messages being received via CAN-bus. During start-up (after wake-up or reset) any message is used to calibrate the bit time until the calibration is sufficient to receive messages correctly. From this time on, the bit time is calibrated and fine-tuned by calibration messages with a special Identifier transmitted by the crystal-operated host.

Only P82C150 nodes being calibrated by calibration messages can transmit messages. The first message is transmitted directly after entering the calibrated state ("sign-on" message). Since the P82C150 is not able to transmit as long as the bit time is not calibrated, it cannot wake-up other CAN nodes via the bus line. Hence to keep the network alive, the calibration message must be transmitted regularly by a crystal-operated (host) node with a maximum repetition period of 8192 bit (bith length measured by the 82C150). It is recommended to select a repetition period between 3800 and maximum 8000 bit times.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

Table 7 Example of a suitable calibration message. The two important I/O transitions are marked by underlines (1).

SOF	ARBITRATION FIELD	CONTROL FIELD	DATA BYTE 1	DATA BYTE 2	CRC FIELD
0	000 1010 1010 0	0001010	<u>1</u> 010 1010	00001 0100	00010 1011 1000 00 <u>1</u> 0

Note

1. **1 = stuff bit** (recessive);
(1) the total length is 67 bit from start-of-frame to end-of-intermission.

6.2.10 CALIBRATION MESSAGE

The calibration message has to meet the following requirements

- transmitted by a crystal-operated node (host node)
- identifier: 000 1010 1010 (1 = recessive; 0 = dominant)
- RTR bit: 0
- allowed control field: DLC = 2 to 8
- the first recessive to dominant transition after the control field must be followed by another recessive to dominant transition in a distance of exactly 32 bit (stuff bits included).

Example of a suitable calibration message (there are others using different data bytes; see Table 7):

- data length code: 0010
- 1st data byte: 1010 1010 (AAh)
- 2nd data byte: 0000 0100 (04h)

6.3 Initialization

6.3.1 IDENTIFIER PROGRAMMING

Most of the P82C150 identifier bits are fixed. Four bits are programmable via port pins P3 to P0. All output drivers are disabled at reset, also P3 to P0. Thus the outputs are floating unless the input level is defined by external components to define identifier bits. They are latched at the end of reset, and P3 to P0 can be used as port pins. It is not allowed, according to the CAN protocol specification, that multiple bus nodes transmit the same identifier bit combination. Therefore a P82C150 must have one of the 16 possible identifier bit combinations, one that is not yet occupied.

6.3.2 RESET FUNCTION

RST = HIGH disables all output drivers P16 to P0, TX0 and TX1. All I/O Registers are automatically cleared and set to "0". The bit time is set greater than 50 μ s.

After RESET:

status bits	identifier bits
RSTD = 1	ID.8 equals P3
EW = 1	ID.5 equals P2
BM1 = 0	ID.4 equals P1
BM0 = 0	ID.3 equals P0

If a particular clock period is necessary, e.g. for a dedicated DPM output frequency, this can be achieved by feeding an external clock signal into P0. RST and TEST must be permanently HIGH for this special mode. A reset is then performed as usual (RST = HIGH; TEST = LOW).

6.3.3 BIT TIME CALIBRATION

The P82C150 must receive at least three messages to calibrate its bit time after reset or change of bus mode. The first message is used to detect the bit time length (rough calibration) between two consecutive falling edges at the output of the CAN input comparator. Therefore the bit stream should contain a sequence of "1010".

After rough calibration the P82C150 can receive any valid CAN message correctly and executes respective commands without giving an acknowledge. With another valid CAN message and additionally with one valid calibration message the P82C150 is fully calibrated and sends its "sign-on" message. As long as the P82C150 is fully calibrated the P82C150 acts as an active CAN node.

The P82C150 treats any CAN message (including the calibration message) as a valid message, when these messages are terminated by an error passive frame because of a missing acknowledge. This situation may occur

whenever a host node works together with P82C150's and the host node doesn't receive an acknowledge as long as the P82C150's are not fully calibrated.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

Sign-on message:

This special Data Frame is transmitted once by the P82C150 after entering the calibrated state. It indicates to the host node that the P82C150 is ready for transmission.

The sign-on message returns the contents of the Data Input Register, and can be recognized by the host mode by checking the RSTD status bit.

Sign-on message RSTD = 1; other Data Frames RSTD = 0.

Note that in the sign-on message the EW bit is "1". Nevertheless the P82C150 status with the error counters are set to "0".

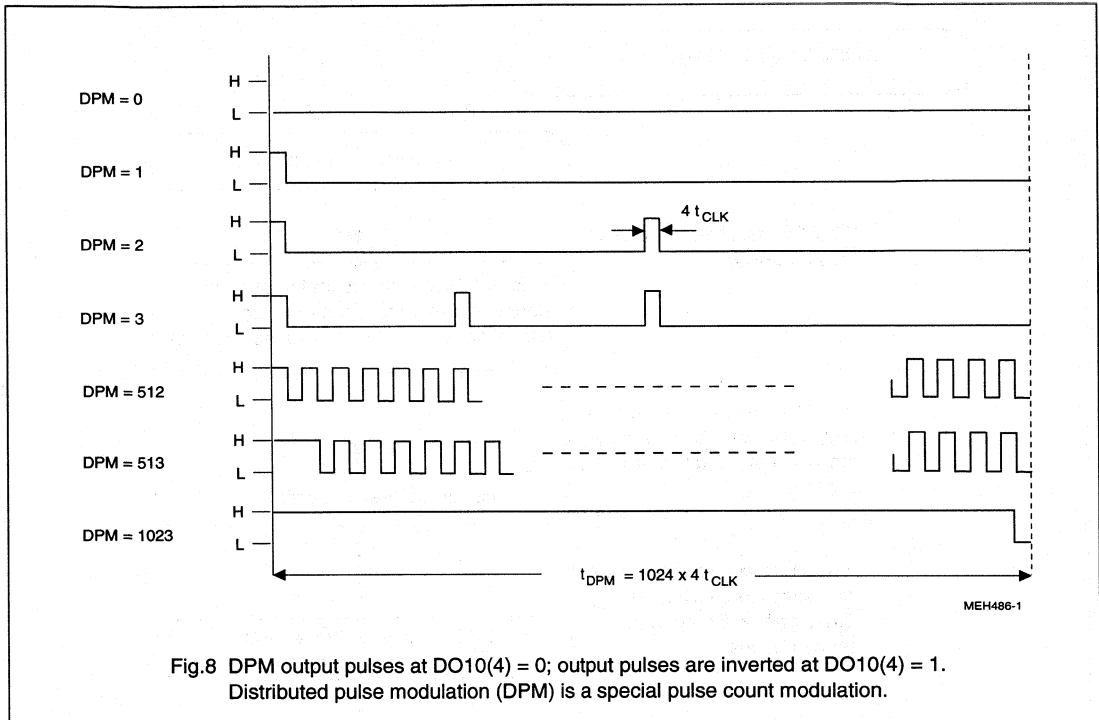


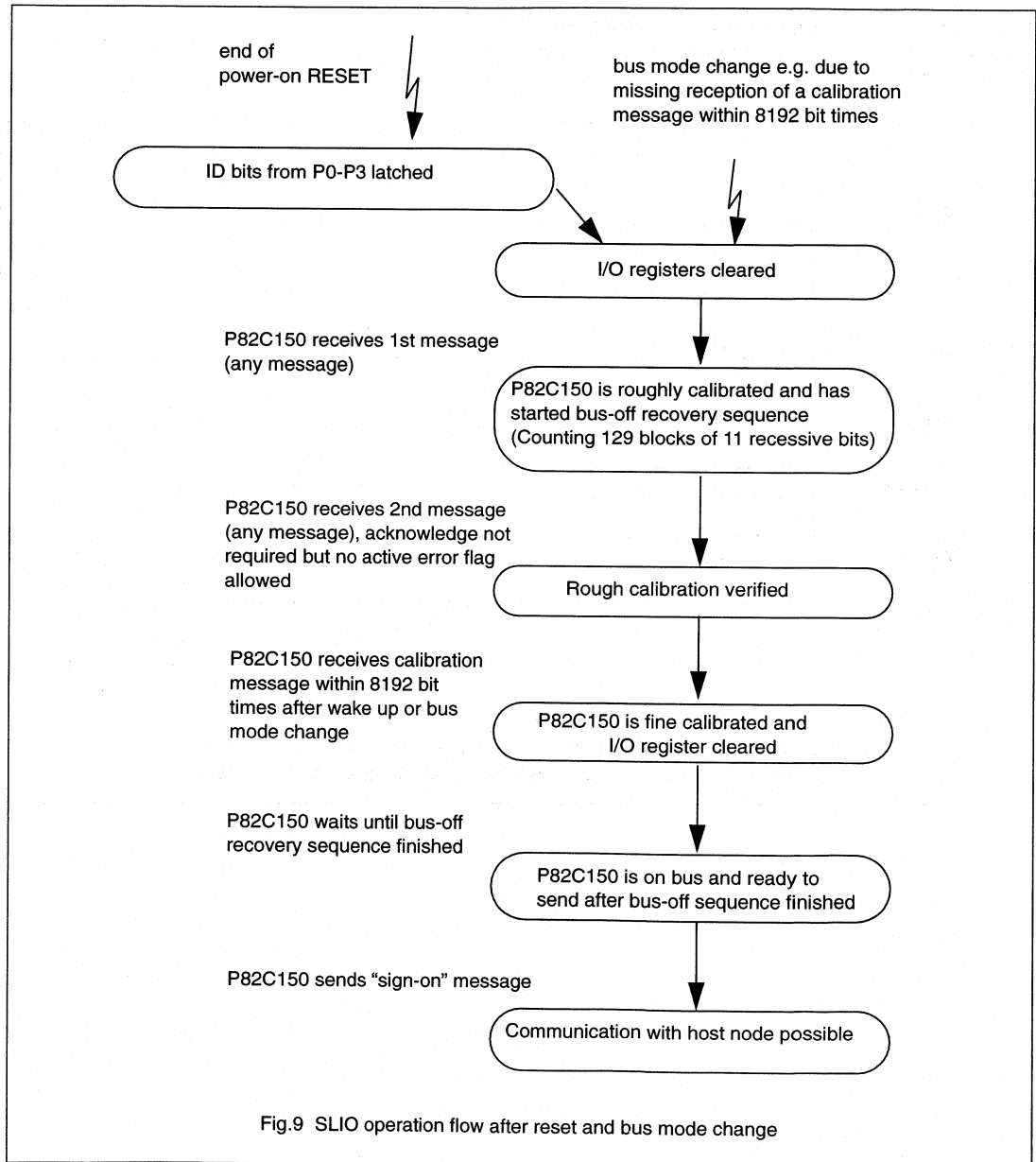
Fig.8 DPM output pulses at DO10(4) = 0; output pulses are inverted at DO10(4) = 1. Distributed pulse modulation (DPM) is a special pulse count modulation.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

6.4 P82C150 operation after RESET or change of bus mode

The following diagram describes the calibration procedure of the P82C150 after power on reset or after a bus mode change.



CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

7 LIMITING VALUES

In accordance with the Absolute Maximum Rating System (IEC 134).

SYMBOL	PARAMETER	MIN.	MAX.	UNIT
V_{DD}	supply voltage on V_{DD} pin	-0.5	+6.5	V
V_I	DC input voltage on any pin (RX0, RX1, TX0, TX1 excluded)	-0.5	$V_{DD} + 0.5$	V
I_I	RX1 and RX0 input current	-	± 2	mA
I_{REF}	reference output current	-	± 2	mA
I_O	port output current at port enabled (pins P0 to P15)	-	± 5	mA
	port output current at analog switch enabled (OE-bits = 0; pins P5 to P9, P13, P14)	-	7.5	mA
	TX0 and TX1 output current	-	30	mA
$P_{O\ tot}$	total power dissipation (port outputs together)	-	200	mW
T_{amb}	operating ambient temperature range:	-40	+125	°C
T_{stg}	storage temperature range	-65	+150	°C
P_{tot}	total power dissipation	-	1	W

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

8 DC CHARACTERISTICS

$V_{DD} = 5\text{ V} \pm 4\%$; $V_{SS} = 0\text{ V}$ and $T_{amb} = -40$ to $+125^\circ\text{C}$ unless otherwise specified.

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
Supply					
V_{DD}	supply voltage range		4.8	5.2	V
I_{DD}	supply current	$V_{RST} = V_{DD}$; all port inputs connected via $1\text{ M}\Omega$ to GND	–	22	mA
I_{SM}	sleep mode current	Ports P15, P13 and P11 connected to V_{DD} ; Ports P12 and P10 connected to V_{SS} ; all other port inputs connected via $1\text{ M}\Omega$ to GND		1	mA
CAN input comparators RX0 and RX1 $1.5\text{V} < V_I < (V_{DD2} - 1.5\text{V})$					
V_{DIF}	differential input voltage	note 1	± 25	–	mV
V_{HYST}	input voltage hysteresis	note 1	8	30	mV
I_I	input current	$0.45 < V_I < V_{DD} - 0.45$	–	± 400	nA
CAN output driver TX0 and TX1 port pins P0 to P16 unloaded					
V_{OLT}	TX0 output voltage low; note 3	$I_{OLT} = 1.5\text{ mA}$ $I_{OLT} = 10\text{ mA}$	–	0.1	V
V_{OHT}	TX1 output voltage high; note 4	$I_{OHT} = -1.5\text{ mA}$ $I_{OHT} = -10\text{ mA}$	$V_{DD} - 0.1$ $V_{DD} - 1$	–	V
Reference voltage REF					
V_{REF}	reference output voltage	$I_O < \pm 75\text{ }\mu\text{A}$	$0.5V_{DD2} - 0.25$	$0.5V_{DD2} + 0.25$	V
Control inputs RST, XMOD and digital port inputs P0/CLK, P1 to P15					
V_{IL}	input voltage LOW		–	$0.2V_{DD}$	V
V_{IH}	input voltage HIGH		$0.7V_{DD}$	–	V
V_{HYST}	input voltage hysteresis	note 1	0.5	–	V
I_{IL1}	input leakage current	$0.45 < V_I < V_{DD} - 0.45$		± 10	μA
Digital port outputs P0/CLK, P1 to P16 OE bits set					
V_{OL}	output voltage LOW	$I_{OL} = 4\text{ mA}$ (sink)	–	1	V
V_{OH}	output voltage HIGH	$I_{OH} = -4\text{ mA}$ (source)	$V_{DD} - 1$	–	V
OC2 comparator P12, P13 and OC3 comparator P10, P11 $1.5\text{V} < V_I < (V_{DD2} - 1.5\text{V})$					
V_{DIF1}	differential input voltage	note 1	± 10	–	mV

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
OC1 comparator input P15		1.5 V < V _I < (V _{DD2} -1.5 V)			
V _{I sw}	input switch-over voltage	note 1			
	lower threshold		0.5V _{DD} -0.01		V
	upper threshold		–	0.5V _{DD} +0.01	V
I _{LI2}	input leakage current	0.45 < V _I < V _{DD} – 0.45	–	±400	nA
C _{IA}	analog input capacitance	note 1	–	20	pF
Analog switches		I _{ON} = ±4 mA			
R _{ON}	On resistance	between P5 to P9, P13 and P14; note 1	20	200	Ω

Notes to the “DC characteristics”

1. These values are characterized but not 100% production tested.
2. Alteration of V_{DD} between two calibration messages should not exceed 0.2V to avoid failures during CAN message transfer. If CAN devices according to CAN specification V1.0 or V1.1 (like the 82C200 V0 or V1) are in the same network with the 82C150, then this alteration of V_{DD} should be limited to 0.1V for the 82C150.
3. The TX0 output pin is an open drain pull-down driver (no pull-up driver included).
4. The TX1 output pin is an open drain pull-up driver (no pull-down driver included).

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

9 AC CHARACTERISTICS

$V_{DD} = 5\text{ V} \pm 4\%$; $V_{SS} = 0\text{V}$; $C_L = 100\text{ pF}$ (output pins); $T_{amb} = -40\text{ to }+125^\circ\text{C}$; unless otherwise specified.

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
f_{CLK_INT}	system clock frequency on-chip	internal oscillator	4	10	MHz
t_{bit}	bit time on CAN-bus	note 2	8	50	μs
t_{RST1}	min. RST pulse width after power on	note 1	10	–	ms
t_{RST2}	min. RST pulse width during operation	note 1	1	–	μs
t_{hold}	ID hold time after end of reset	note 1	–	100	ns
t_d	total signal delay of CAN input comparator and CAN output driver	$1.5\text{ V} < V_I < (V_{DD}-1.5\text{ V})$; $V_{DIF} = \pm 25\text{mV}$; note 1	–	60	ns
t_{rep}	max. time without recalibration message		–	8000	bit
A/D comparator input P15					
t_{cyc}	A/D conversion cycle time		0.4	1.1	ms
t_{init}	initialization time of A/D conversion		0.4	2.1	ms
OC2 comparator P12, P13 and OC3 comparator P10, P11					
t_{resp}	response time	$V_{DIF1} = \pm 100\text{ mV}$, note 1		1	μs
DPM1 and DPM2 outputs					
t_{DPM}	repetition time of DPM cycle		0.4	1.1	ms

Notes to the AC characteristics:

1. These values are characterized but not 100% production tested.
2. Other bit time values are possible with the external oscillator mode (refer to chapter 9.3).

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

10 APPLICATION INFORMATION

10.1 Maximum bus length

The bit timing parameters refer to using a P8xCE598 or P8xC592 microcontroller with on-chip CAN interface as a host node (see Fig.21).

Assumptions

- the total in/out delay of external transceiver circuit is 170 ns (e.g. PCA82C250 CAN transceiver; (see Fig.20).
- the propagation delay on the transmission medium is 5.5 ns/m.

Table 8 Maximum bus length for CAN-bus systems with P82C150 nodes.

BIT RATE		INDICATION FOR MAXIMUM BUS LENGTH	BIT TIMING (P8XCE598/P8XC592)		
(kbit/s)	t _{prop} (note 1)		f _{CLK} (MHz)	BTR0 (note 2)	BTR1 (note 2)
125	0.8 μs	15 m	15	C5 h	34 h
100	1 μs	35 m	16	C7 h	34 h
50	2 μs	120 m	16	CE h	34 h
20	5 μs	400 m	16	E7 h	34 h

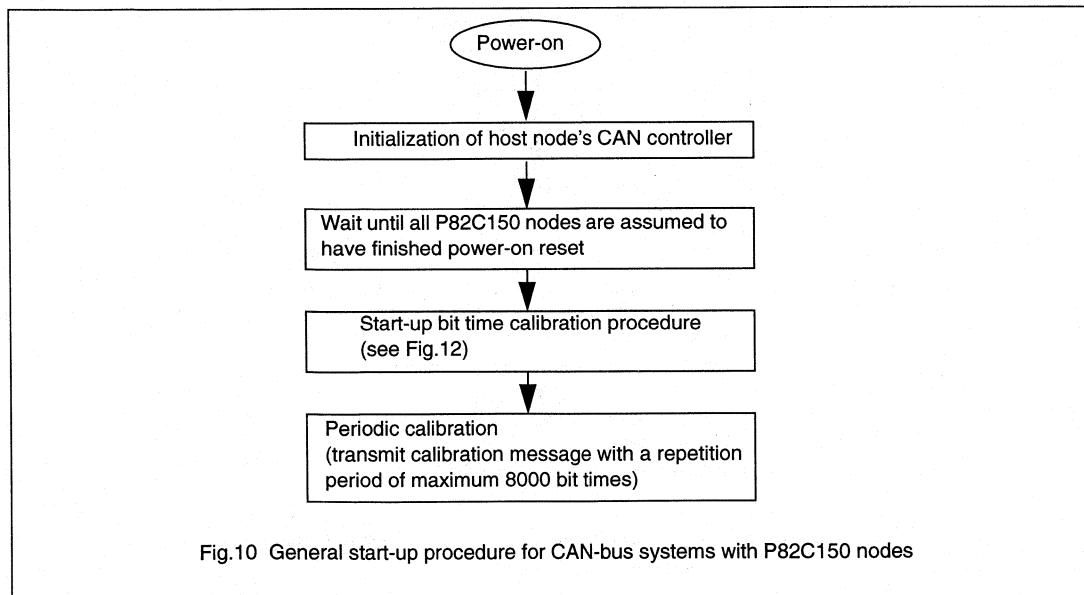
Notes

1. t_{prop} is the maximum propagation delay between two CAN-bus nodes (delays of on- and off-chip transceiver circuits included)
2. BTR0 and BTR1 (hex values) are particular configuration registers referring to bit timing.

10.2 Start up sequence

The following start-up sequence (Fig.11 and Fig.12) shows a simple example how P82C150 nodes can be controlled from a host node. This application example works with different system configurations:

- One conventional crystal-driven CAN node and one or more P82C150 nodes
- More than one conventional crystal-driven CAN node and one or more P82C150 nodes



CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

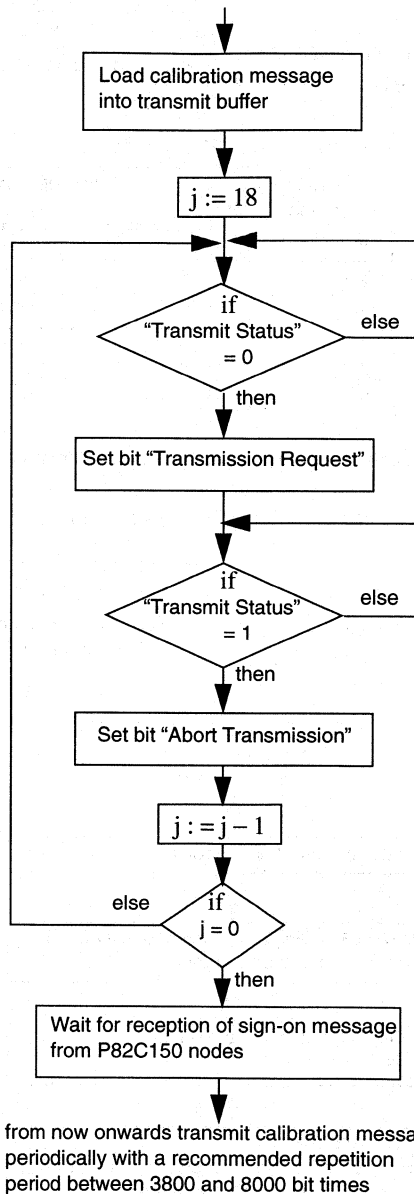


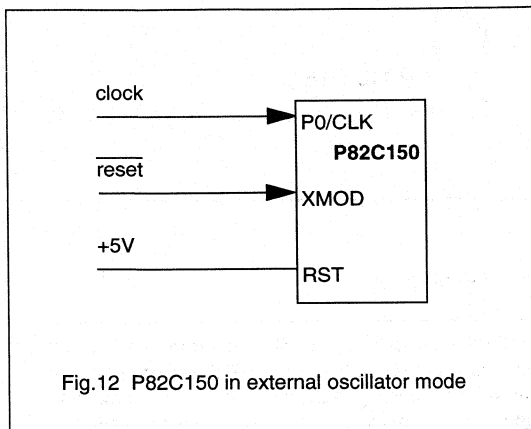
Fig.11 P82C150 start-up bit time calibration procedure for host node (e.g. P8xC592 or P8xC598)

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

10.3 External oscillator mode

In this mode the P82C150 operates with an external clock instead with the on-chip RC-oscillator. The following figure shows the application with an external clock:

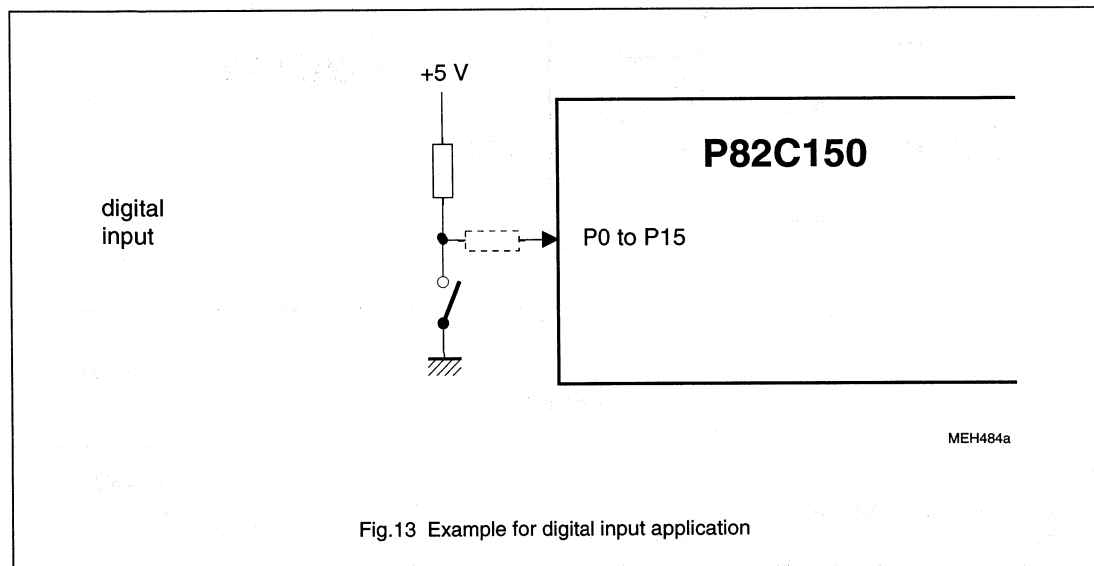


In this mode the P82C150 can achieve bit rates below 20 kbit/s and above 125 kbit/s. The DPM pulse width is $4 \times t_{CLK}$ of the external clock. The corresponding CAN identifier bit at port P0 is set to the value 0. Therefore only eight P82C150 based CAN nodes operate within the same network in external oscillator mode

Note that this mode is not the normal operation mode.

10.4 Using digital I/O port functions

The following figure shows the principle application for digital input and output.



CAN serial linked I/O device (SLIO)
with digital and analog port functions

82C150

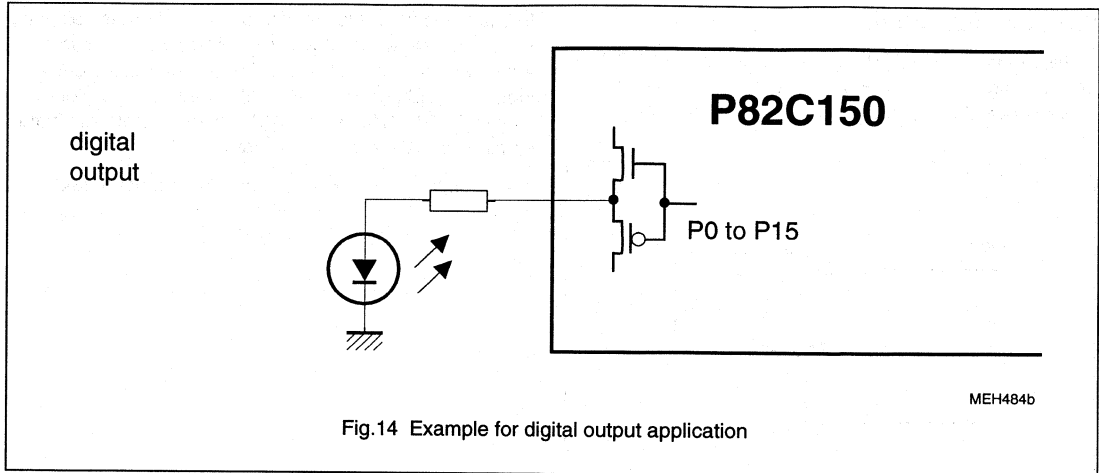


Fig.14 Example for digital output application

10.5 Using DPM

The most simple way to generate an analog voltage using the P82C150 is to apply an external low pass filter at one of the DPM (Distributed Pulse Modulation) outputs. The most simple implementation concept is a RC-filter of

first order (refer to Fig.16). Regarding the selection of the time constant (edge frequency) of this filter, a trade-off between minimizing of the $r_{ip_{le}}$ voltage for maximum accuracy and minimum of the settling time has to be considered.

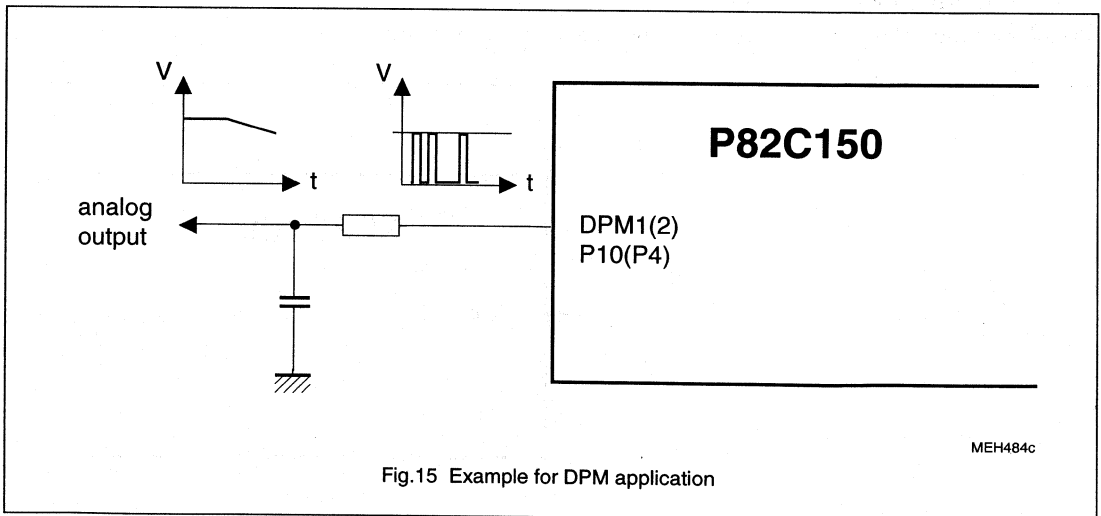


Fig.15 Example for DPM application

Note:

If the output is loaded by a resistive load, this will decrease the accuracy due to the voltage drop across the series resistor. In these cases a low value for the series resistor should be chosen.

The repetition time of one DPM cycle can be derived from:

$$t_{cyc} = \frac{4096}{f_{osc}}$$

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

10.6 Using ADC

The application in Fig.18 can be used for analog to digital conversion for only one analog input signal. With the values $R1 = R2 = 100\text{ k}\Omega$ and $C = 3.3\text{ nF}$ the implemented

ADC can achieve an accuracy of 8 bit. The external components should be connected close to the port pins P15 and P16 with short wiring to avoid disturbances at the analog input port pin P15.

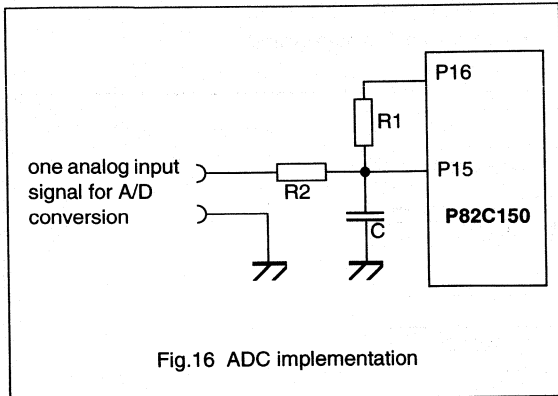


Fig.16 ADC implementation

Using the on-chip multiplex function the P82C150 provides up to six input port pins to convert analog input signals to digital values (see Fig.19).

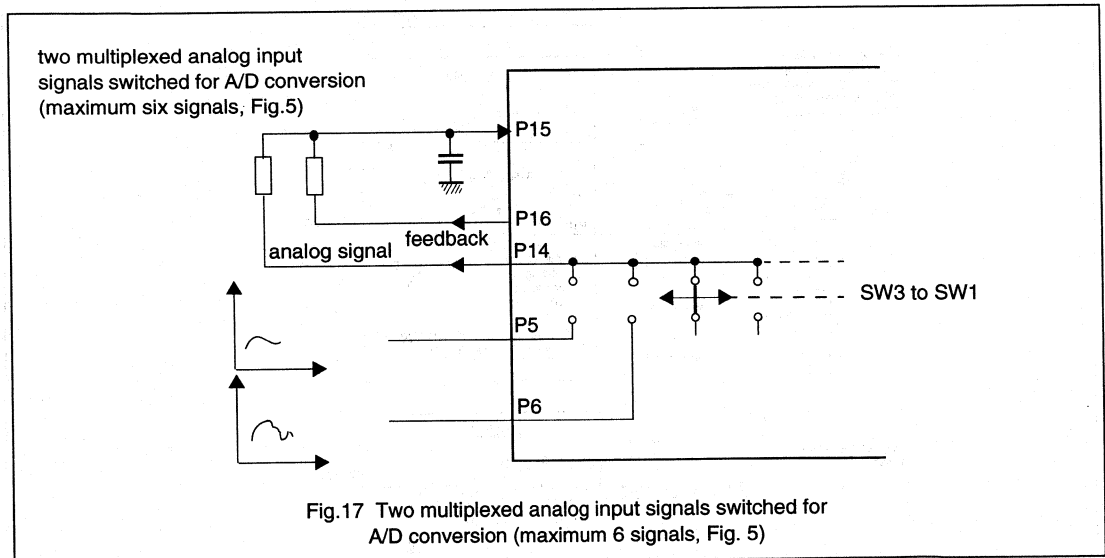


Fig.17 Two multiplexed analog input signals switched for A/D conversion (maximum 6 signals, Fig. 5)

The period for one ADC cycle is identical to the length of one DPM cycle.

CAN serial linked I/O device (SLIO) with digital and analog port functions

82C150

10.7 Using analog input port functions

The following figure shows the wide range of analog input applications:

- comparison of two analog input signals
- comparison of one analog input signal against a fixed threshold

- window comparator including monitoring the comparator outputs at the port pins P8 and P9; additional automatically generated messages, when the corresponding port bits in the negative and/or positive edge register are set
- local control two-step system.

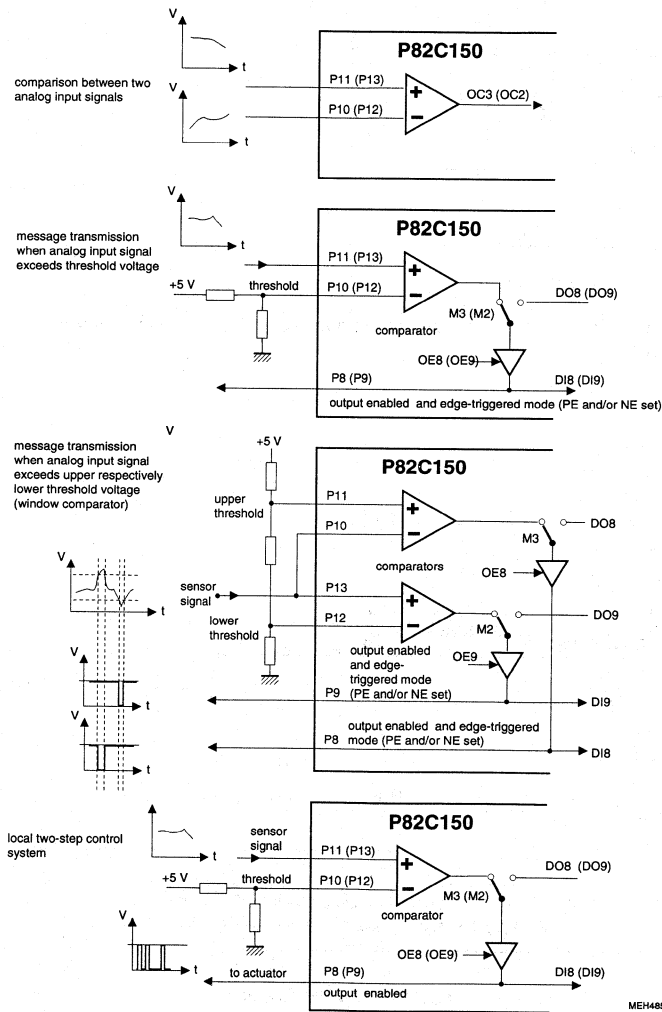


Fig.18 Examples of comparator applications

CAN serial linked I/O device (SLIO)
with digital and analog port functions

82C150

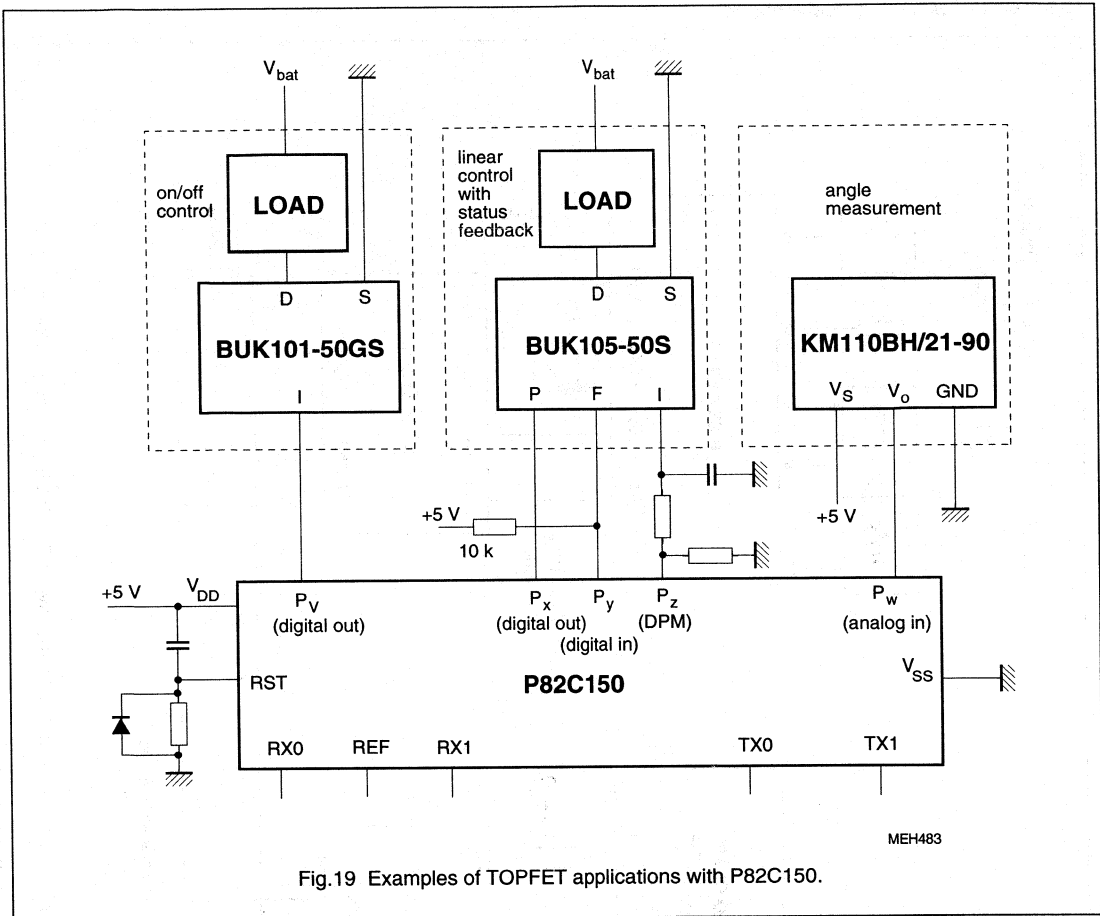


Fig.19 Examples of TOPFET applications with P82C150.

CAN serial linked I/O device (SLIO)
with digital and analog port functions

82C150

10.8 CAN-bus system applications

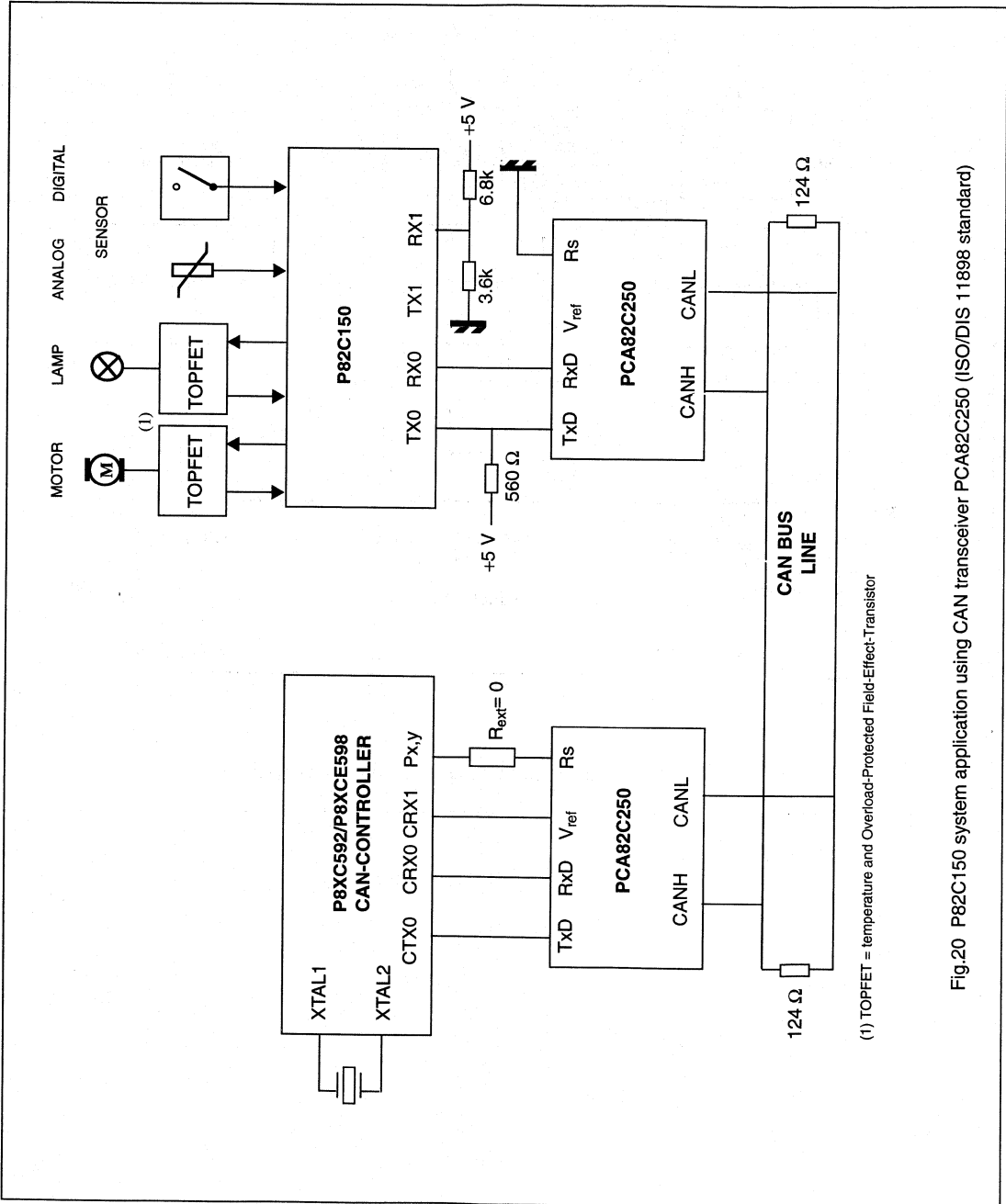


Fig.20 P82C150 system application using CAN transceiver PCA82C250 (ISO/DIS 11898 standard)

Stand-alone CAN-controller

82C200

1 FEATURES

- Multi-master architecture
- Interfaces with a large variety of microcontrollers
- Bus access priority (determined by the message identifier)
- 2032 message identifiers
- Guaranteed latency time for high priority messages
- Powerful error handling capability
- Data length from 0 to 8 bytes
- Configurable bus interface
- Programmable clock output
- Multicast and Broadcast message facility
- Non destructive bit-wise arbitration
- Non-return-to-zero (NRZ) coding/decoding with bit-stuffing
- Programmable transfer rate (up to 1 Mbit/s)
- Programmable output driver configuration
- Suitable for use in a wide range of networks including the SAE networks Class A, B and C
- 16 MHz clock frequency
- -40 to +85/125 °C operating temperature.

2 GENERAL DESCRIPTION

The PCA82C200; PCF82C200 (hereafter generically referred to as PCX82C200) is a highly integrated stand-alone controller for the controller area network (CAN) used within automotive and general industrial environments. The temperature range includes an automotive temperature range version (PCA82C200) of -40 to +125 °C and a -40 to +85 °C version (PCF82C200) for general applications.

The PCX82C200 contains all necessary features required to implement a high performance communication protocol. The PCX82C200 with a simple bus line connection performs all the functions of the physical and data-link layers. The application layer of an Electronic Control Unit (ECU) is provided by a microcontroller, to which the PCX82C200 provides a versatile interface. The use of the PCX82C200 in an automotive or general industrial environment, results in a reduced wiring harness and an enhanced diagnostic and supervisory capability.

3 ORDERING AND PACKAGE INFORMATION

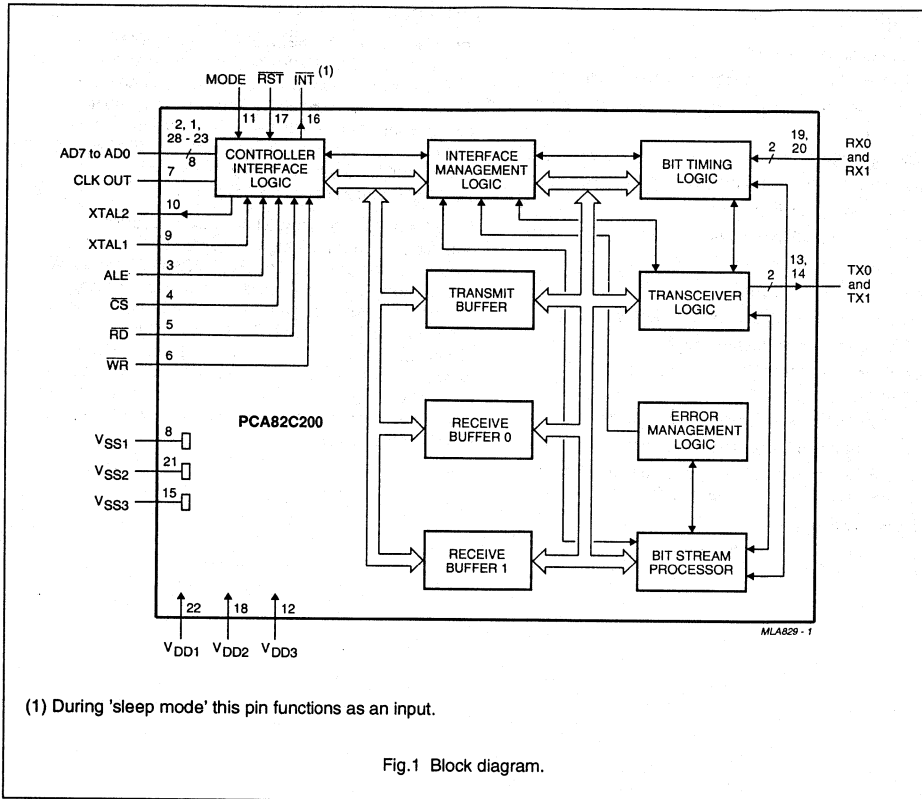
PHILIPS PART ORDER NUMBER PART MARKING	PHILIPS NORTH AMERICA PART ORDER NUMBER ¹	PACKAGE				TEMPERATURE RANGE (°C)
		PINS	PIN POSITION	MATERIAL	CODE	
PCA82C200P	PCA82C200PN	28	DIL	plastic	SOT117	-40 to +125
PCA82C200T	PCA82C200TD	28	SO28	plastic	SOT136A	-40 to +125
PCF82C200P	PCF82C200PN	28	DIL	plastic	SOT117	-40 to +85
PCF82C200T	PCF82C200TD	28	SO28	plastic	SOT136A	-40 to +85

NOTE:

1. Parts ordered by the Philips North America part number will be marked with the Philips part marking.

Stand-alone CAN-controller

82C200



Stand-alone CAN-controller

82C200

4 PINNING

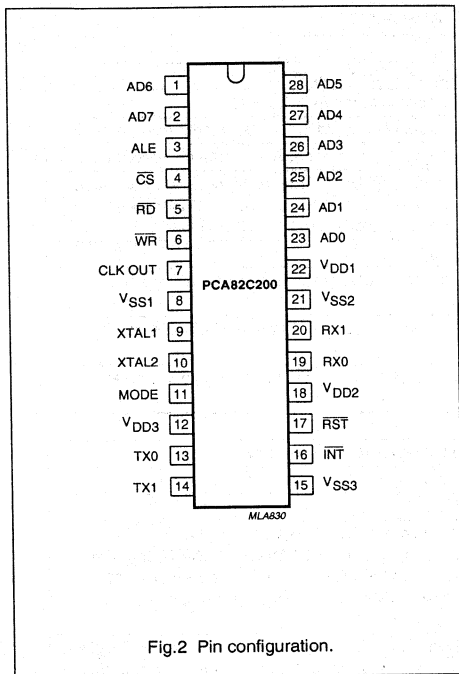


Fig.2 Pin configuration.

Stand-alone CAN-controller

82C200

Pinning description

SYMBOL	PIN	DESCRIPTION
AD7-AD0	2, 1, 28-23	Multiplexed address/data bus.
ALE	3	ALE signal (Intel mode) or AS input signal (Motorola mode).
$\overline{\text{CS}}$	4	Chip select input, LOW level allows access to the PCX82C200.
$\overline{\text{RD}}$	5	$\overline{\text{RD}}$ signal (Intel mode) or E enable signal (Motorola mode) from the microcontroller.
$\overline{\text{WR}}$	6	$\overline{\text{WR}}$ signal (Intel mode) or RD/ $\overline{\text{WR}}$ signal (Motorola mode) from the microcontroller.
CLK OUT	7	Clock output signal produced by the PCX82C200 for the microcontroller. The clock signal is derived from the built-in oscillator, via the programmable divider (see section 6.5). This output is capable of driving one CMOS or NMOS load.
V_{SS1}	8	Ground potential for the logic circuits.
XTAL1 (note 1)	9	Input to the oscillator's amplifier. External oscillator signal is input via this pin.
XTAL2 (note 1)	10	Output from the oscillator's amplifier. Output must be left open when an external oscillator signal is used.
MODE	11	Mode select input: connected to V_{DD} selects Intel mode; connected to V_{SS} selects Motorola mode.
V_{DD3}	12	5 V power supply for the output driver.
TX0	13	Output from the output-driver 0 to the physical bus-line.
TX1	14	Output from the output-driver 1 to the physical bus-line.
V_{SS3}	15	Ground potential for the output driver.
$\overline{\text{INT}}$	16	Interrupt output, used to interrupt the microcontroller (see section 6.2.4). $\overline{\text{INT}}$ is active if the Interrupt Register contains a logic HIGH bit (present). $\overline{\text{INT}}$ is an open drain output and is designed to be a wired-OR with other $\overline{\text{INT}}$ outputs within the system. A LOW level on this pin will reactivate the IC from the sleep mode (see section 6.2.2).
$\overline{\text{RST}}$	17	Reset input, used to reset the CAN interface (LOW level). Automatic power-ON reset can be obtained by connecting $\overline{\text{RST}}$ via a capacitor to V_{SS} and via a resistor to V_{DD} (e.g. $C = 1 \mu\text{F}$; $R = 50 \text{ k}\Omega$).
V_{DD2}	18	5 V power supply for the input comparator.
RX0-RX1	19, 20	Input from the physical bus-line to the input comparator of the PCX82C200. A dominant level will wake-up the PCX82C200. A recessive level is read if RX0 is higher than RX1 and vice versa for the dominant level.
V_{SS2}	21	Ground potential for the input comparator.
V_{DD1}	22	5 V power supply for the logic circuits.

Note

1. XTAL1 and XTAL2 pins should be connected to V_{SS} via 15 pF capacitors.

Stand-alone CAN-controller

82C200

5 FUNCTIONAL DESCRIPTION

The PCX82C200 contains all necessary hardware for a high performance serial network communication (see Fig.1). The PCX82C200 controls the communication flow through the area network using the CAN-protocol. The PCX82C200 meets the following automotive requirements:

- short message length
- guaranteed latency time for urgent messages
- bus access priority, determined by the message identifier
- powerful error handling capability
- configuration flexibility to allow area network expansion.

The latency time defines the period between the initiation (Transmission Request) and the start of the transmission on the bus. Latency time is dependent on a variety of bus related conditions. In the case of a message being transmitted on the bus and one distortion the latency time can be up to 149 bit times (worst case). For more information see section 7.

5.1 Interface Management Logic (IML)

The IML interprets commands from the microcontroller, allocates the message buffers (TBF, RBF0 and RBF1) and provides interrupts and status information to the microcontroller.

5.2 Transmit Buffer (TBF)

The TBF is a 10 byte memory into which the microcontroller writes messages which are to be transmitted over the CAN network.

5.3 Receive Buffers (RBF0 AND RBF1)

The RBF0 and RBF1 are each 10 byte memories which are alternatively used to store messages received from the CAN network. The CPU can process one message while another is being received.

5.4 Bit Stream Processor (BSP)

The BSP is a sequencer, controlling the data stream between the Transmit Buffer, the Receive Buffer (parallel data) and the CAN-bus (serial data).

5.5 Bit Timing Logic (BTL)

The BTL synchronizes the PCX82C200 to the bitstream on the CAN-bus.

5.6 Transceiver Control Logic (TCL)

The TCL controls the output driver.

5.7 Error Management Logic (EML)

The EML performs the error confinement according to the CAN-protocol.

5.8 Controller Interface Logic (CIL)

The CIL is the interface to the external microcontroller. The PCX82C200 can directly interface with a variety of microcontrollers.

6 CONTROL SEGMENT AND MESSAGE BUFFER DESCRIPTION

The PCX82C200 appears to a microcontroller as a memory-mapped I/O device due to the on-chip RAM, guaranteeing the independent operation of both devices.

6.1 Address allocation

The address area of the PCX82C200 consists of the Control Segment and the message buffers. The Control Segment is programmed during an initialization download in order to configure communication parameters (e.g. bit timing). Communication over the CAN-bus is also controlled via this segment by the CPU. During initialization the CLOCK OUT signal may be programmed to a value determined by the microcontroller (see Fig.1). A message which is to be transmitted, must be written to the Transmit Buffer. After a successful reception the microcontroller may read the message from the Receive Buffer and then release it for further use.

6.2 Control Segment layout

The exchange of status, control and command signals between the microcontroller and the PCX82C200 is performed in the control segment. The layout of this segment is shown in Fig.3. After an initial down-load, the contents of the registers Acceptance Code, Acceptance Mask, Bus Timing Registers 0 and 1, and Output Control should not be changed. These registers may only be accessed when the Reset Request bit in the Control Register, is set HIGH (see section 6.2.1).

Stand-alone CAN-controller

82C200

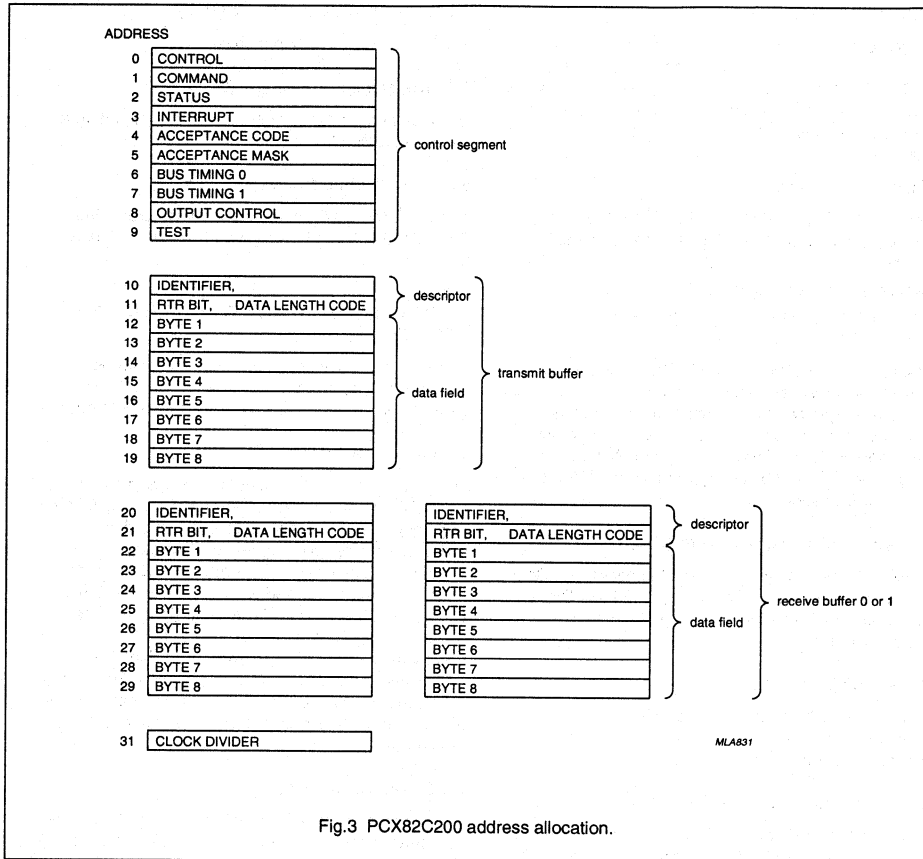


Fig.3 PCX82C200 address allocation.

Stand-alone CAN-controller

82C200

Table 1 Register map

TITLE	ADDR	7	6	5	4	3	2	1	0
Control Segment									
Control Register	0	Test Mode	Sync	reserved	Overrun Interrupt Enable	Error Interrupt Enable	Transmit Interrupt Enable	Receive Interrupt Enable	Reset Request
Command Register	1	reserved	reserved	reserved	Goto Sleep	Clear Overrun Status	Release Receive Buffer	Abort Transmission	Transmission Request
Status Register	2	Bus Status	Error Status	Transmit Status	Receive Status	Transmission Complete Status	Transmit Buffer Access	Data Overrun	Receive Buffer Stat
Interrupt Register	3	reserved	reserved	reserved	Wake-Up Interrupt	Overrun Interrupt	Error Interrupt	Transmit Interrupt	Receive Interrupt
Acceptance Code Register	4	AC.7	AC.6	AC.5	AC.4	AC.3	AC.2	AC.1	AC.0
Acceptance Mask Register	5	AM.7	AM.6	AM.5	AM.4	AM.3	AM.2	AM.1	AM.0
Bus Timing Register 0	6	SJW.1	SJW.0	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0
Bus Timing Register 1	7	SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0
Output Control Register	8	OCTP1	OCTN1	OCPOL1	OCTP0	OCTN0	OCPOL0	OCMODE1	OCMODE0
Test Register (note 1)	9	reserved	reserved	Map Internal Register	Connect RX Buffer 0 CPU	Connect TX Buffer CPU	Access Internal Bus	Normal RAM Connect	Float Output Driver
Transmit Buffer									
Identifier	10	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
RTR, Data Length Code	11	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
bytes 1-8	12-19	Data	Data	Data	Data	Data	Data	Data	Data
Receive Buffer 0/1									
Identifier	20	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
RTR, Data Length Code	21	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
bytes 1-8	22-29	Data	Data	Data	Data	Data	Data	Data	Data
Clock Divider	31	reserved	reserved	reserved	reserved	reserved	CD.2	CD.1	CD.0

Notes

1. The Test Register is used for production testing only.
2. Register 30 is not implemented.

Stand-alone CAN-controller

82C200

6.2.1 CONTROL REGISTER (CR)

The contents of the Control Register are used to change the behaviour of the PCX82C200. Control bits may be set or reset by the attached microcontroller which uses the Control Register as a read/write memory.

Table 2 Description of the Control Register bits

CR	ADDRESS 0			
BIT	SYMBOL	NAME	VALUE	FUNCTION
CR.7	TM	Test Mode (note 1)	HIGH (enabled) LOW (disabled)	PCX82C200 enters Test Mode (normal operation impossible). Normal operating mode.
CR.6	S	Sync (note 2)	HIGH (2 edges) LOW (1 edge)	Bus-line transitions from recessive-to-dominant and vice versa are used for resynchronization (see sections 7.2 and 8). Only transitions from recessive-to-dominant are used for resynchronization.
CR.5	–	–	–	Reserved.
CR.4	OIE	Overrun Interrupt Enable	HIGH (enabled) LOW (disabled)	If the Data Overrun bit is set (see section 6.2.3), the microcontroller receives an Overrun Interrupt signal. Microcontroller receives no Overrun Interrupt signal from the PCX82C200.
CR.3	EIE	Error Interrupt Enable	HIGH (enabled) LOW (disabled)	If the Error or Bus Status change (see section 6.2.3), the microcontroller receives an Error Interrupt signal. Microcontroller receives no Error Interrupt signal.
CR.2	TIE	Transmit Interrupt Enable	HIGH (enabled) LOW (disabled)	When a message has been successfully transmitted or the transmit buffer is accessible again, (e.g. after an Abort Transmission command) the PCX82C200 transmits a Transmit Interrupt signal to the microcontroller. No transmission of the Transmit Interrupt signal by the PCX82C200 to the microcontroller.
CR.1	RIE	Receive Interrupt Enable	HIGH (enabled) LOW (disabled)	When a message has been received without errors, the PCX82C200 transmits a Receive Interrupt signal to the microcontroller. No transmission of the Receive Interrupt signal by the PCX82C200 to the microcontroller.
CR.0	RR	Reset Request (note 3)	HIGH (present) LOW (absent)	Detection of a Reset Request results in the PCX82C200 aborting the current transmission or reception of a message and entering the reset state. On the HIGH-to-LOW transition of the Reset Request bit, the PCX82C200 returns to its normal operating state.

Notes

- The Test Mode is intended for factory testing and not for customer use.
- The Sync bit should only be modified if the Reset Request bit is set HIGH (present), otherwise it is ignored. It is possible to set the Sync bit while the Reset Request bit is changed from HIGH to LOW.
- During an external reset ($\overline{\text{RST}} = \text{LOW}$) or when the Bus Status bit is set HIGH (Bus-Off), the IML forces the Reset Request HIGH (present). During an external reset the microcontroller cannot set the Reset Request bit LOW (absent). Therefore, after having set the Reset Request bit LOW (absent), the microcontroller must check this bit to ensure that the external reset pin is not being held HIGH (present). After the Reset Request bit is set LOW (absent) the PCX82C200 will wait for:
 - one occurrence of the Bus-Free signal (11 recessive bits, see section 8.9.6), if the preceding reset (Reset Request = HIGH) was due to an external reset or a microcontroller initiated reset
 - 128 occurrences of Bus-Free, if the preceding reset (Reset Request = HIGH) was due to a PCX82C200 initiated Bus-Off, before re-entering the Bus-On mode (see section 8.9).
 When Reset Request is set HIGH (present), for whatever reason, the control, command, status and interrupt bits are affected, see Table 3. When Reset Request is set HIGH (present) the registers at addresses 4 to 8 are accessible but the TBF is not.

Stand-alone CAN-controller

82C200

Table 3 Effects of setting the Reset Request bit HIGH (present)

TYPE	BIT	FUNCTION	EFFECT
Control	CR.7	Test Mode	LOW (disabled)
Command	CMR.4	Goto Sleep	LOW (wake-up)
	CMR.3	Clear Overrun Status	HIGH (clear)
	CMR.2	Release Receive Buffer	HIGH (released)
	CMR.1	Abort Transmission	LOW (absent)
	CMR.0	Transmission Request	LOW (absent)
Status	SR.7	Bus Status	LOW (Bus-On) (note 1)
	SR.6	Error Status	LOW (no error) (note 1)
	SR.5	Transmit Status	LOW (idle)
	SR.4	Receive Status	LOW (idle)
	SR.3	Transmission Complete Status	HIGH (complete)
	SR.2	Transmit Buffer Access	HIGH (released)
	SR.1	Data Overrun	LOW (absent)
	SR.0	Receive Buffer Status	LOW (empty)
Interrupt	IR.3	Overrun Interrupt	LOW (reset)
	IR.1	Transmit Interrupt	LOW (reset)
	IR.0	Receive Interrupt	LOW (reset)

Note

1. Only after an external reset; see note 1 to Table 5 "Description of the Status Register bits".

Stand-alone CAN-controller

82C200

6.2.2 COMMAND REGISTER (CMR)

A command bit initiates an action within the transfer layer of the PCX82C200. The Command Register appears to the microcontroller as a write only memory. If a read access is performed to this address the byte 11111111 (binary) is returned.

Table 4 Description of the Command Register bits

CMR	ADDRESS 1			
BIT	SYMBOL	NAME	VALUE	FUNCTION
CMR.7	–	–	–	Reserved.
CMR.6	–	–	–	Reserved.
CMR.5	–	–	–	Reserved.
CMR.4	GTS	GoTo Sleep (note 1)	HIGH (sleep) LOW (wake up)	The PCX82C200 enters sleep mode, if the $\overline{\text{INT}} = \text{HIGH}$ (no interrupt signal from the PCX82C200 to the microcontroller pending or external source pending) and there is no bus activity. The PCX82C200 functions normally.
CMR.3	COS	Clear Overrun Status (note 2)	HIGH (clear) LOW (no action)	The Data Overrun status bit is set to LOW (see section 6.2.3). No action.
CMR.2	RRB	Release Receive Buffer (note 3)	HIGH (released) LOW (no action)	The receive buffer attached to the microcontroller is released. No action.
CMR.1	AT	Abort Transmission (note 4)	HIGH (present) LOW (absent)	If not already in progress, a pending Transmission Request is cancelled. No action.
CMR.0	TR	transmission Request (note 5)	HIGH (present) LOW (absent)	A message shall be transmitted. No action.

Notes

1. The PCX82C200 will enter sleep mode, if Goto Sleep is set HIGH (sleep), there is no bus activity and $\overline{\text{INT}} = \text{HIGH}$ (inactive). After sleep mode is set, the CLK OUT signal continues until at least 15 bit times have passed. The PCX82C200 will wake up when one of the three previously mentioned conditions is negated: after Goto Sleep is set LOW (wake up), there is bus activity or $\overline{\text{INT}}$ is driven LOW (active). On wake up, the oscillator is started and a Wake-Up Interrupt (see section 6.2.4) is generated. A PCX82C200 which is sleeping and then awakened by bus activity will not be able to receive this message until it detects a Bus-Free signal (see section 8.9.6).
2. This command bit is used to acknowledge the Data Overrun condition signalled by the Data Overrun status bit. It may be given or set at the same time as a Release Receive Buffer command bit.
3. After reading the contents of the Receive Buffer (RBF0 or RBF1) the microcontroller must release this buffer by setting the Release Receive Buffer bit HIGH (released). This may result in another message becoming immediately available.
4. The Abort Transmission bit is used when the microcontroller requires the suspension of the previously requested transmission, for example to transmit an urgent message. A transmission already in progress is not stopped. In order to determine if the original message had been transmitted successfully, or aborted, the Transmission Complete status bit should be checked after the Transmit Buffer Access bit has been set HIGH (released) or a Transmit Interrupt has been generated (see section 6.2.4).
5. If the Transmission Request bit was set HIGH in a previous command, it cannot be cancelled by setting the Transmission Request bit LOW (absent). Cancellation of the requested transmission may be performed by setting the Abort Transmission bit HIGH (present).

Stand-alone CAN-controller

82C200

6.2.3 STATUS REGISTER (SR)

The contents of the Status Register reflect the status of the PCX82C200 bus controller. The Status Register appears to the microcontroller as a read only memory.

Table 5 Description of the Status Register bits

SR	ADDRESS 2			
BIT	SYMBOL	NAME	VALUE	FUNCTION
SR.7	BS	Bus Status (note 1)	HIGH (Bus-Off) LOW (Bus-On)	The PCX82C200 is not involved in bus activities. The PCX82C200 is involved in bus activities.
SR.6	ES	Error Status	HIGH (error) LOW (ok)	At least one of the Error Counters (see section 8.10.3) has reached the microcontroller Warning Limit. Both Error Counters have not reached the Warning Limit.
SR.5	TS	Transmit Status (note 2)	HIGH (transmit) LOW (idle)	The PCX82C200 is transmitting a message. No message is transmitted.
SR.4	RS	Receive Status (note 2)	HIGH (receive) LOW (idle)	The PCX82C200 is receiving a message. No message is received.
SR.3	TCS	Transmission Complete Status (note 3)	HIGH (complete) LOW (incomplete)	Last requested transmission has been successfully completed. Previously requested transmission is not yet completed.
SR.2	TBS	Transmit Buffer Access (note 3)	HIGH (released) LOW (locked)	The microcontroller may write a message into the TBF. The microcontroller cannot access the Transmit Buffer. A message is either waiting for transmission or is in the process of being transmitted.
SR.1	DO	Data Overrun (note 4)	HIGH (overrun) LOW (absent)	This bit is set HIGH (Overrun), when both Receive Buffers are full and the first byte of another message should be stored. No data overrun has occurred since the Clear Overrun command was given.
SR.0	RBS	Receive Buffer Status (note 5)	HIGH (full) LOW (empty)	This bit is set when a new message is available. No message has become available since the last Release Receive Buffer command bit was set.

Notes

- When the Bus Status bit is set HIGH (Bus-Off), the PCX82C200 will set the Reset Request bit HIGH (present). It will stay in this state until the microcontroller sets the Reset Request bit LOW (absent). Once this is completed the PCX82C200 will wait the minimum protocol-defined time (128 occurrences of the Bus-Free signal) before setting the Bus Status bit LOW (Bus-On), the Error Status bit LOW (ok) and resetting the Error Counters.
- If both the Receive Status and Transmit Status bits are LOW (idle) the CAN-bus is idle.
- If the microcontroller tries to write to the Transmit Buffer when the Transmit Buffer Access bit is LOW (locked), the written bytes will not be accepted and will be lost without this being signalled. The Transmission Complete Status bit is set LOW (incomplete) whenever the Transmission Request bit is set HIGH (present). If an Abort Transmission command is issued, the Transmit Buffer will be released. If the message, which was requested and then aborted, was not transmitted, the Transmission Complete Status bit will remain LOW.
- If Data Overrun = HIGH (Overrun) is detected, the currently received message is dropped. A transmitted message, granted acceptance, is also stored in a Receive Buffer. This occurs because it is not known if the PCX82C200 will lose arbitration and so become a receiver of the message. If no Receive Buffer is available, Data Overrun is signalled.
- If the command bit Release Receive Buffer is set HIGH (released) by the microcontroller, the Receive Buffer Status bit is set LOW (empty) by IML. When a new message is stored in any of the receive buffers, the Receive Buffer Status bit is set HIGH (full) again.

Stand-alone CAN-controller

82C200

6.2.4 INTERRUPT REGISTER (IR)

The Interrupt Register allows the identification of an interrupt source. When one or more bits of this register are set, the INT pin is activated. All bits are reset by the PCX82C200 after this register is read by the microcontroller. This register appears to the microcontroller as a read only memory.

Table 6 Description of the Interrupt Register bits

IR		ADDRESS 3		
BIT	SYMBOL	NAME	VALUE	FUNCTION
IR.7	–	–	–	Reserved.
IR.6	–	–	–	Reserved.
IR.5	–	–	–	Reserved.
IR.4	WUI	Wake-Up Interrupt	HIGH (set) LOW (reset)	The Wake-Up Interrupt bit is set HIGH, when the sleep mode is left (see section 6.2.2). Wake-Up Interrupt bit is reset by a read access of Interrupt Register by the microcontroller.
IR.3	OI	Overrun Interrupt (note 1)	HIGH (set) LOW (reset)	This bit is set HIGH, if both Receive Buffers contain a message and the first byte of another message should be stored (passed acceptance), and the Overrun Interrupt Enable is HIGH (enabled). Overrun Interrupt bit is reset by a read access of Interrupt Register by the microcontroller.
IR.2	EI	Error Interrupt	HIGH (set) LOW (reset)	This bit is set on a change of either the Error Status or Bus Status bits (see section 6.2.3) if the Error Interrupt Enable is HIGH (enabled). The Error Interrupt bit is reset by a read access of the Interrupt Register by the microcontroller.
IR.1	TI	Transmit Interrupt	HIGH (set) LOW (reset)	This bit is set on a change of the Transmit Buffer Access bit from LOW to HIGH (released) and Transmit Interrupt Enable is HIGH (enabled). Transmit Interrupt bit will be reset after a read access of the Interrupt Register by the microcontroller.
IR.0	RI	Receive Interrupt (note 2)	HIGH (set) LOW (reset)	This bit is set when a new message is available in the Receive Buffer and the Receive Interrupt Enable bit is HIGH (enabled). Receive Interrupt bit is automatically reset by a read access of Interrupt Register by the microcontroller.

Notes

1. Overrun Interrupt bit (if enabled) and Data Overrun bit (see section 6.2.3) are set at the same time.
2. Receive Interrupt bit (if enabled) and Receive Buffer Status bit (see section 6.2.3) are set at the same time.

Stand-alone CAN-controller

82C200

6.2.5 ACCEPTANCE CODE REGISTER (ACR)

The Acceptance Code Register is part of the acceptance filter of the PCX82C200. This register can be accessed (read/write), if the Reset Request bit is set HIGH (present). When a message is received which passes the acceptance test and if there is an empty Receive Buffer, then the respective Descriptor and Data Field (see Fig.4) are sequentially stored in this empty buffer. In the case that there is no empty Receive Buffer, the Data Overrun bit is set HIGH (overrun), see sections 6.2.3 and 6.2.4. When the complete message has been correctly received the following occurs:

- the Receive Buffer Status bit is set HIGH (full)
- if the Receive Interrupt Enable bit is set HIGH (enabled), the Receive Interrupt is set HIGH (set).

The Acceptance Code bits (AC.7-AC.0) and the eight most significant bits of the message's Identifier (ID.10-ID.3) must be equal to those bit positions which are marked relevant by the Acceptance Mask bits (AM.7-AM.0). If the following equation is satisfied, acceptance is given:

$$[(ID.10 .. ID.3) = (AC.7 .. AC.0)] \text{ or } (AM.7 .. AM.0) = 1111 \text{ 1111 binary}$$

During transmission of a message which passes the acceptance test, the message is also written to its own Receive Buffer. If no Receiver Buffer is available, Data Overrun is signalled because it is not known at the start of a message whether the PCX82C200 will lose arbitration and so become a receiver of the message.

Table 7 Acceptance Code Register bits

ACR	ADDRESS 4						
7	6	5	4	3	2	1	0
AC.7	AC.6	AC.5	AC.4	AC.3	AC.2	AC.1	AC.0

6.2.6 ACCEPTANCE MASK REGISTER (AMR)

The Acceptance Mask Register is part of the acceptance filter of the PCX82C200. This register can be accessed (read/write) if the Reset Request bit is set HIGH (present). The Acceptance Mask Register qualifies which of the corresponding bits of the acceptance code are "relevant" or "don't care" for acceptance filtering.

Table 8 Acceptance Mask Register bits

AMR	ADDRESS 5						
7	6	5	4	3	2	1	0
AM.7	AM.6	AM.5	AM.4	AM.3	AM.2	AM.1	AM.0

Table 9 Description of the Acceptance Mask Register bits

ACCEPTANCE MASK BIT	VALUE	COMMENTS
AM.7 to AM.0	HIGH (don't care)	This bit position is "don't care" for the acceptance of a message.
	LOW (relevant)	This bit position is "relevant" for acceptance filtering.

Stand-alone CAN-controller

82C200

6.2.7 BUS TIMING REGISTER 0 (BTR0)

The contents of Bus Timing Register 0 defines the values of Baud Rate Prescaler (BRP) and the Synchronization Jump Width (SJW). This register can be accessed (read/write) if the Reset Request bit is set HIGH (present).

Table 10 Bus Timing Register 0 bits

BTR0	ADDRESS 6						
7	6	5	4	3	2	1	0
SJW.1	SJW.0	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0

Baud Rate Prescaler (BRP)

The period of the system clock t_{SCL} is programmable and determines the individual bit timing. The system clock is calculated using the following equation:

$$t_{SCL} = 2t_{CLK} (32BRP.5 + 16BRP.4 + 8BRP.3 + 4BRP.2 + 2BRP.1 + BRP.0 + 1)$$

t_{CLK} = time period of the PCX82C200 oscillator.

Synchronization Jump Width (SJW)

To compensate for phase shifts between clock oscillators of different bus controllers, any bus controller must resynchronize on any relevant signal edge of the current transmission. The synchronization jump width defines the maximum number of clock cycles a bit period may be shortened or lengthened by one resynchronization:

$$t_{SJW} = t_{SCL} (2SJW.1 + SJW.0 + 1)$$

For further information on bus timing see sections 6.2.8 and 7.

6.2.8 BUS TIMING REGISTER 1 (BTR1)

The contents of Bus Timing Register 1 defines the length of the bit period, the location of the sample point and the number of samples to be taken at each sample point. This register may be accessed (read/write) if the Reset Request bit is set HIGH (present).

Table 11 Bus Timing Register 1 bits

BTR1	ADDRESS 7						
7	6	5	4	3	2	1	0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

Sampling (SAM)

Table 12 Selection of sampling

BIT	VALUE	COMMENTS
SAM	HIGH (3 samples)	Three samples are taken.
	LOW (1 sample)	The bus is sampled once.

SAM = LOW (logic 0) is recommended for high speed buses (SAE class C), while SAM = HIGH (logic 1) is recommended for slow/medium speed buses (class A and B) where filtering of spikes on the bus-line is beneficial (see section 7.1.6).

Stand-alone CAN-controller

82C200

Time Segment 1 (TSEG1) and Time Segment 2 (TSEG2)

TSEG1 and TSEG2 determine the number of clock cycles per bit period and the location of the sample point:

$$t_{TSEG1} = t_{SCL} (8TSEG1.3 + 4TSEG1.2 + 2TSEG1.1 + TSEG1.0 + 1)$$

$$t_{TSEG2} = t_{SCL} (4TSEG2.2 + 2TSEG2.1 + TSEG2.0 + 1)$$

For further information on bus timing see sections 6.2.7 and 7.

6.2.9 OUTPUT CONTROL REGISTER (OCR)

The Output Control Register allows, under software control, the set-up of different output driver configurations. This register may be accessed (read/write) if the Reset Request bit is set HIGH (present).

Table 13 Output Control Register bits

OCR		ADDRESS 8					
7	6	5	4	3	2	1	0
OCTP1	OCTN1	OCPOL1	OCTP0	OCTN0	OCPOL0	OCMODE1	OCMODE0

If the PCX82C200 is in the sleep mode (Goto Sleep = HIGH) a recessive level is output on the TX0 and TX1 pins. If the PCX82C200 is in the reset state (Reset Request = HIGH) the output drivers are floating.

Normal Output Mode

In Normal Output Mode the bit sequence (TXD) is sent via TX0 and TX1. The voltage levels on the output driver pins TX1 and TX0 depend on both the driver characteristic programmed by OCTPx, OCTNx (float, pull-up, pull-down, push-pull) and the output polarity programmed by OCPOLx (see Fig.4).

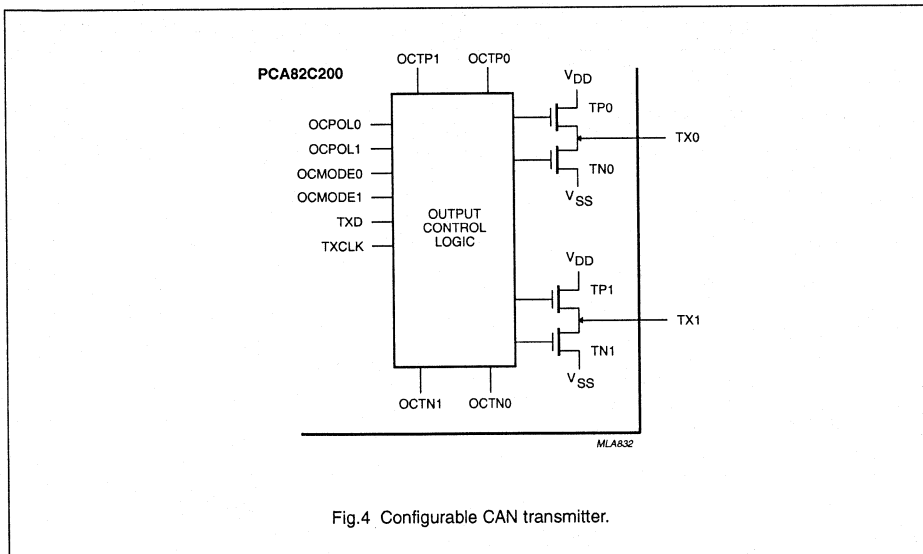


Fig.4 Configurable CAN transmitter.

Stand-alone CAN-controller

82C200

Clock Output Mode

For the TX0 pin this is the same as in Normal Output Mode. However, the data stream to TX1 is replaced by the transmit clock (TXCLK). The rising edge of the transmit clock (non inverted) marks the beginning of a bit period. The clock pulse width is t_{scl} .

Bi-phase Output Mode

In contrast to Normal Output Mode the bit representation is time variant and toggled. If the bus controllers are galvanically decoupled from the bus-line by a transformer, the bit stream is not allowed to contain a DC component. This is achieved by the following scheme. During recessive bits all outputs are deactivated (3-state). Dominant bits are sent alternatingly on TX0 and TX1, i.e. the first dominant bit is sent on TX0, the second is sent on TX1, and the third one is sent on TX0 again, etc.

Test Output Mode

For the TX0 pin this is the same as in Normal Output Mode. To measure the delay time of the transmitter and receiver this mode connects the output of the input comparator (COMP OUT) with the input of the output driver TX1. This mode is used for production testing only.

The following two tables, Table 14 and Table 15, show the relationship between the bits of the Output Control Register and the two serial output pins TX0 and TX1 of the PCX82C200, connected to the serial bus (see Fig. 1).

Table 14 Description of the Output Mode bits

OCMODE1	OCMODE0	DESCRIPTION
1	0	Normal Output Mode; TX0, TX1: bit sequence (TXD; note 1).
1	1	Clock Output Mode; TX0: bit sequence, TX1: bus clock (TXCLK).
0	0	Bi-phase Output Mode.
0	1	Test Output Mode; TX0: bit sequence, TX1: COMP OUT.

Note

1. TXD is the data bit to be transmitted.

Stand-alone CAN-controller

82C200

Table 15 Output pin set-up

DRIVE	OCTPx	OCTNx	OCPOLx	TXD	TPx (note 1)	TNx (note 2)	TXx (note 3)
Float	0	0	0	0	OFF	OFF	float
	0	0	0	1	OFF	OFF	float
	0	0	1	0	OFF	OFF	float
	0	0	1	1	OFF	OFF	float
Pull-down	0	1	0	0	OFF	ON	LOW
	0	1	0	1	OFF	OFF	float
	0	1	1	0	OFF	OFF	float
	0	1	1	1	OFF	ON	LOW
Pull-up	1	0	0	0	OFF	OFF	float
	1	0	0	1	ON	OFF	HIGH
	1	0	1	0	ON	OFF	HIGH
	1	0	1	1	OFF	OFF	float
Push/Pull	1	1	0	0	OFF	ON	LOW
	1	1	0	1	ON	OFF	HIGH
	1	1	1	0	ON	OFF	HIGH
	1	1	1	1	OFF	ON	LOW

Notes

1. TPx is the on-chip output transistor x, connected to V_{DD} ; x = 0 or 1.
2. TNx is the on-chip output transistor x, connected to V_{SS} ; x = 0 or 1.
3. TXx is the serial output level on pin TX0 or TX1. It is required that the output level on the CAN-bus is dominant with TXD = 0 and recessive with TXD = 1 (see section 8.1.1).

6.2.10 TEST REGISTER (TR)

The Test Register is used for production testing only.

Stand-alone CAN-controller

82C200

6.3 Transmit Buffer layout

The global layout of the Transmit Buffer is shown in Fig.3. This buffer serves to store a message from the microcontroller to be transmitted by the PCX82C200. It is subdivided into Descriptor and Data Field. The Transmit Buffer can be written to and read from by the microcontroller (see note 3 to Table 2).

6.3.1 DESCRIPTOR**Table 16** Descriptor Byte 1 (DSCR1)

DSCR1	ADDRESS 10						
7	6	5	4	3	2	1	0
ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3

Table 17 Descriptor Byte 2 (DSCR2)

DSCR2	ADDRESS 11						
7	6	5	4	3	2	1	0
ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0

Identifier (ID)

The Identifier consists of 11 bits (ID.10 to ID.0). ID.10 is the most significant bit, which is transmitted first on the bus during the arbitration process. The Identifier acts as the message's name, used in a receiver for acceptance filtering and also determines the bus access priority during the arbitration process. The lower the binary value of the Identifier the higher the priority. This is due to the larger number of leading dominant bits during arbitration (see section 8.7 "Bus organization").

*Remote Transmission Request bit (RTR)***Table 18** Description of the RTR bit

BIT	VALUE	COMMENTS
RTR	HIGH (remote)	Remote Frame will be transmitted by the PCX82C200.
	LOW (data)	Data Frame will be transmitted by the PCX82C200.

Data Length Code (DLC)

The number of bytes (Data Byte Count) in the Data Field of a message is coded by the Data Length Code. At the start of a Remote Frame transmission the Data Length Code is not considered due to the RTR bit being HIGH (remote). This forces the number of transmitted/received data bytes to be 0. Nevertheless, the Data Length Code must be specified correctly to avoid bus errors, if two CAN-controllers start a Remote Frame transmission simultaneously.

The range of the Data Byte Count is 0 to 8 bytes and coded as follows:

$$\text{Data Byte Count} = 8\text{DLC.3} + 4\text{DLC.2} + 2\text{DLC.1} + \text{DLC.0}$$

For reasons of compatibility no Data Byte Counts other than 0 to 8 should be used.

6.3.2 DATA FIELD

The number of transferred data bytes is determined by the Data Length Code. The first bit transmitted is the most significant bit of data byte 1 at address 12.

Stand-alone CAN-controller

82C200

6.4 Receive Buffer layout

The layout of the Receive Buffer and the individual bytes correspond to the definitions given for the Transmit Buffer layout, except that the addresses start at 20 instead of 10 (see Fig.3).

6.5 Clock Divider Register (CDR)

The Clock Divider Register controls the CLK OUT frequency for the microcontroller (see Fig.1). It can be written to or read by the microcontroller. The default state of the register is divide by 12 for Motorola mode and divide by 2 for Intel mode. Values from 0 to 7 may be written into this register and will result in the CLK OUT frequencies shown in Table 20.

Table 19 Clock Divider Register bits

CDR	ADDRESS 31						
	6	5	4	3	2	1	0
-	-	-	-	-	CD.2	CD.1	CD.0

Note

Bits CDR.7 to CDR.3 are reserved.

Table 20 CLK OUT frequency selection

CD.2	CD.1	CD.0	CLK OUT FREQUENCY
0	0	0	$f_{CLK}/2$
0	0	1	$f_{CLK}/4$
1	1	0	$f_{CLK}/6$
0	1	1	$f_{CLK}/8$
1	0	0	$f_{CLK}/10$
1	0	1	$f_{CLK}/12$
1	1	0	$f_{CLK}/14$
1	1	1	f_{CLK}

Note

1. f_{CLK} is the frequency of the oscillator.

Stand-alone CAN-controller

82C200

7 BUS TIMING/SYNCHRONIZATION

The Bus Timing Logic (BTL) monitors the serial bus-line via the on-chip input comparator and performs the following functions (see section 5):

- monitors the serial bus-line level
- adjusts the sample point, within a bit period (programmable)
- samples the bus-line level using majority logic (programmable, 1 or 3 samples)
- synchronization to the bit stream:
 - hard synchronization at the start of a message
 - resynchronization during transfer of a message.

The configuration of the BTL is performed during the initialization of the PCX82C200. The BTL uses the following three registers:

- Control register (Sync)
- Bus Timing Register 0
- Bus Timing Register 1.

7.1 Bit timing

A bit period is built up from a number of system clock cycles (t_{SCL}), see section 6.2.7. One bit period is the result of the addition of the programmable segments TSEG1 and TSEG2 and the general segment SYNCSEG (see sections 6.2.7 to 6.2.8).

7.1.1 SYNCHRONIZATION SEGMENT (SYNCSEG)

The incoming edge of a bit is expected during this state; this state corresponds to one system clock cycle ($1 \times t_{SCL}$).

7.1.2 TIME SEGMENT 1 (TSEG1)

This segment determines the location of the sampling point within a bit period, which is at the end of TSEG1. TSEG1 is programmable from 1 to 16 system clock cycles (see section 6.2.8).

The correct location of the sample point is essential for the correct functioning of a transmission. The following points must be taken into consideration:

- a Start-Of-Frame (see section 8.2.1) causes all PCX82C200's to perform a 'hard synchronization' (see section 7.2.1) on the first recessive-to-dominant edge. During arbitration, however, several PCX82C200's may simultaneously transmit. Therefore it may require twice the sum of bus-line, input comparator and the output driver delay times until the bus is stable. This is the propagation delay time.
- to avoid sampling at an incorrect position, it is necessary to include an additional synchronization buffer on both sides of the sample point. The main reasons for incorrect sampling are:
 - incorrect synchronization due to spikes on the bus-line
 - slight variations in the oscillator frequency of each PCX82C200 in the network, which results in a phase error.

Time Segment 1 consists of the segment for compensation of propagation delays and the synchronization buffer directly before the sample point (see Fig.5).

7.1.3 TIME SEGMENT 2 (TSEG2)

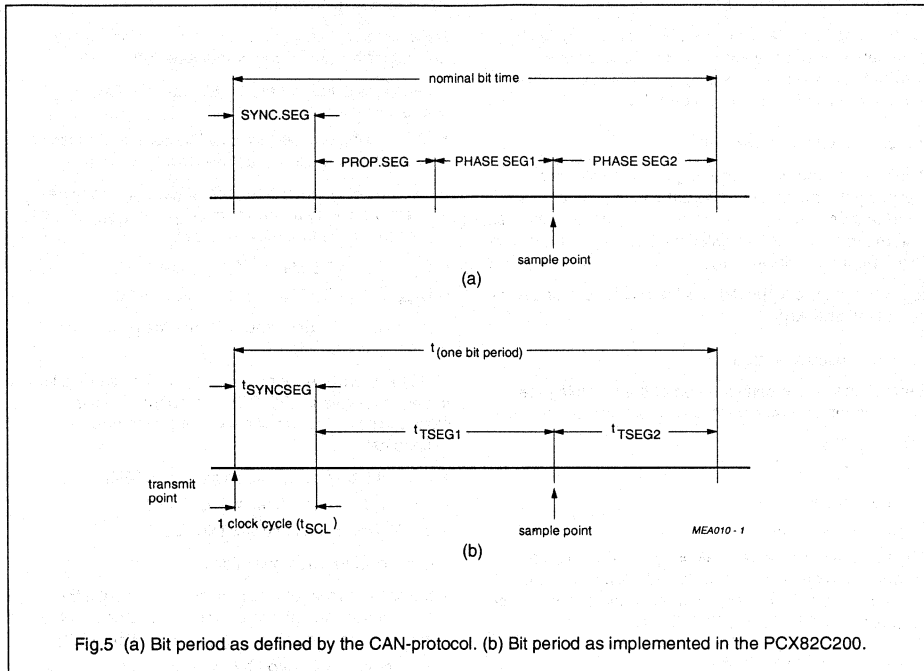
This time segment provides:

- additional time at the sample point for calculation of the subsequent bit levels (e.g. arbitration)
- synchronization buffer segment directly after the sample point (see section 7.1.2).

TSEG2 is programmable from 1 to 8 system clock cycles (see section 6.2.8).

Stand-alone CAN-controller

82C200



Stand-alone CAN-controller

82C200

7.1.4 SYNCHRONIZATION JUMP WIDTH (SJW)

SJW defines the maximum number of clock cycles (t_{SCL}) a bit period may be reduced or increased by one resynchronization. SJW is programmable from 1 to 4 system clock cycles (see section 6.2.7).

7.1.5 PROPAGATION DELAY TIME

The propagation delay time (t_{prop}) is calculated by summing the maximum propagation delay times of the physical bus, the input comparator and the output driver. The resulting sum is multiplied by 2 and then rounded up to the nearest multiple of t_{SCL} .

$$t_{prop} = 2 \times (\text{physical bus delay} + \text{input comparator delay} + \text{output driver delay})$$

7.1.6 BIT TIMING RESTRICTIONS

Restrictions on the configuration of the bit timing are based on internal processing. The restrictions are:

- $t_{TSEG2} \geq 2t_{SCL}$
- $t_{TSEG2} \geq t_{SJW}$
- $t_{TSEG1} \geq t_{TSEG2}$
- $t_{TSEG1} \geq t_{SJW} + t_{prop}$

The three sample mode (SAM = 1) has the effect of introducing a delay of one system clock cycle on the bus-line. This must be taken into account for the correct calculation of TSEG1 and TSEG2:

- $t_{TSEG1} \geq t_{SJW} + t_{prop} + 2t_{SCL}$
- $t_{TSEG2} \geq 3t_{SCL}$

7.2 Synchronization

Synchronization is performed by a state machine which compares the incoming edge with its actual bit timing and adapts the bit timing by hard synchronization or resynchronization.

7.2.1 HARD SYNCHRONIZATION

This type of synchronization occurs only at the beginning of a message. The PCX82C200 synchronizes on the first incoming recessive-to-dominant edge of a message (being the leading edge of a message's Start-Of-Frame bit; see section 7.1).

7.2.2 RESYNCHRONIZATION

Resynchronization occurs during the transmission of a message's bit stream to compensate for:

- variations in individual PCX82C200 oscillator frequencies
- changes introduced by switching from one transmitter to another (e.g. during arbitration).

As a result of resynchronization either t_{TSEG1} may be increased by up to a maximum of t_{SJW} or t_{TSEG2} may be decreased by up to a maximum of t_{SJW} :

- $t_{TSEG1} \leq t_{SCL} ((TSEG1 + 1) + (SJW + 1))$
- $t_{TSEG2} \geq t_{SCL} ((TSEG2 + 1) - (SJW + 1))$

Note: TSEG1, TSEG2 and SJW are the programmed numerical values.

The phase error (e) of an edge is given by the position of the edge relative to SYNCSEG, measured in system clock cycles (t_{SCL}). The value of the phase error is defined as:

- $e = 0$, if the edge occurs within SYNCSEG
- $e > 0$, if the edge occurs within TSEG1
- $e < 0$, if the edge occurs within TSEG2.

The effect of resynchronization is:

- the same as that of a hard synchronization, if the magnitude of the phase error (e) is less or equal to the programmed value of t_{SJW} (see section 6.2.7)
- to increase a bit period by the amount of t_{SJW} , if the phase error is positive and the magnitude of the phase error is larger than t_{SJW}
- to decrease a bit period by the amount of t_{SJW} , if the phase error is negative and the magnitude of the phase error is larger than t_{SJW} .

7.2.3 SYNCHRONIZATION RULES

The synchronization rules are as follows:

- only one synchronization within one bit time is used
- an edge is used for synchronization only if the value detected at the previous sample point differs from the bus value immediately after the edge
- hard synchronization is performed whenever there is a recessive-to-dominant edge during Bus-Idle (see section 7)

Stand-alone CAN-controller

82C200

- all other edges (recessive-to-dominant and optionally dominant-to-recessive edges if the Sync bit is set HIGH; see section 6.2.1) which are candidates for resynchronization will be used with the following exception:
 - a transmitting PCX82C200 will not perform a resynchronization as a result of a recessive-to-dominant edge with positive phase error, if only these edges are used for resynchronization. This ensures that the delay times of the output driver and input comparator do not cause a permanent increase in the bit time.

8 COMMUNICATION PROTOCOL

8.1 Frame types

The PCX82C200 bus controller supports the four different CAN-protocol frame types for communication:

- Data Frame, to transfer data
- Remote Frame, request for data
- Error Frame, globally signal a (locally) detected error condition
- Overload Frame, to extend delay time of subsequent frames (an Overload Frame is not initiated by the PCX82C200).

8.1.1 BIT REPRESENTATION

There are two logical bit representations used in the CAN-protocol:

- a recessive bit on the bus-line appears only if all connected PCX82C200's send a recessive bit at that moment
- dominant bits always overwrite recessive bits i.e. the resulting bit level on the bus-line is dominant.

8.2 Data Frame

A Data Frame carries data from a transmitting PCX82C200 to one or more receiving PCX82C200's. A Data Frame is composed of seven different bit-fields:

- Start-Of-Frame
- Arbitration Field
- Control Field
- Data Field (may have a length of zero)
- CRC Field
- Acknowledge Field
- End-Of-Frame.

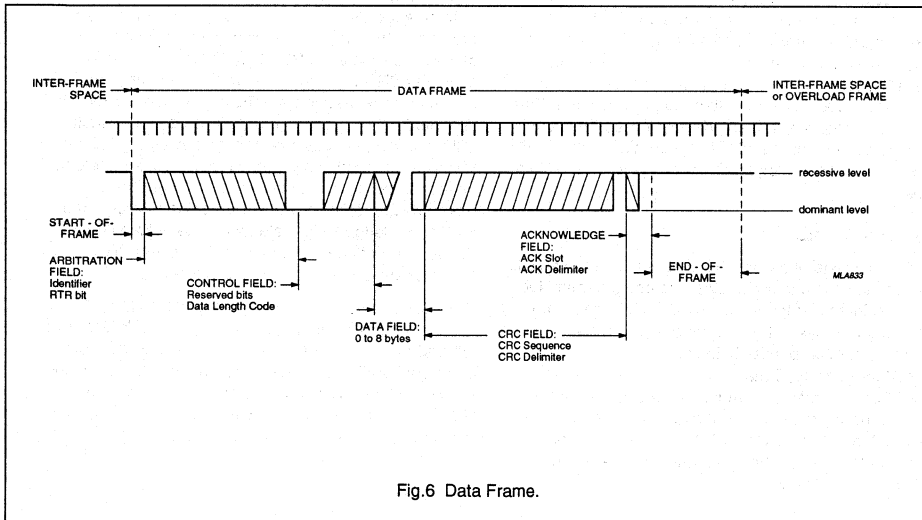


Fig.6 Data Frame.

Stand-alone CAN-controller

82C200

8.2.1 START-OF-FRAME BIT

Signals the start of a Data Frame or Remote Frame. It consists of a single dominant bit used for hard synchronization of a PCX82C200 in receive mode.

8.2.2 ARBITRATION FIELD

Consists of the message Identifier and the RTR bit (see section 6.3.1). In the event of simultaneous message transmissions by two or more PCX82C200's the bus access conflict is solved by bit-wise arbitration, which is active during the transmission of the Arbitration Field.

Identifier

This 11-bit field is used to provide information about the message, as well as the bus access priority. It is transmitted in the order ID.10 to ID.0 (LSB). The situation that the seven most significant bits (ID.10 to ID.4) are all recessive must not occur.

An Identifier does not define which particular PCX82C200 will receive the frame, because a CAN based communication network does not discriminate between a point-to-point, multicast or broadcast communication.

Remote Transmission Request bit (RTR)

A PCX82C200, acting as a receiver for certain information may initiate the transmission of the respective data by transmitting a Remote Frame to the network, addressing the data source via the Identifier and setting the RTR bit HIGH (remote; recessive bus level). If the data source simultaneously transmits a Data Frame containing the requested data, it uses the same Identifier. No bus access conflict occurs due to the RTR bit being set LOW (data; dominant bus level) in the Data Frame.

8.2.3 CONTROL FIELD

This field consists of six bits. It includes two reserved bits (for future expansions of the CAN-protocol), transmitted with a dominant bus level, and is followed by the Data Length Code (4 bits). The number of bytes in the (destuffed; number of data bytes to be transmitted/received) Data Field is indicated by the Data Length Code. Admissible values of the Data Length Code and hence the number of bytes in the (destuffed) Data Field, are 0 to 8. A logic 0 (logic 1) in the Data Length Code is transmitted as a dominant (recessive) bus level, respectively.

8.2.4 DATA FIELD

The data, stored within the Data Field of the Transmit Buffer, are transmitted according to the Data Length Code. Conversely, data of a received Data Frame will be stored in the Data Field of a Receive Buffer. Data is stored by byte-wise both for transmission by the microcontroller and on reception by the PCX82C200. The most significant bit of the first data byte (lowest address) is transmitted/received first.

8.2.5 CYCLIC REDUNDANCY CODE FIELD (CRC)

The CRC Field consists of the CRC Sequence (15 bits) and the CRC Delimiter (1 recessive bit). The Cyclic Redundancy Code (CRC) encloses the destuffed bit stream of the Start-Of-Frame, Arbitration Field, Control Field, Data Field and CRC Sequence. The most significant bit of the CRC Sequence is transmitted/received first. This frame check sequence, implemented in the PCX82C200, is derived from a cyclic redundancy code best suited for frames with a total bit count of less than 127 bits, see section 8.8.3 With Start-Of-Frame (dominant bit) included in the code word, any rotation of the code word can be detected by the absence of the CRC Delimiter (recessive bit).

8.2.6 ACKNOWLEDGE FIELD (ACK)

The Acknowledge Field consists of two bits, the Acknowledge Slot and the Acknowledge Delimiter, which are transmitted with a recessive level by the transmitter of the Data Frame. All PCX82C200's having received the matching CRC Sequence, report this by overwriting the transmitter's recessive bit in the Acknowledge Slot with a dominant bit (see section 8.9.2). Thereby a transmitter, still monitoring the bus level recognizes that at least one receiver within the network has received a complete and correct message (i.e. no error was found). The Acknowledge Delimiter (recessive bit) is the second bit of the Acknowledge Field. As a result, the Acknowledge Slot is surrounded by two recessive bits: the CRC Delimiter and the Acknowledge Delimiter.

All nodes within a CAN network may use all the information coming to the network by the PCX82C200's (shared memory concept). Therefore, acknowledgement and error handling are defined to provide all information in a consistent way throughout this shared memory. Hence, there is no reason to discriminate different receivers of a message in the acknowledge field. If a

Stand-alone CAN-controller

82C200

node is disconnected from the network due to bus failure, this particular node is no longer part of the shared memory. To identify a 'lost node' additional and application specific precautions are required.

8.2.7 END-OF-FRAME

Each Data Frame or Remote Frame is delimited by the End-Of-Frame bit sequence which consists of seven recessive bits (exceeds the bit stuff width by two bits). Using this method a receiver detects the end of a frame independent of a previous transmission error because the receiver expects all bits up to the end of the CRC sequence to be coded by the method of bit-stuffing (see section 8.7.3). The bit-stuffing logic is deactivated during the End-Of-Frame sequence.

8.3 Remote Frame

A PCX82C200, acting as a receiver for certain information may initiate the transmission of the respective data by transmitting a Remote Frame to the network, addressing the data source via the Identifier and setting the RTR bit HIGH (remote; recessive bus level). The Remote Frame is similar to the Data Frame with the following exceptions:

- RTR bit is set HIGH
- Data Length Code is ignored
- no Data Field contained.

Note that the Data Length Code value should be the same as for the corresponding Data Frame (although this is ignored for a Remote Frame).

A Remote Frame is composed of six different bit fields:

- Start-Of-Frame
- Arbitration Field
- Control Field
- CRC-Field
- Acknowledge Field
- End-Of-Frame.

See section 8.2 for a more detailed explanation of the Remote Frame bit fields.

8.4 Error Frame

The Error Frame consists of two different fields. The first field is accomplished by the superimposing of Error Flags contributed from different PCX82C200s. The second field is the Error Delimiter (see Fig.7).

8.4.1 ERROR FLAG

There are two forms of an Error Flag:

- Active Error Flag, consists of six consecutive dominant bits
- Passive Error Flag, consists of six consecutive recessive bits unless it is overwritten by dominant bits from other PCX82C200's.

An error-active PCX82C200 (see section 8.9) detecting an error condition signals this by transmission of an Active Error Flag. This Error Flag's form violates the bit-stuffing law (see section 8.7.3) applied to all fields, from Start-Of-Frame to CRC Delimiter, or destroys the fixed form of the fields Acknowledge Field or End-Of-Frame (see Fig.6). Consequently, all other PCX82C200's detect an error condition and start transmission of an Error Flag. Therefore the sequence of dominant bits, which can be monitored on the bus, results from a superposition of different Error Flags transmitted by individual PCX82C200's. The total length of this sequence varies between six (minimum) and twelve (maximum) bits.

An error-passive PCX82C200 (see section 8.9) detecting an error condition tries to signal this by transmission of a Passive Error Flag. The error-passive PCX82C200 waits for six consecutive bits with identical polarity, beginning at the start of the Passive Error Flag. The Passive Error Flag is complete when these six identical bits have been detected.

8.4.2 ERROR DELIMITER

The Error Delimiter consists of eight recessive bits and has the same format as the Overload Delimiter. After transmission of an Error Flag, each PCX82C200 monitors the bus-line until it detects a transition from a dominant-to-recessive bit level. At this point in time, every PCX82C200 has finished sending its Error Flag and all PCX82C200's start transmission of seven recessive bits (plus the recessive bit at dominant-to-recessive transition, results in a total of eight recessive bits). After this event and an Intermission Field all error-active PCX82C200's within the network can start a transmission simultaneously.

If a detected error is signalled during transmission of a Data Frame or Remote Frame, the current message is spoiled and a retransmission of the message is initiated.

Stand-alone CAN-controller

82C200

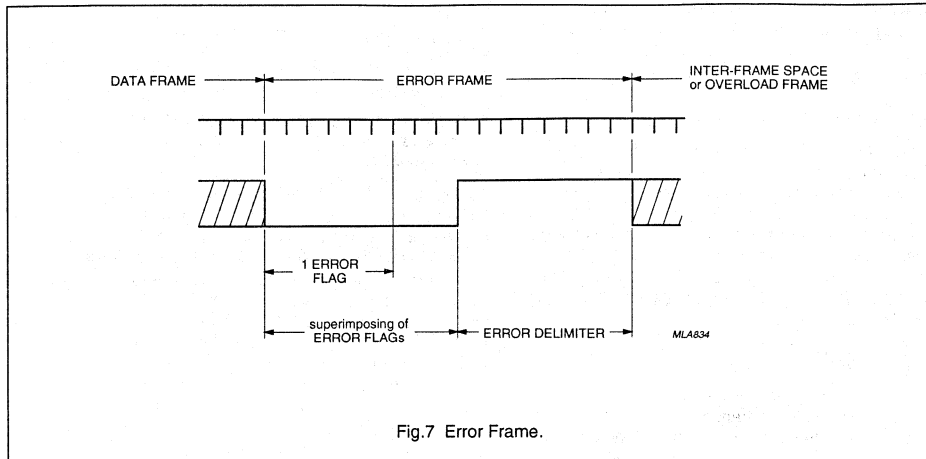


Fig.7 Error Frame.

If a PCX82C200 monitors any deviation of the Error Frame, a new Error Frame will be transmitted. Several consecutive Error Frame's may result in the PCX82C200 becoming error-passive and leaving the network unblocked.

In order to terminate an Error Flag correctly, an error-passive CAN-bus controller requires the bus to be Bus-Idle (see section 8.6.2) for at least three bit periods (if there is a local error at an error-passive receiver). Therefore a CAN-bus should not be 100% permanently loaded.

8.5 Overload Frame

The Overload Frame consists of two fields, the Overload Flag and the Overload Delimiter. There are two conditions in the CAN-protocol which lead to the transmission of an Overload Flag:

- condition 1; receiver circuitry requires more time to process the current data before receiving the next frame (receiver not ready)
- condition 2; detection of a dominant bit during Intermission Field (see section 8.6.1).

The transmission of an Overload Frame may only start:

- condition 1; during the first bit period of an expected Intermission Field

- condition 2; one bit period after detecting the dominant bit during Intermission Field.

The PCX82C200 will never initiate transmission of a condition 1 Overload Frame and will only react on a transmitted condition 2 Overload Frame, according to the CAN-protocol. No more than two Overload Frames are generated to delay a Data Frame or a Remote Frame. Although the overall form of the Overload Frame corresponds to that of the Error Frame, an Overload Frame does not initiate or require the retransmission of the preceding frame.

8.5.1 OVERLOAD FLAG

The Overload Flag consists of six dominant bits and has a similar format to the Error Flag.

The Overload Flag's form corrupts the fixed form of the Intermission Field. All other PCX82C200's detecting the overload condition also transmit an Overload Flag (condition 2).

8.5.2 OVERLOAD DELIMITER

The Overload Delimiter consists of eight recessive bits and takes the same form as the Error Delimiter. After transmission of an Overload Flag, each PCX82C200 monitors the bus-line until it detects a transition from a

Stand-alone CAN-controller

82C200

dominant-to-recessive bit level. At this point in time, every PCX82C200 has finished sending its Overload Flag and all PCX82C200's start simultaneously transmitting seven more recessive bits.

8.6 Inter-Frame Space

Data Frames and Remote Frames are separated from preceding frames (all types) by an Inter-Frame Space, consisting of an Intermission Field and a Bus-Idle. Error-passive PCX82C200's also send a Suspend Transmission (see section 8.9.5) after transmission of a message. Overload Frames and Error Frames are not preceded by an Inter-Frame Space.

8.6.1 INTERMISSION FIELD

The Intermission Field consists of three recessive bits. During an Intermission period, no frame transmissions will be started by any PCX82C200. An Intermission is required to have a fixed time period to allow a CAN-controller to execute internal processes prior to the next receive or transmit task.

8.6.2 BUS-IDLE

The Bus-Idle time may be of arbitrary length (minimum 0 bit). The bus is recognized to be free and a CAN-controller having information to transmit may access the bus. The detection of a dominant bit level during Bus-Idle on the bus is interpreted as the Start-Of-Frame.

8.7 Bus organization

Bus organization is based on five basic rules described in the following paragraphs.

8.7.1 BUS ACCESS

PCX82C200's only start transmission during the Bus-Idle state. All PCX82C200's synchronize on the leading edge of the Start-Of-Frame (hard synchronization).

8.7.2 ARBITRATION

If two or more PCX82C200's simultaneously start transmitting, the bus access conflict is solved by a bit-wise arbitration process during transmission of the Arbitration Field.

During arbitration every transmitting PCX82C200 compares its transmitted bit level with the monitored bus level. Any PCX82C200 which transmits a recessive bit and monitors a dominant bus level immediately becomes

the receiver of the higher priority message on the bus without corrupting any information on the bus. Each message contains a unique Identifier and a RTR bit describing the type of data within the message.

The Identifier together with the RTR bit implicitly define the message's bus access priority. During arbitration the most significant bit of the Identifier is transmitted first and the RTR bit last. The message with the lowest binary value of the Identifier and RTR bit has the highest priority. A Data Frame has higher priority than a Remote Frame due to its RTR bit having a dominant level.

For every Data Frame there is a unique transmitter. For reasons of compatibility with other CAN-bus controllers, use of the Identifier binary bit pattern ID = 1111111XXXX (X being bits of arbitrary level) is forbidden. The number of available different Identifiers is 2032 ($2^{11} - 2^4$).

8.7.3 CODING/DECODING

The following bit fields are coded using the bit-stuffing technique:

- Start-Of-Frame
- Arbitration Field
- Control Field
- Data Field
- CRC Sequence.

When a transmitting PCX82C200 detects five consecutive bits of identical polarity to be transmitted, a complementary (stuff) bit is inserted into the transmitted bit-stream.

When a receiving PCX82C200 has monitored five consecutive bits with identical polarity in the received bit streams of the above described bit fields, it automatically deletes the next received (stuff) bit. The level of the deleted stuff bit has to be the complement of the previous bits; otherwise a Stuff Error will be detected and signalled (see section 8.8.2).

The remaining bit fields or frames are of fixed form and are not coded or decoded by the method of bit-stuffing.

The bit-stream in a message is coded according to the Non-Return-to-Zero (NRZ) method, i.e. during a bit period, the bit level is held constant, either recessive or dominant.

Stand-alone CAN-controller

82C200

8.7.4 ERROR SIGNALLING

A PCX82C200 which detects an error condition, transmits an Error Flag. Whenever a Bit Error, Stuff Error, Form Error or an Acknowledgement Error is detected, transmission of an Error Flag is started at the next bit. Whenever a CRC Error is detected, transmission of an Error Flag starts at the bit following the Acknowledge Delimiter, unless an Error Flag for another error condition has already started. An Error Flag violates the bit-stuffing law or corrupts the fixed form bit fields. A violation of the bit-stuffing law affects any PCX82C200 which detects the error condition. These devices will also transmit an Error Flag.

An error-passive PCX82C200 (see section 8.9) which detects an error condition, transmits a Passive Error Flag. A Passive Error Flag is not able to interrupt a current message at different PCX82C200's, but this type of Error Flag may be cancelled by other PCX82C200's. After having detected an error condition, an error-passive PCX82C200 will wait for six consecutive bits with identical polarity and when monitoring them, interpret them as an Error Flag.

After transmission of an Error Flag, each PCX82C200 monitors the bus-line until it detects a transition from a dominant-to-recessive bit level. At this point in time, every PCX82C200 has finished transmitting its Error Flag and all PCX82C200's start transmitting seven additional recessive bits (Error Delimiter, see section 8.4.2).

The message format of a Data Frame or Remote Frame is defined in such a way, that all detectable errors can be signalled within the message transmission time and therefore, it is very simple for a PCX82C200 to associate an Error Frame to the corresponding message and to initiate retransmission of the corrupted message.

If a PCX82C200 monitors any deviation of the fixed form of an Error Frame, it transmits a new Error Frame.

8.7.5 OVERLOAD SIGNALLING

Some CAN-controllers (but not the PCX82C200) require to delay the transmission of the next Data Frame or Remote Frame by transmitting one or more Overload Frames. The transmission of an Overload Frame must start during the first bit of an expected Intermission. Transmission of Overload Frames which are reactions on a dominant bit during an expected Intermission Field, start one bit after this event.

Though the format of Overload Frame and Error Frame are identical, they are treated differently. Transmission of an Overload Frame during Intermission Field does not initiate the retransmission of any previous Data Frame or Remote Frame.

If a CAN-controller which transmitted an Overload Frame monitors any deviation of its fixed form, it transmits an Error Frame.

8.8 Error detection

The processes described in the following paragraphs are implemented in the PCX82C200 for error detection.

8.8.1 BIT ERROR

A transmitting PCX82C200 monitors the bus on a bit-by-bit basis. If the bit level monitored is different from the transmitted one, a Bit Error is signalled. The exceptions being:

- during the Arbitration Field, a recessive bit can be overwritten by a dominant bit. In this case, the PCX82C200 interprets this as a loss of arbitration
- during the Acknowledge Slot, only the receiving PCX82C200's are able to recognize a Bit Error.

8.8.2 STUFF ERROR

The following bit fields are coded using the bit-stuffing technique:

- Start-Of-Frame
- Arbitration Field
- Control Field
- Data Field
- CRC Sequence.

There are two possible ways of generating a Stuff Error:

- the disturbance generates more than the allowed five consecutive bits with identical polarity. These errors are detected by all PCX82C200's
- a disturbance falsifies one or more of the five bits preceding the stuff bit. This error situation is not recognized as a Stuff Error by the receivers. Therefore, other error detection processes may detect this error condition such as: CRC check, format violation at the receiving PCX82C200's or Bit Error detection by the transmitting PCX82C200.

Stand-alone CAN-controller

82C200

8.8.3 CRC ERROR

To ensure the validity of a transmitted message all receivers perform a CRC check. Therefore, in addition to the (destuffed) information digits (Start-Of-Frame up to Data Field), every message includes some control digits (CRC Sequence; generated by the transmitting PCX82C200 of the respective message) used for error detection.

The code used for the PCX82C200 bus controller is a (shortened) BCH code, extended by a parity check and has the following attributes:

- 127 bits as maximum length of the code
- 112 bits as maximum number of information digits (maximum 83 bits are used by PCX82C200)
- length of the CRC Sequence amounts to 15 bits
- Hamming distance $d = 6$.

As a result, (d-1) random errors are detectable (some exceptions exist).

The CRC Sequence is calculated by the following procedure:

1. the destuffed bit stream consisting of Start-Of-Frame up to the Data Field (if present) is interpreted as a polynomial with coefficients of 0 or 1
2. this polynomial is divided (modulo-2) by the following generator polynomial:

$$f(X) = (X^{14} + X^9 + X^8 + X^6 + X^5 + X^4 + X^2 + X + 1) (X + 1) = 1100010110011001 \text{ binary.}$$

The remainder of this polynomial division is the CRC Sequence which includes a parity check. Burst errors are detected up to a length of 15 [degree of $f(X)$]. Multiple errors (number of disturbed bits at least $d = 6$) are not detected with a residual error probability of 2^{-15} ($\approx 3 \times 10^{-5}$) by CRC check only.

8.8.4 FORM ERROR

Form Errors result from violation of the fixed form of the following bit fields:

- End-Of-Frame
- Intermission
- Acknowledge Delimiter
- CRC Delimiter.

During the transmission of these bit fields an error condition is recognized if a dominant bit level instead of a recessive one is detected.

8.8.5 ACKNOWLEDGEMENT ERROR

This is detected by a transmitter whenever it does not monitor a dominant bit during the Acknowledge Slot.

8.8.6 ERROR DETECTION BY AN ERROR FLAG OF ANOTHER PCX82C200

The detection of an error is signalled by transmitting an Error Flag. An Active Error Flag causes a Stuff Error, a Bit Error or a Form Error at all other PCX82C200's.

8.8.7 ERROR DETECTION CAPABILITIES

Errors which occur at all PCX82C200's (global errors) are 100% detected. For local errors, i.e. for errors occurring at some PCX82C200's only, the shortened BCH code, extended by a parity check, has the following error detection capabilities:

- up to five single bit errors are 100% detected, even if they are distributed randomly within the code
- all single bit errors are detected if their total number (within the code) is odd
- the residual error probability of the CRC check amounts to 3×10^{-5} . As an error may be detected not only by CRC check but also by other detection processes described in sections 8.8.1 to 8.8.5, the residual error probability is several magnitudes less than 3×10^{-5} for undetected errors.

8.9 Error confinement (definitions)

8.9.1 BUS-OFF

A PCX82C200 which has too many unsuccessful transmissions, relative to the number of successful transmissions, will enter the Bus-Off state. It remains in this state, neither receiving nor transmitting messages until the Reset Request bit is set LOW (absent) and both Error Counters are set to '0' (see note 1 to Table 5 and section 8.10.3).

8.9.2 ACKNOWLEDGE (ACK)

A PCX82C200 which has received a valid message correctly, indicates this to the transmitter by transmitting a dominant bit level on the bus during the Acknowledge Slot, independent of accepting or rejecting the message.

Stand-alone CAN-controller

82C200

8.9.3 ERROR-ACTIVE

An error-active PCX82C200 is in its normal operating state able to receive and to transmit normally and also to transmit an active Error Flag (see section 8.10.3).

8.9.4 ERROR-PASSIVE

An error-passive PCX82C200 may transmit or receive messages normally. In the case of a detected error condition it transmits a Passive Error Flag, instead of an Active Error Flag. Hence the influence on bus activities by an error-passive PCX82C200 (e.g. due to a malfunction) is reduced.

8.9.5 SUSPEND TRANSMISSION

After an error-passive PCX82C200 has transmitted a message, it sends eight recessive bits after the Intermission Field and then checks for Bus-Idle. If during Suspend Transmission another PCX82C200 starts transmitting a message the suspended PCX82C200 will become the receiver of this message; otherwise being in Bus-Idle it may start to transmit a further message.

8.9.6 START-UP

A PCX82C200 which was either switched off or is in the Bus-Off state, must run a start-up routine in order to:

- synchronize with other available PCX82C200's, before starting to transmit. Synchronizing is achieved, when 11 recessive bits, equivalent to Acknowledge Delimiter, End-Of-Frame and Intermission Field, have been detected (Bus-Free)
- wait for other PCX82C200s without passing into the Bus-Off state (due to a missing acknowledge), if there is no other PCX82C200 currently available.

8.10 Aims of error confinement

8.10.1 DISTINCTION OF SHORT AND LONG-LASTING DISTURBANCES

The microcontroller must be informed when there are long-lasting disturbances and when bus activities have returned to normal operation. During long-lasting disturbances, a PCX82C200 enters the Bus-Off state and the microcontroller may use default values.

Minor disturbances of bus activities will not affect a PCX82C200. In particular, a PCX82C200 does not enter the Bus-Off state or inform the microcontroller of a short-lasting bus disturbance.

8.10.2 DETECTION AND LOCALIZATION OF HARDWARE DISTURBANCES AND DEFECTS

The rules for error confinement are defined by the CAN-protocol specification (and implemented in the PCX82C200), in that the PCX82C200, being nearest to the error-locus, reacts with a high probability, the quickest (i.e. becomes error-passive or Bus-Off), hence errors can be localized and their influence on normal bus activities is minimized.

8.10.3 ERROR CONFINEMENT

All PCX82C200's contain a Transmit Error Counter and a Receive Error Counter, which registers errors during the transmission and the reception of messages, respectively.

If a message is transmitted or received correctly, the count is decreased. In the event of an error, the count is increased. The Error Counters have a non-proportional method of counting: an error causes a larger counter increase than a correctly transmitted/received message causes the count to decrease. Over a period of time this may result in an increase in error counts, even if there are fewer corrupted messages than uncorrupted ones. The level of the Error Counters reflect the relative frequency of disturbances. The ratio of increase/decrease depends on the acceptable ratio of invalid/valid messages on the bus and is hardware implemented to eight.

If one of the Error Counters exceeds the Warning Limit of 96 error points, indicating a significant accumulation of error conditions, this is signalled by the PCX82C200 (Error Status, Error Interrupt).

A PCX82C200 operates in the error-active mode until it exceeds 127 error points on one of its Error Counters. At this point it will enter the error-passive state.

A transmit error which exceeds 255 error points results in the PCX82C200 entering the Bus-Off state.

Stand-alone CAN-controller

82C200

9 LIMITING VALUES

Limiting values in accordance with the Absolute Maximum Rating System (IEC134)

SYMBOL	PARAMETER	MIN.	MAX.	UNIT
V_{DD}	supply voltage range	4.5	5.5	V
I_I	input/output current on any pin except from TX0 and TX1	–	±10	mA
I_{OT}	sink current of TX0 and TX1 together (note 1)	–	28	mA
I_{OT}	source current of TX0 and TX1 together (note 1)	–	–20	mA
T_{amb}	operating ambient temperature range:			
	PCA82C200	–40	+125	°C
	PCF82C200	–40	+85	°C
T_{stg}	storage temperature range	–65	+150	°C
P_{tot}	total power dissipation (note 2)	–	1	W

Notes

- I_{OT} is allowed in case of a bus failure condition because then the TX-outputs are switched off automatically after a short time (Bus-Off state). During normal operation I_{OT} is a peak current, permitted for $t < 100$ ms. The average output current must not exceed 10 mA for each TX-output.
- The value is based on the maximum allowable die temperature and the thermal resistance of the package, not on device power consumption.

10 DC CHARACTERISTICS

$V_{DD} \leq 5$ V ±10%; $V_{SS} = 0$ V; $T_{amb} = -40$ to $+125$ °C for the PCA82C200 and $T_{amb} = -40$ to $+85$ °C for the PCF82C200. All voltages measured with respect to V_{SS} unless otherwise specified

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
Supply					
V_{DD}	supply voltage range		4.5	5.5	V
I_{DD}	supply current: operating	$\overline{RST} = V_{SS}$; $f_{CLK} = 16$ MHz (note 1)	–	15	mA
I_{sm}	sleep mode	oscillator inactive (note 2)	–	40	µA
Inputs					
V_{IL1}	LOW level input voltage (except XTAL1, RX0 and RX1)		–0.5	0.8	V
V_{IL2}	XTAL1 LOW level input voltage		–	$0.2V_{DD}$	V
V_{IH1}	HIGH level input voltage (except XTAL1, RST, RX0 and RX1)		3.2	$V_{DD} + 0.5$	V
V_{IH2}	XTAL1 HIGH level input voltage		$0.7V_{DD}$	–	V
V_{IH3}	\overline{RST} HIGH level input voltage		3.3	$V_{DD} + 0.5$	V
I_{LI}	input leakage current (except XTAL1, RX0 and RX1)	0.45 V $< V_i < V_{DD}$	–	±10	µA

Stand-alone CAN-controller

82C200

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
Outputs					
V_{OL}	LOW level output voltage (except XTAL2, TX0 and TX1)	$I_{OL} = 1.6 \text{ mA}$	–	0.45	V
V_{OH1}	HIGH level output voltage (except TX0, TX1, $\overline{\text{INT}}$ and CLK OUT)	$I_{OH} = -80 \mu\text{A}$	2.4	–	V
V_{OH2}	CLK OUT HIGH level output voltage	$I_{OH} = -80 \mu\text{A}$	$0.8V_{DD}$	–	V
CAN input comparator					
V_{DIF}	differential input voltage	$V_{DD} = 5 \text{ V} \pm 5\%$; $1.4 \text{ V} < V_i < V_{DD} - 1.4 \text{ V}$ note 3	± 42	–	mV
V_{HYST}	hysteresis voltage	note 3	12	45	mV
I_i	input current		–	± 400	nA
CAN output driver					
V_{OLT}	TX0 and TX1 output voltage LOW	$V_{DD} = 5 \text{ V} \pm 5\%$ $I_o = 1.2 \text{ mA}$ (note 3)	–	0.1	V
V_{OHT}	TX0 and TX1 output voltage HIGH	$I_o = 10 \text{ mA}$	–	1.0	V
		$I_o = 1.2 \text{ mA}$ (note 3)	$V_{DD} - 0.1$	–	V
		$I_o = 10 \text{ mA}$	$V_{DD} - 1.0$	–	V

Notes

- $(AD0 - AD7) = \overline{\text{ALE}} = \overline{\text{RD}} = \overline{\text{WR}} = \overline{\text{CS}} = V_{DD}$; $\text{MODE} = V_{SS}$; $\text{RX0} = 2.7 \text{ V}$; $\text{RX1} = 2.3 \text{ V}$; $\text{XTAL1} = 0.5V_{DD} - 0.5 \text{ V}$; all outputs unloaded.
- $(AD0 - AD7) = \text{ALE} = \overline{\text{RD}} = \overline{\text{WR}} = \overline{\text{INT}} = \overline{\text{RST}} = \overline{\text{CS}} = \text{MODE} = \text{RX0} = V_{DD}$; $\text{RX1} = \text{XTAL1} = V_{SS}$; all outputs unloaded.
- Not tested during production.

Stand-alone CAN-controller

82C200

11 AC CHARACTERISTICS

 $V_{DD} = 5\text{ V} \pm 10\%$; $V_{SS} = 0\text{ V}$; $C_L = 50\text{ pF}$ (output pins); $T_{amb} = -40$ to $+85/125\text{ }^\circ\text{C}$; unless otherwise specified (note 1)

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
f_{CLK}	oscillator frequency		3	16	MHz
t_{SU1}	address set-up to ALE/AS LOW		10	–	ns
t_{HD1}	address hold time		22	–	ns
t_{PW1}	ALE/AS pulse width		35	–	ns
t_{VD1}	\overline{RD} LOW to valid data output	Intel mode	–	60	ns
t_{VD2}	E HIGH to valid data output	Motorola mode	–	60	ns
t_{DF1}	data float after \overline{RD} HIGH	Intel mode	10	55	ns
t_{DF2}	data float after E LOW	Motorola mode	10	55	ns
t_{SU2}	input data set-up to \overline{WR} HIGH	Intel mode	30	–	ns
t_{HD2}	input data hold after \overline{WR} HIGH	Intel mode	13	–	ns
t_{H1}	\overline{WR} HIGH to next ALE HIGH		23	–	ns
t_{LH3}	E LOW to next AS HIGH	Motorola mode	23	–	ns
t_{SU3}	input data set-up to E LOW	Motorola mode	30	–	ns
t_{HD3}	input data hold after E LOW	Motorola mode	25	–	ns
t_{LL1}	ALE LOW to \overline{WR} LOW	Intel mode	10	–	ns
t_{LL2}	ALE LOW to \overline{RD} LOW	Intel mode	10	–	ns
t_{LH1}	AS LOW to E HIGH	Motorola mode	10	–	ns
t_{SU4}	set-up time of $\overline{RD}/\overline{WR}$ to E HIGH	Motorola mode	20	–	ns
t_{PW2}	\overline{WR} pulse width	Intel mode	170	–	ns
t_{PW3}	\overline{RD} pulse width	Intel mode	170	–	ns
t_{PW4}	E pulse width	Motorola mode	170	–	ns
t_{LL3}	\overline{CS} LOW to \overline{WR} LOW	Intel mode	0	–	ns
t_{LL4}	\overline{CS} LOW to \overline{RD} LOW	Intel mode	0	–	ns
t_{LH2}	\overline{CS} LOW to E HIGH	Motorola mode	0	–	ns
Input comparator/output driver					
t_{sd}	sum of the input and output delays	$V_{DD} = 5\text{ V} \pm 5\%$; $V_{DIF} = \pm 42\text{ mV}$; $1.4\text{ V} < V_I < V_{DD} - 1.4\text{ mV}$	–	62	ns

Note

- AC characteristics are not tested.

Stand-alone CAN-controller

82C200

11.1 AC timing diagrams

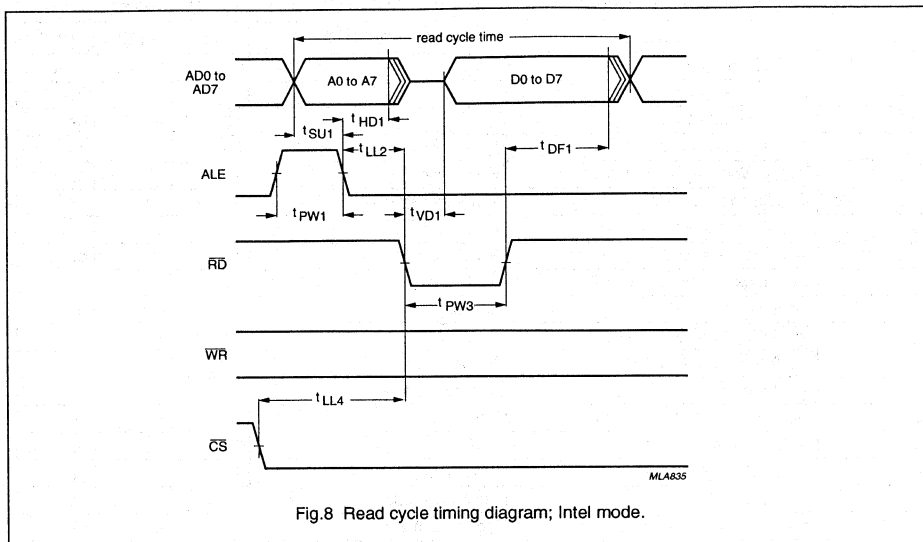


Fig.8 Read cycle timing diagram; Intel mode.

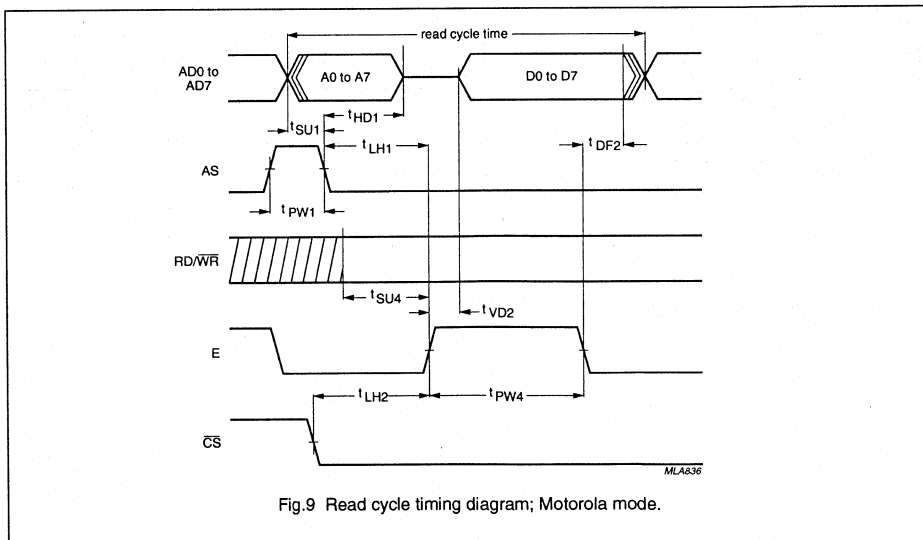


Fig.9 Read cycle timing diagram; Motorola mode.

Stand-alone CAN-controller

82C200

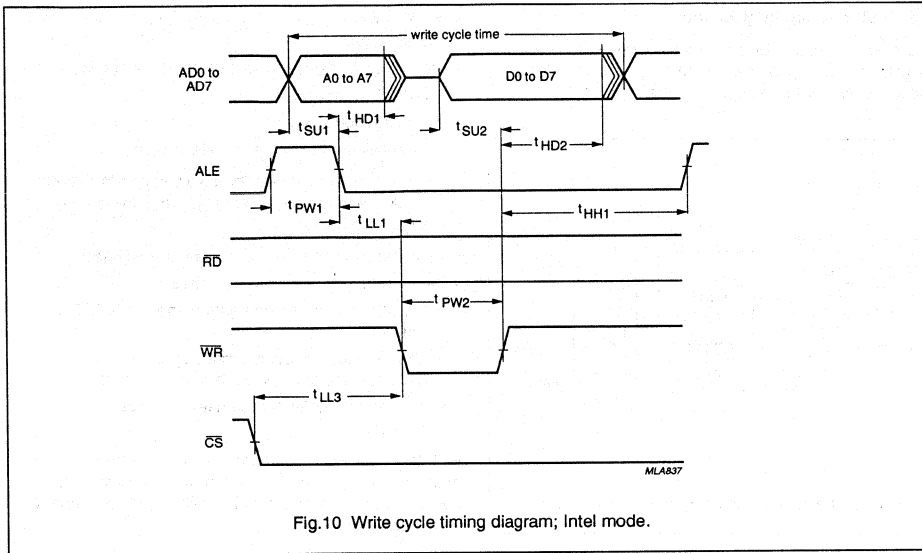


Fig.10 Write cycle timing diagram; Intel mode.

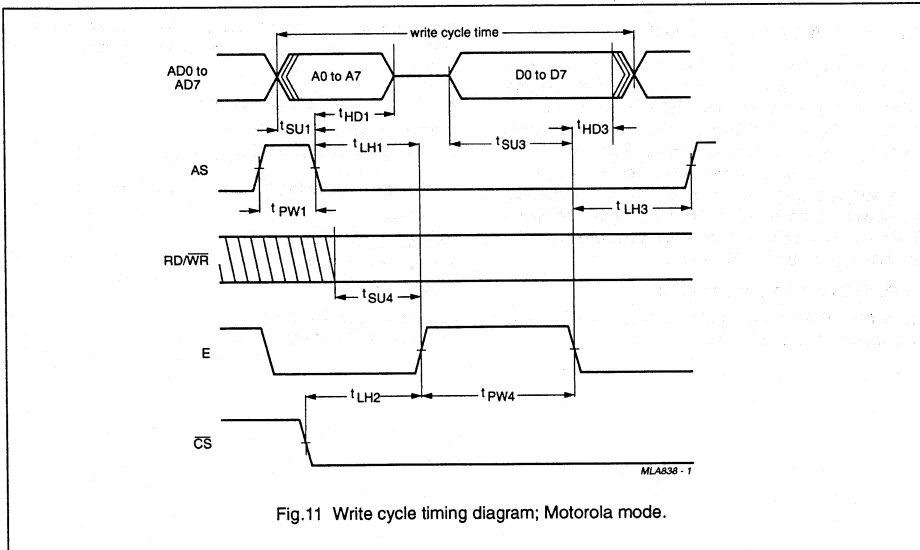


Fig.11 Write cycle timing diagram; Motorola mode.

Stand-alone CAN-controller

82C200

11.2 Additional AC information

To provide optimum noise immunity under worse case conditions, the chip is powered by three separate pins and grounded by three separate pins, see Fig.12.

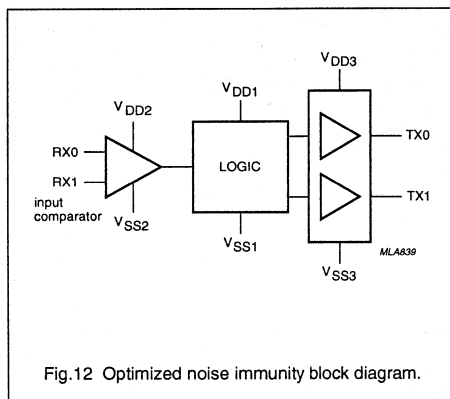


Fig.12 Optimized noise immunity block diagram.

12 DEVELOPMENT SUPPORT AND TOOLS

12.1 The PCX82C200 Evaluation Board

Philips offers powerful support during the design and test stages of CAN networks, working closely with customers to develop their systems. The 'Philips Stand-alone CAN-Controller (PSCC) Evaluation Board' is a versatile tool being a ready-to-use hardware and software module, very similar to a real CAN module. Since a 5 V power supply is provided, the board can be used in any vehicle without modification. An RS232 interface allows a terminal or a PC with terminal-emulation software to be connected to the board. The board comprises:

- a PCX82C200 CAN-bus controller
- a PCA80C552 microcontroller with up to 32K x 8 bits external RAM and EPROM

- a 5 V power supply with protection against car battery disturbances
- two different physical CAN-bus interfaces (selectable)
- an RS232 interface
- demonstration hardware
- a wrap field for customer-specific circuitry.

The software provided with the board supports "learning about CAN" and assists in prototype (e.g. in-vehicle) networks. It provides:

- demonstration software (automatically-initiated)
- the menu-driven software comprises:
 - a facility to alter the contents of the PCX82C200 registers
 - a bus monitor to receive messages from the CAN-bus and to display them on a terminal
 - a download facility for the user's application software.

With these facilities the board is a basis for prototype modules; when using entirely your own software, the board can be used as a custom, debugged and proven hardware module.

12.2 Advanced support

For further development support, Philips subcontractor I+ME offers a complete set of development tools including:

- a CAN simulator; CAN/Net Sim
- an emulator; CAN/Net Emu
- a network analyzer; CAN/Net Anal.

I+ME can be contacted through the following address:

I+ME GmbH
Ferdinandstrasse 15 A
D-3340 Wolfenbuettel
West Germany.

Phone: ++49-5331-72066

Fax: ++49-5331-32455

CAN controller interface

PCA82C250

FEATURES

- Fully compatible with the "ISO/DIS 11898" standard
- High speed (up to 1 Mbaud)
- Bus lines protected against transients in an automotive environment
- Slope control to reduce radio frequency interference (RFI)
- Differential receiver with wide common-mode range for high immunity against electromagnetic interference (EMI)
- Thermally protected
- Short-circuit proof to battery and ground
- Low current standby mode
- An unpowered node does not disturb the bus lines
- At least 110 nodes can be connected.

APPLICATIONS

- High-speed applications (up to 1 Mbaud) in cars.

GENERAL DESCRIPTION

The PCA82C250 is the interface between the CAN protocol controller and the physical bus. The device provides differential transmit capability to the bus and differential receive capability to the CAN controller.

QUICK REFERENCE DATA

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
V_{CC}	supply voltage		4.5	5.5	V
I_{CC}	supply current		–	170	μ A
$1/t_{bit}$	maximum transmission speed	non-return-to-zero	1	–	Mbaud
V_{CAN}	CANH, CANL input/output voltage		–8	+18	V
ΔV	differential bus voltage		1.5	3.0	V
t_{pd}	propagation delay	high-speed mode	–	50	ns
T_{amb}	operating ambient temperature		–40	+125	$^{\circ}$ C

ORDERING INFORMATION

TYPE NUMBER	PACKAGE			
	PINS	PIN POSITION	MATERIAL	CODE
PCA82C250	8	DIP8	plastic	SOT97-1
PCA82C250T	8	SO8	plastic	SOT96-1

CAN controller interface

PCA82C250

BLOCK DIAGRAM

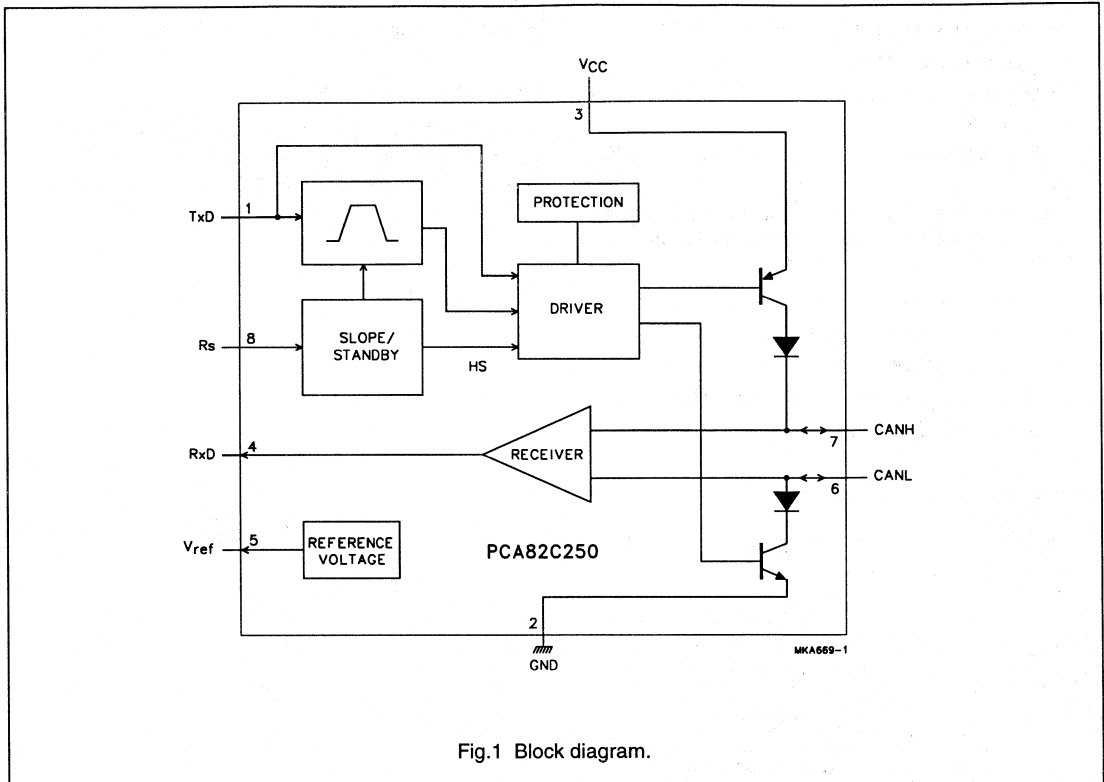


Fig.1 Block diagram.

PINNING

SYMBOL	PIN	DESCRIPTION
TxD	1	transmit data input
GND	2	ground
V _{CC}	3	supply voltage
RxD	4	receive data output
V _{ref}	5	reference voltage output
CANL	6	LOW level CAN voltage input/output
CANH	7	HIGH level CAN voltage input/output
Rs	8	slope resistor input

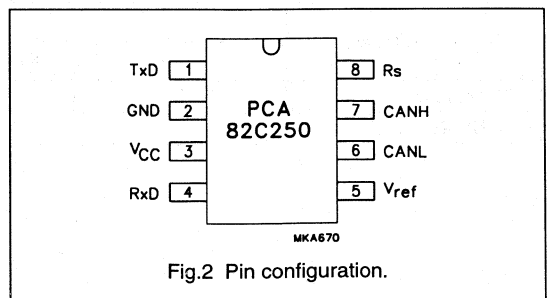


Fig.2 Pin configuration.

CAN controller interface

PCA82C250

FUNCTIONAL DESCRIPTION

The PCA82C250 is the interface between the CAN protocol controller and the physical bus. It is primarily intended for high-speed applications (up to 1 Mbaud) in cars. The device provides differential transmit capability to the bus and differential receive capability to the CAN controller. It is fully compatible with the "ISO/DIS 11898" standard.

A current limiting circuit protects the transmitter output stage against short-circuit to positive and negative battery voltage. Although the power dissipation is increased during this fault condition, this feature will prevent destruction of the transmitter output stage.

If the junction temperature exceeds a value of approximately 160 °C, the limiting current of both transmitter outputs is decreased. Because the transmitter is responsible for the major part of the power dissipation, this will result in a reduced power dissipation and hence a lower chip temperature. All other parts of the IC will remain in operation. The thermal protection is particularly needed when a bus line is short-circuited.

The CANH and CANL lines are also protected against electrical transients which may occur in an automotive environment. Pin 8 (Rs) allows three different modes of operation to be selected: high-speed, slope control or standby.

For high-speed operation, the transmitter output transistors are simply switched on and off as fast as possible. In this mode, no measures are taken to limit the rise and fall slope. Use of a shielded cable is recommended to avoid RFI problems. The high-speed mode is selected by connecting pin 8 to ground.

For lower speeds or shorter bus length, an unshielded twisted pair or a parallel pair of wires can be used for the bus. To reduce RFI, the rise and fall slope should be limited. The rise and fall slope can be programmed with a resistor connected from pin 8 to ground. The slope is proportional to the current output at pin 8.

If a HIGH level is applied to pin 8, the circuit enters a low current standby mode. In this mode, the transmitter is switched off and the receiver is switched to a low current. If dominant bits are detected (differential bus voltage >0.9 V), RxD will be switched to a LOW level. The microcontroller should react to this condition by switching the transceiver back to normal operation (via pin 8). Because the receiver is slow in standby mode, the first message will be lost.

Table 1 Truth table of CAN transceiver.

SUPPLY	TxD	CANH	CANL	BUS STATE	RxD
4.5 to 5.5 V	0	HIGH	LOW	dominant	0
4.5 to 5.5 V	1 (or floating)	floating	floating	recessive	1
<2 V (not powered)	X	floating	floating	recessive	X
2 V < V _{CC} < 4.5 V	>0.75V _{CC}	floating	floating	recessive	X
2 V < V _{CC} < 4.5 V	X	floating if V _{Rs} > 0.75V _{CC}	floating if V _{Rs} > 0.75V _{CC}	recessive	X

Table 2 Rs (pin 8) summary.

CONDITION FORCED AT Rs	MODE	RESULTING VOLTAGE OR CURRENT AT Rs
V _{Rs} > 0.75V _{CC}	standby	I _{Rs} < 110 μA
-10 μA < I _{Rs} < -200 μA	slope control	0.4V _{CC} < V _{Rs} < 0.6V _{CC}
V _{Rs} < 0.3V _{CC}	high-speed	I _{Rs} < -500 μA

CAN controller interface

PCA82C250

LIMITING VALUES

In accordance with the Absolute Maximum Rating System (IEC 134). All voltages are referenced to pin 2; positive input current.

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
V_{CC}	supply voltage		-0.3	+9.0	V
V_n	DC voltage at pins 1, 4, 5 and 8		-0.3	$V_{CC} + 0.3$	V
$V_{6,7}$	DC voltage at pins 6 and 7	$0\text{ V} < V_{CC} < 5.5\text{ V}$; no time limit	-8.0	+18.0	V
V_{trt}	transient voltage at pins 6 and 7	see Fig.8	-150	+100	V
T_{stg}	storage temperature		-55	+150	°C
T_{amb}	operating ambient temperature		-40	+125	°C
T_{vj}	virtual junction temperature	note 1	-40	+150	°C

Note

1. In accordance with "IEC 747-1".

An alternative definition of virtual junction temperature T_{vj} is: $T_{vj} = T_{amb} + P_d \times R_{th\ vj-amb}$,

where $R_{th\ vj-amb}$ is a fixed value to be used for the calculation of T_{vj} .

The rating for T_{vj} limits the allowable combinations of power dissipation (P_d) and ambient temperature (T_{amb}).

HANDLING

Classification A: human body model; C = 100 pF; R = 1500 Ω ; V = ± 2000 V.

Classification B: machine model; C = 200 pF; R = 0 Ω ; V = ± 200 V.

QUALITY SPECIFICATION

Quality specification "SNW-FQ-611 part E" is applicable and can be found in the "Quality reference pocket-book" (ordering number 9398 510 34011).

THERMAL CHARACTERISTICS

SYMBOL	PARAMETER	VALUE	UNIT
$R_{th\ j-a}$	thermal resistance from junction to ambient in free air		
	PCA82C250	100	K/W
	PCA82C250T	160	K/W

CAN controller interface

PCA82C250

CHARACTERISTICS

$V_{CC} = 4.5$ to 5.5 V; $T_{amb} = -40$ to $+125$ °C; $R_L = 60$ Ω; $I_B > -10$ μA; unless otherwise specified.

All voltages referenced to ground (pin 2); positive input current; all parameters are guaranteed over the ambient temperature range by design, but only 100% tested at $+25$ °C.

SYMBOL	PARAMETER	CONDITIONS	MIN.	TYP.	MAX.	UNIT
Supply						
I_3	supply current	dominant; $V_1 = 1$ V	–	–	70	mA
		recessive; $V_1 = 4$ V; $R_B = 47$ kΩ	–	–	14	mA
		recessive; $V_1 = 4$ V; $V_B = 1$ V	–	–	18	mA
		standby; $T_{amb} < 90$ °C; note 1	–	100	170	μA
DC bus transmitter						
V_{IH}	HIGH level input voltage	output recessive	$0.7V_{CC}$	–	$V_{CC} + 0.3$	V
V_{IL}	LOW level input voltage	output dominant	–0.3	–	$0.3V_{CC}$	V
I_{IH}	HIGH level input current	$V_1 = 4$ V	–200	–	+30	μA
I_{IL}	LOW level input voltage	$V_1 = 1$ V	100	–	600	μA
$V_{6,7}$	recessive bus voltage	$V_1 = 4$ V; no load	2.0	–	3.0	V
I_{LO}	off-state output leakage current	-2 V < (V_6, V_7) < 7 V	–2	–	+1	mA
		-5 V < (V_6, V_7) < 18 V	–5	–	+12	mA
V_7	CANH output voltage	$V_1 = 1$ V	2.75	–	4.5	V
V_6	CANL output voltage	$V_1 = 1$ V	0.5	–	2.25	V
$\Delta V_{6,7}$	difference between output voltage at pins 6 and 7	$V_1 = 1$ V	1.5	–	3.0	V
		$V_1 = 1$ V; $R_L = 45$ Ω; $V_{CC} \geq 4.9$ V	1.5	–	–	V
		$V_1 = 4$ V; no load	–500	–	+50	mV
I_{sc7}	short-circuit CANH current	$V_7 = -5$ V; $V_{CC} \leq 5$ V	–	–	105	mA
		$V_7 = -5$ V; $V_{CC} = 5.5$ V	–	–	120	mA
I_{sc6}	short-circuit CANL current	$V_6 = 18$ V	–	–	160	mA
DC bus receiver: $V_1 = 4$ V; pins 6 and 7 externally driven; -2 V < (V_6, V_7) < 7 V; unless otherwise specified						
$V_{diff(r)}$	differential input voltage (recessive)		–1.0	–	0.5	V
		-7 V < (V_6, V_7) < 12 V; not standby mode	–1.0	–	0.4	V
$V_{diff(d)}$	differential input voltage (dominant)		0.9	–	5.0	V
		-7 V < (V_6, V_7) < 12 V; not standby mode	1.0	–	5.0	V
$V_{diff(hys)}$	differential input hysteresis	see Fig.5	–	150	–	mV
V_{OH}	HIGH level output voltage (pin 4)	$I_4 = -100$ μA	$0.8V_{CC}$	–	V_{CC}	V
V_{OL}	LOW level output voltage (pin 4)	$I_4 = 1$ mA	0	–	$0.2V_{CC}$	V
		$I_4 = 10$ mA	0	–	1.5	V
R_i	CANH, CANL input resistance		5	–	25	kΩ

CAN controller interface

PCA82C250

SYMBOL	PARAMETER	CONDITIONS	MIN.	TYP.	MAX.	UNIT
R_{diff}	differential input resistance		20	–	100	k Ω
C_i	CANH, CANL input capacitance		–	–	20	pF
C_{diff}	differential input capacitance		–	–	10	pF
Reference output						
V_{ref}	reference output voltage	$V_B = 1\text{ V};$ $-50\ \mu\text{A} < I_5 < 50\ \mu\text{A}$	$0.45V_{CC}$	–	$0.55V_{CC}$	V
		$V_B = 4\text{ V};$ $-5\ \mu\text{A} < I_5 < 5\ \mu\text{A}$	$0.4V_{CC}$	–	$0.6V_{CC}$	V
Timing (see Figs 4, 6 and 7)						
t_{bit}	minimum bit time	$V_B = 1\text{ V}$	–	–	1	μs
t_{onTxD}	delay TxD to bus active	$V_B = 1\text{ V}$	–	–	50	ns
t_{offTxD}	delay TxD to bus inactive	$V_B = 1\text{ V}$	–	40	80	ns
t_{onRxD}	delay TxD to receiver active	$V_B = 1\text{ V}$	–	55	120	ns
t_{offRxD}	delay TxD to receiver inactive	$V_B = 1\text{ V}; V_{CC} < 5.1\text{ V};$ $T_{amb} < +85\text{ }^\circ\text{C}$	–	82	150	ns
		$V_B = 1\text{ V}; V_{CC} < 5.1\text{ V};$ $T_{amb} < +125\text{ }^\circ\text{C}$	–	82	170	ns
		$V_B = 1\text{ V}; V_{CC} < 5.5\text{ V};$ $T_{amb} < +85\text{ }^\circ\text{C}$	–	90	170	ns
		$V_B = 1\text{ V}; V_{CC} < 5.5\text{ V};$ $T_{amb} < +125\text{ }^\circ\text{C}$	–	90	190	ns
t_{onRxD}	delay TxD to receiver active	$R_B = 47\text{ k}\Omega$	–	390	520	ns
		$R_B = 24\text{ k}\Omega$	–	260	320	ns
t_{offRxD}	delay TxD to receiver inactive	$R_B = 47\text{ k}\Omega$	–	260	450	ns
		$R_B = 24\text{ k}\Omega$	–	210	320	ns
ISRI	differential output voltage slew rate	$R_B = 47\text{ k}\Omega$	–	14	–	V/ μs
t_{WAKE}	wake-up time from standby (via pin 8)		–	–	20	μs
t_{dRxDL}	bus dominant to RxD LOW	$V_B = 4\text{ V};$ standby mode	–	–	3	μs
Standby/slope control (pin 8)						
V_B	input voltage for high-speed		–	–	$0.3V_{CC}$	V
I_B	input current for high-speed	$V_B = 0\text{ V}$	–	–	–500	μA
V_{stb}	input voltage for standby mode		$0.75V_{CC}$	–	–	V
I_{slope}	slope control mode current		–10	–	–200	μA
V_{slope}	slope control mode voltage		$0.4V_{CC}$	–	$0.6V_{CC}$	V

Note

- $I_1 = I_4 = I_5 = 0\text{ mA}; 0\text{ V} < V_6 < V_{CC}; 0\text{ V} < V_7 < V_{CC}; V_8 = V_{CC}.$

CAN controller interface

PCA82C250

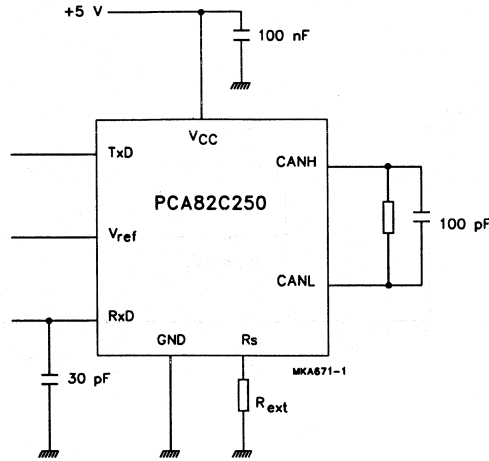


Fig.3 Test circuit for characteristics.

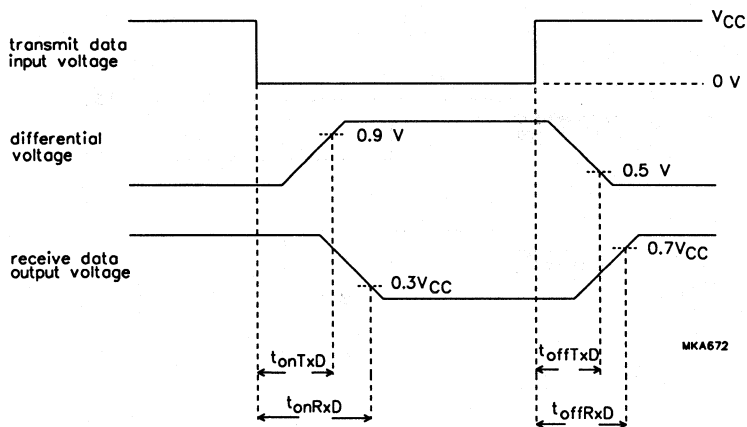


Fig.4 Timing diagram for dynamic characteristics.

CAN controller interface

PCA82C250

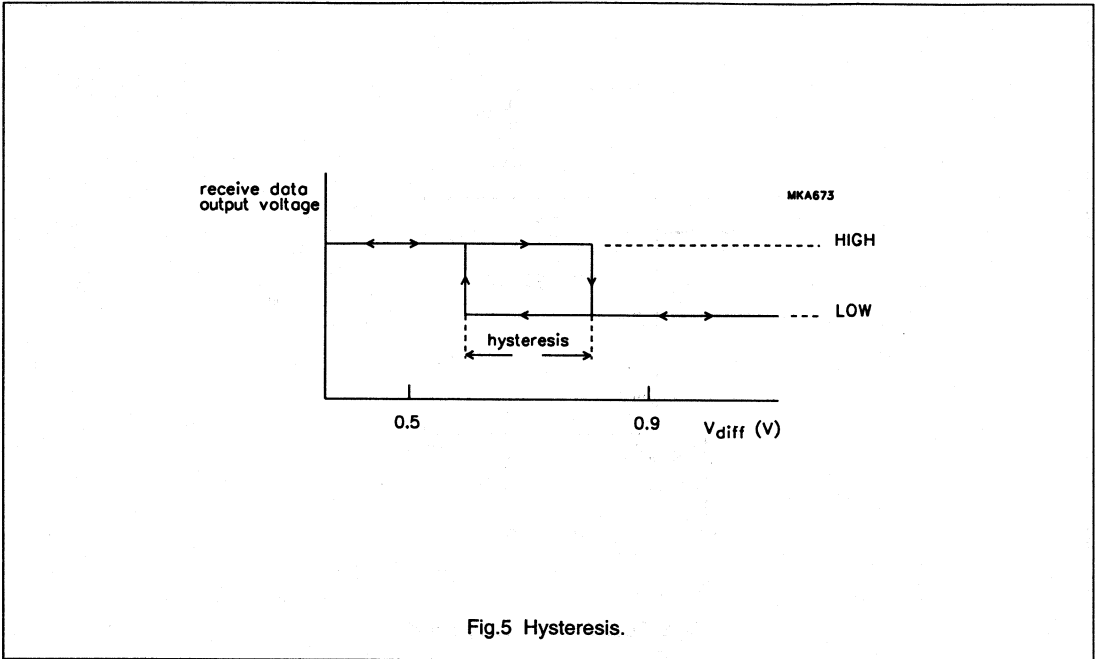
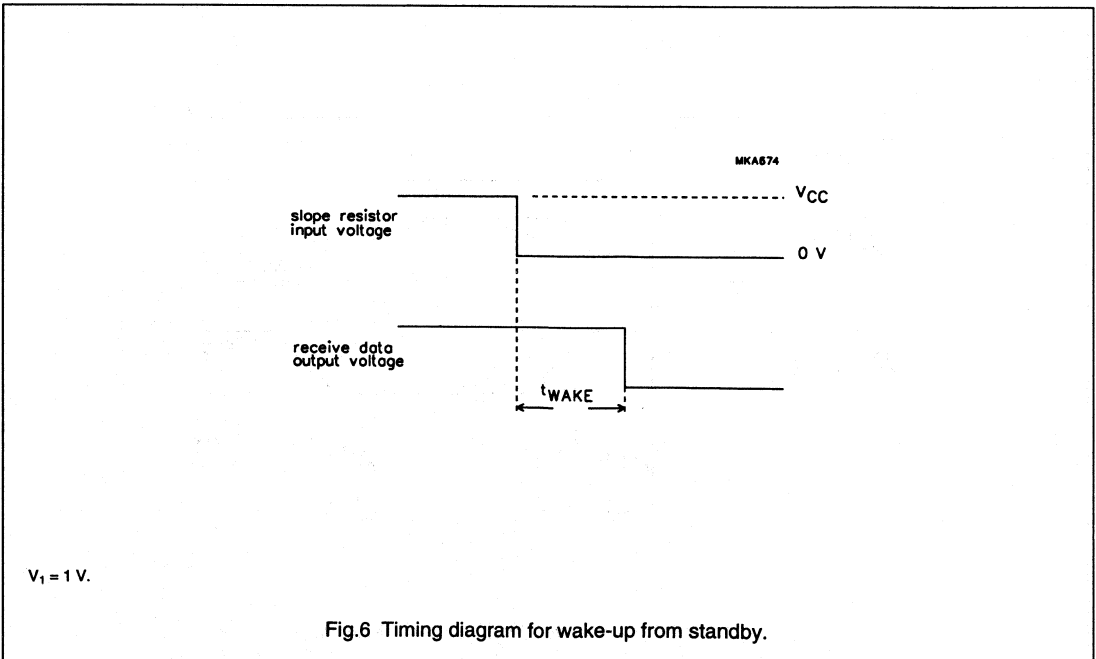


Fig.5 Hysteresis.

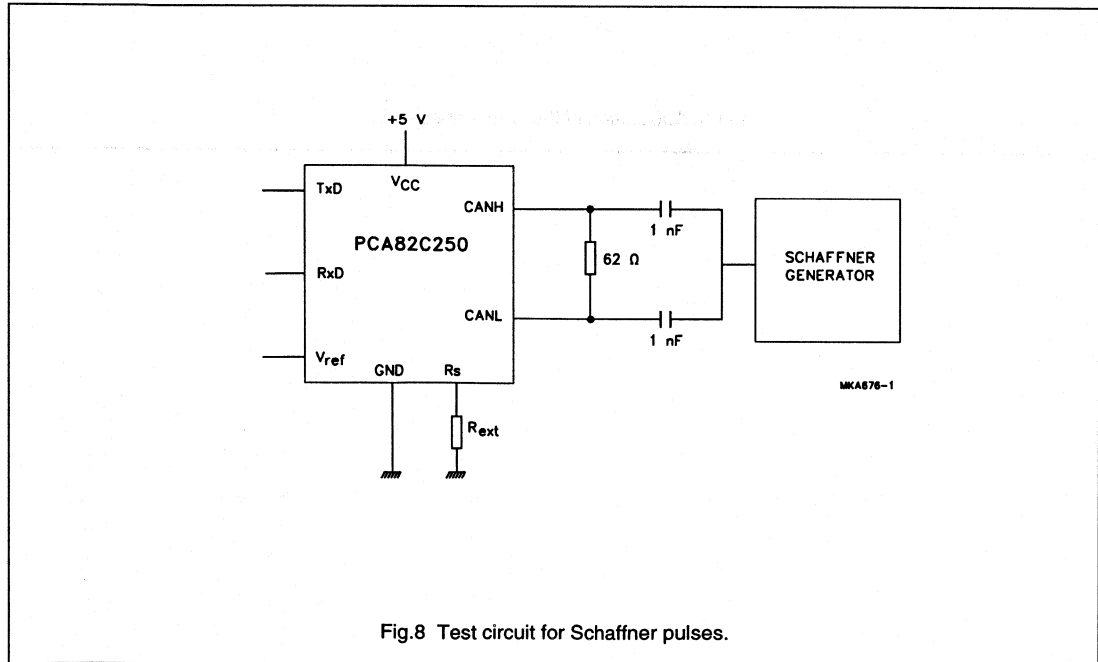
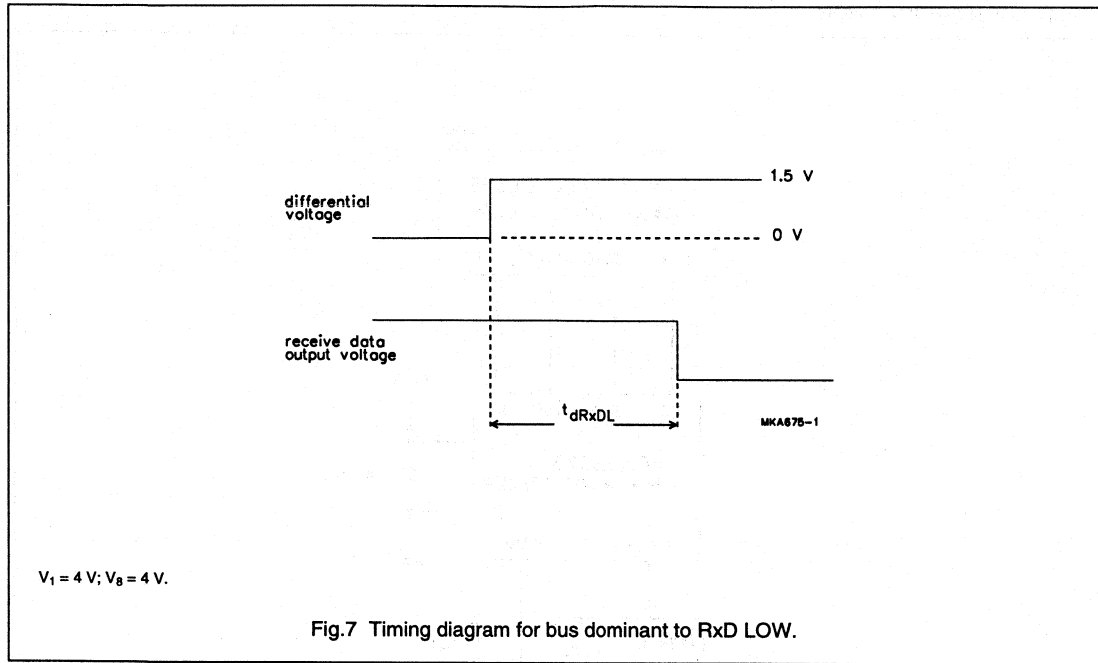


$V_1 = 1 V.$

Fig.6 Timing diagram for wake-up from standby.

CAN controller interface

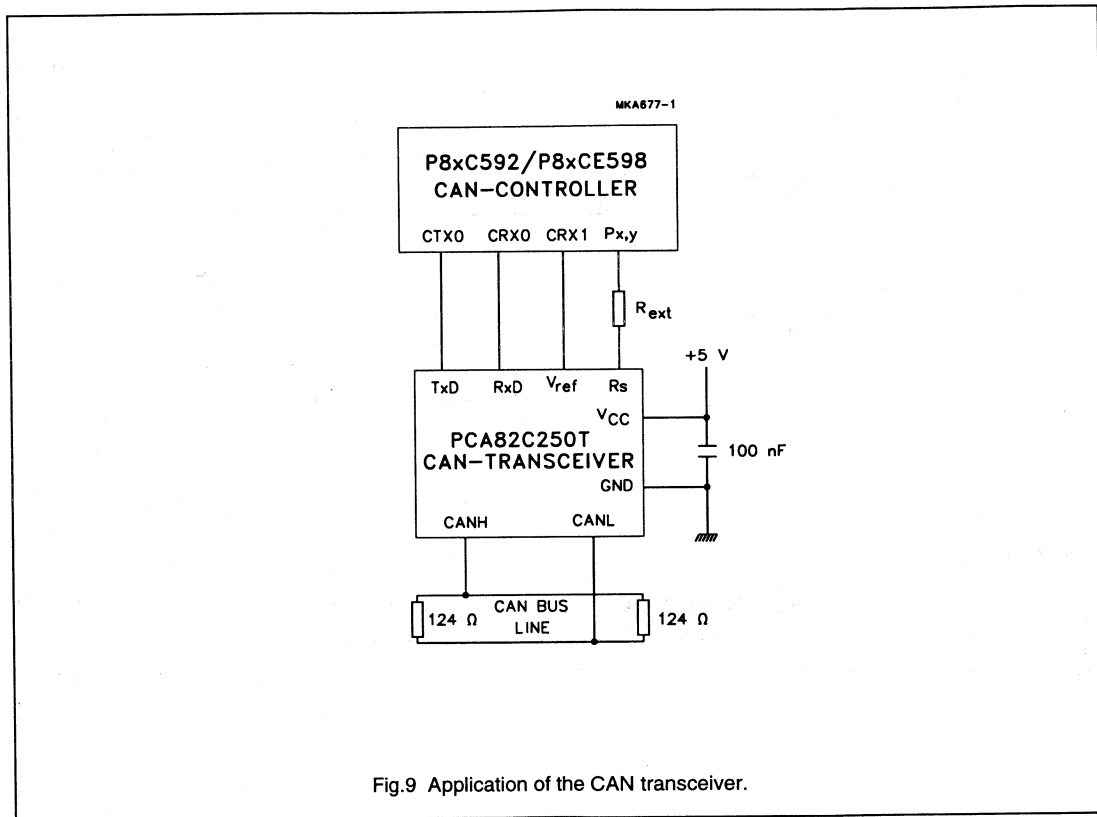
PCA82C250



CAN controller interface

PCA82C250

APPLICATION INFORMATION



CAN controller interface

PCA82C250

INTERNAL PIN CONFIGURATION

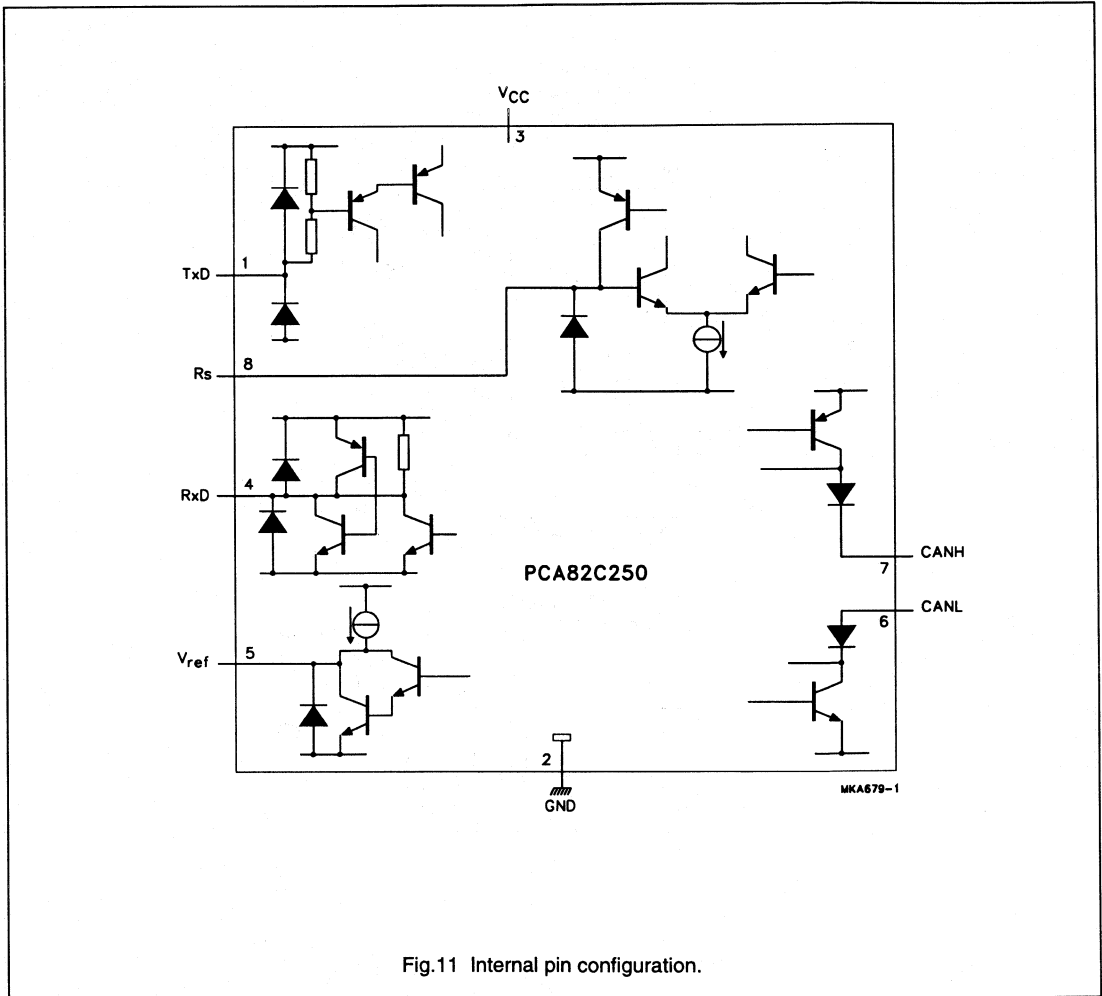


Fig.11 Internal pin configuration.

Section 7

87C750, 8XC751, 8XC752

Application Notes

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

AN422	Using the 8XC751 microcontroller as an I ² C bus master	See Section 3
AN423	Software driven serial communication routines for the 83C751 and 83C752 microcontrollers	415
AN426	Controlling air core meters with the 87C751 and SA5775	420
AN427	Timer I for the 83/87C748/749 and the 83/87C751/752 (non-I ² C applications) microcontrollers	434
AN428	Using the ADC and PWM of the 83C752/87C752	440
AN429	Airflow measurement using the 83/87C752 and "C"	447
AN430	Using the 8XC751/752 in multimaster I ² C applications	See Section 3
AN433	I ² C slave routines for the 83C751	See Section 3
AN436	"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller	466
AN439	87C751 fast NiCad charger	476
AN442	(BCM) 87C751 Specification for a bus-controlled monitor	488
AN445	ACCESS.bus mouse application code for the 8XC751 microcontroller	505
AN446	A software duplex UART for the 751/752	535
AN453	Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs	543
AN454	Interfacing the 83C576/87C576 to the ISA bus	562
EIE/AN91007	I ² C driver routines for 8XC751/2 microcontrollers	See Section 3

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

DESCRIPTION

The need often arises to make use of a serial port in connection with a microcontroller that does not have a hardware UART on-chip. Aside from the obvious cases where the microcontroller application intrinsically requires RS-232 communications to achieve its purpose, a serial output may often be a simple and convenient method of providing detailed diagnostic information to the outside world while using only a single I/O port pin. In many cases, the solution may be to implement the UART function in software. The routines included here demonstrate a method to add such a function to a microcontroller without the benefit of a hardware UART.

Examples of microcontrollers that do not have on-chip UARTs are the 83C751 and 83C752. While it is possible to connect an external UART chip to these microcontrollers, it tends to use up many I/O port pins and begins to become less economical than simply using a standard 80C51. The are several factors to be considered in deciding if the software UART method will be usable in a particular application. The first is whether the serial communication channel is to be simplex (transmit only or receive only), half-duplex (transmit and receive, but not simultaneously), or full-duplex (simultaneous transmit and receive). Both simplex and half-duplex operation are fairly easy to implement in software on an 80C51-type microcontroller, and will be covered by this application note. Full-duplex operation is more difficult to implement in software and can use up a large portion of the microcontroller's time and resources.

A second consideration to be taken into account is the amount of system resources that will be "used up" by the serial communication software. First of all, such software routines will almost always require the use of at least one counter/timer to generate the time slices for the serial bit cells. Next, the physical connection to the outside world will require one I/O port pin each for the serial input and the serial output. Moreover, the port pin used for serial input should be an external interrupt input pin. This allows the software to be interrupted automatically at the beginning of an incoming start bit and synchronizes the timer accurately to the serial data stream. Additional port pins may be used to implement signals such as Request to Send (RTS), Clear to Send (CTS), etc.

Finally, serial communication software will take up a certain amount of CPU time, more than would be required to operate a hardware UART. The overhead of software implemented serial communication may or

may not be an issue, depending on the application, the throughput of the serial channel(s), the baud rate, other tasks the CPU is handling and how time-critical they are, etc.

The program listing that is included here is a demonstration of half-duplex serial routines on the 83C751 or 83C752 microcontrollers. The operation of the software would be the same on other 80C51 derivatives, except that the counter/timer operation is slightly different. The program, as listed, will send a canned message to the serial output (port pin P1.0 in this case), then wait for data on the serial input (port pin P1.5/INT0). When a character has been received on the serial input, it will be echoed through the serial output. Since the software is inherently half-duplex, the rate at which characters are received must be less than half the rate that would be possible on a full-duplex channel. This example has been set up to receive and transmit at 9600 baud when run with a 16 MHz crystal.

The operation of the routines is fairly straightforward. Beginning with a start bit occurring on the serial input line, an interrupt (external interrupt 0) will occur. At the interrupt service routine Int0, the counter/timer is loaded with a value that will result in a time delay that is approximately equivalent to half a bit cell time for the baud rate being used, less some constant to account for the elapsed time between a timer interrupt and the point where the serial input is actually sampled. The timer reload register is loaded with a value that will result in a time delay that is as close as can be calculated to one full bit cell time. The program then starts the timer and simply returns to the main program, waiting for the timer to time out, generating another interrupt.

At that point, the serial start bit should be about halfway through its nominal duration. When the first timer interrupt occurs, the timer interrupt routine Timr0 calls the receive bit routine RxBit which checks to make sure that the start bit is still valid and flags an error if it is not. The RxBit routine will then return control to the main program routine, waiting for the next timer interrupt.

On the second timer interrupt, the RxBit routine reads the serial input line and shifts the value into the serial holding register RxDat. This process is repeated until 8 bits have been read in on consecutive timer interrupts. Finally, on the tenth timer interrupt, the receive routine looks for a valid stop bit and flags an error if one is not detected. At this point, the RcvRdy flag is set to inform the main program that a character is waiting in the holding register.

The transmit routine works in a somewhat similar fashion, beginning with a call to the byte transmit routine XmtByte, which first checks to make sure that a byte receive operation is not already in progress. The RSXmt routine will then set up the timer and timer reload registers to correspond to one bit cell time, start the timer, and assert a start bit.

At each subsequent timer interrupt, the routine TxBit shifts out the next bit from the transmit holding register XmtDat, until all 8 bits have been transmitted. Once all of the data has been sent, the stop bit is asserted on the next timer interrupt. A final timer interrupt is required to insure that the stop bit lasts at least one full bit cell time. At this point, transmit flag TxFlag is cleared in order to inform the main program that the transmission is completed.

Different baud rates and different crystal frequencies can be handled easily by simply changing the values in the program for BaudVal and StrtVal. BaudVal is simply the number of 8xC751 machine cycles that occur during one bit cell time in the serial transmission. So, it may be calculated as the crystal frequency /12, divided by the baud rate. This number is rounded to the nearest integer and expressed as a negative number in the code. For example, the value shown in the listing is for 9600 baud with a 16MHz crystal. $16\text{ million} / 12$, divided by $9600 = 138.8889$. This is stated in the program as -139.

StrtVal is used to position the sampling of the serial input during a receive operation to the approximate middle of the bit cell. This is nominally half the value of BaudVal, minus some overhead to account for the interrupt latency, plus the time it takes to get to the instruction that actually reads the input. StrtVal may be calculated as $\text{BaudVal} / 2 + 30$. For example, using the BaudVal value from the listing: $-139 / 2 + 30 = -39$.

A few other useful routines are embedded in the sample program: PrByte, which converts a byte of data to hexadecimal form and transmits it; HexAsc, which converts one nibble of raw data to hexadecimal form; and Mess, which transmits an absolute string of data (usually a text message) which is terminated by a 0 byte.

This demonstration of software driven serial port routines uses 5 bytes of microcontroller RAM, two port bits (including one external interrupt input), one counter/timer, and about 256 bytes of code space, excluding the message string at the end of the listing.

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

```

RS751      Half-Duplex Serial Communication Routines      11/14/89
1
2          ;*****
3
4          ;      Software Driven Half-Duplex Serial Communication Routines
5          ;      for 83C751 and 83C752 series Microcontrollers
6
7          ;
8
9          ;*****
10
11         $Title(Half-Duplex Serial Communication Routines)
12         $Date(11/14/89)
13         $MOD751
14
15         ;*****
16
17         FF75      BaudVal  EQU      -139          ;Timer value for 9600 baud @ 16 MHz.
18                                     ;(one bit cell time)
19         FFD9      StrtVal  EQU      -39          ;Timer value to start receive.
20                                     ;(half of one bit cell time, minus the
21                                     ;time it takes the code to sample RxD)
22
23         0010      XmtDat   DATA    10h          ;Data for RS-232 transmit routine.
24         0011      RcvDat   DATA    11h          ;Data from RS-232 receive routine.
25         0012      BitCnt   DATA    12h          ;RS-232 transmit & receive bit count.
26         0013      LoopCnt  DATA    13h          ;Loop counter for test routine.
27
28         0020      Flags    DATA    20h
29         0000      TxFlag   BIT      Flags.0      ;Receive-in-progress flag.
30         0001      RxFlag   BIT      Flags.1      ;Transmit-in-progress flag.
31         0002      RxErr    BIT      Flags.2      ;Receiver framing error.
32         0003      RcvRdy   BIT      Flags.3      ;Receiver ready flag.
33
34         0090      TxD      BIT      P1.0         ;Port bit for RS-232 transmit.
35         0095      RxD      BIT      P1.5         ;Port bit for RS-232 receive (INT0).
36
37         ;*****
38
39         ; Interrupt Vectors
40
41         0000      41          ORG      0          ;Reset vector.
42         0000 0124  42          AJMP   Reset
43
44         0003      44          ORG      03H        ;External interrupt 0.
45         0003 019F  45          AJMP   ExInt0     ;Indicates RS-232 start bit received.
46
47         000B      47          ORG      0BH        ;Timer 0 interrupt.
48         000B 0175  48          AJMP   Timr0      ;Baud rate generator.
49
50         0013      50          ORG      13H        ;External interrupt 1 (not used).
51         0013 32    51          RETI
52
53         001B      53          ORG      1BH        ;Timer I interrupt (not used).
54         001B 32    54          RETI
55
56         0023      56          ORG      23H        ;I2C interrupt (not used).
57         0023 32    57          RETI
58
59         ;*****
60
61         ;Simple test of RS-232 transmit and receive.
62
63         0024 758130 63      Reset:  MOV     SP,#30h
64         0027 752000 64          MOV     Flags,#0      ;Clear RS-232 flags.
65         002A C201   65          CLR     RxFlag
66         002C 758800 66          MOV     TCON,#00h    ;Set up timer controls.
67         002F 75A882 67          MOV     IE,#82h     ;Enable timer 0 interrupts.
68
69         0032 751310 69          MOV     LoopCnt,#16   ;Test transmit first.
70         0035 7900   70          MOV     R1,#0       ;Zero line count.
71         0037 90010C 71          MOV     DPTR,#Msg1   ;Point to message string.

```

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

```

003A 11FB      72   Loop1:  ACALL   Mess           ;Send an RS-232 message repeatedly.
003C 743A      73           MOV     A,#':'
003E 1154      74           ACALL   XmtByte
0040 E9        75           MOV     A,R1
0041 11DD      76           ACALL   PrByte           ;Print R1 contents.
0043 09        77           INC     R1               ;Advance R1 value.
0044 D513F3    78           DJNZ   LoopCnt,Loop1
                                79
0047 D2A8      80   Loop2:  SETB   EX0           ;Enable interrupt 0 (RS-232 receive).
0049 3003FD    81           JNB    RcvRdy,$         ;Wait for data available.
004C C203      82           CLR    RcvRdy
004E E511      83           MOV    A,RcvDat        ;Echo same byte.
0050 1154      84           ACALL   XmtByte
0052 80F3      85           SJMP   Loop2
                                86
                                87
                                88   ; Send a byte out RS-232 and wait for completion before returning.
                                89   ; (use if there is nothing else to do while RS-232 is busy)
                                90
0054 2001FD    91   XmtByte: JB     RxFlag,$         ;Wait for receive complete.
0057 115D      92           ACALL   RSXmt          ;Send ACC to RS-232 output.
0059 2000FD    93           JB     TxFlag,$       ;Wait for transmit complete.
005C 22        94           RET
                                95
                                96
                                97   ; Begin RS-232 transmit.
                                98
005D F510      99   RSXmt:  MOV     XmtDat,A       ;Save data to be transmitted.
005F 75120A    100          MOV    BitCnt,#10      ;Set bit count.
0062 758CFF    101          MOV    TH,#High BaudVal ;Set timer for baud rate.
0065 758A75    102          MOV    TL,#Low BaudVal
0068 758DFF    103          MOV    RTH,#High BaudVal ;Also set timer reload value.
006B 758B75    104          MOV    RTL,#Low BaudVal
006E D28C     105          SETB  TR              ;Start timer.
0070 C290     106          CLR   TxD            ;Begin start bit.
0072 D200     107          SETB  TxFlag         ;Set transmit-in-progress flag.
0074 22       108          RET
                                109
                                110
                                111   ; Timer 0 timeout: RS-232 receive bit or transmit bit.
                                112
0075 C0E0     113   Timr0:  PUSH   ACC
0077 C0D0     114          PUSH  PSW
0079 20013E    115          JB    RxFlag,RxBit    ;Is this a receive timer interrupt?
007C 200007    116          JB    TxFlag,TxBit   ;Is this a transmit timer interrupt?
007F C28C     117          CLR   TR            ;Stop timer.
0081 D0D0     118          T0Ex1: CLR   TR
0083 D0E0     119          T0Ex2: POP    PSW
0085 32       120          POP    ACC
                                121
                                122
                                123   ; RS-232 transmit bit routine.
                                124
0086 D51204    125   TxBit:  DJNZ   BitCnt,TxBusy ;Decrement bit count, test for done.
0089 C200     126          CLR   TxFlag        ;End of stop bit, release timer.
008B 80F2     127          SJMP  T0Ex1        ;Stop timer and exit.
                                128
008D E512     129   TxBusy: MOV    A,BitCnt      ;Get bit count.
008F B40104    130          CJNE  A,#1,TxNext    ;Is this a stop bit?
0092 D290     131          SETB  TxD           ;Set stop bit.
0094 80EB     132          SJMP  T0Ex2        ;Exit.
                                133
0096 E510     134   TxNext: MOV    A,XmtDat    ;Get data.
0098 13       135          RRC   A             ;Advance to next bit.
0099 F510     136          MOV    XmtDat,A
009B 9290     137          MOV    TxD,C        ;Send data bit.
009D 80E2     138          SJMP  T0Ex2        ;Exit.
                                139
                                140
                                141   ;Begin RS-232 receive (after external interrupt 0).
                                142
009F 75120A    143   ExInt0: MOV    BitCnt,#10    ;Set receive bit count.
00A2 758CFF    144          MOV    TH,#High StrtVal ;First timeout in HALF a bit time.

```

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

```

00A5 758AD9      145      MOV      TL,#Low StrtVal
00A8 758DFE      146      MOV      RTH,#High BaudVal      ;Set timer reload for baud rate.
00AB 758B75      147      MOV      RTL,#Low BaudVal
00AE 751100      148      MOV      RcvDat,#0              ;Initialize received data to 0.
00B1 C2A8        149      CLR      EX0                    ;Disable external interrupt 0.
00B3 C202        150      CLR      RxErr                 ;Clear error flag.
00B5 D28C        151      SETB    TR                     ;Start timer.
00B7 D201        152      SETB    RxFlag                 ;Set receive-in-progress flag.
00B9 32          153      RETI
154
155
156      ; RS-232 receive bit routine.
157
00BA D5120D      158      RxBit:   DJNZ    BitCnt,RxBusy   ;Decrement bit count, test for stop.
00BD 209502      159      JB      RxD,RxBitEx            ;Valid stop bit?
00C0 D202        160      RxBitErr: SETB   RxErr          ;Bad stop bit, tell mainline.
00C2 C201        161      RxBitEx: CLR    RxFlag         ;Release timer for other purposes.
00C4 D2A8        162      SETB    EX0                    ;Re-enable external interrupt 0.
00C6 D203        163      SETB    RcvRdy                ;Tell mainline that a byte is ready.
00C8 80B5        164      SJMP   T0Ex1                  ;Stop timer and exit.
165
00CA E512        166      RxBusy:  MOV    A,BitCnt        ;Get bit count.
00CC B40905      167      CJNE   A,#9,RxNext           ;Is this a start bit?
00CF 2095EE      168      JB      RxD,RxBtErr          ;Valid start bit?
00D2 80AD        169      SJMP   T0Ex2                  ;Exit.
170
00D4 E511        171      RxNext:  MOV    A,RcvDat        ;Get partial receive byte.
00D6 A295        172      MOV    C,RxD                 ;Get receive pin value.
00D8 13          173      RRC    A                      ;Shift in new bit.
00D9 F511        174      MOV    RcvDat,A              ;Save updated receive byte.
00DB 80A4        175      SJMP   T0Ex2                  ;Exit.
176
177
178      ; Print byte routine: print ACC contents as ASCII hexadecimal.
179
00DD C0E0        180      PrByte:  PUSH   ACC
00DF C4          181      SWAP   A
00E0 11EB        182      ACALL  HexAsc
00E2 1154        183      ACALL  XmtByte
00E4 D0E0        184      POP    ACC
00E6 11EB        185      ACALL  HexAsc                ;Print nibble in ACC as ASCII hex.
00E8 1154        186      ACALL  XmtByte
00EA 22          187      RET
188
189
190
191      ; Hexadecimal to ASCII conversion routine.
192
00EB 540F        192      HexAsc:  ANL    A,#0FH          ;Convert a nibble to ASCII hex.
00ED 30E308      193      JNB    ACC.3,NoAdj
00F0 20E203      194      JB     ACC.2,Adj
00F3 30E102      195      JNB    ACC.1,NoAdj
00F6 2407        196      Adj:    ADD    A,#07H
00F8 2430        197      NoAdj:  ADD    A,#30H
00FA 22          198      RET
199
200
201      ; Message string transmit routine.
202
00FB C0E0        203      Mess:   PUSH   ACC
00FD 7800        204      MOV    R0,#0                  ;R0 is character pointer (string
00FF E8          205      Mesl:  MOV    A,R0              ; length is limited to 256 bytes).
0100 93          206      MOV    A,@A+DPTR             ;Get byte to send.
0101 B40003      207      CJNE   A,#0,Send            ;End of string is indicated by a 0.
0104 D0E0        208      POP    ACC
0106 22          209      RET
210
0107 1154        211      Send:  ACALL  XmtByte         ;Send a character.
0109 08          212      INC    R0                     ;Next character.
010A 80F3        213      SJMP   Mesl
214
010C 0D0A        215      Msg1:  DB     0Dh, 0Ah

```

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

```

010E 54686973      216      DB      'This is a test of the software serial routines.', 0
0112 20697320
0116 61207465
011A 7374206F
011E 66207468
0122 6520736F
0126 66747761
012A 72652073
012E 65726961
0132 6C20726F
0136 7574696E
013A 65732E00

      217
      218      END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

```

ACC . . . . . D ADDR 00E0H PREDEFINED
ADJ . . . . . C ADDR 00F6H
BAUDVAL . . . . . NUMB FF75H
BITCNT . . . . . D ADDR 0012H
EX0 . . . . . B ADDR 00A8H PREDEFINED
EXINT0 . . . . . C ADDR 009FH
FLAGS . . . . . D ADDR 0020H
HEXASC . . . . . C ADDR 00EBH
IE . . . . . D ADDR 00A8H PREDEFINED
LOOP1 . . . . . C ADDR 003AH
LOOP2 . . . . . C ADDR 0047H
LOOPCNT . . . . . D ADDR 0013H
MESL . . . . . C ADDR 00FFH
MESS . . . . . C ADDR 00FBH
MSG1 . . . . . C ADDR 010CH
NOADJ . . . . . C ADDR 00F8H
P1 . . . . . D ADDR 0090H PREDEFINED
PRBYTE . . . . . C ADDR 00DDH
PSW . . . . . D ADDR 00D0H PREDEFINED
RCVDAT . . . . . D ADDR 0011H
RCVRDY . . . . . B ADDR 0003H
RESET . . . . . C ADDR 0024H
RSXMT . . . . . C ADDR 005DH
RTH . . . . . D ADDR 008DH PREDEFINED
RTL . . . . . D ADDR 008BH PREDEFINED
RXBIT . . . . . C ADDR 00BAH
RXBITEX . . . . . C ADDR 00C2H
RXBTERR . . . . . C ADDR 00C0H
RXBUSY . . . . . C ADDR 00CAH
RXD . . . . . B ADDR 0095H
RXERR . . . . . B ADDR 0002H
RXFLAG . . . . . B ADDR 0001H
RXNEXT . . . . . C ADDR 00D4H
SEND . . . . . C ADDR 0107H
SP . . . . . D ADDR 0081H PREDEFINED
STRTVAL . . . . . NUMB FFD9H
TOEX1 . . . . . C ADDR 007FH
TOEX2 . . . . . C ADDR 0081H
TCON . . . . . D ADDR 0088H PREDEFINED
TH . . . . . D ADDR 008CH PREDEFINED
TIMRO . . . . . C ADDR 0075H
TL . . . . . D ADDR 008AH PREDEFINED
TR . . . . . B ADDR 008CH PREDEFINED
TXBIT . . . . . C ADDR 0086H
TXBUSY . . . . . C ADDR 008DH
TXD . . . . . B ADDR 0090H
TXFLAG . . . . . B ADDR 0000H
TXNEXT . . . . . C ADDR 0096H
XMTBYTE . . . . . C ADDR 0054H
XMTDAT . . . . . D ADDR 0010H

```

Controlling air core meters with the 87C751 and SA5775

AN426

Author: Bill Houghton

INTRODUCTION

Often, certain classes of microcontroller applications surface where large amounts of on-chip resources such as a large program memory space and numerous I/O pins are not required. These applications are typically cost sensitive and desirable attributes of the MCU include low cost and modest on-chip resources such as program and data memory, I/O, and timer-counters. Substantial benefits of reduced design cycle time can be realized by using an industry-standard architecture having software compatibility with existing popular microcontrollers.

THE 87C751

The Philips 87C751 is one such microcontroller that easily meets these requirements. This device, shown in Figure 1, has a 2k x 8 program memory, 64 bytes of RAM, 19 parallel I/O lines, and a 16-bit autoreload timer-counter. It also includes an I²C serial interface and a fixed rate timer. The 87C751 is based on the 80C51 core and thus uses an industry-standard architecture and instruction set. The device is available in both ROM (83C751) and EPROM (87C751) versions. The EPROM version is available in both UV erasable and OTP packages. References to the 87C751 in this document also apply to the 83C751, unless explicitly stated.

TYPICAL APPLICATION

A typical example of such an application is the interface between the 87C751 and the Philips SA5775 Air Core Meter Driver shown in Figure 2. This circuit includes the 87C751 microcontroller, the SA5775 air core meter driver, an NE555 timer, and discrete support components.

An air core meter differs from a conventional (d'Arsonval) meter movement in that it has no spring to return the needle to a predetermined position, no zeroing adjustment, and no permanent magnet in the classical sense. Instead, it consists of two coils of wire wound in quadrature with each other around a central core in which there is a disc magnetized along its diameter. A shaft is placed through the center of this disc so that the shaft rotates with the disc. An indicating needle attached to this shaft will rotate with it.

SA5775 Air Core Meter Driver

The SA5775 is a monolithic driver for controlling air core meters typically used in automotive instrument clusters and is shown

in Figure 3. The SA5775 receives a 10-bit serial word and converts that word to four voltage outputs that appear at the SINE+, SINE-, COSINE+, and COSINE- outputs. The differential voltage at the SINE outputs are applied to one coil of the meter and the COSINE outputs are applied to the other coil of the meter.

The currents through these coils produce a resultant magnetic force which is the vector sum of the magnetic forces produced by each of the two coils. Since the currents through the coils are bidirectional this magnetic vector can rotate through a full 360 degrees. The magnetized disc within the air core meter will follow the rotating vector and the needle will indicate the vector's current position. Since 10 bits are used, there are 1024 discrete words available resulting in an angular displacement of 0.3516 degrees per bit. This is small enough to provide an apparently smooth movement of the needle. The smoothness of the motion will depend greatly on the damping factor of the meter movement.

A simplified block diagram of the SA5775 is shown in Figure 4. This device consists of a serial-in/parallel-out shift register, a data latch, a D/A converter, buffers, and an internal voltage reference.

A logic high must be present on the chip select (CS) input to clock in the data. Data appearing on the data input (DI) pin is clocked into the shift register on the rising edge of the clock (CLK) input. The data output (DO) pin is the overflow from the shift register, allowing the user to daisy chain multiple SA5775 devices. Note that data is clocked out of this pin on the falling edge of the clock. The CS pin is also used to latch the parallel outputs of the shift register into the data latch. The outputs of the data latch feed the inputs to the D/A converter. The D/A converter outputs are buffered to form the drive signals for the meter coils.

A voltage reference for the D/A converter is provided internally. It is possible to externally provide different values for these voltages and pins are provided for this purpose. However, this is not generally recommended as this could lead to increased power dissipation.

The D/A converter circuits and its associated output buffers are purposely designed such that the span of these circuits does not include the power supply rails. This is to avoid inaccuracies that would otherwise occur if the output were to become very close to either supply rail. With a supply voltage of

14 volts (VIGN), the positive reference (VREF+) is approximately 8 volts and the negative reference (VREF-) is approximately 1 volt. The outputs will then span a range from 1 volt to approximately 11 volts. The maximum output is $[(VREF+) + 0.41(VREF+) - (VREF-)]$. The SA5775 is designed to drive air core meters having a minimum winding impedance of 200 ohms.

The clock high and low time requirements are each 200 ns minimum, implying a maximum data rate approaching 2.5 megabits per second. At this rate it would require approximately 4 ms to ramp from zero to full scale if all binary codes were loaded into the SA5775. However, the air core meter cannot respond to such data rates.

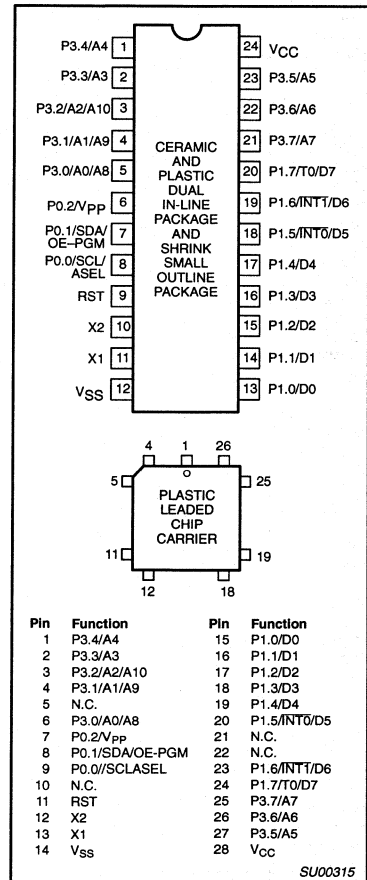


Figure 1. Pin Configuration

Controlling air core meters with the 87C751 and SA5775

AN426

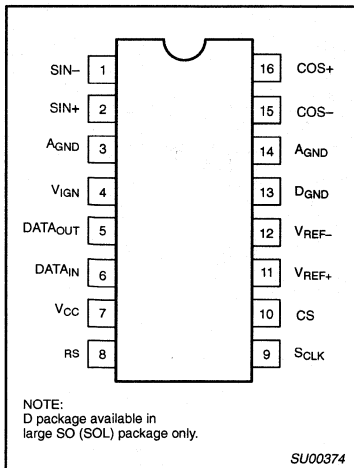


Figure 3. D and N Packages

87C751 Microcontroller

The 87C751 microcontroller provides all of the intelligence in this application. It samples various input ports to determine which demonstration programs to run, the incremental step sizes for angular displacement of the meter core, and the time delay between increments. In one of the demonstration modes, it also samples a variable frequency input and positions the meter core in response to the frequency of that input. The 87C751 also transmits the 10-bit serial data to the SA5775. Data input (DI), Clock (CLK), and Chip Select (CS) lines are driven from the 87C751.

Port 0 of the 87C751 is a 3-bit wide port and is used for communicating data to the ACMD. Data is transmitted, MSB first, in a serial stream clocked into the DI of the SA5775 on the rising edge of the clock. In order to clock in data, the CS pin of the SA5775 must be high. The data in the input register is shifted into a latch that drives the DAC on the high to low transition of the CS line. As data is shifted into the ACMD, it overflows through the Data Out (DO) pin on the falling edge of the clock. With this facility, multiple ACMDs can be daisy-drained with DO of one ACMD being connected to DI of the next one, and common clock and chip select lines may be used. This simplifies the interfacing to multiple meter drivers.

The 78L05 regulator (Q2) provides 5 Volt power for the board so that single supply of +14 volts can be applied to the board.

Three rotary switches are used on this board. The PROGRAM SELECT switch (S3) is used to select the program routine that is

executed, the INC SELECT (S2) switch selects the incremental step sizes of the two of the routines, and the DELAY switch (S4) is used to set the delay between successive word transmissions in one of the routines.

The START/COUNT button (S5) is used to begin execution of a routine, and to cause the next incremental step in Routine #1.

The COUNT UP/DOWN switch (S6) is used in Routine #1 to determine whether the count is increased or decreased with transmission of successive words.

NE555 Timer

The NE555 timer shown in this application example is used as a free running squarewave generator used to simulate sensor inputs such as those which might be found in an automobile, etc. The NE555 timer (U4) operates in the astable mode to produce an output frequency that can be varied from about 1Hz to about 200 Hz. Three of the program routines measure the input period and produce an output code that is proportional to the frequency present at pin 20 (TO) of the microcontroller. A RATE switch (S7) is used to select between the on board oscillator or an external source.

The program listing is included at the end of this application note.

Program Entry

The program starts at address 030(hex) on line 21 of the program listing. The first task is to write 1's to all pins of each port.

Lines 25 and 26 clear registers 6 and 7.

These registers are used in this program only to hold the data that is sent out to the ACMD. The registers are cleared to be sure that the starting value is zero.

At line 27 the program waits until the START/COUNT button (S5) is depressed before continuing. Lines 28 and 29 set the timer to overflow after 10ms. This is done by setting the timer registers for a count of 10,000 microseconds less than full scale. When the timer counter overflows the timer flag is set, and the timer is reloaded with the value in the timer register. By examining the timer flag we know when 10ms has expired.

Line 30 calls subroutine RPS (Read Port Selected), which reads Port 3 to determine which routine has been selected. Since the PROGRAM SELECT switch (S3) is connected to port pins P3.2 through P3.4, subroutine RPS (lines 507 through 511 at the end of the program) first reads Port 3 into the accumulator, then complements it because the switches used are complementary binary. The reading is then rotated right once and the upper nibble and the LSB (least significant

bit) are masked off, leaving twice the value of the port selected in the accumulator. Twice the read value is needed for the next few main program lines that determine which routine to execute.

Line 31 moves the address of label JMPTBL (Jump Table) to the 16-bit Data Pointer (DPTR) register. Line 32 causes a program jump to the address that is the sum of the value in the accumulator (two times the routine number selected) plus the DPTR register. Since each of the commands on lines 33 through 40 are two byte commands, these addresses are all separated by two bytes; hence, the need for the accumulator to contain a number that is twice the number of the selected routine.

Routine 0

This routine begins on line 41 by incrementing the 10-bit word in registers 7 and 6 by the amount indicated by the setting of the INCREMENT SELECT switch, then sending that word to the SA5775. When a full scale overflow is detected, a full scale code (3FF hex) is sent out, followed by a delay of 500 ms, then successive output codes are sent out, decremented by an amount indicated by the INCREMENT SELECT switch. When an underflow is detected a code of zero scale is sent and the routine returns to the beginning of the program. This routine is implemented with a series of subroutine calls.

The SO subroutine begins on line 356 and starts by sending out whatever ten bits that in the two LSBs of register 7 (R7) plus the 8 bits of R6 by calling the SENDIT subroutine. Then it calls the UP subroutine, which increases the word value to be sent out. The program then jumps to the beginning of this subroutine, repeating the process of sending out a word and incrementing to the next word until an overflow from the tenth bit (bit 2 of R7) is detected at line 362.

The SENDIT subroutine (beginning on line 476) brings the CS line high, sets a bit counter (R1) to 2 (to send out two bits of R7), brings the value of R7 to the accumulator, rotates the accumulator to the right three times through the carry bit to bring the two LSBs to the position of the two MSBs, calls the SEND1 routine, which sends the number of bits in the accumulator, starting with the MSB, indicated by R1. Counter R1 is then set to 8 to send out all 8 bit of R6 and the accumulator is loaded with the contents of R6. The SEND1 routine is again called to send out the final 8 bits, and, on line 491, the CS line is brought low, loading the SA5775 internal parallel latch with the contents of the input shift register.

Controlling air core meter meters with the 87C751 and SA5775

AN426

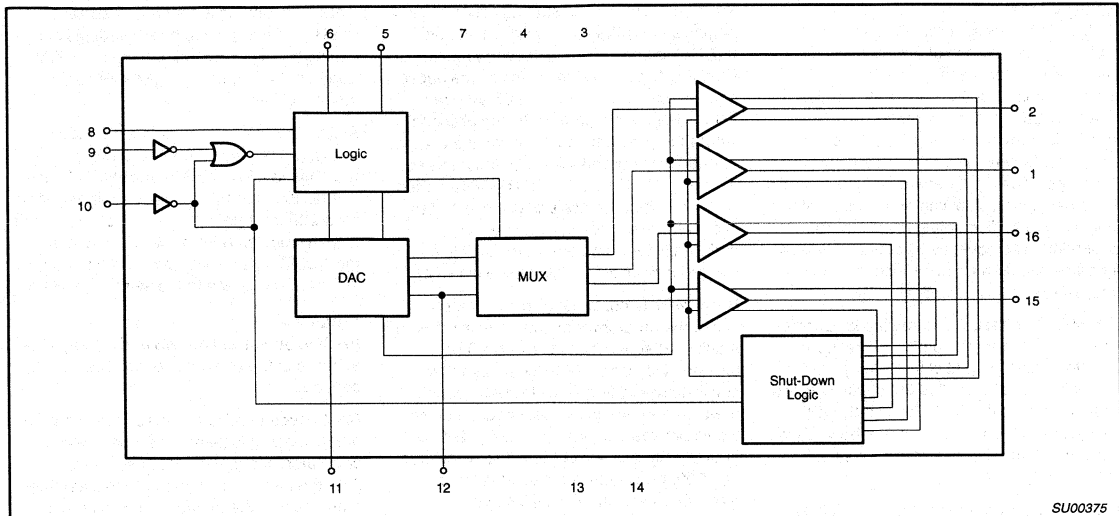


Figure 4. Block Diagram of the SA5775

The SEND1 routine rotates the accumulator left through the carry bit, moves the value of the carry bit to port pin PO.1 (SDA—Serial Data pin), waits to provide a setup time, brings the clock low, waits, brings the clock high, waits, then decrements bit counter sends the next bit if the counter is not zero. A return is executed when the counter becomes zero.

The UP subroutine, beginning at line 364, reads the delay selected by switch S4 at port pin P1, complements it (again, because the rotary switches are complementary binary), masks off the upper four bits (because the delay switch has just four positions and is connected to the lower four bits of the port), multiplies it by 4 (rotates left twice), then moves the result to R1. If R1 is not zero, the program jumps around line 376 and calls a 10ms delay (subroutine DLY10MS) the number of times entered into R1.

The 10ms delay subroutine (starting at line 436) sets the timer for 10ms, waits at line 446 for the timer flag to be set, clears the timer flag, stops the timer, and returns, in this case, to line 379, where the program decrements R1 and repeats the 10ms delay until R1 is zero.

If the selected delay was zero, the program jumps from line 376 to line 380 and reads port 3 to determine the amount the sent out word is to change from the value previously sent out. The accumulator is complemented and the upper 6 bits masked off to recover only the two bits of the selected increment amount. Since increments of 1, 2, 3, or

4 LSBs are hardly noticeable, the program then multiplies the result by 8 (rotate left three times). To insure a minimum change amount, the accumulator is increment by one at line 386. This all means that the increment amounts that can be selected are 1, 9, 17, or 25 LSBs. This amount is added, in lines 387 through 391, to the word previously sent out and we return from this subroutine.

After calling the S0 subroutine, PR0GO call the FULLSC (full scale) subroutine, which sends out the full scale code of 3E8(hex). Although a 10-bit full scale code would be 3FF(hex), going only to 3E8 allows an easy distinction between zero scale and full scale when looking at the display. The FULLSC subroutine is found at line 352.

After advancing to full scale, there is a 500ms delay, found at line 464 and called from line 48, then 49 calls the SOD subroutine to send out decreasing word values.

The SOD subroutine begins at line 393 and begins by sending out the current word in R7 and R6 from line 398, then calling subroutine DOWN, which calculates the next (decreasing) word to send out. DOWN begins at line 402. It essentially does the same thing as the UP subroutine, but subtracts the INCREMENT SELECT value from the previously sent word rather than adding to it.

At line 50 subroutine ZEROSC is called to send a zero scale code to the SA5775, then the program branches back to the beginning.

Routine 1

This routine is selected with the PROGRAM SELECT switch is in position 1 or position 9. Routine 1 (PROG1) increments or decrements the word send out, depending upon the setting of the COUNT UP/COUNT DOWN switch, S6. The amount of change is determined by the setting of the INC SELECT switch, S2.

At line 63, the program examines S6 at port pin P3.6 and jumps to the decrement portion of the routine if the pin is low. If this pin is high, the UP subroutine is called from line 64 to increase the R7/R6 word value. The UP subroutine was previously described.

If pin P3.6 is low, the DOWN subroutine (line 402) decreases the previous word sent out by the amount determined from the INC SELECT switch setting.

To insure enough delay to allow the user time to release the START/COUNT button (S5), a delay of 200ms is included at line 66 before jumping to line 27, where another depression of the START/COUNT button is awaited. If S3 (PROGRAM SELECT) is still set to 1 or 9, depression of S5 will cause a jump back to line 52. If another program is selected, the program will jump to the selected routine.

Holding down S5 with PROGRAM SELECT set at position 1 or 9 will cause increasing or decreasing word values to be sent to the SA5775.

Controlling air core meters with the 87C751 and SA5775

AN426

Routine 2

PROG2 is the most complex of all these routines. The purpose of this routine is to cause the air core meter deflection to represent the frequency presented at the timer/counter input to the microcontroller. This is done by measuring the period of the input square wave and taking the inverse of the period. The input here must be a square wave because a slow rise and fall time at this input will cause fluctuating readings. To determine the frequency by counting pulses for a time would require a much longer time and, therefore, is impractical.

The MEAS (measure) subroutine is called at line 79 to measure the period of the input waveform and the CALC (calculate) subroutine is called at line 80 to calculate the code to send to the SA5775. The SENDIT subroutine is then called to send the word to the SA5775 and the program jumps back to line 28.

The MEAS subroutine begins at line 83 by being sure the timer is not running and clearing the timer (overflow) flag, then entering zero into both high and low bytes of the timer and the timer register. The carry bit is then cleared (line 90) and the timer started and the timer interrupt enabled.

Lines 93 and 94 form a short loop that waits until either the carry bit is set or until the TO input is low. The carry bit is set when the timer has gone beyond one second. This is done by the timer interrupt subroutine, found at lines 16 through 19. If the TO input never goes low, we know the frequency is at or near zero and the program jumps to GZS (line 108) where R3 is loaded with a 1F (hex) to cause the CALC subroutine to load zero scale into R7/R6.

When (and if) TO is found to be low, the program jumps to line 95 and waits for that input to go high. Time out process is the same as above.

Now that the TO input is found high (if is before the one second time out), the timer and carry bit are cleared in lines 97 through 100 (R3 is an extension of the timer).

At lines 101 through 107 we wait for one complete cycle at the TO input, with the timer/counter measuring that period, then return to line 80, where the CALC subroutine is called.

The CALC subroutine, starting at line 113, begins by initializing the word to send out (R7/R6) to zero, clearing the carry bit, checking to see if R3 indicates a time above one second, returning to line 81 if it does. Otherwise the program continues at line 26, where the program checks to see if the input frequency is beyond full scale (timer reading above 00 12 88 hex). If it is, R7/R6 is loaded with 12 88 hex (full scale of decimal 1,000). This value was chosen because it is sufficiently far from zero scale that it is easily discerned from zero scale on the display.

If the result is not to be full scale or zero scale, the program continues at line 140 with a shift and subtract divide routine. The dividend would be 1,000,000 (decimal) to convert back to frequency in Hertz (period measurements is in microseconds), but that would provide a maximum count of 200 at 200Hz, only one fifth of the full scale desired of 1,000. So we made the dividend to be 5,000,000 decimal, or 4C 4B 40 hex.

This algorithm is found in lines 156 through 192 and works as follows:

1. Clear a counter.
2. Rotate dividend until the first one is in the second MSB position. Since a code of 4C has already provides that, no shifting is necessary.
3. Rotate the divisor (the period in microseconds in this case) left until the first one is in the second MSB position, but the first byte is LESS THAN the first byte of the dividend. Increment the counter each time the divisor is rotated.
4. Initialize a counter to zero.
5. Rotate the quotient (answer) and dividend one bit left.
6. If first byte of quotient is smaller than the first byte of the quotient, jump to step 8.
7. Add one to the quotient and subtract the divisor from the dividend.
8. Decrement the counter and go to step 5 if it is not zero.

Once the CALC subroutine is completed, the program calls SENDIT from line 81 and jumps, ultimately, to the selected routine.

Routine 3

PROG3, beginning at line 194, measures the input period four times, then calculates the code to display that is the average of these four readings.

It starts by setting a counter for three readings, taking those three readings and storing them in memory, beginning at RAM address 20 hex, using register RO as an index register.

At line 212 the program takes a fourth reading, then adds the three previous readings to it in lines 213 through 227; and divides the sum by four (rotates right twice) in lines 229 through 239. The word to send out is then calculated from line 240 and sent to the ACMD, after which the program then looks for and jumps to the selected routine.

Routine 4

PROG4 begins at line 243 and displays the average of the current and last three words sent out.

RAM space used is first initialized to zero and a new reading is taken and a new word is calculated and saved. At lines 264 through 284, the new word is added to the last three readings and the average calculated and stored in RAM locations 28 and 29 (hex), and the average word is sent out.

At line 286, the program reads for the program selected and jumps to line 254 if this routine is selected, otherwise it goes to line 28.

Routine 5

PROG5 begins at line 293 and, very simply, send in sequence the codes for 1/8 through full scale in 1/8 scale steps, with 500ms between steps. It then steps down to zero scale in 1/8 scale steps, then returns to line 28.

Routine 6

PROG6 begins at line 314 and does the same as PROG5, but steps in 1/4 scale increments.

Routine 7

PROG7 loads the code for 3/8 scale into R7/R6, sends it, waits 500ms, changes r& for 5/8 scale, sends it, waits for 500ms, then repeats this sequence 9 more times (for a total of ten times), waits 500ms, then returns the output to zero scale and the program jumps to line 28.

Controlling air core meters with the 87C751 and SA5775

AN426

```

1 ; ACMD V3 DEMO TT.20
2 ; PROCESSOR: 87C751
3 ; 7-29-89
4 ;
5 ; The purpose of this program is to drive version 3 of the ACMD (SA5775)
6 ; demonstration board. The PROGRAM SELECT switch is used to select from
7 ; a choice of four routines. Registers R7 and R6 contain the 10-bit word
8 ; that is sent to the SA5775.
9 ;
10 $MOD751
0000 11 ORG 0
12 ;
0000 B02E 13 SJMP START ;RESET VECTOR
14 ;
000B 15 ORG 00BH ;TIMER/COUNTER INTERRUPT ROUTINE
000B 0B 16 INC R3 ;INCREMENT R3 (3rd BYTE OF TIMER)
000C 740F 17 MOV A,#0FH ;TEST FOR TIME OUT (R3 > 0F)
000E 9B 18 SUBB A,R3 ;IF R3 > 0F, CARRY IS SET
000F 32 19 RETI
20 ;
0030 21 ORG 30H ;START OF PROGRAM
0030 7580FF 22 START: MOV P0,#0FFH ;SET PORTS HIGH
0033 7590FF 23 MOV P1,#0FFH
0036 75B0FF 24 MOV P3,#0FFH
0039 7F00 25 MOV R7,#0 ;CLEAR WORD TO SEND OUT
003B 7E00 26 MOV R6,#0
003D 20B6FD 27 W: JB P3.6,W ;WAIT FOR START BUTTON DEPRESS
0040 758BF0 28 READY: MOV RTL,#LOW(0-10000) ;SET TIMER REGISTER
0043 758DD8 29 MOV RTH,#HIGH(0-10000) ;FOR 10ms TIME
0046 51D2 30 ACALL RPS ;READ PORT 3 FOR PROG SELECT
0048 90004C 31 MOV DPTR,#JMPTBL ;JMP ADDRESS TO DATA POINTER
004B 73 32 JMP @A+DPTR ;GOTO APPROPRIATE ROUTINE
004C 015C 33 JMPTBL: AJMP PROG0 ;RAMP UP AND BACK DOWN
004E 0168 34 AJMP PROG1 ;STEP UP/DOWN W/ start PRESS
0050 017A 35 AJMP PROG2 ;READ & DISPLAY SPEED
0052 2145 36 AJMP PROG3 ;DISPLAY AVERAGE OF 4 NEW READINGS
0054 2186 37 AJMP PROG4 ;DISPLAY AVERAGE OF LAST 4 READINGS
0056 21D3 38 AJMP PROG5 ;ADVANCE TO FULL SCALE AND BACK IN 45 DEGREE STEPS
0058 21F3 39 AJMP PROG6 ;ADVANCE TO FULL SCALE AND BACK IN 90 DEGREE STEPS
005A 4107 40 AJMP PROG7 ;ALTERNATE DISPLAY BETWEEN 3/8 AND 5/8 SCALE TEN TIMES
005C 41 PROG0:
42 ; This routine increases word sent at the selected step size (INCREMENT SELECT)
43 ; and delay time (DELAY), up to full scale, waits 500ms, then decreases the
44 ; word sent at the selected step size and delay times until zero scale is reached.
45
005C 5128 46 ACALL SO ;SEND OUT INCREASING WORDS
005E 5121 47 ACALL FULLSC ;SET TO FULL SCALE
0060 51A5 48 ACALL DLY500 ;WAIT 500ms
0062 5152 49 ACALL SOD ;SEND OUT DECREASING WORDS
0064 511B 50 ACALL ZERO SC ;RESET TO ZERO SCALE
0066 0130 51 AJMP START ;GO TO BEGINNING OF PROGRAM
006B 52 PROG1:
53 ;
54 ; MANUAL INCREMENT/DECREMENT ROUTINE
55 ;
56 ; This routine increases or decreases the sent out word, depending upon
57 ; the setting of the UP/DOWN switch, by an amount set by the INCREMENT
58 ; SELECT switch. There is a wait of 200ms before again looking for
59 ; depression of the START/COUNT button to allow time to release this
60 ; button and switch bounce to settle. The program then looks to see which
61 ; routine is selected and goes to that routine.
62 ;
0068 30B50B 63 JNB P3.5,DCX ;GO AND COUNT DOWN IF SELECTED
006B 5130 64 ACALL UP ;INCREASE WORD
006D 51B5 65 DP1: ACALL SENDIT ;SEND THE WORD
006F 519D 66 ACALL DLY200 ;WAIT 200ms
0071 013D 67 AJMP W ;WAIT FOR COUNT BUTTON DEPRESS & SELECTED ROUTINE
0073 20B5F2 68 DCX: JB P3.5,PROG1 ;GO AND COUNT UP IF SELECTED
0076 515A 69 ACALL DOWN ;DECREASE WORD

```

Controlling air core meters with the 87C751 and SA5775

AN426

```

0078 80F3    70          SJMP  DP1
007A          71      PROG2:
                72      ;
                73      ;           READ TIME INPUT AND DISPLAY "SPEED"
                74      ;
                75      ;           This routine measures the period of the square wave at the T0 input and
                76      ;           sends out a word that is inversely proportional to 5 times that period,
                77      ;           providing a display proportional to frequency.
                78      ;
007A 1182    79          ACALL  MEAS          ;MEASURE THE INPUT PERIOD
007C 11C5    80          ACALL  CALC          ;CALCULATE THE WORD TO SEND
007E 51B5    81          ACALL  SENDIT       ;SEND OUT THE WORD
0080 0140    82          AJMP   READY
0082 C28C    83      MEAS:  CLR    TR          ;HALT TIMER
0084 C28D    84          CLR    TF          ;CLEAR TIMER FLAG
0086 758B00  85          MOV    RTL,#0        ;SET TIMER REGISTERS
0089 758D00  86          MOV    RTH,#0
008C 758A00  87          MOV    TL,#0          ;SET TIMER
008F 758C00  88          MOV    TH,#0
0092 7B00    89          MOV    R3,#0        ;CLEAR TIMER 3RD BYTE
0094 C3      90          CLR    C
0095 D28C    91          SETB   TR          ;START TIMER
0097 75A882  92          MOV    IE,#82H      ;ENABLE TIMER INTERRUPT
009A 4021    93      W20:   JC     GZS          ;JUMP IF R3 > 0F
009C 2097FB  94          JB     P1.7,W20      ;WAIT FOR T0 INPUT LOW
009F 401C    95      W21:   JC     GZS          ;JUMP IF R3 > 0F
00A1 3097FB  96          JNB   P1.7,W21      ;WAIT FOR T0 INPUT HIGH
00A4 758A00  97          MOV    TL,#0          ;RESET TIMER
00A7 758C00  98          MOV    TH,#0
00AA 7B00    99          MOV    R3,#0
00AC C3      100         CLR    C          ;CLEAR CARRY/BORROW
00AD 4008    101      W22:   JC     HT          ;JUMP IF TIME UP (CARRY SET)
00AF 2097FB  102         JB     P1.7,W22      ;WAIT FOR T0 LOW
00B2 4003    103      W23:   JC     HT          ;JUMP IF TIME UP (CARRY SET)
00B4 3097FB  104         JNB   P1.7,W23      ;WAIT FOR T0 HIGH AGAIN
00B7 C28C    105      HT:    CLR    TR          ;HALT TIMER
00B9 75A800  106         MOV    IE,#0          ;DISABLE ALL INTERRUPTS
00BC 22      107         RET
00BD 7B1F    108      GZS:   MOV    R3,#1FH      ;SET FOR ZERO SCALE
00BF 22      109         RET
00C0 7F03    110      GFS:   MOV    R7,#03
00C2 7EE8    111         MOV    R6,#0E8H
00C4 22      112         RET
00C5          113      CALC:
                114      ;
                115      ;           This subroutine calculates the 10-bit word to send as a function fo what
                116      ;           is in R3, TH & TL. The 10-bit word is developed and left in registers
                117      ;           R7 and R6 for use by SENDIT subroutine.
                118      ;
00C5 7F00    119         MOV    R7,#0          ;INITIALIZE QUOTIENT
00C7 7E00    120         MOV    R6,#0
00C9 C3      121         CLR    C          ;CLEAR CARRY/BORROW
00CA 740F    122         MOV    A,#0FH      ;CHECK FOR ZERO SCALE
00CC 9B      123         SUBB   A,R3
00CD 5001    124         JNC   NZS          ;JUMP IF NOT ZERO SCALE
00CF 22      125         RET
00D0 E58A    126      NZS:   MOV    A,TL          ;CHECK FOR FULL SCALE
00D2 9488    127         SUBB   A,#88H
00D4 E58C    128         MOV    A,TH
00D6 9413    129         SUBB   A,#13H
00D8 EB      130         MOV    A,R3
00D9 9400    131         SUBB   A,#0
00DB 40E3    132         JC     GFS
00DD 752E4C  133         MOV    2EH,#4CH      ;SET DIVIDEND TO 5,000,000
00E0 752F4B  134         MOV    2FH,#4BH
00E3 753040  135         MOV    30H,#40H
00E6 7C00    136         MOV    R4,#0          ;CLEAR DIVIDE COUNTER
00E8 8B2B    137         MOV    2BH,R3        ;MOVE READING TO MEMORY (DIVISOR)
00EA 858C2C  138         MOV    2CH,TH

```

Controlling air core meters with the 87C751 and SA5775

AN426

```

00ED 858A2D 139      MOV    2DH,TL
00F0 C3 140      ROTL: CLR    C                ;BRING DIVISOR BE JUST LESS THAN DIVIDEND
00F1 E52E 141      MOV    A,2EH
00F3 952B 142      SUBB  A,2BH
00F5 4014 143      JC     DIV24                ;JUMP IF SHIFTING WOULD MAKE DIVISOR > DIVIDEND
00F7 6012 144      JZ     DIV24                ;JUMP IF DIVISOR & DIVIDEND MS BYTES EQUAL BEFORE SHIFT
00F9 E52D 145      MOV    A,2DH                ;SHIFT DIVISOR TO LEFT
00FB 33 146      RLC   A
00FC F52D 147      MOV    2DH,A
00FE E52C 148      MOV    A,2CH
0100 33 149      RLC   A
0101 F52C 150      MOV    2CH,A
0103 E52B 151      MOV    A,2BH
0105 33 152      RLC   A
0106 F52B 153      MOV    2BH,A
0108 0C 154      INC   R4
0109 80E5 155      SJMP  ROTL
010B C3 156      DIV24: CLR   C
010C EE 157      MOV    A,R6                ;ROTATE QUOTIENT LEFT
010D 33 158      RLC   A
010E FE 159      MOV    R6,A
010F EF 160      MOV    A,R7
0110 33 161      RLC   A
0111 FF 162      MOV    R7,A
0112 C3 163      CLR   C                ;ROTATE DIVIDEND LEFT
0113 E530 164      MOV    A,30H
0115 33 165      RLC   A
0116 F530 166      MOV    30H,A
0118 E52F 167      MOV    A,2FH
011A 33 168      RLC   A
011B F52F 169      MOV    2FH,A
011D E52E 170      MOV    A,2EH
011F 33 171      RLC   A
0120 F52E 172      MOV    2EH,A
0122 C3 173      CLR   C                ;TEST SUBTRACT MOST SIGNIFICANT BYTES
0123 952B 174      SUBB  A,2BH
0125 401B 175      JC     ZERO                ;JUMP IF QUOTIENT MS BYTE < DIVISOR MS BYTE
0127 7401 176      MOV    A,#1                ;ADD 1 TO QUOTIENT
0129 2E 177      ADD   A,R6
012A FE 178      MOV    R6,A
012B EF 179      MOV    A,R7
012C 3400 180      ADDC  A,#0
012E FF 181      MOV    R7,A
012F C3 182      CLR   C                ;SUBTRACT DIVISOR FROM DIVIDEND
0130 E530 183      MOV    A,30H
0132 952D 184      SUBB  A,2DH
0134 F530 185      MOV    30H,A
0136 E52F 186      MOV    A,2FH
0138 952C 187      SUBB  A,2CH
013A F52F 188      MOV    2FH,A
013C E52E 189      MOV    A,2EH
013E 952B 190      SUBB  A,2BH
0140 F52E 191      MOV    2EH,A
0142 DCC7 192      ZERO: DJNZ  R4,DIV24
0144 22 193      RET
0145 194      PROG3:
195      ;
196      ;          DISPLAY AVERAGE OF FOUR NEW READINGS
197      ;
198      ; This routine reads the period of the T0 input four times, then displays the
199      ; "speed" corresponding to the average of these four readings.
200      ;
0145 7903 201      MOV    R1,#3                ;SET FOR 3 READINGS
0147 7820 202      MOV    R0,#20H            ;SET INDEX REGISTER FOR BOTTOM
0149 1182 203      P30: ACALL MEAS            ;TAKE 3 READINGS AND SAVE THEM
014B EB 204      MOV    A,R3
014C F6 205      MOV    @R0,A
014D 08 206      INC   R0
014E A68C 207      MOV    @R0,TH

```

Controlling air core meters with the 87C751 and SA5775

AN426

```

0150 08      208      INC      R0
0151 A68A    209      MOV      @R0,TL
0153 08      210      INC      R0
0154 D9F3    211      DJNZ    R1,P30
0156 1182    212      ACALL   MEAS      ;TAKE A 4TH READING, LEAVING IN R3,TH,TL
0158 7828    213      MOV      R0,#28H  ;SET INDEX REGISTER FOR TOP
015A 7903    214      MOV      R1,#3    ;SET COUNTER TO ADD FIRST 3 READINGS TO LAST ONE
015C E58A    215      P31:    MOV      A,TL     ;ADD FIRST THREE READINGS TO THE LAST ONE
015E 26      216      ADD     A,@R0
015F F58A    217      MOV      TL,A
0161 18      218      DEC     R0
0162 E58C    219      MOV      A,TH
0164 36      220      ADDC   A,@R0
0165 F58C    221      MOV      TH,A
0167 18      222      DEC     R0
0168 EB      223      MOV      A,R3
0169 36      224      ADDC   A,@R0
016A FB      225      MOV      R3,A
016B 18      226      DEC     R0
016C D9EE    227      DJNZ    R1,P31
016E 7902    228      MOV      R1,#2
0170 EB      229      P32:    MOV      A,R3     ;DIVIDE BY 4 (ROTATE RIGHT TWICE) FOR AVERAGE
0171 C3      230      CLR     C
0172 13      231      RRC    A
0173 FB      232      MOV      R3,A
0174 E58C    233      MOV      A,TH
0176 13      234      RRC    A
0177 F58C    235      MOV      TH,A
0179 E58A    236      MOV      A,TL
017B 13      237      RRC    A
017C F58A    238      MOV      TL,A
017E D9F0    239      DJNZ    R1,P32
0180 11C5    240      ACALL   CALC     ;CALCULATE THE WORD
0182 51B5    241      ACALL   SENDIT   ;SEND OUT THE WORD
0184 0140    242      AJMP   READY    ;GO TO SELECTED ROUTINE
0186         243      PROG4:
0186         244      ;
0186         245      ;      DISPLAY AVERAGE OF LAST FOUR WORDS SENT OUT
0186         246      ;
0186         247      ;      This routine sends out the average of the last four readings sent out.
0186         248      ;
0186 7827    249      MOV      R0,#27H
0188 7600    250      P4:    MOV      @R0,#0
018A 18      251      DEC     R0
018B B81FFA  252      CJNE   R0,#1FH,P4
018E 7820    253      P4A:   MOV      R0,#20H
0190 1182    254      P40:   ACALL   MEAS   ;MEASURE PERIOD
0192 11C5    255      ACALL   CALC     ;CALCULATE THE CODE
0194 EF      256      MOV      A,R7    ;SAVE THE CODE
0195 F6      257      MOV      @R0,A
0196 08      258      INC     R0
0197 EE      259      MOV      A,R6
0198 F6      260      MOV      @R0,A
0199 752800  261      MOV      28H,#0  ;INITIALIZE THE WORD TO SEND
019C 752900  262      MOV      29H,#0
019F 7927    263      MOV      R1,#27H
01A1 E529    264      P41:   MOV      A,29H   ;ADD TOGETHER LAST 4 RESULTS
01A3 C3      265      CLR     C
01A4 27      266      ADD     A,@R1
01A5 F529    267      MOV      29H,A
01A7 E528    268      MOV      A,28H
01A9 19      269      DEC     R1
01AA 37      270      ADDC   A,@R1
01AB F528    271      MOV      28H,A
01AD 19      272      DEC     R1
01AE B91FF0  273      CJNE   R1,#1FH,P41
01B1 7902    274      MOV      R1,#2
01B3 C3      275      P42:   CLR     C
01B4 E528    276      MOV      A,28H

```

Controlling air core meters with the 87C751 and SA5775

AN426

```

01B6 13      277      RRC      A
01B7 F528    278      MOV      28H,A
01B9 E529    279      MOV      A,29H
01BB 13      280      RRC      A
01BC F529    281      MOV      29H,A
01BE D9F3    282      DJNZ    R1,P42
01C0 AF28    283      MOV      R7,28H
01C2 AE29    284      MOV      R6,29H
01C4 51B5    285      ACALL   SENDIT      ;SEND OUT THE WORD
01C6 51D2    286      ACALL   RPS         ;READ PROGRAM SELECT
01C8 B40806  287      CJNE    A,#8,N4     ;JUMP TO N4 (& "READY") IF PROGRAM 4 NOT SELECTED
01CB 08      288      INC      R0
01CC B828C1  289      CJNE    R0,#28H,P40 ;GOTO P40 IF R0 NOT 28 (HEX)
01CF 80BD    290      SJMP   P4A
01D1 0140    291      N4:     AJMP   READY
        292      ;
        293      PROG5:
        294      ;
        295      ; This routine advances the display in 45 degree steps to full scale, then steps down
        296      ; to zero in 45 degree steps. There is a 500ms delay between steps.
        297      ;
01D3 7F00    298      MOV      R7,#0
01D5 7E7F    299      P5:     MOV      R6,#07FH
01D7 51B1    300      ACALL   SD500      ;SEND THE WORD AND WAIT 500ms
01D9 7EFF    301      MOV      R6,#0FFH
01DB 51B1    302      ACALL   SD500      ;SEND THE WORD AND WAIT 500ms
01DD 0F      303      INC      R7
01DE BF04F4  304      CJNE    R7,#4,P5
01E1 7F03    305      MOV      R7,#3
01E3 7EFF    306      LP5:    MOV      R6,#0FFH
01E5 51B1    307      ACALL   SD500      ;SEND THE WORD AND WAIT 500ms
01E7 7E7F    308      MOV      R6,#7FH
01E9 51B1    309      ACALL   SD500
01EB 1F      310      DEC      R7
01EC BFFFF4  311      CJNE    R7,#0FFH,LP5
01EF 511B    312      ACALL   ZEROSC     ;RETURN TO ZERO
01F1 013D    313      AJMP   W           ;WAIT FOR KEY PRESS
01F3        314      PROG6:
        315      ;
        316      ; This routine advances the display in 90 degree steps to full scale, then steps down
        317      ; to zero in 90 degree steps. There is a 500ms delay between steps.
        318      ;
01F3 7EFF    319      MOV      R6,#0FFH
01F5 7F00    320      MOV      R7,#0
01F7 51B1    321      LP6:    ACALL   SD500      ;SEND THE WORD AND WAIT 500ms
01F9 0F      322      INC      R7
01FA BF04FA  323      CJNE    R7,#4,LP6
01FD 1F      324      LP6A:  DEC      R7
01FE 51B1    325      ACALL   SD500      ;SEND THE WORD AND WAIT 500ms
0200 BF00FA  326      CJNE    R7,#0,LP6A
0203 511B    327      ACALL   ZEROSC     ;RETURN TO ZERO
0205 013D    328      AJMP   W           ;WAIT FOR KEY PRESS
0207        329      PROG7:
        330      ;
        331      ; This routine alternates between 3/8 and 5/8 scale ten times with 300ms delay
        332      ; between steps, then waits 500ms before returning display to zero scale.
        333      ;
0207 7A0A    334      MOV      R2,#10     ;SET COUNTER
0209 7E7F    335      PR7:   MOV      R6,#07FH
020B 7F01    336      MOV      R7,#1
020D 51AD    337      ACALL   SD300      ;SEND OUT THE WORD AND WAIT 300ms
020F 7F02    338      MOV      R7,#2
0211 51AD    339      ACALL   SD300      ;SEND OUT THE WORD AND WAIT 300ms
0213 DAF4    340      DJNZ    R2,PR7     ;DO IT 10 TIMES
0215 51A5    341      ACALL   DLY500     ;WAIT 500ms
0217 511B    342      ACALL   ZEROSC     ;RESET TO ZERO SCALE
0219 0130    343      AJMP   START      ;LOOK FOR VALID PROGRAM
        344      ;
        345      ;

```

Controlling air core meters with the 87C751 and SA5775

AN426

```

346 ;           SUBROUTINES
347 ;
348 ;
021B 7F00 349 ZEROSC: MOV   R7,#0           ;RESET METER TO ZERO SCALE
021D 7E00 350       MOV   R6,#0
021F 4125 351       AJMP  RST
0221 7F03 352 FULLSC: MOV   R7,#03H        ;SET METER TO FULL SCALE
0223 7EFF 353       MOV   R6,#0FFH
0225 51B5 354 RST:  ACALL SENDIT
0227 22    355
0228      356 SO:
357 ;
358 ; This subroutine sends increasing 10-bit words in registers R7 & R6 to the ACMD.
359 ;
0228 51B5 360       ACALL SENDIT           ;WRITE THE 10-BIT WORD TO ACMD
022A 5130 361       ACALL UP           ;INCREASE THE WORD VALUE
022C 30E2F9 362      JNB  ACC.2,SO           ;JUMP IF BIT 2 NOT SET
022F 22    363       RET
0230      364 UP:
365 ;
366 ; This subroutine waits for a period of time = 10ms X DELAY read un, then
367 ; increases the 10-bit word by the INCREMENT SELECT amount.
368 ;
0230 E590 369       MOV   A,P1           ;READ DELEY
0232 F4    370       CPL   A           ;COMPLEMENT ACC
0233 540F 371       ANL  A,#0FH        ;MASK OFF UPPER 4 BITS
0235 23    372       RL   A
0236 23    373       RL   A
0237 F9    374       MOV   R1,A
0238 B90002 375      CJNE  R1,#0,D10      ;JUMP IF DELAY SET FOR ZERO
023B 8006 376      SJMP  NODLY
023D 7B01 377 D10:  MOV   R3,#1           ;SET FOR 1 X 10ms DELAY
023F 5195 378 D10A: ACALL DLY10MS          ;DELAY 10MS x DELAY
0241 D9FC 379      DJNZ  R1,D10A
0243 E5B0 380 NODLY: MOV   A,P3           ;READ INCREMENT SELECT
0245 F4    381       CPL   A           ;COMPLEMENT ACC
0246 5403 382       ANL  A,#3           ;MASK OFF UPPER 6 BITS
0248 23    383       RL   A
0249 23    384       RL   A
024A 23    385       RL   A
024B 04    386       INC  A
024C 2E    387       ADD  A,R6          ;ADD INCREMENT TO R6
024D FE    388       MOV   R6,A          ;SAVE IT
024E E4    389       CLR  A
024F 3F    390       ADDC A,R7           ;ADD CARRY TO R7
0250 FF    391       MOV   R7,A          ;SAVE IT
0251 22    392       RET
0252      393 SOD:
394 ;
395 ; This subroutine sends out decreasing words at the rate set by DELAY and
396 ; step size determined by INCREMENT SELECT.
397 ;
0252 51B5 398       ACALL SENDIT           ;SEND OUT THE PRESENT WORD
0254 515A 399       ACALL DOWN          ;DECREASE THE WORD
0256 50FA 400      JNC  SOD           ;DO IT AGAIN IF CARRY NOT SET
0258 411B 401      AJMP  ZEROSC
025A      402 DOWN:
403 ;
404 ; Waits for 10ms x DELAY pot setting, then sends out decreasing values of words
405 ; in step sizes of 8 x INCREMENT SELECT + 1.
406 ;
025A E590 407       MOV   A,P1           ;READ DELAY
025C F4    408       CPL   A           ;COMPLEMENT ACC
025D 540F 409       ANL  A,#0FH        ;MASK OFF UPPER FOUR BITS
025F 23    410       RL   A
0260 23    411       RL   A
0261 F9    412       MOV   R1,A          ;SAVE DELAY
0262 B90002 413      CJNE  R1,#0,D10S      ;JUMP IF DELAY SET FOR ZERO
0265 8004 414      SJMP  NDD

```


Controlling air core meters with the 87C751 and SA5775

AN426

```

0267 5195 415 D10S: ACALL DLY10MS ;DELAY 10ms x (DELAY +1)
0269 D9FC 416 DJNZ R1,D10S
026B E5B0 417 NDD: MOV A,P3 ;READ INCREMENT SELECT
026D F4 418 CPL A ;COMPLEMENT ACC
026E 5403 419 ANL A,#3 ;MASK OFF UPPER 6 BITS
0270 23 420 RL A ;MULTIPLY BY 8
0271 23 421 RL A
0272 23 422 RL A
0273 04 423 INC A ;INSURE MINIMUM STEP
0274 C3 424 CLR C ;CLEAR CARRY FOR SUBTRACTION
0275 CE 425 XCH A,R6
0276 9E 426 SUBB A,R6 ;SUBTRACT INCREMENT FROM R6
0277 CE 427 XCH A,R6 ;SAVE IT
0278 E4 428 CLR A ;CLEAR ACCUM FOR SUBTRACTION
0279 CF 429 XCH A,R7
027A 9F 430 SUBB A,R7 ;SUBTRACT BORROW FROM R7
027B 5403 431 ANL A,#3 ;INSURE MAXIMUM WORD
027D CF 432 XCH A,R7 ;SAVE IT
027E 22 433 RET
027F 00 434 DELAY: NOP ;30s DELAY
0280 22 435 RET
0281 436 DMS10:
437 ;
438 ; Produces a delay of 10ms x the value in R3.
439 ; Destroys R3 and timer readings.
440 ;
441 ;
0281 758AF0 442 MOV TL,#LOW,(0-10000) ;LOAD TIMER FOR 10ms DELAY
0284 758CD8 443 MOV TH,#HIGH(0-10000)
0287 C28D 444 CLR TF ;CLEAR TIMER FLAG
0289 D28C 445 SETB TR ;START TIMER
028B 308DFD 446 MS10W: JNB TF,MS10W ;WAIT FOR TIMER FLAG TO BE SET
028E C28D 447 CLR TF ;CLEAR TIMER FLAG
0290 DBF9 448 DJNZ R3,MS10W ;WAIT RS x 10ms
0292 C28C 449 CLR TR ;STOP TIMER
0294 22 450 RET
451 ;
0295 7B01 452 DLY10MS: MOV R3,#1 ;SET R3 FOR 10ms WAIT
0297 80EB 453 SJMP DMS10 ;WAIT 10ms
454 ;
0299 7B0A 455 DLY100: MOV R3,#10 ;SET R3 FOR 100ms WAIT
029B 80E4 456 SJMP DMS10 ;WAIT 100ms
457 ;
029D 7B14 458 DLY200: MOV R3,#20 ;SET R3 FOR 200ms WAIT
029F 80E0 459 SJMP DMS10 ;WAIT 200ms
460 ;
02A1 7B1E 461 DLY300: MOV R3,#30 ;SET R3 FOR 300ms WAIT
02A3 80DC 462 SJMP DMS10 ;WAIT 300ms
463 ;
02A5 7B32 464 DLY500: MOV R3,#50 ;SET R3 FOR 500ms WAIT
02A7 80D8 465 SJMP DMS10 ;WAIT 500ms
466 ;
02A9 51B5 467 SD200: ACALL SENDIT ;SEND THE WORD
02AB 80F0 468 SJMP DLY200 ;WAIT 200ms
469 ;
02AD 51B5 470 SD300: ACALL SENDIT ;SEND THE WORD
02AF 80F0 471 SJMP DLY300 ;WAIT 200ms
472 ;
02B1 51B5 473 SD500: ACALL SENDIT ;SEND THE WORD
02B3 80F0 474 SJMP DLY500 ;WAIT 500ms
475 ;
02B5 476 SENDIT:
477 ;
478 ; This subroutine sends out a single word locate4d in R7 and R6.
479 ; Accumulator, R0 and R1 are destroyed.
480 ;
02B5 D282 481 SETB P0.2 ;SET CS HIGH
02B7 7902 482 MOV R1,#02 ;SET COUNTER FOR 2 BITS OF R7
02B9 EF 483 MOV A,R7 ;MOVE R7 TO A FOR SEND OUT

```

Controlling air core meters with the 87C751 and SA5775

AN426

```

02BA 13      484      RRC      A              ;ALIGN R7 FOR SEND OUT
02BB 13      485      RRC      A
02BC 13      486      RRC      A
02BD 51C7    487      ACALL    SEND1      ;SEND OUT UPPER TWO BITS
02BF 7908    488      MOV      R1,#8      ;SET COUNTER FOR R6 SEND OUT
02C1 EE      489      MOV      A,R6      ;MOVE R6 TO ACCUM
02C2 51C7    490      ACALL    SEND1      ;SEND OUT LOWER 8 BITS
02C4 C282    491      CLR      P0.2      ;LOAD ACMD
02C6 22      492      RET
02C7         493      SEND1:
                    494      ;
                    495      ; This subroutine sends [R1] number of bits of the accumulator, starting
                    496      ; with the MSB over the IIC port.
                    497      ; Accumulator, R0 and R1 are destroyed.
                    498      ;
02C7 33      499      RLC      A              ;ROTATE BIT TO CARRY
02C8 9281    500      MOV      P0.1,C      ;MOVE CARRY TO DATA OUT
02CA C280    501      CLR      P0.0      ;CLOCK LOW
02CC 00      502      NOP
02CD D280    503      SETB    P0.0      ;CLOCK HIGH
02CF D9F6    504      DJNZ   R1,SEND1    ;SEND NEXT BIT TILL DONE
02D1 22      505      RET
                    506      ;
02D2 E5B0    507      RPS:   MOV      A,P3      ;READ PORT 3 FOR PROGRAM SELECT
02D4 F4      508      CPL      A              ;COMPLEMENT ACC
02D5 03      509      RR      A              ;ROTATE TO LSB's & MULT BY 2
02D6 540E    510      ANL      A,#0EH      ;MASK FOR PROGRAM SELECT * 2
02D8 DD      511      RET
                    512      END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

Controlling air core meters with the 87C751 and SA5775

AN426

ACC	D	ADDR	00E0H	PREDEFINED
CALC	C	ADDR	00C5H	
D10	C	ADDR	023DH	
D10A	C	ADDR	023FH	
D10S	C	ADDR	0267H	
DCX	C	ADDR	0073H	
DELAY	C	ADDR	027FH	NOT USED
DIV24	C	ADDR	010BH	
DLY100	C	ADDR	0299H	NOT USED
DLY10MS	C	ADDR	0295H	
DLY200	C	ADDR	029DH	
DLY300	C	ADDR	02A1H	
DLY500	C	ADDR	02A5H	
DMS10	C	ADDR	0281H	
DOWN	C	ADDR	025AH	
DP1	C	ADDR	006DH	
FULLSC	C	ADDR	0221H	
GFS	C	ADDR	00C0H	
GZS	C	ADDR	00BDH	
HT	C	ADDR	00B7H	
IE	D	ADDR	00A8H	PREDEFINED
JMPTBL	C	ADDR	004CH	
LP5	C	ADDR	01E3H	
LP6	C	ADDR	01F7H	
LP6A	C	ADDR	01FDH	
MEAS	C	ADDR	0082H	
MS10W	C	ADDR	028BH	
N4	C	ADDR	01D1H	
NDD	C	ADDR	026BH	
NODLY	C	ADDR	0243H	
NZS	C	ADDR	00D0H	
P0	D	ADDR	0080H	PREDEFINED
P1	D	ADDR	0090H	PREDEFINED
P3	D	ADDR	00B0H	PREDEFINED
P30	C	ADDR	0149H	
P31	C	ADDR	015CH	
P32	C	ADDR	0170H	
P4	C	ADDR	0188H	
P40	C	ADDR	0190H	
P41	C	ADDR	01A1H	
P42	C	ADDR	01B3H	
P4A	C	ADDR	018EH	
P5	C	ADDR	01D5H	
PR7	C	ADDR	0209H	
PROG0	C	ADDR	005CH	
PROG1	C	ADDR	0068H	
PROG2	C	ADDR	007AH	
PROG3	C	ADDR	0145H	
PROG4	C	ADDR	0186H	
PROG5	C	ADDR	01D3H	
PROG6	C	ADDR	01F3H	
PROG7	C	ADDR	0207H	
READY	C	ADDR	0040H	
ROTL	C	ADDR	00F0H	
RPS	C	ADDR	02D2H	
RST	C	ADDR	0225H	
RTH	D	ADDR	008DH	PREDEFINED
RTL	D	ADDR	008BH	PREDEFINED
SD200	C	ADDR	02A9H	NOT USED
SD300	C	ADDR	02ADH	
SD500	C	ADDR	02B1H	
SEND1	C	ADDR	02C7H	
SENDIT	C	ADDR	02B5H	
SO	C	ADDR	0228H	
SOD	C	ADDR	0252H	
START	C	ADDR	0030H	
TF	B	ADDR	008DH	PREDEFINED
TH	D	ADDR	008CH	PREDEFINED
TL	D	ADDR	008AH	PREDEFINED
TR	B	ADDR	008CH	PREDEFINED
UP	C	ADDR	0230H	
W	C	ADDR	003DH	
W20	C	ADDR	009AH	
W21	C	ADDR	009FH	
W22	C	ADDR	00ADH	
W23	C	ADDR	00B2H	
ZERO	C	ADDR	0142H	
ZEROSC	C	ADDR	021BH	

Timer 1 for the 83/87C748/749 and the 83/87C751/752 (non-I²C applications) microcontrollers

AN427

The small package 83/87C748, 83/87C749, 83/87C751 and 83/87C752 microcontrollers include two hardware-implemented timers: a 16-bit programmable timer, and a 10-bit fixed-rate timer. The programmable timer is available for the application program, and its operation is similar to the timer/counter of the 80C51 timer in mode 2. The fixed-rate timer, Timer 1, is typically employed as a watchdog timer for the I²C port communications and is not available for other uses.

In applications which do not take advantage of the I²C communications capability, the "silicon real estate" taken by Timer 1 is not necessarily lost—it can be used as a fixed-rate timer by the application. This timer can become useful in various cases, such as simple control applications that need a delay while doing some software activities in parallel, or generating a free-running repetitive waveform where the exact timing is not important. Another type of application is a watchdog timer prompting the user about unexpected operation of a system or its hardware, or resetting a program that "lost track."

TIMER 1 IMPLEMENTATION

Timer 1 is clocked once per machine cycle, which is the oscillator frequency divided by 12. The timer operation is enabled by setting the TIRUN bit (bit 4) in the I2CFG register. Writing a 0 into the TIRUN bit will stop and clear the timer. The timer is 10 bits wide, and when it reaches the terminal count of 1024 it carries out and sets the Timer 1 interrupt flag. An interrupt will occur if the Timer 1 interrupt is enabled by bit ETI (bit 4) of the Interrupt Enable (IE) register, and global interrupts are enabled by bit EA (bit 7) of the same IE register.

The vector address for the Timer 1 interrupt is 1B hex, and the interrupt service routine must start at this address. As with all 8051 family microcontrollers, only the Program Counter is pushed onto the stack upon interrupt (other registers that are used both by the interrupt

service routine and elsewhere must be explicitly saved). The Timer 1 interrupt flag is cleared by setting the CLRTI bit (bit 5) of the I2CFG register.

Note that when the I²C interface is not operating—SLAVEN, MASTRQ, and MASTER bits are all 0—the I²C hardware does not affect Timer 1. The SCL and SDA pins can be used as I/O pins, and the activity of these pins will not cause the timer to run, stop, or reset. Upon hardware reset of the microcontroller, the SLAVEN, MASTRQ, and MASTER bits are all reset, so the programmer does not have to worry about interaction between the SDA/SCL pins and the timer.

FIXED-RATE TIMER

The first programming example demonstrates simple fixed-rate operation. Upon reset, interrupts are enabled, and Timer 1 is started. A wait loop simulates the "application" program. The demonstration service routine simply sets a flag—in real life it could do something more useful, such as toggling an output pin. Note that the interrupt flag is cleared by setting CLRTI prior to returning from the service routine. Upon overflow, the timer will go on running, as the TIRUN bit is still set, so the interrupts will be spaced exactly 1024 clock cycles apart. If the service routine would toggle an output pin instead of setting a flag, its output would be a square wave with a period of 2048 cycles. For an application that demands a "one-shot" delay only, the service routine should clear the TIRUN bit in order to avoid subsequent interrupts.

WATCHDOG TIMER

A watchdog timer mechanism is typically applied in order to detect "abnormal" behavior of hardware. If the microcontroller operates in a very noisy environment, there might be a fear of the program "running wild" as a result of extremely violent EMI interference. In such

a case, a watchdog may take care to reset the microcontroller when the Timer 1 interrupt occurs. This could be applied in application programs with a repetitive nature—the software needs to reset the timer within 1024 machine cycles of the last reset.

In a system where something is supposed to occur regularly—for example, an interrupt for an external event—the watchdog is designed to "bite" when the hardware "sleeps" and the expected "something" does not happen for too long a time. The timer is allowed to run continuously, but when the expected event occurs, it resets the timer back to 0. When the timer is reset within 1024 cycles of the last reset, the application runs normally. If the event does not occur, the Timer 1 interrupt service routine will be activated to take care of the exception.

The second programming example demonstrates the watchdog. Upon Reset, the TIRUN bit, ETI, and global interrupts are enabled. The watchdog timer is reset and restarted by the small subroutine WdRst. The application is simulated by a loop of delays. Delay 1 is less than 1024 cycles, and when WdRst is called within Delay 1 intervals, no Timer 1 interrupt occurs. This represents normal operation of a "real life" application. When the delay from last reset is greater than 1024 cycles—representing a hardware exception—the interrupt will occur. The service routine for the watchdog is somewhat unusual, as it does not return to the program location where the interrupt occurred. Instead, the operation of the microcontroller is restarted at Reset. Upon entering the service routine, the interrupt is cleared and the timer is reset. Because execution does not return to the interrupted program with a RETI instruction, the interrupt pending flag is cleared by a call to a dummy subroutine XRETI. The program is restarted at Reset with a regular AJMP instruction. The stack pointer is explicitly reinitialized for the warm reset, so there is no danger of stack overflow upon repeated watchdog invocations.

Timer I for the 83/87C748/749 and the 83/87C751/752 (non-I²C applications) microcontrollers

AN427

TIINT

Timer I Fixed Rate Timer

11/06/90 PAGE 1

```

1 ;
2 *****
3 ;
4           Timer I Fixed Rate Timer Usage
5 ;This program demonstrates how to activate Timer I on the 8XC748/8XC749/83C751
6 ;or 83C752 microcontrollers as a fixed rate timer when the I2C port is not
7 ;used. Once activated, Timer I will generate an interrupt every 1024
8 ;machine cycles. The I2C bus pins SCL and SDA may be used as open drain
9 ;outputs.
10
11 ;
12 *****
13 $MOD7 751
14 $Title(Timer I Fixed Rate Timer)
15 $Date(11/06/90)
16 $Debug
17
18
19 0020  Flags      DATA  20h          ;Flag byte
20 0000  TstFlag   BIT    Flags.0     ;Timer I flag.
21
22
23 0000          ORG    0
24 0000 0120    AJMP   Reset
25
26 001B          ORG    1Bh           ;Timer I interrupt.
27 001B D200    TimerI: SETB   TstFlag ;Set flag to indicate a Timer I interrupt.
28 001D D2DD    SETB   CLRTI        ;Clear Timer I to allow it to restart.
29 001F 32      RETI
30
31
32 0020 D2AB    Reset:  SETB   ETI      ;Enable Timer I interrupt.
33 0022 D2AF    SETB   EA          ;Enable global interrupts.
34 0024 D2DC    SETB   TIRUN       ;Start Timer I.
35
36 0026 C200    Loop:   CLR    TstFlag ;Initialize interrupt flag.
37 0028 3000FD Wait:   JNB   TstFlag,Wait ;Wait for Timer I interrupt.
38 002B 80F9    SJMP   Loop
39
40          END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

**Timer I for the 83/87C748/749 and the 83/87C751/752
(non-I²C applications) microcontrollers**

AN427

TIINT

Timer I Fixed Rate Timer

11/06/90 PAGE 2

CLRTI	B ADDR	00DDH	PREDEFINED
EA	B ADDR	00AFH	PREDEFINED
ETI	B ADDR	00ABH	PREDEFINED
FLAGS	D ADDR	0020H	
LOOP	C ADDR	0026H	
RESET	C ADDR	0020H	
TIMERI	C ADDR	001BH	NOT USED
TIRUN	B ADDR	00DCH	PREDEFINED
TSTFLAG	B ADDR	0000H	
WAIT	C ADDR	0028H	

Timer I for the 83/87C748/749 and the 83/87C751/752 (non-I²C applications) microcontrollers

AN427

TIWD

Timer I Watchdog

11-06-90 PAGE 1

```

1 ;*****
2
3 ;               Timer I Watchdog Timer Usage
4
5 ;This program demonstrates how to use Timer I on the 83C751 or 83C752
6 ;microcontrollers as a watchdog timer when the I2C port is not used.
7 ;Once started, Timer I must be cleared more often than once every 1024
8 ;machine cycles. If Timer I is allowed to overflow, a Timer I
9 ;interrupt will be generated. Thus, if global interrupts or the Timer
10 ;I interrupt are inhibited, the watchdog function will be disabled.
11 ;Also, if the watchdog interrupt occurs during another interrupt
12 ;service, it will be delayed until an RETI (return from interrupt)
13 ;instruction is executed. The I2C bus pins SCL and SDA may be used as
14 ;open drain outputs.
15
16 ;*****
17
18 $MOD751
19 $Title(Timer I Watchdog)
20 $Date(11-06-90)
21 $Debug
22
0000 0000 23          ORG      0
0000 0126 24          AJMP    Reset
25
001B 001B 26          ORG      1Bh      ;Timer I interrupt.
001B C2AF 27  TimerI:  CLR      EA          ;Get here only if watchdog overflows.
001D C2DC 28          CLR      TIRUN     ;Turn off Timer I.
001F D2DD 29          SETB   CLRTI     ;Clear Timer I interrupt.
0021 1125 30          ACALL  XRETI     ;Force interrupt pending to clear.
0023 0126 31          AJMP    Reset     ;Do a warm start.
0025 32 32  XRETI:  RETI
33
0026 758107 34  Reset:  MOV     SP,#7h      ;Initialize the stack pointer.
35
36 ;Note: it is important to force the stack pointer to a particular
37 ;starting value in this application because we may be re-starting
38 ;after a watchdog interrupt, with the stack in an unknown condition.
39
0029 75D800 40  MOV     I2CFG,#0          ;Initialize I2CFG (set up CT0, CT1).
002C D2DC 41  SETB   TIRUN          ;Enable Timer I run.
002E D2AB 42  SETB   ETI           ;Enable Timer I interrupt.
0030 D2AF 43  SETB   EA            ;Enable interrupt system.
44
45
46 ;The following is a "dummy" main program to test the watchdog timer.
47
0032 1153 48  Loop:  ACALL  Delay1     ;Wait 901 machine cycles.
0034 114E 49          ACALL  WdRst    ;Reset Watchdog.
0036 1153 50          ACALL  Delay1     ;Wait 901 machine cycles.
0038 114E 51          ACALL  WdRst    ;Reset Watchdog.
003A 1153 52          ACALL  Delay1     ;Wait 901 + 4 for ACALL & prior RET.
003C 1157 53          ACALL  Delay2     ;Wait 108 + 2 for ACALL.
003E 00 54          NOP          ;1016
003F 00 55          NOP          ;1017
0040 00 56          NOP          ;1018
0041 00 57          NOP          ;1019
0042 00 58          NOP          ;1020

```

Timer I for the 83/87C748/749 and the 83/87C751/752 (non-I²C applications) microcontrollers

AN427

```

TIWD                                     Timer I Watchdog                                     11-06-90  PAGE 2
0043 00          59          NOP          ;1021
0044 00          60          NOP          ;1022
0045 00          61          NOP          ;1023
0046 00          62          NOP          ;1024          : Should get 'bitten' here.
0047 00          63          NOP          ;1025
0048 00          64          NOP          ;1026
0049 00          65          NOP          ;1027
004A 00          66          NOP          ;1028
004B 00          67          NOP          ;1029
004C 0132        68          AJMP         Loop          ;Should never get here.
          69
004E C2DC        70          WdRst:   CLR          TIRUN          ;Reset Watchdog timer (Timer I).
0050 D2DC        71          SETB         TIRUN
0052 22          72          RET
          73
0053 7480        74          Delay1:  MOV          A,#128          ;Wait 901 machine cycles (1).
0055 8002        75          SJMP         DLoop          ;(2)
0057 740F        76          Delay2:  MOV          A,#15          ;Wait 108 machine cycles (1).
0059 A3          77          DLoop:   INC          DPTR          ;Delay = (ACC * 7) + 2 mach. cyc (2).
005A A3          78          INC          DPTR          ;(2)
005B 14          79          DEC          A          ;(1)
005C 70FB        80          JNZ         Dloop          ;(2)
005E 22          81          RET          ;(2)
          82
          83          END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

Timer I for the 83/87C748/749 and the 83/87C751/752
(non-I²C applications) microcontrollers

AN427

TIWD

Timer I Watchdog

11-06-90 PAGE 3

CLRTI	B ADDR	00DDH	PREDEFINED
DELAY1	C ADDR	0053H	
DELAY2	C ADDR	0057H	
DLOOP	C ADDR	0059H	
EA	B ADDR	00AFH	PREDEFINED
ETI	B ADDR	00ABH	PREDEFINED
I2CFG	D ADDR	00D8H	PREDEFINED
LOOP	C ADDR	0032H	
RESET	C ADDR	0026H	
SP	D ADDR	0081H	PREDEFINED
TIMERI	C ADDR	001BH	NOT USED
TIRUN	B ADDR	00DCH	PREDEFINED
WDRST	C ADDR	004EH	
XRETI	C ADDR	0025H	

Using the ADC and PWM of the 83C752/87C752

AN428

The Philips 83C752/87C752 is a single-chip control-oriented microcontroller. It is an 80C51 derivative, having the same basic architecture and powerful instruction set in a small 28-pin package. As "add-on" functions to a standard microcontroller, it offers an I²C small area network port, a five-channel multiplexed 8-bit analog-to-digital converter (ADC), and a pulse width modulation (PWM) output. The part is essentially the popular 8XC751 with the addition of the ADC and the PWM output.

There are many control applications for which this microcontroller can provide an almost-complete, low-cost solution. The A/D converter can monitor analog voltages of up to five sources. The PWM output can be used to generate an analog control voltage with the addition of a simple integrator circuit. Another potential use for the PWM output is as a driver of power-switching circuits for DC motor speed control.

The analog-to-digital converter has 8-bit resolution, and the conversion takes 40 machine cycles. A multiplexer selects one out of five input pins. The operation of the A/D

converter and the multiplexer is controlled by the ADCON register.

The repetition frequency of the PWM output pulses is determined by an 8-bit prescaler, programmed at register PWMP. The duty cycle of these pulses is determined by the contents of a compare register, PWM. In order to implement the pulse width modulator, the prescaler output drives an 8-bit counter. When the counter value matches the contents of the compare (PWM) register, the PWM output is set high, and when the counter reaches zero, the output is set low. The counter is modulo 255, so the duty cycle generated will be the PWM contents multiplied by 1/255.

The enclosed listing demonstrates usage of the A/D converter and the PWM. In order to communicate with the outside world, the program sends messages on a software-driven RS-232 port. The routines for sending messages via a software-controlled serial port can be quite useful, and for further discussion on those, please refer to Application Note 423: "Software Driven Serial Communication Routines for the 83C751 and 83C752 Microcontrollers."

Bit 5 of port 1 is used for the RS-232 communications, and in order to hook the microcontroller to a terminal, a buffer (e.g., MC1488) is needed. Timer 0 is used as the baud rate generator, where the timer value is defined by the symbol BaudVal. The programmed value will generate a 9600 baud rate with a 16MHz crystal.

The program, after initialization and sending a message to the terminal, scans all five A/D channel inputs and outputs the voltage read on the serial port, as a hexadecimal value. Circuit operation can be verified by comparing channel voltages with the reading at the terminal. The program follows with an infinite loop in which channel 0 of the A/D converter is read, and its value is used to program the PWM. A simple verification of the duty cycle can be done with a voltmeter: since it acts as an integrator, its reading will be proportional to the duty cycle. Reading of a voltmeter on the PWM output should be proportional to the channel 0 input voltage. If the analog reference voltage AV_{CC}, which is full-scale of the A/D measurement, is set to be exactly as V_{CC}, the PWM output will track channel 0 within about 20mV.

Using the ADC and PWM of the 83C752/87C752

AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90

PAGE 1

```

1 ;
2
3 ;*****
4
5 ;           87C752 A/D and PWM Demonstration Program
6
7 ; This program first reads all five A/D channels and outputs the values in
8 ; hexadecimal as RS-232 data. Next, the PWM output is set to reflect the
9 ; value on A/D channel 0, and again outputs the A/D value to RS-232. Note
10 ; that the A/D value is inverted before being moved to the PWM compare
11 ; register in order to compensate for the inversion on the PWM output pin.
12 ; This process is repeated continuously.
13
14 ; Thus, a voltage may be applied to ADC0 (P1.0, pin 13) to vary the PWM pulse
15 ; width. A simple test of this function is to measure the voltage on ADC0
16 ; and PWM with a voltmeter. A typical voltmeter will integrate the waveform
17 ; on the PWM output and show a voltage within about 20mV of that on ADC0.
18
19 ; The RS-232 output appears on Port 1 pin 5, which must be buffered with an
20 ; MC1488 or perhaps a MAX232 chip prior to being connected to a terminal.
21 ; The transmission rate will be 9600 baud when the 87C752 is operated from
22 ; 16MHz crystal.
23
24 ;*****
25
26 $Title(87C752 A/D and PWM Demonstration)
27 $Date(12/03/90)
28 $MOD752
29
30 ;*****
31
32 FF75      BaudVal  EQU      -139          ;Timer value for 9600 baud @ 16 MHz.
33                                     ;(one bit cell time)
34
35 0010      XmtDat  DATA    10h          ;Data for RS-232 transmit routine.
36 0012      BitCnt  DATA    12h          ;RS-232 transmit bit count.
37 0013      PWMVal  DATA    13h          ;Holds next value for updating the PWM.
38 0014      ADVal   DATA    14h          ;Holds last A/D conversion result.
39
40 0020      Flags   DATA    20h          ;
41 0000      TxFlag  BIT      Flags.0      ;Transmit-in-progress flag.
42 0001      ADFlag  BIT      Flags.1      ;Indicates A/D conversion complete.
43
44 0095      TxD     BIT      P1.5         ;Port bit for RS-232 transmit.
45
46 ;*****
47
48 ; Interrupt Vectors
49
50 0000      ORG     0                ;Reset vector.
51 0000 0135  AJMP   Reset
52
53 000B      ORG     0BH              ;Timer 0 interrupt.
54 000B 01C5  AJMP   Timr0            ;(used as a baud rate generator)
55
56 002B      ORG     2Bh              ;A/D conversion complete interrupt.
57 002B 0199  AJMP   ADInt
58

```

Using the ADC and PWM of the 83C752/87C752

AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90

PAGE 2

```

0033          59          ORG      33h          ;PWM interrupt.
0033 01A3     60          AJMP     PWMInt
61
62
63          ;*****
64
0035 758130   65          Reset:  MOV     SP,#30h
0038 752000   66          MOV     Flags,#0          ;Clear RS-232 flags.
003B 758800   67          MOV     TCON,#00h        ;Set up timer controls.
003E 75A882   68          MOV     IE,#82h        ;Enable timer 0 interrupt.
69
0041 90011B   70          MOV     DPTR,#Msg1        ;Point to message string.
0044 310A     71          ACALL   Mess          ;Send message.
72
0046 7900     73          MOV     R1,#0          ;Start with A/D channel 0.
0048 E9       74          Loop1:  MOV     A,R1
0049 118D     75          ACALL   ADConv          ;Start A/D conversion.
004B FA       76          MOV     R2,A
77
004C 900152   78          MOV     DPTR,#Msg2        ;Point to message string.
004F 310A     79          ACALL   Mess          ;Send message.
0051 E9       80          MOV     A,R1
0052 11EC     81          ACALL   PrByte          ;Print channel #.
0054 900161   82          MOV     DPTR,#Msg3        ;Point to message string.
0057 310A     83          ACALL   Mess          ;Send message.
84
0059 EA       85          MOV     A,R2
005A 11EC     86          ACALL   PrByte          ;Print A/D value.
005C 09       87          INC     R1          ;Advance R1 value.
005D B905E8   88          CJNE   R1,#5,Loop1
0060 90014F   89          MOV     DPTR,#CRLF        ;Point to message string.
0063 310A     90          ACALL   Mess
91
92          ; Now use A/D channel 0 value to control the PWM.
93
0065 758FFF   94          MOV     PWMP,#0FFh        ;Set PWM slow frequency.
0068 758E00   95          MOV     PWCM,#0          ;Set initial PWM value.
006B 751300   96          MOV     PWMVal,#0        ;Default starting value for the PWM.
006E 75FE01   97          MOV     PWENA,#1         ;Start PWM
0071 75A8CA   98          MOV     IE,#0CAh        ;Now enable the A/D and PWM interrupts.
99
0074 7400     100         Loop2:  MOV     A,#0          ;Read A/D channel 0.
0076 1186     101         ACALL   ADStart        ;Start A/D conversion.
0078 3001FD   102         JNB    ADFlag,$        ;Wait for A/D conversion complete.
007B E514     103         MOV     A,ADVal        ;Get A/D result to print.
007D 11EC     104         ACALL   PrByte          ;Print PWM value.
007F 900165   105         MOV     DPTR,#Msg4        ;Point to message string.
0082 310A     106         ACALL   Mess
0084 80EE     107         SJMP   Loop2
108
109
110          ; A/D Conversion Routines.
111          ; The following shows two ways to use the A/D. Both routines are used by
112          ; different portions of the sample program.
113
114          ; Method 1: This version of the routine starts the conversion and then
115          ; returns. The mainline program can detect when the conversion is
116          ; complete by checking the A/D conversion complete flag (ADFlag) which is

```

Using the ADC and PWM of the 83C752/87C752

AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90

PAGE 3

```

117 ; set by the A/D interrupt service routine. A/D data must be read by the
118 ; calling routine.
119
0086 C201 120 ADStart: CLR      ADFlag      ;Clear A/D conversion complete flag.
0088 4428 121         ORL      A,#28h     ;Add control bits to channel #.
008A F5A0 122         MOV      ADCON,A    ;Start conversion.
008C 22   123         RET
124
125
126 ; Method 2: This is an alternative version of the A/D routine which
127 ; starts the conversion and then waits for it to complete before
128 ; returning. A/D data is returned in the ACC.
129
008D 4428 130 ADConv: ORL      A,#28h     ;Add control bits to channel #.
008F F5A0 131         MOV      ADCON,A    ;Start conversion.
0091 E5A0 132 ADCl:  MOV      A,ADCON
0093 30E4FB 133         JNB     ACC.4,ADC1  ;Wait for conversion complete.
0096 E584 134         MOV      A,ADAT    ;Read A/D.
0098 22   135         RET
136
137
138 ; A/D interrupt service routine.
139
0099 E584 140 ADInt:  MOV      A,ADAT    ;Read A/D data.
009B F514 141         MOV      ADVal,A   ;Save A/D data for print routine.
009D F4   142         CPL      A        ;Complement the value for the PWM.
009E F513 143         MOV      PWMVal,A  ;Set new value for PWM update.
00A0 D201 144         SETB    ADFlag    ;Tell main that new A/D data is ready.
00A2 32   145         RETI
146
147
148 ; PWM interrupt service routine allows updating the PWM synchronously.
149
00A3 85138E 150 PWMInt: MOV      PWCM,PWMVal ;Update PWM duty cycle.
00A6 32   151         RETI
152
153
154 ; Send a byte out RS-232 and wait for completion before returning.
155
00A7 11AD 156 XmtByte: ACALL   RSXmt     ;Send ACC to RS-232 output.
00A9 2000FD 157         JB      TxFlag,$  ;Wait for transmit complete.
00AC 22   158         RET
159
160
161 ; Begin RS-232 transmit.
162
00AD F510 163 RSXmt:  MOV      XmtDat,A   ;Save data to be transmitted.
00AF 75120A 164         MOV      BitCnt,#10 ;Set bit count.
00B2 758CFE 165         MOV      TH,#High BaudVal ;Set timer for baud rate.
00B5 758A75 166         MOV      TL,#Low BaudVal
00B8 758DFE 167         MOV      RTH,#High BaudVal ;Also set timer reload value.
00BB 758B75 168         MOV      RTL,#Low BaudVal
00BE D28C 169         SETB    TR        ;Start timer.
00C0 C295 170         CLR      TxD        ;Begin start bit.
00C2 D200 171         SETB    TxFlag    ;Set transmit-in-progress flag.
00C4 22   172         RET
173
174

```

Using the ADC and PWM of the 83C752/87C752

AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90

PAGE 4

```

175 ; Timer 0 timeout: RS-232 receive bit or transmit bit.
176
00C5 C0E0 177 Timr0:  PUSH  ACC
00C7 C0D0 178         PUSH  PSW
00C9 200007 179         JB    TxFlag,TxBit ;Is this a transmit timer interrupt?
00CC C28C 180 T0Ex1:  CLR    TR          ;Stop timer.
00CE D0D0 181 T0Ex2:  POP    PSW
00D0 D0E0 182         POP    ACC
00D2 32 183         RETI
184
185
186 ; RS-232 transmit bit routine.
187
00D3 D51204 188 TxBit:  DJNZ   BitCnt,TxBusy ;Decrement bit count, test for done.
00D6 C200 189         CLR    TxFlag ;End of stop bit, release timer.
00D8 80F2 190         SJMP  T0Ex1 ;Stop timer and exit.
191
00DA E512 192 TxBusy: MOV    A,BitCnt ;Get bit count.
00DC B40104 193         CJNE  A,#1,TxNext ;Is this a stop bit?
00DF D295 194         SETB  TxD ;Set stop bit.
00E1 80EB 195         SJMP  T0Ex2 ;Exit.
196
00E3 E510 197 TxNext: MOV    A,XmtDat ;Get data.
00E5 13 198         RRC    A ;Advance to next bit.
00E6 F510 199         MOV    XmtDat,A
00E8 9295 200         MOV    TxD,C ;Send data bit.
00EA 80E2 201         SJMP  T0Ex2 ;Exit.
202
203
204 ; Print byte routine: print ACC contents as ASCII hexadecimal.
205
00EC C0E0 206 PrByte: PUSH  ACC
00EE C4 207         SWAP  A
00EF 11FA 208         ACALL HexAsc
00F1 11A7 209         ACALL XmtByte
00F3 D0E0 210         POP   ACC
00F5 11FA 211         ACALL HexAsc ;Print nibble in ACC as ASCII hex.
00F7 11A7 212         ACALL XmtByte
00F9 22 213         RET
214
215
216 ; Hexadecimal to ASCII conversion routine.
217
00FA 540F 218 HexAsc: ANL   A,#0FH ;Convert a nibble to ASCII hex.
00FC 30E308 219         JNB   ACC.3,NoAdj
00FF 20E203 220         JB    ACC.2,Adj
0102 30E102 221         JNB   ACC.1,NoAdj
0105 2407 222 Adj:    ADD   A,#07H
0107 2430 223 NoAdj:  ADD   A,#30H
0109 22 224         RET
225
226
227 ; Message string transmit routine.
228
010A C0E0 229 Mess:   PUSH  ACC
010C 7800 230         MOV   RO,#0 ;RO is character pointer (string
010E E8 231 Mesl:   MOV   A,RO ; length is limited to 256 bytes).
010F 93 232         MOVC  A,@A+DPTR ;Get byte to send.

```

Using the ADC and PWM of the 83C752/87C752

AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90

PAGE 5

```

0110 B40003      233          CJNE     A,#0,Send      ;End of string is indicated by a 0.
0113 D0E0        234          POP      ACC
0115 22          235          RET
                  236
0116 11A7        237      Send:   ACALL    XmtByte      ;Send a character.
0118 08          238          INC      R0          ;Next character.
0119 80F3        239          SJMP    Mes1
                  240
011B 0D0A        241      Msg1:   DB      0Dh, 0Ah,
011D 54686973    242          DB      'This is a demonstration of the 87C752 A/D and PWM.'
0121 20697320
0125 61206465
0129 6D6F6E73
012D 74726174
0131 696F6E20
0135 6F662074
0139 68652038
013D 37433735
0141 3220412F
0145 4420616E
0149 64205057
014D 4D2E
014F 0D0A00      243      CRLF:   DB      0Dh, 0Ah, 0
                  244
0152 0D0A412F    245      Msg2:   DB      0Dh, 0Ah, 'A/D Channel ', 0
0156 44204368
015A 616E6E65
015E 6C2000
                  246
0161 203D2000    247      Msg3:   DB      ' = ', 0
                  248
0165 202000      249      Msg4:   DB      ' ', 0
                  250
                  251          END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

Using the ADC and PWM of the 83C752/87C752

AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90

PAGE 6

ACC	D ADDR	00E0H	PREDEFINED
ADAT	D ADDR	0084H	PREDEFINED
ADC1	C ADDR	0091H	
ADCON	D ADDR	00A0H	PREDEFINED
ADCONV	C ADDR	008DH	
ADFLAG	B ADDR	0001H	
ADINT	C ADDR	0099H	
ADJ	C ADDR	0105H	
ADSTART	C ADDR	0086H	
ADVAL	D ADDR	0014H	
BAUDVAL	NUMB	FF75H	
BITCNT	D ADDR	0012H	
CRLF	C ADDR	014FH	
FLAGS	D ADDR	0020H	
HEXASC	C ADDR	00FAH	
IE	D ADDR	00A8H	PREDEFINED
LOOP1	C ADDR	0048H	
LOOP2	C ADDR	0074H	
MESL	C ADDR	010EH	
MESS	C ADDR	010AH	
MSG1	C ADDR	011BH	
MSG2	C ADDR	0152H	
MSG3	C ADDR	0161H	
MSG4	C ADDR	0165H	
NOADJ	C ADDR	0107H	
P1	D ADDR	0090H	PREDEFINED
PRBYTE	C ADDR	00ECH	
PSW	D ADDR	00D0H	PREDEFINED
PWCM	D ADDR	008EH	PREDEFINED
PWENA	D ADDR	00FEH	PREDEFINED
PWMINT	C ADDR	00A3H	
PWMP	D ADDR	008FH	PREDEFINED
PWMVAL	D ADDR	0013H	
RESET	C ADDR	0035H	
RSXMT	C ADDR	00ADH	
RTH	D ADDR	008DH	PREDEFINED
RTL	D ADDR	008BH	PREDEFINED
SEND	C ADDR	0116H	
SP	D ADDR	0081H	PREDEFINED
T0EX1	C ADDR	00CCH	
T0EX2	C ADDR	00CEH	
TCON	D ADDR	0088H	PREDEFINED
TH	D ADDR	008CH	PREDEFINED
TIMR0	C ADDR	00C5H	
TL	D ADDR	008AH	PREDEFINED
TR	B ADDR	008CH	PREDEFINED
TXBIT	C ADDR	00D3H	
TXBUSY	C ADDR	00DAH	
TXD	B ADDR	0095H	
TXFLAG	B ADDR	0000H	
TXNEXT	C ADDR	00E3H	
XMTBYTE	C ADDR	00A7H	
XMTDAT	D ADDR	0010H	

Airflow measurement using the 83/87C752 and "C"

AN429

INTRODUCTION

This application note describes a low-cost airflow measurement device based on the Philips 83/87C752 microcontroller. Airflow measurement—determining the volume of air transferred per unit time (cubic feet per minute, or cfm)—is intrinsic to a variety of industrial and scientific processes.

Airflow computation depends on three simultaneous physical air measurements—velocity, pressure, and temperature. This design includes circuits and sensors allowing the 8XC752 to measure all three parameters.

The design also includes seven-segment LED displays, discrete LEDs, and pushbutton switches to allow selective display of airflow, temperature, and pressure. Furthermore, airflow is continuously compared with a programmer-defined setpoint. Should the measured airflow exceed the setpoint, an output relay is energized. In actual application, this relay output could be used to signal the setpoint violation (via lamp or audio annunciator) or otherwise control the overall process (e.g., emergency process shutdown). Of course, the setpoint, comparison criteria (greater, less than, etc.) and violation response (relay on, relay off) are easily changed by program modification to meet actual application requirements.

Referring to Figure 1, the overall operation of the airflow device is as follows.

Normally the unit continuously displays the airflow (in cfm) on the seven-segment

displays. The discrete CFM LED is also lit to confirm the parameter being displayed.

Pressing the TEMP pushbutton switches the display to temperature (in degrees C) and lights the TEMP LED. As long as the pushbutton remains pressed, the temperature is displayed. When the pushbutton is released, the display reverts to the default pressure display.

Similarly, pressing the PSI pushbutton displays the atmospheric pressure (in pounds per square inch) and lights the PSI LED. The pressure is displayed as long as the pushbutton is pressed, and the default airflow display resumes when the pushbutton is released.

Finally, pressing the SET-POINT pushbutton displays the programmed airflow setpoint (in cfm) and lights the SET-POINT LED. Again, releasing the pushbutton causes the display to revert to the default airflow measurement.

CONTROL PROGRAMMING IN "C"

While, thanks to advanced semiconductor processing, hardware price/performance continues to improve, software development technology has changed little over time. Thus, given ever-rising costs for qualified personnel, software "productivity" is arguably in decline. Indeed, for low-unit cost and/or low-volume applications, software development has emerged as the major portion of total design cost. Furthermore,

beyond the initial programming cost, "hidden" costs also arise in the form of life-cycle code maintenance and revision and lost revenue/market share due to excessive time-to-market.

Traditionally, control applications have been programmed in assembly language to overcome microcontroller resource and performance constraints. Now, thanks to more powerful microcontrollers and advanced compiler technology, it is feasible to program control applications using a High-Level Language (HLL).

The primary benefit of using an HLL is obvious—one HLL program "statement" can perform the same function as many lines of assembly language. Furthermore, a well-written HLL program will typically be more "readable" than an assembly language equivalent, resulting in reduced maintenance and revision/upgrade costs.

Of the many popular HLLs, the "C" language has emerged as the major contender for control applications. More than other languages, C gives the programmer direct access to, and control of, low-level hardware resources—a requirement for deterministic, real-time I/O applications. Furthermore, C is based on a "minimalist" philosophy in which the language performs only those functions explicitly requested by the programmer. This approach is well-suited for control applications, which are often characterized by strict cost and performance requirements.

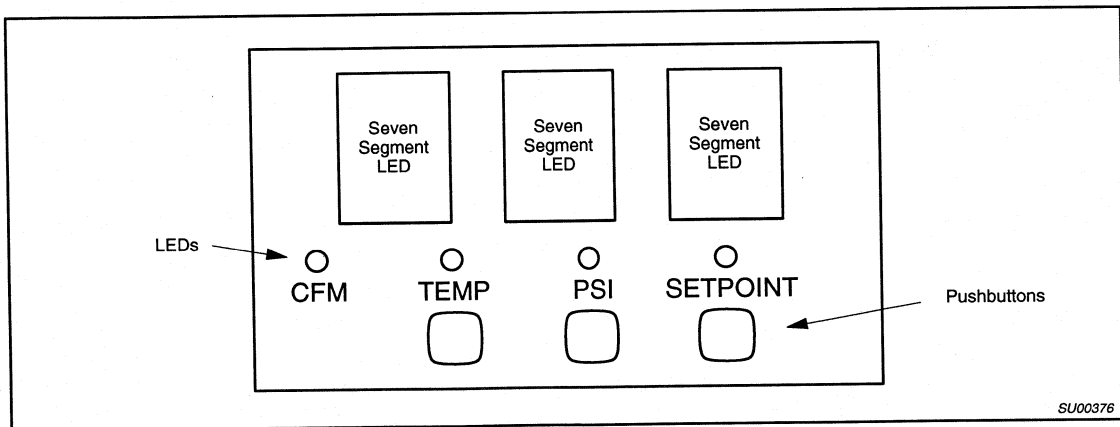


Figure 1. Airflow Meter Front Panel

SU00376

Airflow measurement using the 83/87C752 and "C"

AN429

8XC752 OVERVIEW

The 83C752/87C752 (ROM/EPROM-based) combine the performance advantages of the 8-bit 80C51 architecture with the low cost, power consumption, and size/pin count of a 4-bit microcontroller. Therefore, the 8XC752 is uniquely capable of bringing high processing speed and HLL programming to even the most cost-sensitive applications such as handheld (battery driven) instruments, automotive distributed processing, "smart" appliances, and sophisticated consumer electronics.

Obviously, the 8XC752 can be used for cost-reduced versions of existing 8-bit applications. The device can also replace similarly priced 4-bit devices to achieve benefits of higher performance and, most importantly, easier s/w development including the use of HLL. Indeed, the component and system design costs associated with the 8XC752 are so low that it is a viable candidate for first-time computerization of formerly non-microcontroller-based designs.

Figure 2 shows the block diagram of the 8XC752. Major features of the device include the following.

Full-Function, High-Speed (to 16MHz) 80C51 CPU Core

The popular 80C51 architecture features 8- and 16-bit processing and high-speed execution. Most instructions execute in a single machine cycle (the slowest instructions require only two cycles). Though a streamlined architecture, the CPU core,

unlike 4-bit devices, includes all the basic capabilities (such as stack, multiply instruction, interrupts, etc.) required to support HLL compilation. The CPU core also includes a unique Boolean processor which is well-suited for the bit-level processing and I/O common to control applications.

Low-Power CMOS and Power-Saving Operation Modes

Thanks to the advanced CMOS process, the 8XC752 features extremely low power consumption, which helps to extend battery life in handheld applications and otherwise reduce power supply and thermal dissipation costs and reliability concerns. Low ACTIVE mode (full-speed operation) power consumption—only 11mA typical at 12MHz—is further complemented by two program-initiated power-saving operation modes—IDLE and POWER-DOWN.

In idle mode, CPU instruction processing stops while on-chip I/O and RAM remain powered. Power consumption drops to 1.5 μ A (typical, 12MHz) until processing is restarted by interrupt or reset. Power-down mode cuts power consumption further (to only 10 μ A typical at 12MHz) by stopping both instruction and I/O processing. Return to full-speed operation from power-down mode is via reset.

Note that power consumption can be further cut by reducing the clock frequency as much as application performance requirements allow, as shown in Figure 3.

Another virtue of the CMOS process is superior tolerance to variations in V_{CC} , a requirement inherent in the targeted applications. The EPROM-based device (87C752) operates over a V_{CC} range of 4.5V to 5.5V, while the ROM-based device (83C752) will operate from 4V to 6V.

On-Chip ROM (83C752), EPROM (87C752), and RAM

The 8XC752 integrates 2048 bytes of program ROM/EPROM and 64 bytes of data RAM. This relatively small amount of memory reflects the fact that the targeted applications, though they may require high-speed processing, are typically characterized by simple algorithms and data structures. High code efficiency of the architecture means even this small amount of memory can effectively support the use of C. If necessary, the judicious use of assembly language can help bypass code size (and performance) constraints.

Five-Channel 8-Bit A/D Converter

Most control applications are characterized by the need to monitor "real-world" (i.e., analog) parameters. To this end, the 8XC752 includes a medium-speed (40 clock cycle conversion) 8-bit analog-to-digital (A/D) converter. Five separate input lines are provided along with multiplexer logic to select an input for conversion. The A/D converters speed, resolution, and accuracy are more than adequate to measure temperature, pressure, and other common environmental parameters.

Airflow measurement using the 83/87C752 and "C"

AN429

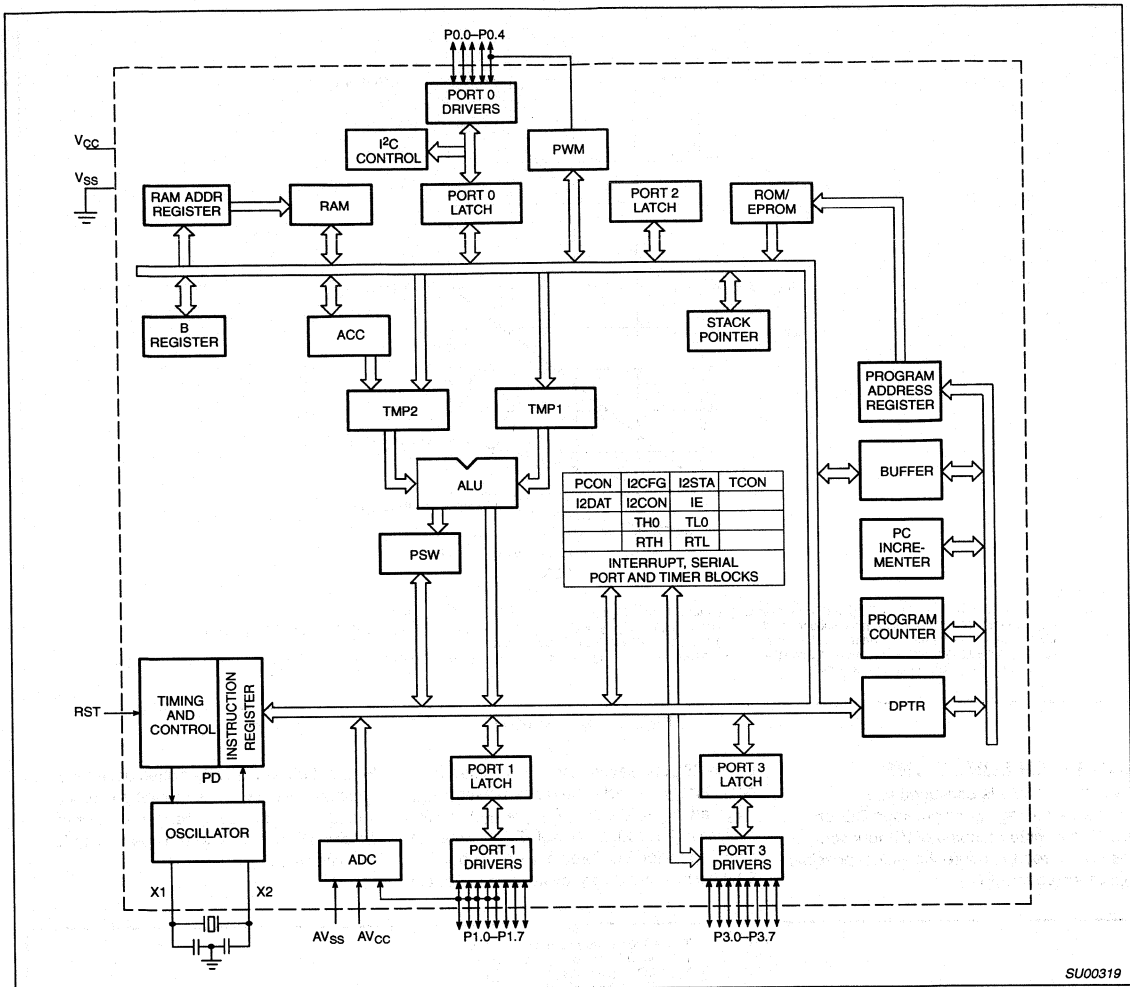


Figure 2. Block Diagram of the 8XC752

Timer/Counters

Control applications, due to their "real-time" nature, invariably call for a variety of timing and counting capabilities. The 8XC752 meets the need by integrating three separate functions—a 16-bit auto-reload counter/timer, an 8-bit pulse width modulator (PWM) output/timer, and a fixed-rate timer for timebase generation. Together, these timing/counting resources can serve a range of tasks, including waveform generation, external event counting, elapsed time

calculation, periodic interrupt generation, and watchdog timer.

I²C Bus

The Inter-Integrated Circuit (I²C) bus is a patented serial peripheral interface. The virtue of I²C is the ability to expand system functionality with acceptable performance and minimum cost. Notably, the pin and interconnect count is radically reduced compared to expansion via a typical microprocessor bus—I²C requires only two

lines, while a parallel bus often consumes 20-30 lines and may call for extra glue logic (decoder, address latch, etc.). The 8XC752 I²C port allows easy connection to a wide variety of compatible peripherals such as LCD drivers, A/D and D/A converters, consumer/telecom and special-purpose memory (e.g., EEPROM). I²C can also be used to build distributed processing systems connecting multiple I²C-compatible microcontrollers.

Airflow measurement using the 83/87C752 and "C"

AN429

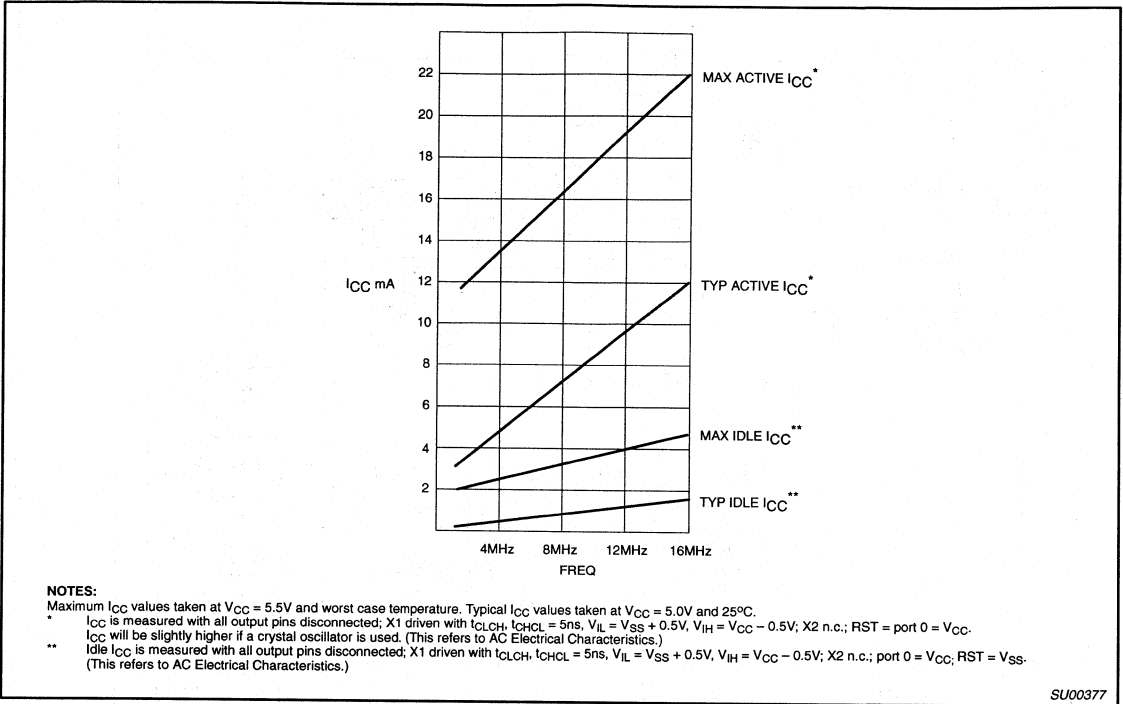


Figure 3. I_{CC} vs. FREQ

8XC752 PIN FUNCTIONS

Since the 8XC752 is packaged in a cost/space-saving 28-pin package DIP or PLCC), a flexible mapping of I/O functions to pins is required to ensure the widest possible application coverage.

Of the 28 pins, seven pins are allocated to basic functions, including digital power (V_{CC}, V_{SS}), analog reference (AV_{CC}, AV_{SS}), clock oscillator (X1, X2), and reset (RST). Thus, 21 pins, organized into three ports (5-bit port 0, 8-bit ports 1 and 3), are available for user I/O.

Figure 4 shows the alternative uses for these 21 lines. As shown, the mapping is quite versatile, which maximizes the access to on-chip I/O functions and helps ensure full pin utilization.

P0.0	TTL IN/OUT (open drain), I ² C clock (SCLK)
P0.1	TTL IN/OUT (open drain), I ² C data (SDA)
P0.2	TTL IN/OUT (open drain)
P0.3	TTL IN/OUT (internal pull-up)
P0.4	TTL IN/OUT (internal pull-up), PWM output
P1.0	TTL IN/OUT (internal pull-up), A/D input channel 0
P1.1	TTL IN/OUT (internal pull-up), A/D input channel 1
P1.2	TTL IN/OUT (internal pull-up), A/D input channel 2
P1.3	TTL IN/OUT (internal pull-up), A/D input channel 3
P1.4	TTL IN/OUT (internal pull-up), A/D input channel 4
P1.5	TTL IN/OUT (internal pull-up), INTO interrupt input
P1.6	TTL IN/OUT (internal pull-up), INT1 interrupt input
P1.7	TTL IN/OUT (internal pull-up), TIMER 0 (T0) input
NOTE: P1.0-P1.4 may only be changed as a group, i.e., either all TTL I/O or all A/D inputs. However, when selected as A/D inputs, P1.0-P1.4 may also be used as TTL inputs.	
P3.0	TTL IN/OUT (internal pull-up)
P3.1	TTL IN/OUT (internal pull-up)
P3.2	TTL IN/OUT (internal pull-up)
P3.3	TTL IN/OUT (internal pull-up)
P3.4	TTL IN/OUT (internal pull-up)
P3.5	TTL IN/OUT (internal pull-up)
P3.6	TTL IN/OUT (internal pull-up)
P3.7	TTL IN/OUT (internal pull-up)

SU00378

Figure 4. 8XC752 I/O Port Description

Airflow measurement using the 83/87C752 and "C"

AN429

AIRFLOW METER CIRCUIT**DESCRIPTION**

Figure 5 is the schematic diagram of the airflow meter circuit. As shown, the 8XC752 is connected to the following function blocks.

Discrete and Seven-Segment LED Display

The seven-segment LEDs display the parameter of interest (airflow, temperature, pressure, or setpoint). A discrete LED associated with each parameter is lit when that parameter is being displayed.

The seven-segment LEDs are identified as X0.1, X1, and X10, reflecting their decimal position (tenths, ones, and tens, respectively). Each display has eight data inputs (the seven segments and a decimal point) and common terminals which allow the display to be enabled or blanked. The eight data inputs, and the four discrete LEDs, are driven from port 3 of the 8XC752 via high-current driver U2 and current limiting resistors RP1.

Since all the segmented and discrete LEDs share common data lines, data display must be time multiplexed. Transistors Q1-Q4 connect to separate output lines of port 0, allowing a particular seven-segment LED or the discrete LEDs (as a group) to be individually enabled for display. This type of LED multiplexing is quite common since, at a fast enough refresh rate, the switching between displays is not perceptible by the operator. The major benefit is the reduction of I/O lines required (without multiplexing, 28, rather than 8, data lines would be required).

Pushbutton Switch Inputs

Three pushbuttons select the parameter to be displayed—temperature, pressure, or setpoint (when no button is pressed, airflow is displayed). The four states (SW1, SW2, SW3, or no button pressed) are effectively encoded onto two port 1 input lines (taking advantage of the capability to use port 1 lines configured as A/D for TTL input) as follows:

	P1.3	P1.4
No button pressed	HIGH	HIGH
SW1 (TEMP) pressed	LOW	HIGH
SW2 (PSI) pressed	HIGH	LOW
SW3 (SETPOINT) pressed	LOW	LOW

The only impact of this encoding scheme is that SW3 has a higher priority than the other pushbuttons—a factor of no concern in this simple application. Similarly, latching, debouncing, rollover, or other conditioning of the pushbutton inputs is not required.

Setpoint Control

This is simply a variable resistor voltage divider which serves to establish an analog voltage corresponding to an airflow threshold at which action is taken. It connects to a port 1 A/D input.

Relay Output

When an airflow setpoint violation is detected, DPDT relay K1 is energized via P1.6, which is configured as a TTL output, buffered by transistor Q5.

Flowmeter Input

Measurement of the air velocity is via an air turbine tachometer connected, via optoisolator U7, to P1.5, which is configured as a TTL input. The tachometer input is

assumed to be a negative-going pulse train with less than 10% duty cycle.

Air Pressure Sensor

To determine airflow, the air velocity must be factored by ambient pressure—for a given velocity (and temperature), lower/higher atmospheric pressure will correspond with lower/higher airflow. The pressure sensor, U3, outputs a voltage differential corresponding to the pressure. Amplifier U4 conditions the pressure sensor output to the range of AV_{SS} to AV_{CC} (the analog references for the 8XC752 A/D converter). The conditioned pressure sensor output is presented to A/D input P1.0.

To calibrate the pressure sensor, press the PSI pushbutton and adjust the gain pot (R1) until the display matches the local atmospheric pressure in pounds per square inch (14.7 at sea level).

Air Temperature Sensor

Similar to pressure, ambient temperature also affects the airflow calculation. For a given air velocity (and pressure), higher/lower temperature will correspond with lower/higher airflow. Temperature sensor U5 outputs an absolute voltage corresponding to temperature. Amplifier U6 conditions the temperature sensor output to the range AV_{SS} to AV_{CC} for connection to A/D input P1.1.

To calibrate the temperature sensor, adjust the gain pot (R5) so that the display (while pressing the TEMP pushbutton) matches the measured output of U5 (LM35).

Figure 6 summarizes the usage of the 8XC752 I/O lines in this application.

Airflow measurement using the 83/87C752 and "C"

AN429

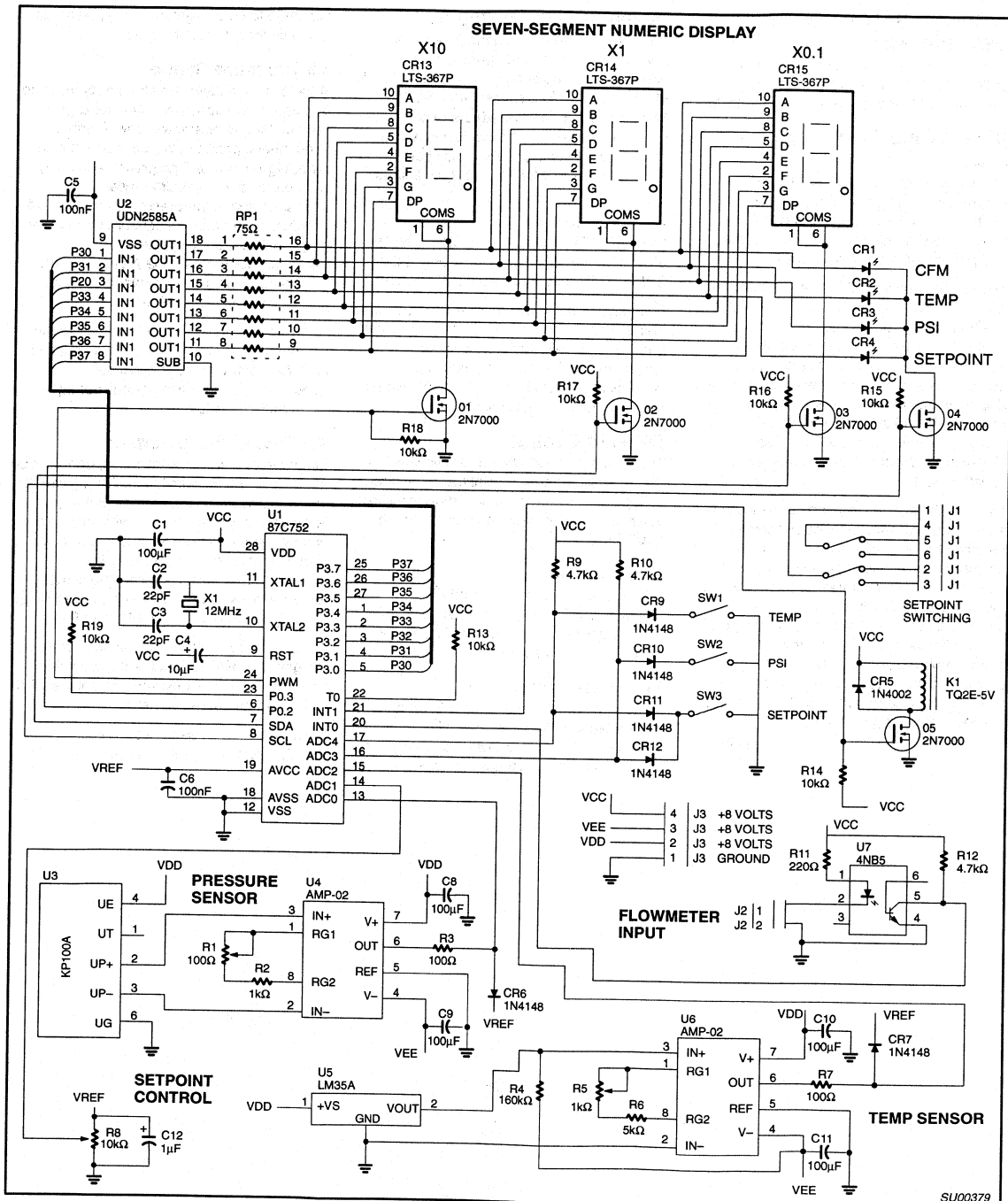


Figure 5. Schematic Diagram of the Airflow Meter Circuit

SU00379

Airflow measurement using the 83/87C752 and "C"

AN429

P0.0	TTL OUT—Enables the discrete LEDs
P0.1	TTL OUT—Enables the tenths digit seven-segment LED
P0.2	TTL OUT—Enables the ones digit seven-segment LED
P0.3	Pulled up
P0.4	TTL OUT—Enables the tens digit seven-segment LED
P1.0	A/D input—Connected to analog air pressure sensor
P1.1	A/D input—Connected to analog air temperature sensor
P1.2	A/D input—Connected to analog setpoint control
P1.3	TTL IN—One of two pushbutton input lines
P1.4	TTL IN—The second pushbutton input line
P1.5	INT0 interrupt input—Air turbine tachometer input
P1.6	TTL OUT—Setpoint relay control
P1.7	Pulled up
NOTE: P1.0–P1.4 may only be changed as a group, i.e., either all TTL I/O or all A/D inputs. However, when selected as A/D inputs, P1.0–P1.4 may also be used as TTL inputs.	
P3.0	TTL OUT—Seven-segment LEDs segment A, CFM discrete LED
P3.1	TTL OUT—Seven-segment LEDs segment B, TEMP discrete LED
P3.2	TTL OUT—Seven-segment LEDs segment C, PSI discrete LED
P3.3	TTL OUT—Seven segment LEDs segment D, SETPOINT discrete LED
P3.4	TTL OUT—Seven segment LEDs segment E
P3.5	TTL OUT—Seven segment LEDs segment F
P3.6	TTL OUT—Seven segment LEDs segment G
P3.7	TTL OUT—Seven segment LEDs segment DP

SU00380

Figure 6. Airflow Meter I/O Port Usage

SOFTWARE DEVELOPMENT PROCEDURE

The airflow meter application software is almost entirely written in C using a development package from Franklin Software. The Franklin Software C compiler is a cross-compiler that runs on the IBM PC (and compatibles) while generating code suitable for execution by any 80C51-based product, including the 8XC752. For more information, contact:

Franklin Software
888 Saratoga Ave., #2
San Jose, CA 95129

The process of developing a C program using the Franklin package (the process is similar for other third-party cross-compilers) is as follows:

1. The program is entered/edited on the PC using the programmer's preferred text editor.
2. The program is compiled on the PC with the Franklin C compiler.

3. Should compile errors (also known as syntax errors) occur, they are corrected by returning to step 1 until an error-free compile is achieved.
4. Before testing the compiled program, it needs to be combined, using the Franklin-supplied linker, with any required assembly language routines. Besides routines explicitly written by the programmer, every Franklin C program requires an assembly language startup routine (supplied by Franklin and, if necessary, edited by the programmer) which performs basic reset initialization and configuration operations before transferring control to the C program.
5. The compiled object code is tested for correct operation. This can either be accomplished by using an 80C51-family simulator running on the PC or by downloading the object code to an in-circuit emulator. The simulator

approach has the virtues of low cost and consolidation of all work on the PC at the cost of non-real-time operation/debug constraints (the simulator may execute 100-1000 times slower than the microcontroller). The in-circuit emulator provides real-time operation and the additional benefit of assisting hardware design debug at somewhat higher cost.

6. Should program execution prove faulty (known as semantic errors), return to step 1 until error-free operation is achieved.
7. The error-free (syntax and semantic) and linked object code, in the form of a .HEX file, is transferred to an EPROM programmer. Fitted with a suitable adaptor, the EPROM programmer can "burn" the object file into the targeted EPROM-based 80C51-family device. For ROM-based devices, the object file is transferred to the factory for custom masking.

Airflow measurement using the 83/87C752 and "C"

AN429

PROGRAM DESCRIPTION

Figure 7 is a flowchart of the program; following the flowchart is the program listing. The flowchart shows the basic processing and flow, while the listing documents the details of the program's implementation.

The program consists of four interrupt-driven (i.e., foreground) routines and a main program (i.e., background). The background program is entered at reset and executes forever, interrupted periodically by the foreground interrupts. Communication between the background program and the foreground handlers is via shared variables.

The four interrupt routines are as follows.

- multiplex () (INT3)

Free-running Timer 1 generates an interrupt at approximately 1000Hz and is used to multiplex the seven-segment and discrete LED display data. In a round-robin manner, at each interrupt, the program turns off the previously enabled display and writes data to, and enables, the next display. Finally, the interrupt routine sets a pointer to the next display—at the next interrupt, that display will be refreshed. Thus, each display (tens, ones, tenths, discrete LEDs) will be refreshed every fourth interrupt, which is more than fast enough for a flicker-free display.

- read_switch () (INT6)

The PWM prescaler is configured to generate a periodic interrupt (INT6) at about 97Hz. The program counts these interrupts, and every 32nd interrupt sets an "update" variable. The main program will change the display data when it detects that "update" is set and clear "update" to prepare for the next display cycle. Thus, display change frequency is about 33Hz (i.e., 33ms), which eliminates display glitches associated with pushbutton switch bounce.

- calc_cfm () (INT0)

The air velocity turbine tachometer drives the 8XC752 INT0 interrupt pin. At each interrupt, the program reads Timer 0, which keeps track of the elapsed time (the low 16 bits of a 24-bit count in microseconds) between INT0 interrupts. The high-order 8-bit elapsed time count is cleared for possible updating by the following routine.

- overflow () (INT1)

When Timer 0 overflows (generating an interrupt), the program increments the high-order 8 bits of a 24-bit variable, counting the microseconds between tachometer interrupts (handled by the previous routine). If this 8-bit value becomes too large (i.e., tachometer interrupts stop), a NOFLOW

variable is set, which will cause the main program to display an EEE out-of-range indicator on the seven-segment LEDs.

With the interrupt handlers executing the low-level timing and I/O, the main program, which is entered on reset and executes forever, consists of only three major steps.

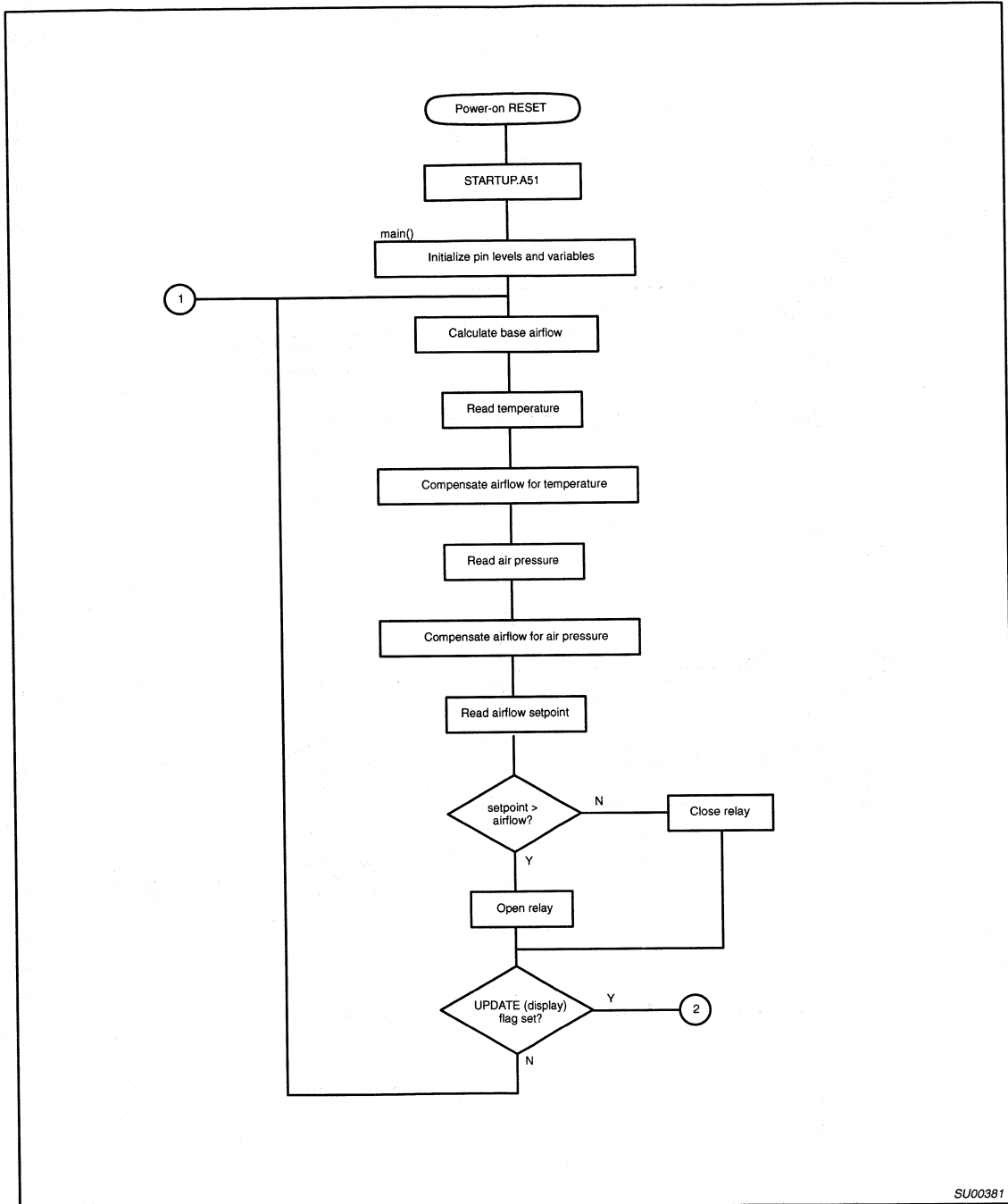
The temperature/pressure compensated airflow is calculated. First, the "base" cfm rate, as tracked by the calc_cfm () tachometer interrupt is adjusted by removing the execution time of the calc_cfm () handler itself. Next, the temperature is determined (A/D channel 1), and airflow is compensated. Similarly, the air pressure is determined (A/D channel 0) and airflow compensated again.

Now that the true airflow is calculated, it is compared with the setpoint (adjusted with the variable resistor), which is determined by reading A/D channel 2. If the airflow is greater than the setpoint, the relay is closed. Otherwise, the relay is opened.

Finally, the UPDATE flag (set by the 33Hz read_switch () interrupt) is checked. If it is time to update, the data to be displayed is determined based on the pushbutton status and the state of the NOFLOW flag. The updated display data is initialized for later display on the LEDs by the multiplex () display refresh interrupt handler.

Airflow measurement using the 83/87C752 and "C"

AN429



SU00381

Figure 7. Program Flowchart

Airflow measurement using the 83/87C752 and "C"

AN429

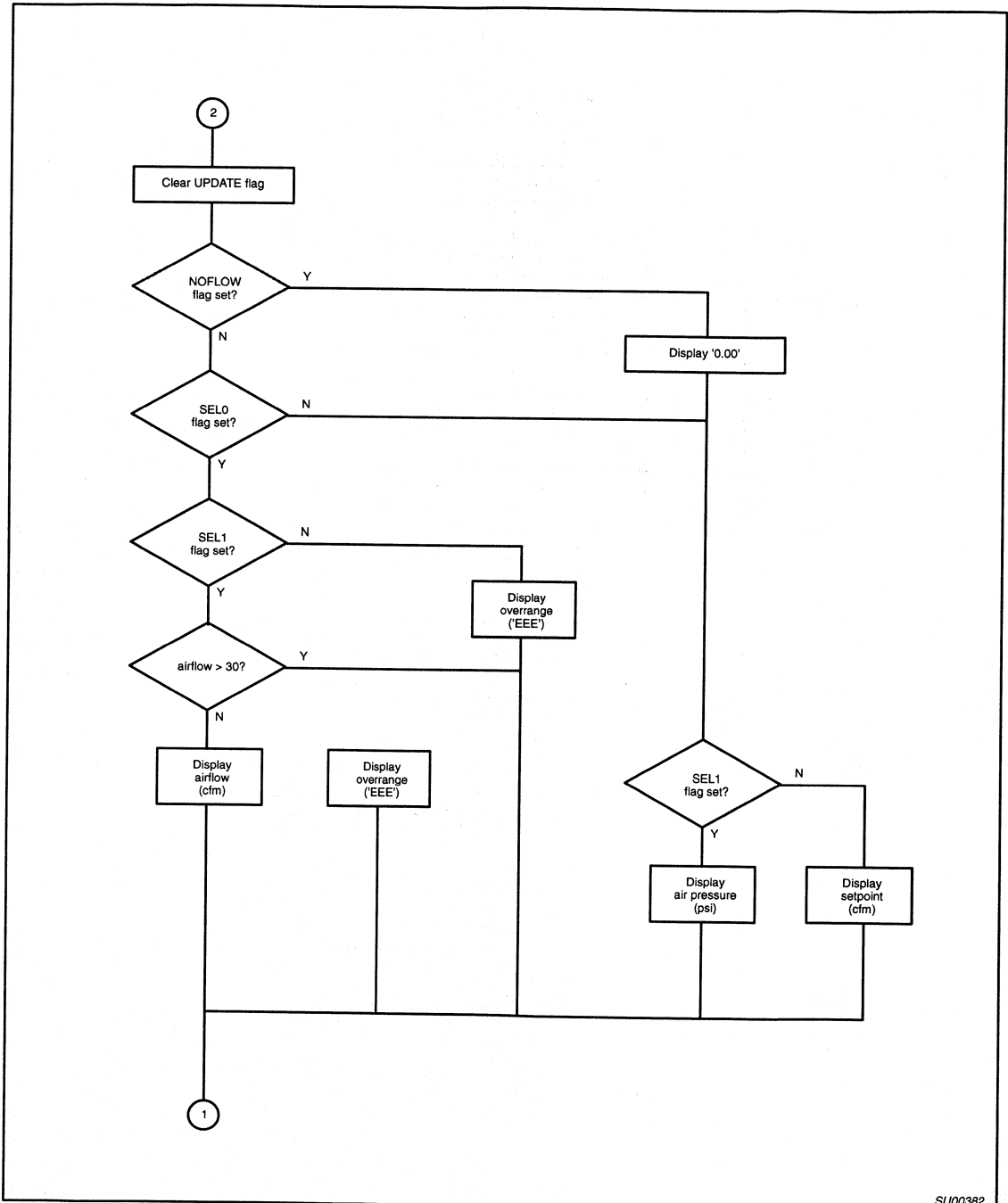


Figure 7. Program Flowchart (Continued)

SU00382

Airflow measurement using the 83/87C752 and "C"

AN429

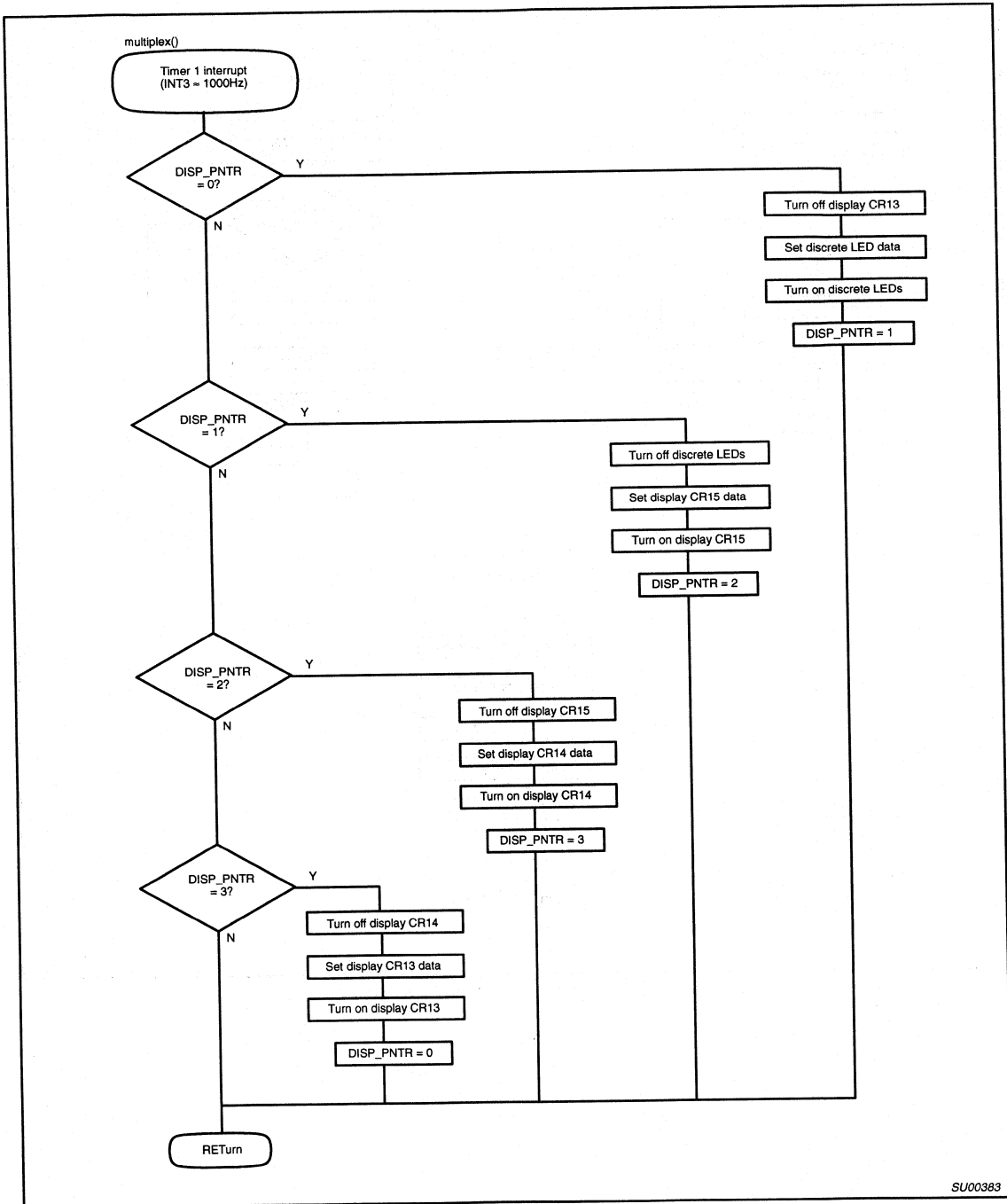
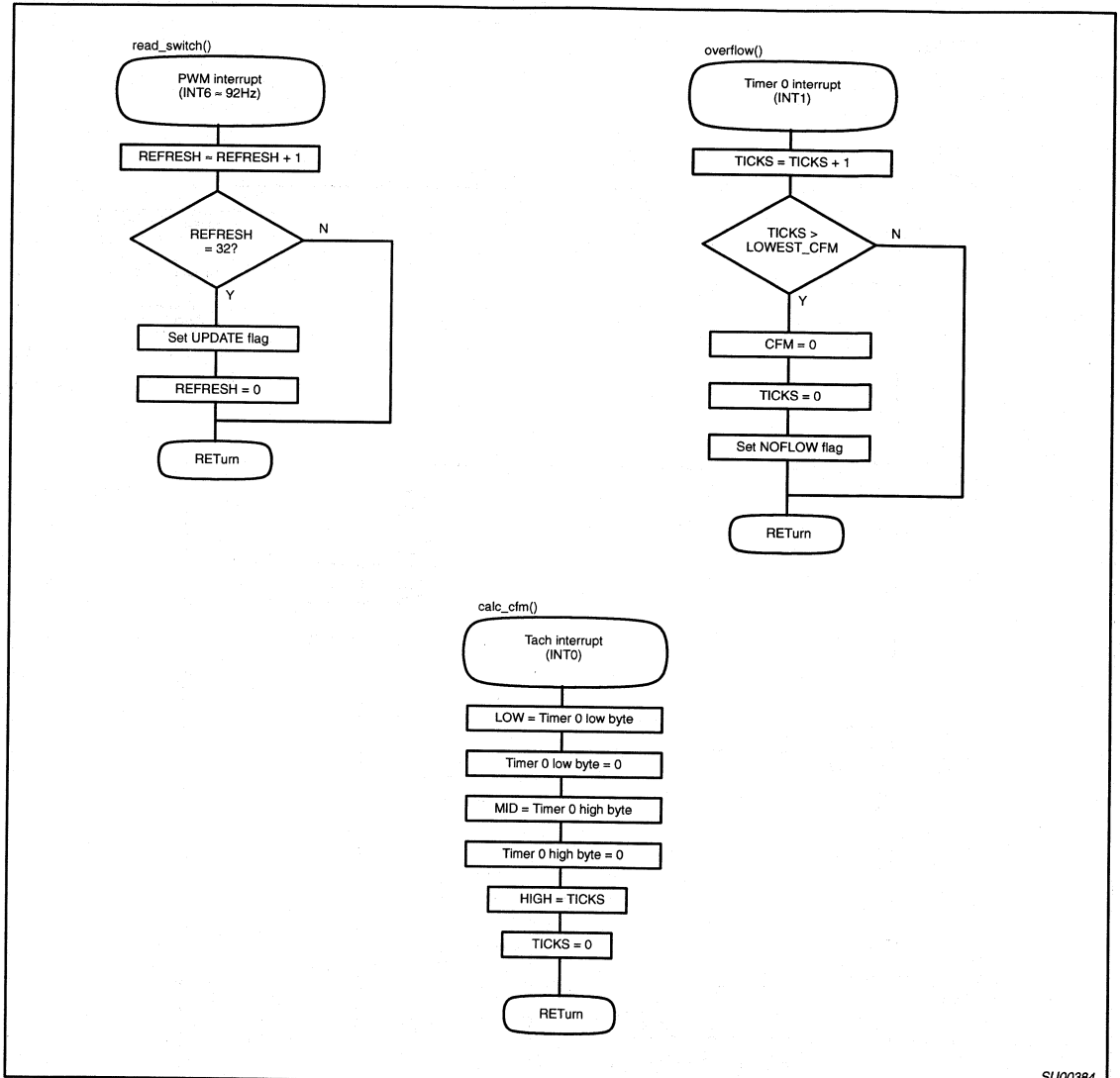


Figure 7. Program Flowchart (Continued)

SU00383

Airflow measurement using the 83/87C752 and "C"

AN429



SU00384

Figure 7. Program Flowchart (Continued)

Airflow measurement using the 83/87C752 and "C"

AN429

```

/*
    this program measures the air flow through a rotary flowmeter
    and displays the calculated cfm. the output of the flowmeter
    tachometer is a small duty cycle pulse train with period
    which is proportional to the flow. the flow is compensated
    for changes in pressure and temperature to maintain
    calibration. if the flow exceeds an adjustable setpoint
    it energizes a 2 form c relay for user application.
*/

/*
    these pragmas specify compiler command line options
*/
#pragma CODE           /* generate code           */
#pragma SYMBOLS        /* and symbols          */
#pragma PL (60)        /* 60 lines per page   */
#pragma PW (120)       /* 120 cols per page   */
#pragma OT (3)         /*                      */
#pragma ROM (SMALL)    /* single-chip mode    */

/*
    include the 8XC752-specific definitions and
    the standard i/o library.
*/
#include                <reg752.h>
#include                <stdio.h>

/*
    define symbolic names for program constants
*/
#define ZERO_K          2730      /* 0 degrees centigrade in 1/10 kelvin */
#define ONE_TENTH_CFM  4444444L   /* 1/10 cfm in microseconds           */
#define STD_TEMP       2980      /* 25 degrees centigrade in 1/10 kelvin */
#define STD_ATM        147       /* one atmosphere in 1/10 psi          */
#define LOWEST_CFM     0x40      /* maximum period from meter 0x400000 */
#define START_ADC0     0x28      /* commands to start appropriate      */
#define START_ADC1     0x29      /* a/d channel conversion cycle       */
#define START_ADC2     0x2a      /*                                     */
#define START_ADC3     0x2b      /*                                     */
#define START_ADC4     0x2c      /*                                     */
#define ADCI            0x10      /* a/d converter status flags         */
#define ADCS            0x08      /*                                     */
#define FREERUN_I      0x10      /*                                     */
#define SEG_A           0x01      /* P3 position for display segment 'a' */
#define CFM             0x01      /* P3 position for 'cfm' led           */
#define SEG_B           0x02      /* P3 position for display segment 'b' */
#define DEGREES        0x02      /* P3 position for 'degrees' led       */
#define SEG_C           0x04      /* P3 position for display segment 'c' */
#define PSI            0x04      /* P3 position for 'psi' led           */
#define SEG_D           0x08      /* P3 position for display segment 'd' */
#define SETPOINT       0x08      /* P3 position for 'setpoint' led      */
#define SEG_E           0x10      /* P3 position for display segment 'e' */
#define SEG_F           0x20      /* P3 position for display segment 'f' */
#define SEG_G           0x40      /* P3 position for display segment 'g' */
#define SEG_DP          0x80      /* P3 position for display decimal pt. */

typedef unsigned char byte; /* byte data type is unsigned 8-bit */
typedef unsigned int word; /* word data type is unsigned 16-bit */
typedef unsigned long l_word; /* l_word data type is unsigned 32-bit */

#define TRUE 1 /* define logical true / false */
#define FALSE 0 /* values for bit variables */

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

/*
    define look-up table of possible seven segment display
    characters. the table consists of 11 elements corresponding
    to the 10 digits ('0'-'9') and error symbol ('E') that can be
    displayed. Each element is defined by ANDING (|) the bit
    mask for each segment (SEG_A - SEG_G) comprising the
    character. the table contents need to be inverted before
    use to be compatible with U2 (udn2585a). for example,
    '~segments[3]' specifies the segment mask to display '3'.
*/
code byte segments [ ] =
{
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F |          /* 0 */
    SEG_B | SEG_C |          /* 1 */
    SEG_A | SEG_B |          /* 2 */
    SEG_A | SEG_B | SEG_C | SEG_D |          /* 3 */
    SEG_A |          SEG_C | SEG_D |          SEG_F | SEG_G | /* 4 */
    SEG_A |          SEG_C | SEG_D | SEG_E | SEG_F | SEG_G | /* 5 */
    SEG_A | SEG_B | SEG_C |          /* 6 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E |          /* 7 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G | /* 8 */
    SEG_A | SEG_B |          SEG_D |          SEG_E |          SEG_F | SEG_G | /* 9 */
    SEG_A |          SEG_D |          SEG_E |          SEG_F | SEG_G | /* E */
};

/*
    define the '752 special function bits which control i/o lines.
    note that i/o line (and constant) names are capitalized
*/
sbit RELAY = 0x96; /* active hi to turn on setpoint relay */
sbit STROBE_0 = 0x80; /* active hi to enable display status led's */
sbit STROBE_1 = 0x81; /* active hi to enable display cr15 (tenths) */
sbit STROBE_2 = 0x82; /* active hi to enable display cr14 (ones) */
sbit NO_FLOW = 0x83; /* flag set when no flow detected */
sbit STROBE_3 = 0x84; /* active hi to enable display cr13 (tens) */
sbit SEL_0 = 0x93; /* active low pushbutton inputs used to */
sbit SEL_1 = 0x94; /* select the display mode */
sbit INTR = 0x95; /* */
sbit UPDATE = 0x97; /* flag set when time to update display */

/*
    define memory variables. note memory variable names are lower case
*/
word cfm; /* gas flow in tenths of a cfm */
word setpoint; /* relay setpoint in tenths of a cfm */
word degree_c /* temperature in tenths centigrade */
data l_word corr; /* intermediate calculation value */
word psi; /* pressure in tenths of a psi */
data display0; /* variables to hold values for the */
data display1; /* displays during refresh. */
data display2; /* display0=status LEDs, display1=CR15, */
data display3; /* display2=CR14, display3=CR13 */
data disp_pntr; /* pointer to next display to enable */
data refresh; /* counter determines display updates */
data high; /* bits 16 - 23 of flow period */
data middle; /* bits 8 - 15 of flow period */
data low; /* bits 0 - 7 of flow period */
data ticks; /* incremented by timer overflow */

/*
    the program consists of four interrupt handlers (multiplex,
    read_switch, overflow, calc_cfm) and a main program.
    multiplex - refresh the seven-segment and discrete status LEDs
    read_switch - signal periodic pushbutton sampling and display update
    overflow - accumulate high order bits of time between tach pulses
    calc_cfm - accumulate low order bits of time between tach pulses
    main - calc airflow, control relay, sample pushbuttons, update display
*/

/*
    multiplex -
    use the free-running I timer to multiplex the seven-segment and
    discrete leds at approx. 1000 hz.
*/

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

void multiplex () interrupt 3
{
    switch (disp_pntr)
    {
        case 0x00:
            STROBE_3 = FALSE; /* turn off display cr13 */
            P3 = 0xff; /* turn off all segments */
            P3 = display0; /* load segments for led's */
            STROBE_0 = TRUE; /* turn on status led's */
            disp_pntr = 1; /* increment ptr to display */
            break;
        case 0x01:
            STROBE_0 = FALSE; /* turn off status led's */
            P3 = 0xff; /* turn off all segments */
            P3 = display1; /* load segments for tenths */
            STROBE_1 = TRUE; /* turn on display cr15 */
            disp_pntr = 2; /* increment ptr to display */
            break;
        case 0x02:
            STROBE_1 = FALSE; /* turn off display cr15 */
            P3 = 0xff; /* turn off all segments */
            P3 = display2; /* load segments for units */
            STROBE_2 = TRUE; /* turn on display cr14 */
            disp_pntr = 3; /* increment ptr to display */
            break;
        case 0x03:
            STROBE_2 = FALSE; /* turn off display cr14 */
            P3 = 0xff; /* turn off all segments */
            P3 = display3; /* load segments for tens */
            STROBE_3 = TRUE; /* turn on display cr13 */
            disp_pntr = 0; /* increment ptr to display */
    }
}

/*
read_switch -
use the free running pwm prescaler to generate
interrupts at 92 hz. every 32nd interrupt set
the UPDATE flag which causes main () to sample
the pushbuttons and update the led displays.
*/

void read_switch () interrupt 6
{
    if (refresh++ == 32)
    {
        UPDATE = TRUE;
        refresh = 0;
    }
}

/*
overflow -
whenever time0 overflows (from 0xffff to 0x0000)
increment the variable 'ticks' which accumulates the
highest order (16 - 23) bits of the gas flow period
in microseconds. if the variable 'ticks' is greater
than the period corresponding to a flow of < 0.1 cfm
then set the NO_FLOW flag which causes main () to
display '00.0'
*/

void overflow () interrupt 1
{
    if (++ticks > LOWEST_CFM)
    {
        cfm = 0;
        ticks = 0;
        NO_FLOW = TRUE;
    }
}

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

/*
    calc_cfm -
    an external interrupt (int0) generated by a tach
    pulse from the flowmeter transfers the current value
    of timer0 into variables 'low' and 'middle', and then
    resets the timers. the 'ticks' variable described
    above is also copied to variable 'high', and then
    reset to zero. the NO_FLOW flag is cleared to
    enable display by main () of the calculated cfm.
*/

void calc_cfm () interrupt 0
{
    low = TLO;
    TLO = 0;
    middle = TH0;
    TH0 = 0;
    high = ticks;
    ticks = 0;
    NO_FLOW = FALSE;
}

/*
    main -
    after initializing pins and variables, enter a continuous loop to...
    - calculate the airflow based on the tach, temp and pressure inputs.
    - compare the airflow to the setpoint input, and control the relay.
    - if the UPDATE flag is set (by the read_switch interrupt handler),
      sample the pushbuttons and update the display data.
*/

void main ()
{
    RELAY      = 0;          /* initialize output pins */
    INTR       = 1;
    UPDATE     = 1;
    STROBE_0   = 0;
    STROBE_1   = 0;
    STROBE_2   = 0;
    STROBE_3   = 0;
    NO_FLOW    = 0;
    I2CFG      = FREERUN_I; /* enable I timer to run, no i2c */
    RTL        = 0;        /* timer 0 period 0x10000 u_seconds */
    RTH        = 0;
    PWMP       = 255;      /* pwm timer interrupt at 923 hz */
    TR         = 1;        /* enable timer 0 */
    ITO        = 1;        /* INT0 is edge active */
    ticks      = 0;        /* initialize variables */
    cfm        = 0;
    low        = 0;
    middle     = 0;
    high       = 0;
    degree_c   = 250;      /* 25.0 tenths degrees c */
    psi        = 147;      /* 14.7 tenths psi */
    corr       = 0;
    refresh    = 0;
    disp_pntr  = 0;
    IE         = 0xab;     /* enable interrupts */

    /*
    main execution loop, executes forever.
    */

    while(1)
    {

```


Airflow measurement using the 83/87C752 and "C"

AN429

```

*/
        calculate base cfm rate - first create long word representing
        flow rate period in microseconds. then subtract the time
        overhead in servicing the routine 'calc_cfm'. then divide the
        period into the period for 1/10 cfm, to get flow rate in 1/10
        cfm resolution.

*/
        corr = high * 0x10000L;
        corr += (middle * 0x100L);
        corr += low;
        corr -= CORRECTION;
        corr = ONE_TENTH_CFM / corr;

/*
        read temperature - measure output from the LM35 sensor,
        scaled by the AMP-02. the scaling results in a range
        of 0 to 51.0 degrees centigrade, in 0.2 degree steps.

*/
        ADCON = START_ADC1;
        while (ADCON & ADCS) ;
        degree_c = ADAT;
        degree_c *= 2;

*/
        compensate cfm rate for temperature - convert temperature
        into degrees kelvin, then divide it into the measured flow
        rate multiplied by the calibration temperature of the flow-
        meter in degrees kelvin. (nominal 25 degrees centigrade)

*/
        corr *= STD_TEMP;
        corr /= (ZERO_K + degree_c);

*/
        read pressure - measure output of the KP100A pressure trans-
        ducer, scaled by the AMP_02. the scaling results in a range
        of 0 to 25.5 psi, in 1/10 psi steps.

*/
        ADCON = START_ADC0;
        while (ADCON & ADCS) ;
        psi = ADAT;

*/
        compensate cfm rate for pressure - multiply measured pres-
        sure and the calculated flow rate, and then divide it by
        the standard atmospheric pressure at sea-level. (nominal
        14.7 psi)

        corr *= psi;
        corr /= STD_ATM;
        cfm = corr;

*/
        read setpoint pot to obtain setpoint in the range of
        0 - 25.5 cfm in 1/10 cfm steps.

*/
        ADCON = START_ADC2;
        while (ADCON & ADCS) ;
        setpoint = ADAT;

*/

```

Airflow measurement using the 83/87C752 and "C"

AN429

```

test if cfm rate greater or equal to the
setpoint, and if so then energize relay
*/

    if (setpoint > cfm)
        RELAY = 0;
    else
        RELAY = 1;

*/

test if UPDATE flag has been set, and if so reset flag.
*/

    if (UPDATE)
    {
        UPDATE = 0;
    }

*/

then test is the NO_FLOW flag has been set. if so then
display '00.0' cfm
*/

    if (NO_FLOW)
    {
        display0 = ~CFM;
        display1 = ~segments[0];
        display2 = ~(segments[0] | SEG_DP);
        display3 = ~segments[0];
    }

*/

if the NO_FLOW flag was not set then read the display
select pushbuttons, and display the appropriate data.
*/

    else if (SEL_0)
    {
        if (SEL_1)
        {

*/

if no pushbutton is depressed then the default display is
the flow rate in cfm. if the flowrate is greater than
or equal to 30 cfm then display the overrange message
'EEE', otherwise display the flow in 'XX.X' format.
*/

        if (cfm <= 300)
        {
            display0 = ~CFM;
            display1 = ~segments[cfm % 10];
            cfm /= 10;
            display2 = !(segments[cfm % 10]);
            cfm /= 10;
            display3 = ~segments [cfm % 10];
        }

        else
        {
            display0 = ~CFM;
            display1 = ~segments[10];
            display2 = ~segments[10];
            display3 = ~segments[10];
        }
    }
}

```

Airflow measurement using the 83/87C752 and "C"

AN429

```
*/
    if the temp pushbutton (SW1) is pressed then display the air temperature.
*/

    else
    {
        display0 = ~DEGREES;
        display1 = ~segments[degree_c % 10];
        degree_c /= 10;
        display2 = ~(segments[degree_c % 10] | SEG_DP);
        degree_c /= 10;
        display3 = ~segments[degree_c % 10];
    }
    }
    else
    {

*/
    if the psi pushbutton (SW2) is pressed then display the air pressure.
*/

        if(SEL_1)
        {
            display0 = ~PSI;
            display1 = ~segments[psi % 10];
            psi /= 10;
            display2 = ~(segments[psi % 10] | SEG_DP) ;
            psi /= 10;
            display3 = ~segments[psi % 10] ;
        }

*/
    if the setpoint pushbutton (SW3) is pressed then display the setpoint.
*/

        else
        {
            display0 = ~SETPOINT;
            display1 = ~segments[setpoint % 10] ;
            setpoint /= 10;
            display2 = ~(segments[setpoint % 10] | SEG_DP) ;
            setpoint /= 10;
            display3 = ~segments[setpoint % 10] ;
        }
    }
}
}
```

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

INTRODUCTION

With conventional microprocessor based systems, the market was primarily concerned with performance, cost and features. With the advent of hand-held and portable computers, the prominent market requirements focus on size, weight and battery life.

Given a mature 386SX/AT architecture that provides more than adequate performance for average notebook application usage, the design challenges for these machines revolve around developing low power systems that maximize battery usage.

The features of a notebook PC are usually characterized as weight and battery life. The heaviest component of a PC is usually the battery and the choice of battery is dictated by the power required. Thus the performance of the power management scheme has a direct bearing on both these parameters.

The battery life targets for notebook machines is around 5 hours (the air commute time from coast to coast USA).

OVERVIEW

Most of the power consumed by a fully powered PC is wasted. The hard disk spins constantly even though data transfers from the disk are very sporadic. PCs may sit unattended for periods where the user is distracted by a telephone call, for instance.

There are two principle challenges in designing a power management system: the ability to power down various devices without affecting other devices on the same bus, and ensuring full compatibility with existing operating systems and applications.

The largest user of power in a PC is the display sub-system (3–5 Watts) followed by the peripherals such as the hard disk (2–4 Watts), the main system memory (0.5–1.5 Watts), and the core logic (1 Watt).

INTEGRATED POWER MANAGEMENT

The conventional PC architecture needs to be extended to support power management. Hardware needs to be added to provide power-down capabilities and software needs to be added to support the hardware and provide DOS compatibility. The software support is usually realized in the BIOS. The hardware support can be implemented with external circuitry. This external circuitry manages the power resources to individual sub-sections of the PC system as these

resources dictate. The external circuitry monitors battery power, system activities and timed events.

Conventionally, the external circuitry is comprised of a digital power management ASIC and associated components. The use of this part increases the chip count of the PC system.

The power management system has to determine how the system resources are being used. The resource usage of a PC can be determined by monitoring events or activities. User activity is usually determined by monitoring the keyboard controller for keystroke events. Keystroke events can be indicated by interrupts to the PC core logic or IO reads to the keyboard controller location.

The power management ASIC solutions on the market today, such as the VADEM/INTEL 82C347, VLSI VL82C312 and INTEL 80C386SL all require external analog support circuitry to completely implement the power management functions. For example, low battery detect is implemented by the use of external comparator chains and complex, close tolerance level detect circuitry. The cost of this external circuitry is usually a significant proportion of the overall cost of the power management solution. The 83/87C752 employs an internal analog-to-digital converter (ADC). The ADC can be used to implement the battery level detection function at no extra cost and with no extra support circuitry.

The 83/87C752 is a member of the Philips 8051 family of high performance 8-bit microcontrollers. These processors have been optimized for sequential real time control applications. The 83/87C752 contains most of the features of the 80C51 and has the following features:

- 2k bytes ROM
- 64 bytes RAM
- Single level interrupt structure
- 16 bit programmable counter/timer
- Two 8-bit and one 5-bit bi-directional IO ports
- I²C serial interface
- PWM with interrupt and overflow capability
- 5 channels of 8-bit A/D
- 28-pin DIP and PLCC.

FLEXIBILITY

ASIC solutions to power management offer rigid schemes which work adequately with a few notebook architectures, but rarely offer

exactly what the designer requires. With the current competitive arena for laptop development, time-to-market and value added features have a significant impact on the sales success of a particular product. The 83/87C752 offers flexibility at a low price. The power management design requirements can be coded and configured in the controller software and One Time Programmable (OTP) devices can offer a quick low-cost implementation of the coded scheme.

With these integrated functions that the 83/87C752 offers, and its ability to provide a complete solution to power resource control, this device is emerging to be the industry standard for power management.

TOPOLOGY

Figure 1 shows a block diagram of a typical system implementation. It employs the integrated power management scheme using the Philips microcontroller to handle the keyboard and power management functions. The CPU and coprocessor reside on the local bus with the system memory. A local bus controller monitors CPU bus cycles to see if they are memory or ISA cycles. It also integrates the interrupt and DMA functions. The ISA bus controller processes non-system memory bus cycles. A frequency generator is used to provide the system clocks and clock multiplexing. The peripheral controller integrates the communications and mass storage sub-system. The VGA sub-system shares the ISA bus with the peripheral controller. The VGA controller has associated VGA memory.

Figure 2 shows the microcontroller with the external support devices. The frequency generator provides the system clocks. It must have the ability to change the frequency of the clocks without violating the minimum high or low times for the core logic. The Philips microcontroller can control the speed of the system clocks via frequency select pins on the frequency generator. Frequency generators such as the Avasem AV9127 can change the processor clock speed gradually and continuously without violating the minimum high or low times.

The integrated controller monitors system activity via its digital input ports. It uses internal timers to time the intervals between activity. The power to the VGA and peripheral sub-systems is controlled by the digital output port pins via MOSFETs.

Battery level and V_{CC} is monitored by the onboard A/D converter.

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

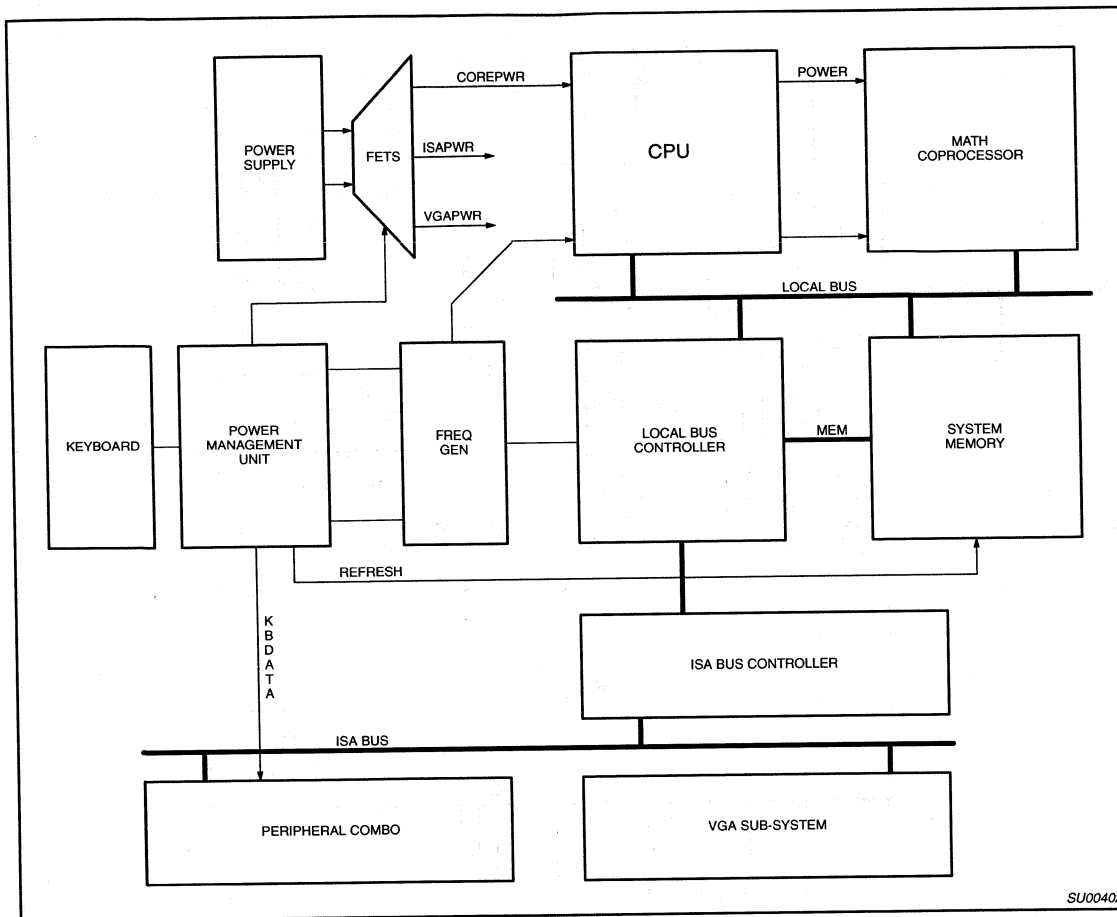


Figure 1. Integrated Power Management Scheme

"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

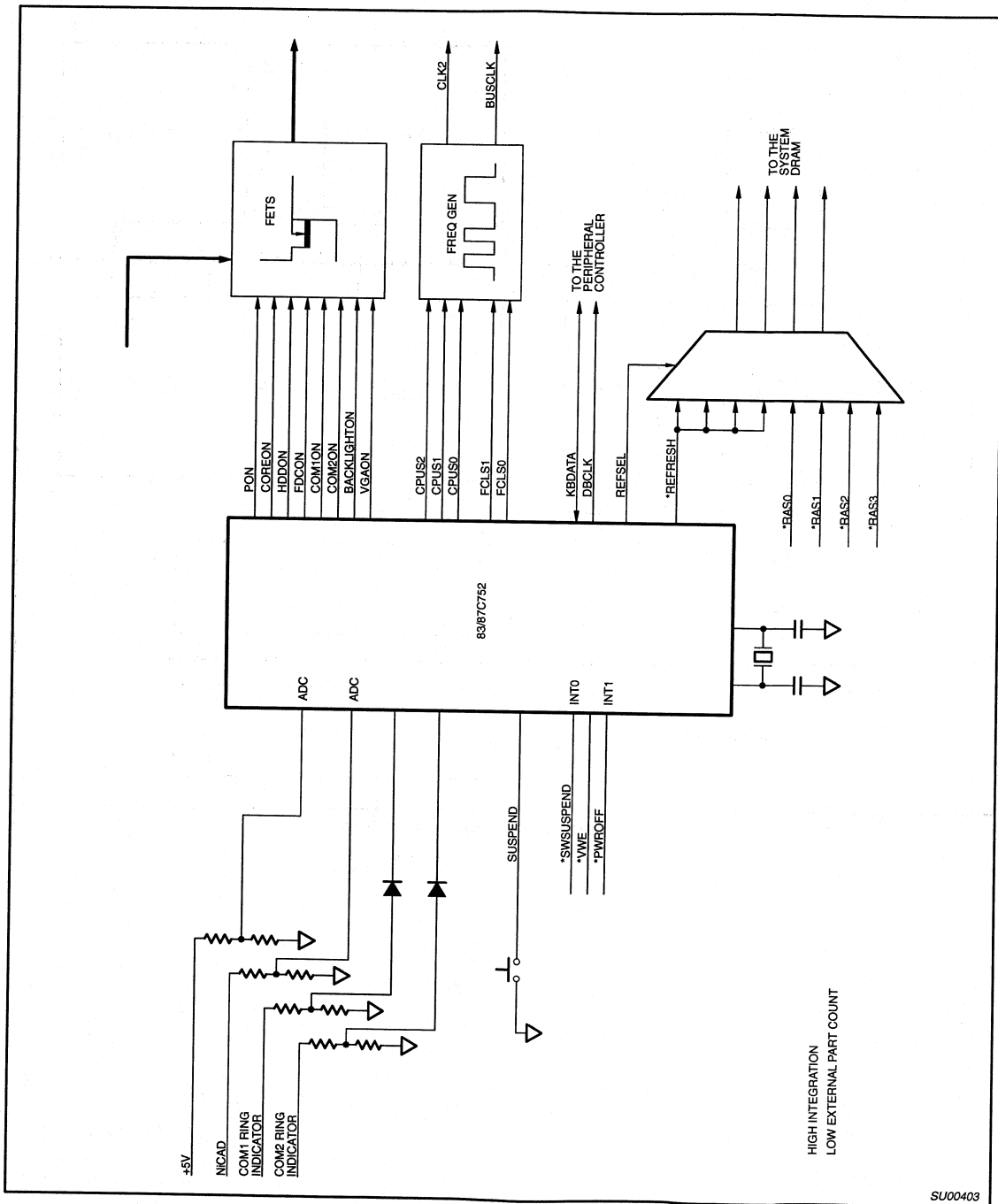


Figure 2. External Support Devices

SU00403

"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

OPERATION

The power management system operates like a state machine. Transitions from state to state are controlled by expiring timers which are retriggered by external events. On entering a state, external power is switched or clocks are modified. A typical power management system would employ six states: Full Power, Doze, Shutdown, Sleep, Suspend, and Off.

The state diagram of Figure 3 shows the power management states and their interrelationships.

Full Power

Entry into this state is controlled by a transition of the ON/OFF switch. In this state all the power control outputs are asserted, and the clock generator is selected for the highest speed. The system runs at full speed and power.

Doze

This state is entered from the FULL POWER state. Entry into this state is controlled by an expired timer (typically 30 secs). The timer expired as a result of not being reloaded by a transition on an activity monitor input pin. In this state the frequency generator is instructed to reduce the clock speed to about half that of the previous state.

Shutdown

This state is also entered from the FULL POWER state and operates in parallel with the DOZE state. Entry into this state is controlled by an expired timer (typically 30 secs). The timer expired as a result of not being reloaded by a transition on an activity monitor pin. In this state the power to a particular peripheral or group of peripherals is removed via an external FET.

Shutdown-Doze

This is an intermediate state which implements the features of both the SHUTDOWN and DOZE states. Entry into this state from the DOZE state is controlled by an expired timer (typically 30 secs). The timer expired as a result of not being reloaded by a transition on an activity monitor pin. Entry into this state from the SHUTDOWN state is controlled by an expired timer also. The timer expired as a result of not being reloaded by a transition on an activity monitor input pin. In this state the power to a particular peripheral or group of peripherals is removed via an external FET and the frequency generator is instructed to reduce the clock speed to about half that of the FULL POWER state.

Sleep

This state is entered from either the DOZE state or SHUTDOWN state. Entry into this state is controlled by an expired timer (typically 30 secs). The length of this timer is usually longer than that employed in the DOZE or SHUTDOWN states. The timer expired as a result of not being reloaded by a transition on an activity monitor pin. The activity monitor may look for keystrokes or video activity as described below. In this state power is removed from the backlight and LCD modulation voltage regulator via external FETs.

Suspend

This state is entered from any of the above states. Entry into this state is controlled by a transition on an external suspend switch or a command from the BIOS. During this state, the microcontroller takes over the task of refreshing the system memory and removes the power from the rest of the system via external FETs.

Off

This state is entered from any of the above states. Entry into this state is controlled by a transition on an external switch or a command from the BIOS.

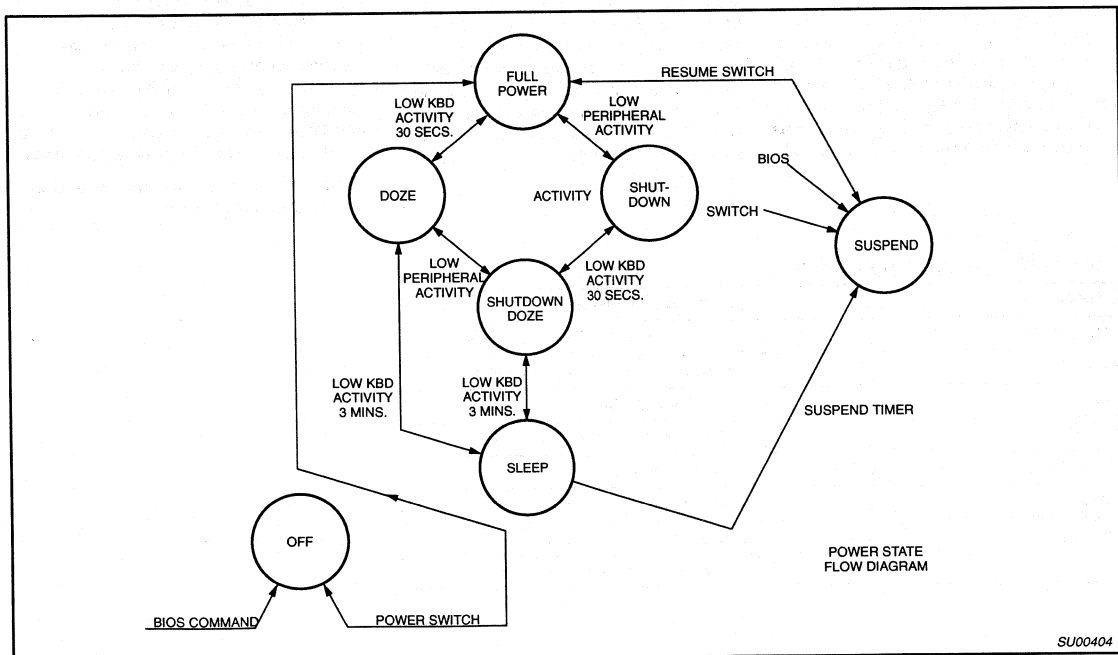


Figure 3. Multi-State Power Management Scheme

"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

POWER MANAGEMENT ELEMENTS

Figure 4 shows the internal power management elements of the Philips controller. The Activity monitors contain combinatorial algorithms to monitor or poll an activity or combination of activities. The activity monitors reload programmable timers which toggle clock control and power control output pins.

Each functional block is discussed in more detail below.

Power Outputs

The power outputs are logic level signals that control MOSFETs via level shifting circuitry.

The MOSFETs switch power to the various blocks under power management control and are chosen to have a low R_{ds} on which reduces the voltage drop across the DRAIN-SOURCE channel.

The level shifting circuitry and MOSFET orientation is shown in Figure 5.

C1 and R1 control the switchon edge and should be chosen to make the edge sufficiently slow to minimize the inrush current. This is necessary where the notebook computer employs solid chip tantalum capacitors which fail short-circuit on switchon current transients.

R1 pulls the gate to the NiCAD supply rail. The NiCAD battery voltage is usually greater than +12 Volts. We can exploit this relatively high voltage to turn the MOSFETs hard on and further reduce the R_{ds} on. The NPN transistor is operated as a cascode and gives no net inversion between the logic level and

the state of the MOSFET. This is necessary to handle the default powerup mode of the port pins.

For a lower performance and cost reduced system, a logic level FET can be used such as a MTM25N06L and driven directly from the microcontroller port. These logic level FETs usually have R_{ds} on specifications in the region of 100 milliohms. The finite drain-source resistance implies that a small amount of power is wasted in this channel while power is applied to the switched group of devices.

Clock Control

The clock control module controls the speed of the system clocks. Where the notebook system has a synchronous ISA clock, it is derived directly from CLK2, the processor clock. The clock control outputs can be fed directly to a frequency generator such as an AVASEM AV9127 where pins are committed to encode a frequency select scheme. The scheme employs two programmable clock generators; one with eight preset frequencies used for the system clock; and the other with four preset frequencies used for the mass storage subsystem. Figure 6 shows the interconnection between the clock control port and frequency generator.

By changing the assignments of the encoded select lines, the system clock frequency can be reduced. Frequency generators employ analog voltage controlled oscillators which, when instructed to change frequency, will steadily and gradually change frequency in a smooth transition. This scheme does not violate the minimum high and low times for the core logic devices.

Timers

The timers should run independently of the keyboard scanning function. The timers are used as timeouts for a combination of external events or activities. The timers are constructed of reloadable timers and reloaded by transitions or conditions on external events. A typical timeout period is between 1 and 4 minutes, therefore the timeout must be constructed from both timer hardware and support software. Figure 7 shows the interrelationships between hardware and software. The software must record the instances of timeout cycles. If the number of cycles is allowed to reach a predetermined number, the timeout elapses and the assigned power control output is negated, or the clock control outputs proceed to the next state. The count of the number of cycles is reset by a command from the activity monitor.

The activity monitor asserts flags during the background and interrupt tasks. The timer software processes these flags to determine the state of the timeout. The software uses a count variable to measure the instances of the timer elapsing and a flag to determine whether activity has occurred.

Activity Monitor

The activity monitor sets the activity flags for the timers. The monitors contain combinatorial elements which poll an external activity or a number of external activities. External activity can be detected by transitions or levels on input port pins. The interrupt pins would be better suited for transition detect while the general input ports could be used to poll for external conditions.

There are several key activity indicators on the PC. See Table 1 below.

Table 1. Key Activity Indicators

SIGNAL	TYPE	DEVICE/PERIPHERAL
*IDECs1	LEVEL	Hard Disk IDE interface chip select
*IDESC0	LEVEL	Hard Disk IDE interface chip select
**VWE	EDGE	VGA Memory Write enable signal
*FDCS	LEVEL	Floppy disk digital control register
*LPTRDY	LEVEL	Parallel printer interface flag
*GPRD	EDGE	Accessory interface select
*53C90SEL	EDGE	SCSI interface chip select
*LIDSW	LEVEL	Notebook lid switch

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

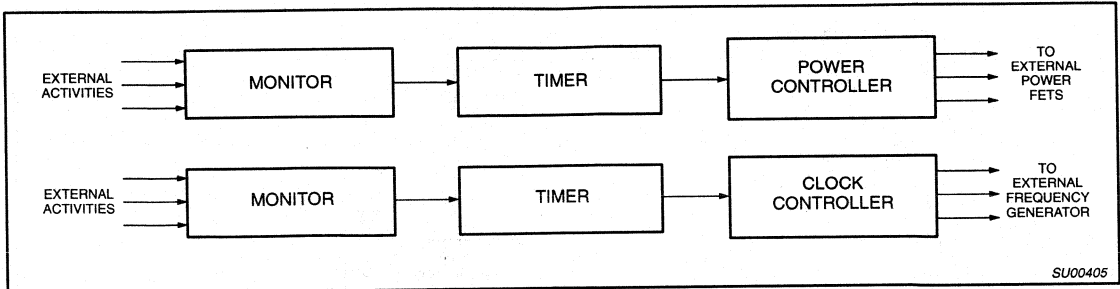


Figure 4. Internal Power Management Elements

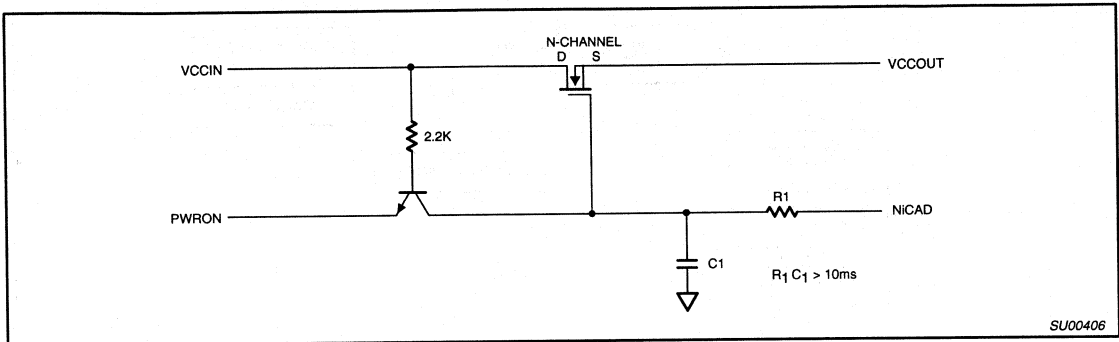


Figure 5. Power Control Circuitry

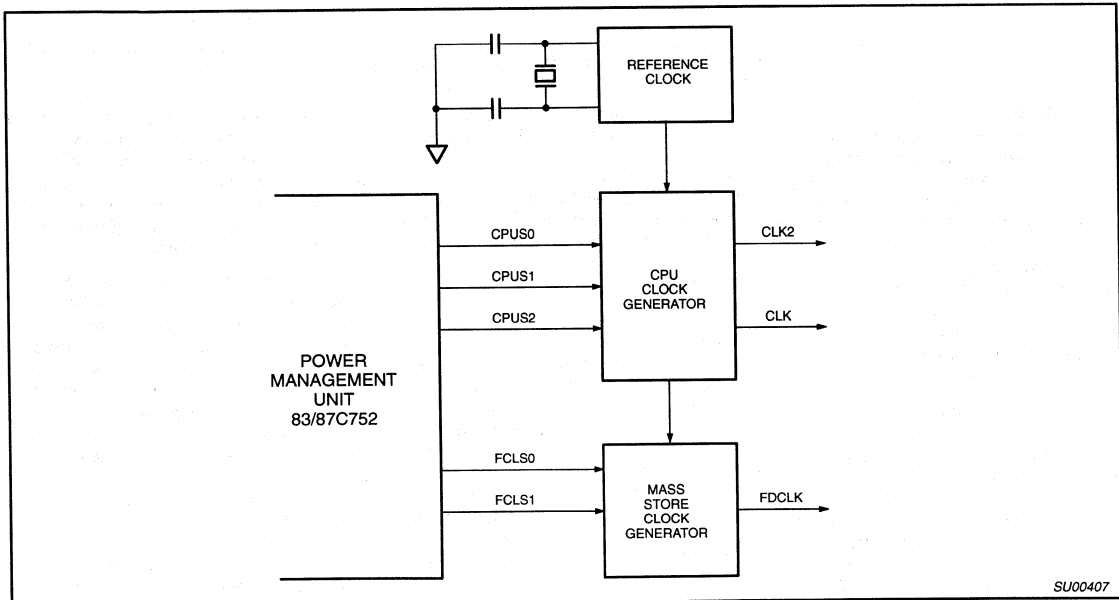


Figure 6. Clock Control Circuitry

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

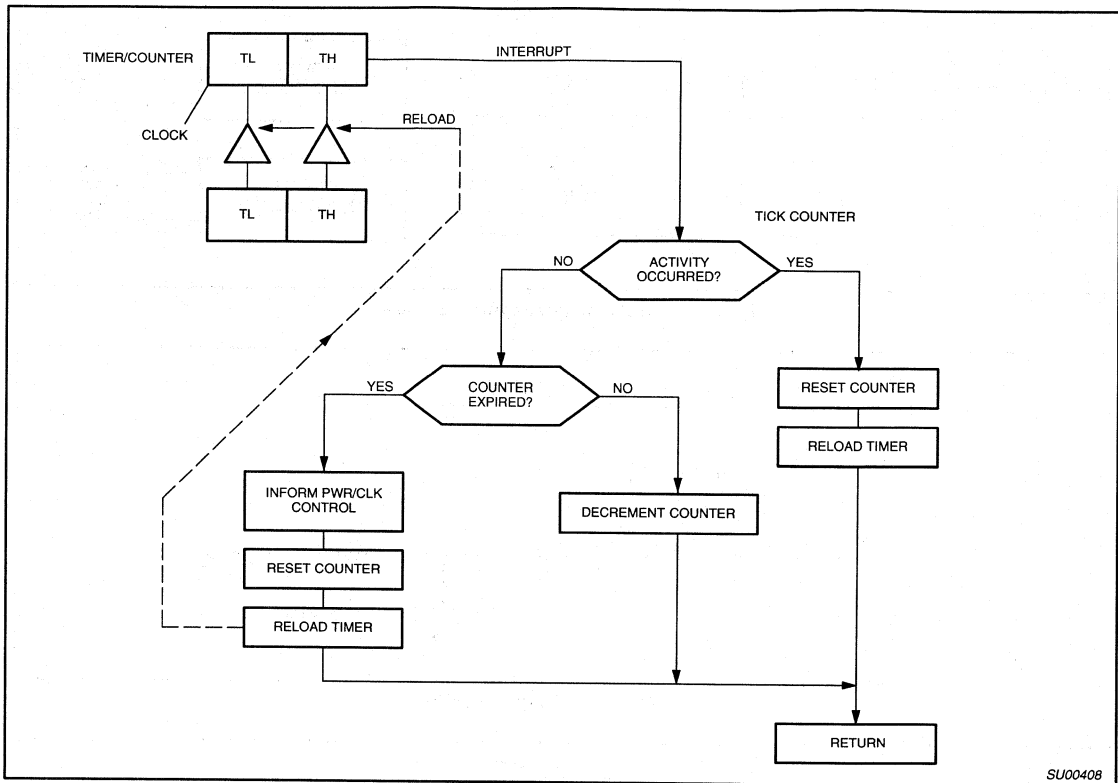


Figure 7. Hardware-Software Interrelationships

SU00408

Suspend Control

During the suspend state, power is removed from all the system devices except the power management controller and system memory. Before the system is allowed to enter the suspend state, the BIOS empties all the DMA holding registers, makes a copy of the CPU registers and stack pointer into reserved main memory. The VGA memory is also copied into main memory. Once this transaction has been completed, the system BIOS instructs the power management controller to enter the suspend state.

At the start of the suspend state, the microcontroller takes over the memory refresh. This is achieved with the use of an external refresh mux. The mux is switched over to the microcontroller refresh lines when REFSEL is asserted. Switchover must only be executed after the system memory controller has performed a complete refresh cycle (512 / 1024 rows). The external multiplexer asserts all the individual *CAS lines while switched over to the controller.

The *RAS lines are all driven by a single *REFRESH command from the controller.

Battery Monitor

The ADC can be used to monitor the battery condition via an external resistor divider. This monitor can be used to assess the condition of the system batteries during system use and while being charged.

A simple “minus delta V” algorithm may be implemented to measure the slope of the battery voltage over time. The charging curve of Figure 8 shows the characteristic of the voltage across the NiCAD cells with a constant current applied. As the NiCADs become charged, the internal temperature of the cells rises and thus their internal resistance increases. This results in a droop of the voltage across the cells. The charging current must be terminated when a droop of greater than 50mV per cell over 1 second is detected.

This minus delta V algorithm can be implemented in the microcontroller software.

The analog to digital controller can be used to sample the NiCAD battery voltage level at regular intervals. The sample is compared against the last sample for a negative result. If the magnitude of the result is greater than 50mV per cell, then the charging circuit must be given a command to switch off.

A battery condition detector can also be implemented in the microcontroller. This detector can be used to give an early indication of an exhausted battery. When a NiCAD cell reaches its supply capacity, the voltage across the cell drops rapidly, thus the system supply can only sustain the core logic for a matter of seconds. The battery condition monitor should give the system and user an early indication of this condition. This can be implemented by monitoring the battery voltage. When a preset slope or level is reached or exceeded, the controller can issue an NMI to the core logic. The core logic can execute a shutdown routine which saves the state of the machine on the hard disk and preserves the integrity of the user’s data.

"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

PERFORMANCE

Table 2 shows the power performance of the 83/87C752 in a typical notebook system.

During each power management state, the core logic system and peripherals demand lower and lower power as the clock speeds are reduced, resets are asserted and power is removed from the device or peripheral.

During the suspend mode, power is completely removed from the core logic devices and peripherals, only the DRAMs remain energized so that the system state can be restored to the next post suspend cycle on the detection of an activity.

The FULL-ON figures are maximums, in reality, the processor executes sporadic scripts and then idles in tight loops awaiting IO. This means that the actual power required by the system under normal conditions will be lower than that estimated in that column of the table.

Table 2.

DEVICE	STATE	FULL-ON POWER (W) (MAX)	SHUTDOWN-DOZE (W)	SLEEP POWER (W)	SUSPEND POWER (W)
Am386SX		2.14	0.7	0.5	0
DRAM + Controller		1.7	0.6	0.4	0.025
Local bus controller		0.3	0.3	0.2	0
ISA bus controller		0.2	0.18	0.07	0
87C752		0.08	0.08	0.08	0.015
BIOS ROM		0.11	0.002	0.002	0
Floppy drive		3.1	0.035	0.035	0
IDE drive		3.3	0.68	0.68	0
VGA controller		2.1	1.6	1.2	0.015
Keyboard controller		0.7	0.6	0.4	0
Keyboard		0.15	0.15	0.15	0
Oscillators		0.1	0.1	0.1	0
RS232 buffers		0.11	0.002	0.002	0
LCD panel		0.7	0.62	0.2	0
Backlight		1.5	1.5	0.1	0
TOTALS		15.69	6.709	4.119	0.055

The gradient of power performance across the states is quite steep, especially when transitioning from the sleep to the suspend states.

Table 3.

	FULL-ON	1/2 CLK2 SHUTDOWN-DOZE	1/4 CLK2 SLEEP	SUSPEND
CPU	ON	ON	ON	OFF
DRAM	ON	ON	ON	ON
Local bus	ON	ON	ON	OFF
ISA bus	ON	ON	OFF	OFF
87C752	ON	ON	ON	ON
BIOS ROM	ON	OFF	OFF	OFF
Floppy	ON	ON	OFF	OFF
IDE drive	ON	IDLE	OFF	OFF
VGA	ON	ON	ON	OFF
Keyboard controller	ON	ON	ON	ON
Keyboard	ON	ON	ON	OFF
Oscillators	ON	ON	ON	OFF
RS232 buffers	ON	OFF	OFF	OFF
LCD panel	ON	ON	OFF	OFF
Backlight	ON	ON	OFF	OFF

Note that the panel and backlight are off during the sleep state.

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

OTHER INTEGRATION OPPORTUNITIES

The 83/87C752 offers a complete power management solution as described above. The functionality and IO capability of this device can be used as a common denominator for other, larger Philips microcontrollers. The 83/87C552 offers the same internal functionality while providing more integration capabilities with its increased IO, memory and timer functions.

An example of an integration opportunity would be to combine the keyboard scanner function with the power management

function. The Philips 83/87C552 microcontroller is ideal for this task. The microcontroller can implement the scanning and code generation schemes associated with the keyboard function, implement the activity monitors, timers and power control scheme required for power management as well as provide an integrated solution to battery condition detection by exploiting the onboard A/D converters.

The PC keyboard scanner is traditionally an 8051 microcontroller. The keyboard scanning and code generation can be performed by an 8051 running at 6MHz. A 12 or 16MHz

83/87C552 microcontroller would have the bandwidth to take on other tasks.

Figure 9 shows the 83/87C552 integrated as a power management unit and keyboard scanner.

Other members of the Philips family of 8051 derivative microcontrollers can also provide an integrated solution to power management (see Table 4).

As can be seen, the 83/87C550 provides an intermediate solution to the 83/87C552 and 83/87C752. It has more memory and IO than the 83/87C752 and has three more ADC channels.

Table 4. Microcontrollers with A/D

DEVICE	ROM	RAM	A/D	I/O	PWM	TIMERS
83/87C550	4k	128	8 8-bit	24	0	2
83/87C552	8k	256	8 10-bit	48	2	3
83/87C752	2k	64	5 8-bit	21	1	1

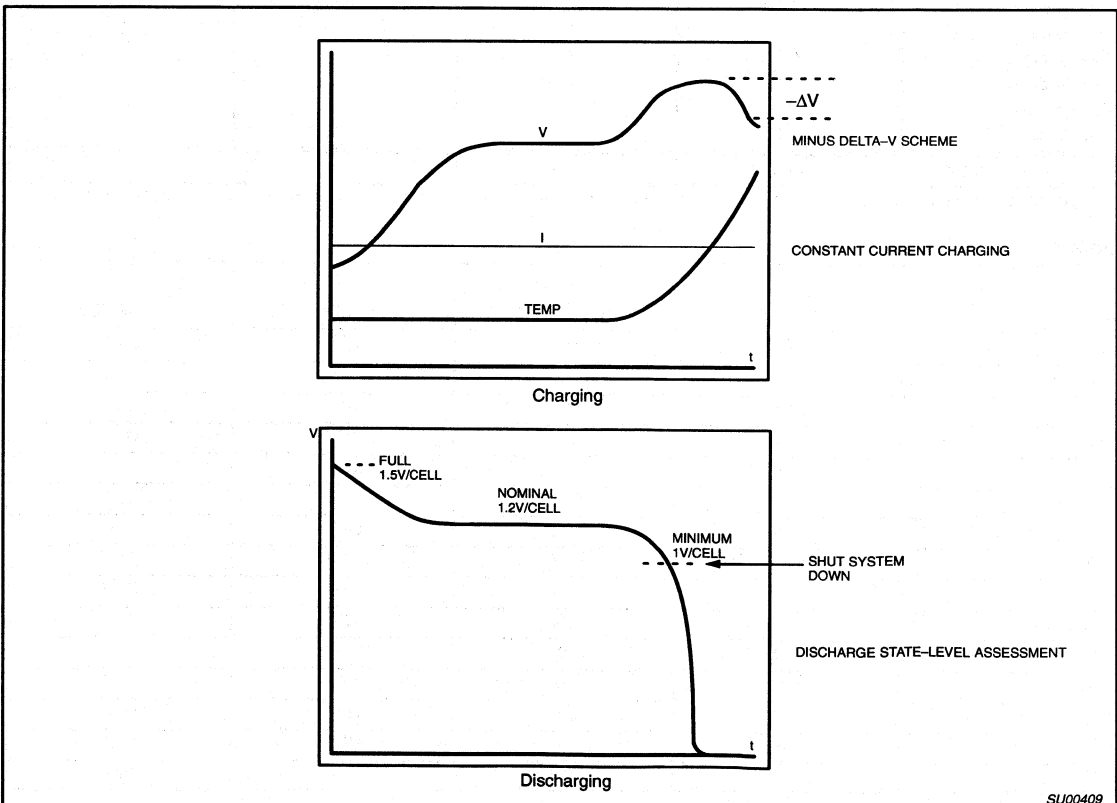


Figure 8. Battery Characteristics

SU00409

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

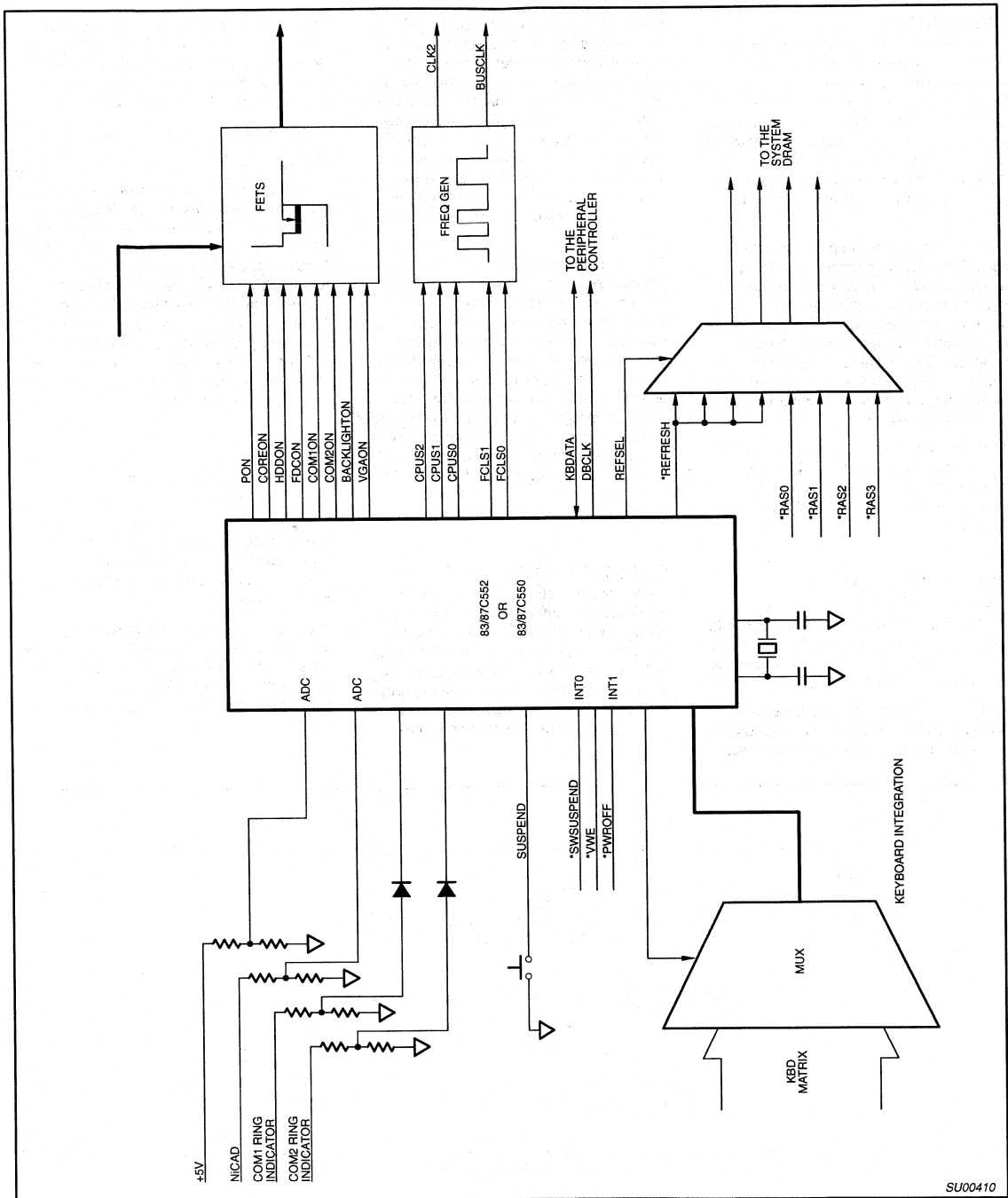


Figure 9. Further Integration Opportunities

SU00410

87C751 fast NiCad charger

AN439

DESCRIPTION

This application note describes a portable standalone, automatic constant current NiCad battery charger using the Philips Semiconductors 87C751 microcontroller. This unit will fast charge NiCad batteries from a 12V source such as an automobile battery or a DC power supply. The charge current is 2.5A which is suitable for charging NiCad cells of 1200mAh capacity in little over 1/2 hour.

Use of a microcontroller provides complete flexibility of design parameters such as the number of cells, their capacity and charge rate requirements. A number of key microcontroller techniques described in this application note are a state machine, a low-cost, high-resolution single slope analog-to-digital converter comprised of the microcontroller and a comparator as well as an analog control system.

As shown in the block diagram (Figure 10), the charger consists of the 87C751, a switching charge current regulator, an analog-to-digital converter and some LED indicators.

DESIGN OBJECTIVES

The goal of this design is to achieve a maximum charge rate without damage to the NiCad cells. To determine the requirements for such a charger, we need to examine the most important characteristics of NiCad cells.

As a NiCad cell charges, gas bubbles are released from the electrolyte and accumulate on the plates, reducing the effective plate

area and increasing cell impedance. When the cell gets near full charge the rate of gas generation and temperature rise increase, since the charge current produces gas rather than stored charge. At that point the process goes into thermal runaway. The cell pressure rises sharply causing the case to vent. A large enough venting can destroy the cell immediately. If the venting is of a lesser magnitude, as would be the case with a relatively low charge current, the cell capacity is reduced.

The standard technique is to charge the cells at such low current that there is no risk of thermal runaway. The electrolyte then reabsorbs the gas bubble at the same rate as they are generated. Usually this implies a charge current of 0.1 times the cell capacity and a charge time of 16 hours. We want to achieve full charge in about half an hour. Slow charging also increases the likelihood of dendrite formation. Dendrites are crystalline fingers that can propagate through the plate separators and short the cell internally. Fast charging tends to clear these shorts before they have a chance to become significant.

CHARGE TERMINATION

There are two methods commonly used for terminating NiCad fast charges: delta-peak voltage detection and delta-temperature detection. This charger uses the delta-peak voltage method. It is simpler to implement, especially if interchangeable battery packs are to be charged, because the temperature sensing method requires a temperature

sensor to be attached to the battery pack. The temperature change in the battery pack depends on how well the pack is thermally coupled to its surroundings, which also makes temperature sensing somewhat tricky.

The voltage on a NiCad pack rises during charging, steeply at first, and then at a lower rate. When the pack is nearly full, the voltage rate of rise increases a little, then falls to zero as the voltage peaks. As the pack goes into over-charge the voltage starts to drop and the internal temperature and pressure rise. A typical voltage drop used for charge termination is 1% of the peak voltage as shown in the battery voltage waveform (Figure 11). This charger uses 1% of the measured peak voltage as the threshold, which means that it can charge any number of cells from 1 to 6 without any external input to select a number of cells.

THE 87C751 MICROCONTROLLER

The Philips Semiconductors 87C751 contains a 2k byte ROM, a 64 byte RAM, 19 I/O lines, a 16 bit auto-reload timer, a five-source fixed-priority interrupt structure, a bidirectional I²C serial bus interface and an on-chip oscillator.

In this application note, the timer is used as a 'tick-timer', scheduling events and measuring periods. The ports are used to control and monitor the external analog circuitry. The software is written in C using the Franklin C compiler.

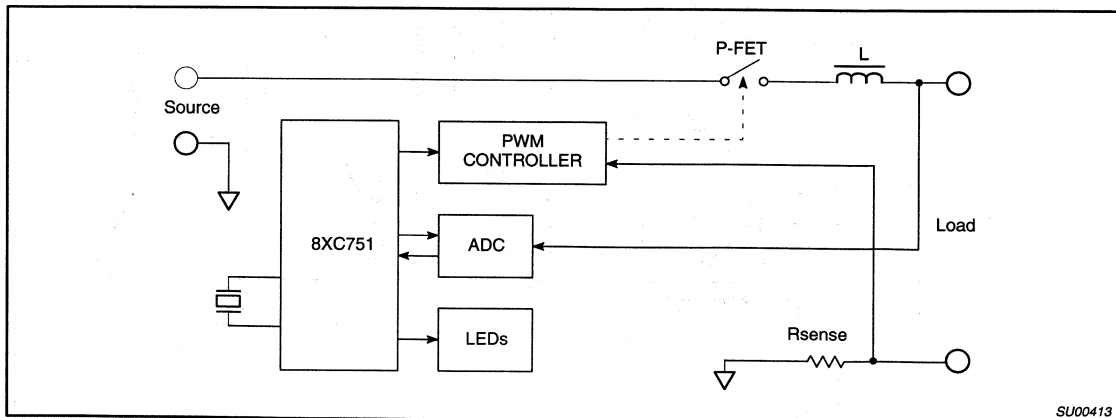


Figure 10. Block Diagram

87C751 fast NiCad charger

AN439

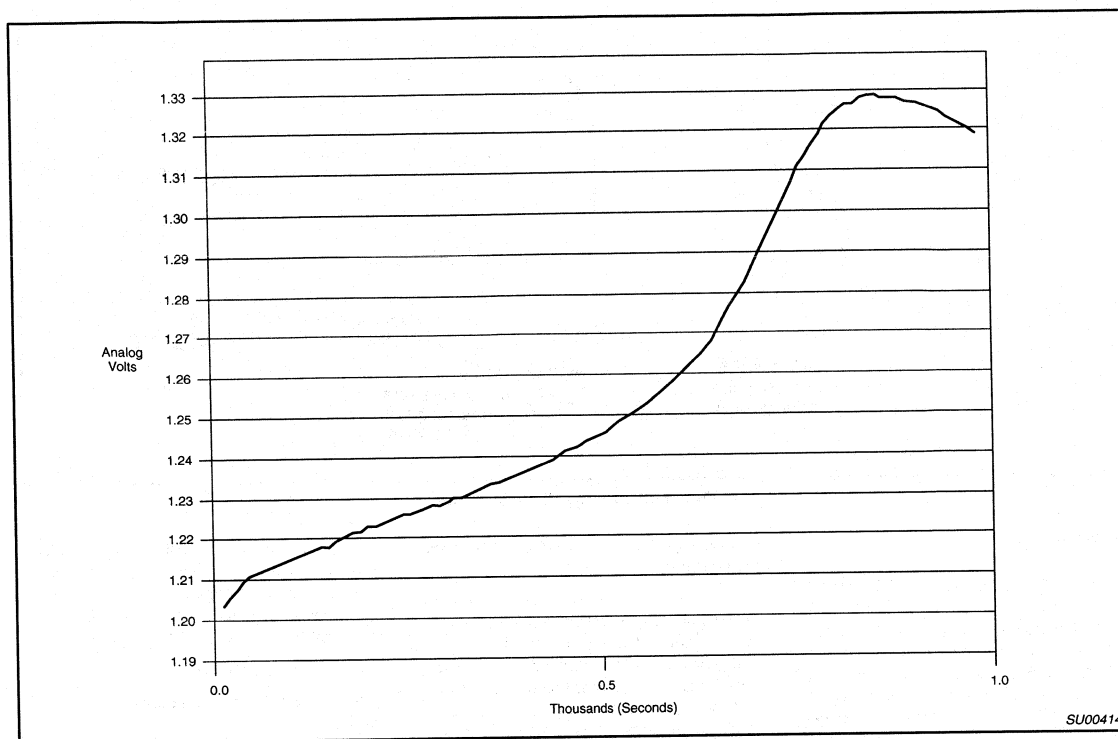


Figure 11. Battery Voltage Waveform During Charging

SOFTWARE

Scheduler

The timer generates events at a fixed interval by updating the variable **tick** in the timer interrupt service routine. The timer is initialized and reloaded with $ffff_{hex}$, which corresponds to a period of 71 milliseconds. The choice of this period is related to the analog-to-digital converter, which is described below.

The variable **tick** is incremented once every 71 ms so that 14 **ticks** make a second. The scheduler calls the **charger** procedure when **tick** equals zero, the **leds** procedure when **tick** equals 7 and it resets **tick** to zero as well as incrementing the **seconds** variable when **tick** reaches 14.

State Machine

The state machine in the **charger** procedure is the heart of the system. The state machine processes the current state and the filtered battery voltage. Figure 12 shows the topology of the state machine. There are four states:

FAULT, INITIALIZING, CHARGING and DONE. The variable state keeps track of the current state. The operation of the states is as follows:

FAULT

This is the default state. When the system powers up and is reset, the state variable is initialized with this state. The filtered battery voltage is checked using the **check_limits** procedure (see Figure 12). If the measured voltage is within a predetermined range, it is assumed that a battery has been connected and the state variable is set to the **INITIALIZING** state. A voltage out of range will always set the state variable to **FAULT** no matter what the current state is.

INITIALIZING

During this state the running average noise rejection filter and other variables used by the charging algorithm are initialized by sampling the battery voltage for a preset number of passes through the state machine without making any decisions about the charging process. The number of passes is defined in

the constant **INITIALIZATION_TIME** which is an integer number of seconds. When the initialization time is over, the state variable is updated to the **CHARGING** state.

CHARGING

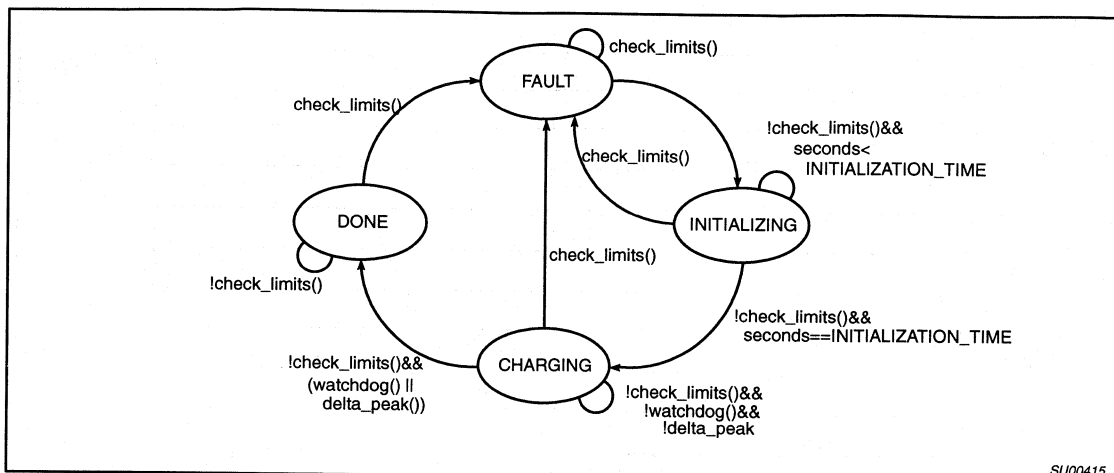
During this state the battery voltage is processed in the **delta_peak** procedure. A watchdog timer is also used to ensure that the battery is not overcharged in the unlikely event that the **delta_peak** procedure misses the termination conditions. Until the battery peaks or the watchdog timer times out, the state machine remains in the **CHARGING** state. If a voltage peak is detected and the voltage drops below the peak by 1%, or the watchdog timer expires, the state variable is updated to the **DONE** state.

DONE

After **MAINTENANCE_PERIOD** seconds, charging current is applied to the battery for 1 second. If the battery voltage falls out of range (i.e., it has been disconnected), the state variable is updated to **FAULT**, the default state.

87C751 fast NiCad charger

AN439



SU00415

Figure 12. Charger State Machine

HARDWARE

Buck Converter Current Source

There are two basic converter topologies from which the others are derived. They are the buck and boost converters. As the names imply, the buck converter produces an output lower than its input and the boost converter does the opposite. The transformer isolated versions of the buck and boost circuits are known as the forward and flyback converters. Since the input supply voltage is greater than the maximum expected output voltage and no isolation is required, the buck topology is a good choice for simplicity and high efficiency.

The required values of inductors and capacitors for a switching converter are inversely proportional to operating frequency, while switching losses are proportional to frequency, so that a compromise is made in the choice of operating frequency. To keep the converter compact and avoid excessive switching losses, the switching frequency was chosen at 100kHz.

The Unitorde UC3843D is a low-cost surface mount current-mode switching power supply controller. In this application, the voltage feedback loop is left open and the controller is driven from its compensation input (pin 1). The UC3843 data sheet shows that this node in the IC is driven by an active pulldown and a fixed 0.5 mA pullup current source. If the voltage feedback, pin 3, is grounded, the internal voltage error amplifier will turn off its

output, allowing control of the node voltage by pulling current out of pin 1.

The ICL7667 is an inverting MOS gate driver. It provides the correct polarity for the mosfet gate signal and its 1.5A peak output improves efficiency by switching the fet quickly. The Amobead is an optional component which reduces EMI at the expense of increased drain voltage swing. The scope traces were taken without the Amobead in place.

Analog-to-Digital Converter

The analog-to-digital converter consists of Q104, C118, R114, U104 as well as R115, R116 and C119 operating in conjunction with the 87C751. In this application a simple, low-cost ADC with relatively high resolution is required, but it does not need linearity, absolute accuracy or long term stability because the detection method is relative to the peak voltage and happens over a period less than one hour. The circuit functions as a voltage-to-period converter. Although the capacitor voltage follows an exponential curve, it is nearly linear in the operating region from 0V to the comparison threshold of 454mV since the battery voltage which drives it is typically 5 to 6V for a four cell pack under charge. With a single cell the period can stretch to near the 71ms tick period. The capacitor voltage is described by the equation

$$V_C = V_{BATT}(1 - e^{-t/RC})$$

where $C=0.1\mu\text{F}$ and $R=1\text{M}$ in this circuit. The straight line approximation we are using is the tangent to the actual curve at $t=0$. This can be found by differentiating the above equation with respect to time

$$dV_C/dt = V_{BATT}/RC \cdot e^{-t/RC}$$

and setting $t=0$. Then the expression becomes

$$dV_C/dt = V_{BATT}/RC$$

Integrating the above expression yields a straight line

$$V_C = V_{BATT}/RC \cdot t$$

from the origin to the point (RC, V_{BATT}) . If you solve for t using the straight line and $V_C=454\text{mV}$, $V_{BATT}=5.50\text{V}$, you get $t=8.255\text{ms}$. Substituting this back into the first equation and solving for V_{BATT} yields 5.73V, which is within 5% of the actual voltage. The absolute accuracy of the conversion is only important when using the battery voltage measurement to detect a battery connected to the output, versus a short circuit or an open circuit. All the critical sensing is done within 1% of the peak voltage and is relative to the peak.

To avoid contamination of the battery voltage reading by switching noise, the voltage sensing is done during a quiet period when the switching current source is turned off by the processor.

87C751 fast NiCad charger

AN439

Waveforms

The first trace (Figure 13) shows drain to ground voltage on the P-channel FET. Note that the peak negative voltage is -17.66V and there is minimal flyback ringing. The positive spike on turn-on may be due to sense resistor inductance.

The second trace (Figure 14) shows the voltage across the 0.1Ω sense resistor at a DC output current of 2.5A and an input

voltage of 12.2V . Voltage spikes due to trace and sense resistor inductance are filtered out by R105, C112 to prevent false triggering of the UC3843 current comparator.

The third trace (Figure 15) shows the FET gate voltage. FET gates are usually rated at $\pm 20\text{V}$ maximum, but reliability is enhanced if they are kept within $\pm 15\text{V}$. In this case the gate voltage is well within range for reliable operation.

REFERENCES

1. Billings, Keith H.: Switchmode Power Supply Handbook, McGraw Hill 1989
2. Unitrode I.C. Data Handbook, Unitrode Corp. 1990
3. Panasonic NiCad Battery Handbook

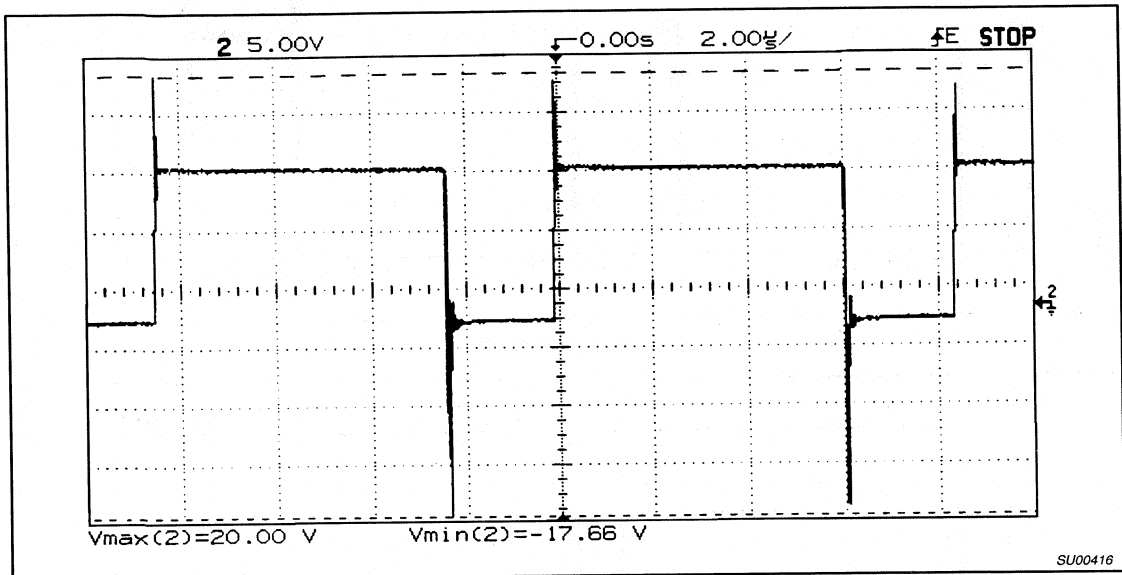


Figure 13. First Trace

SU00416

87C751 fast NiCad charger

AN439

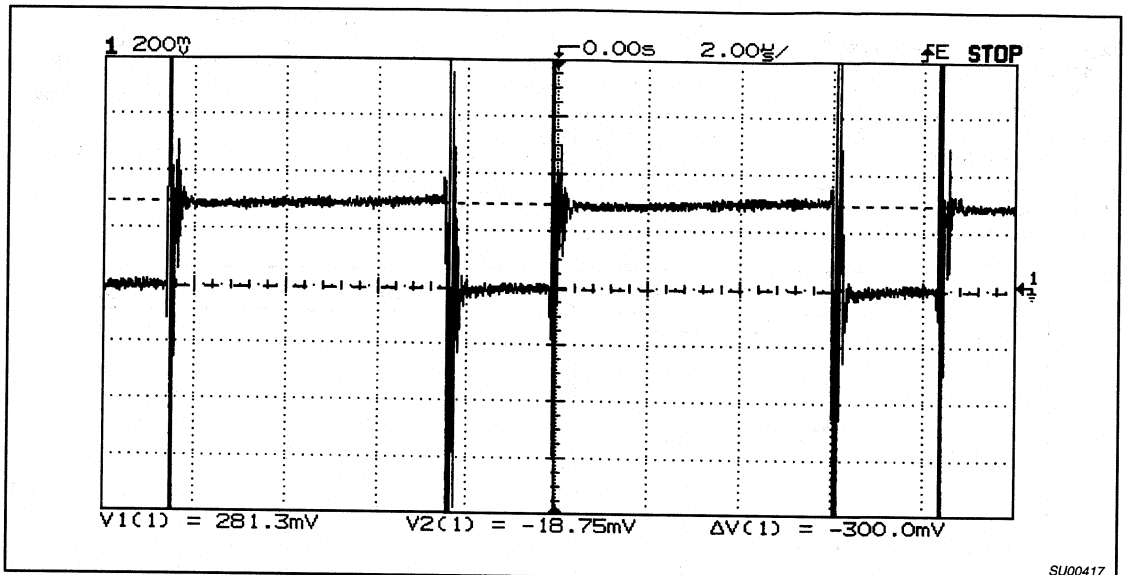


Figure 14. Second Trace

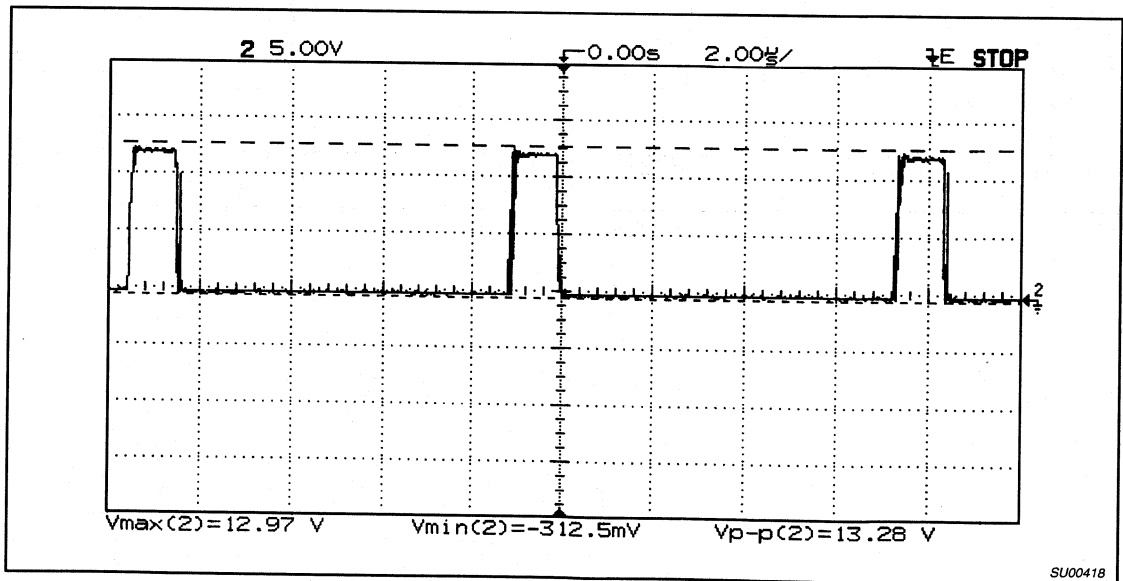
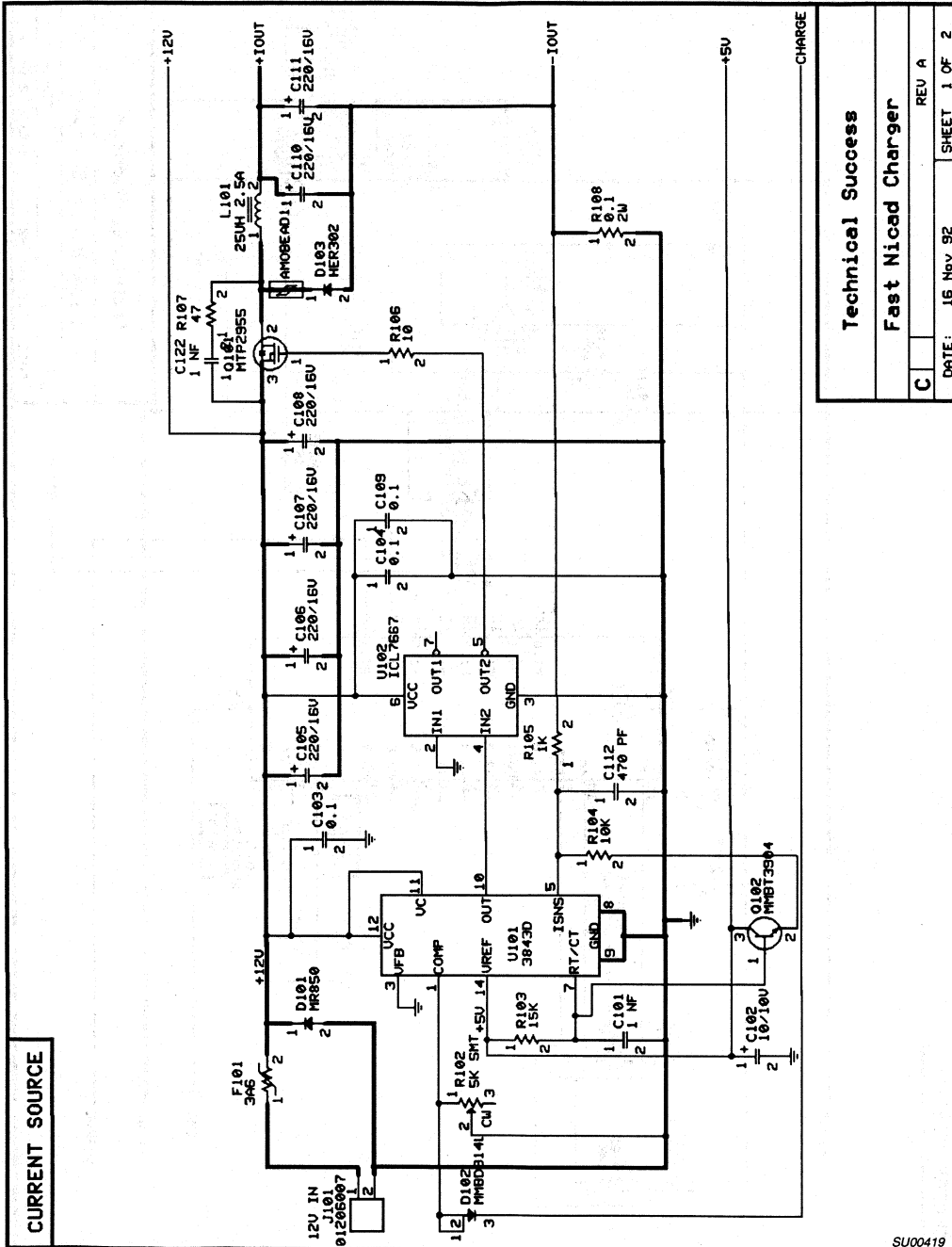


Figure 15. Third Trace

87C751 fast NiCad charger

AN439



Technical Success	
Fast Nicad Charger	
C	REV A
DATE: 16 Nov 92	SHEET 1 OF 2

Figure 16. Schematic Diagram of Battery Charger (1 of 2)

SU00419

87C751 fast NiCad charger

AN439

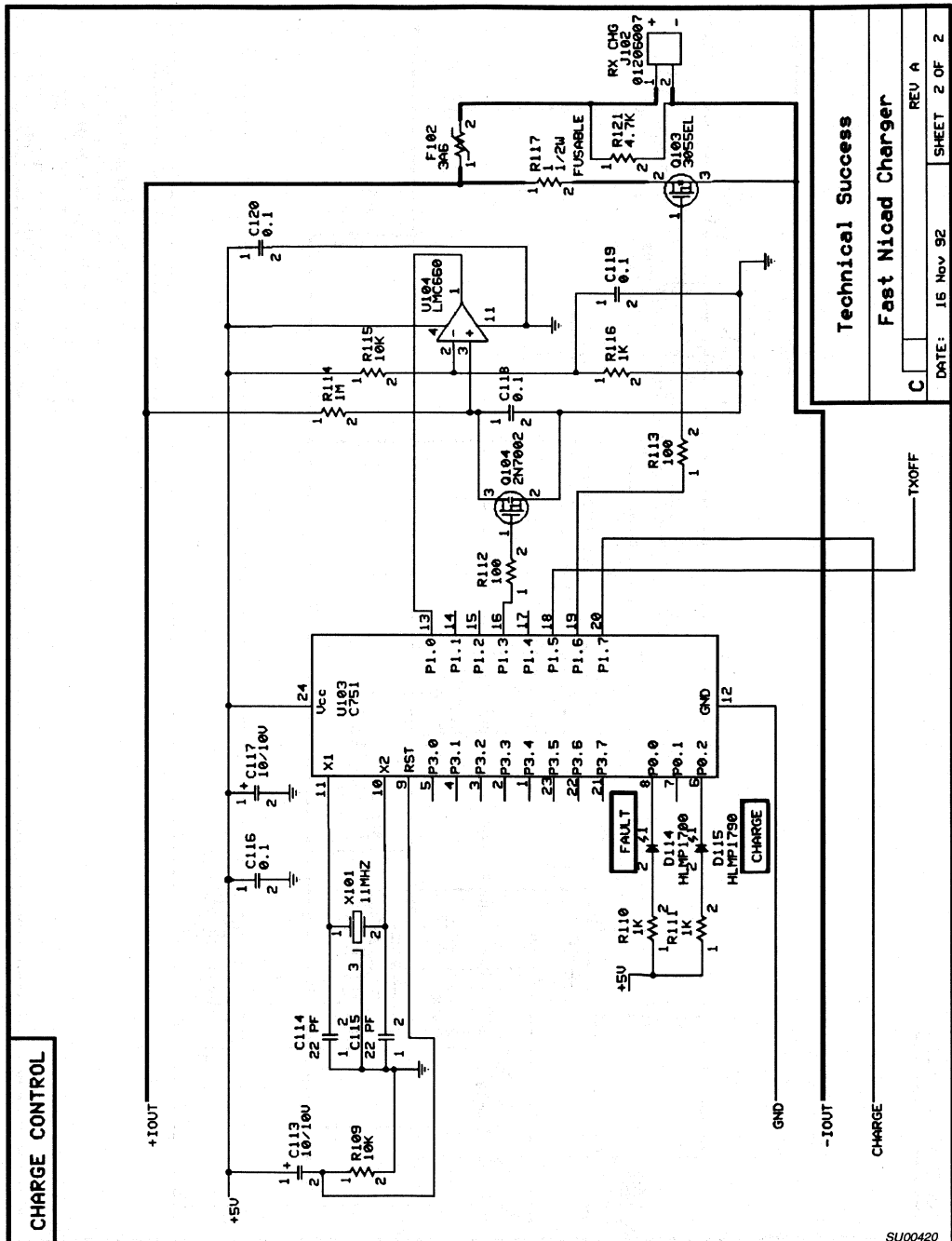


Figure 16. Schematic Diagram of Battery Charger (2 of 2)

SU00420

87C751 fast NiCad charger

AN439

```

#pragma PL (60)
#pragma PW (120)
#pragma OT (3)
#pragma ROM (SMALL)

#include <reg751.h>

typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned long dword;

#define TRUE      1
#define FALSE    0
#define ON       0
#define OFF      1
#define FLASH    2

/* parameters */
#define ONE_OVER_DELTA    128
#define ONE_SECOND       14

/* timers */
/* 1 cpu cycle = 1.085uS */
/* 1 tic = 0xffff cpu cycles */
/* 1 "second" = 14 tics = 995.5uS */
#define MAX_BATT_PERIOD    5530 /* 60ms period, 0.9 volts */
#define MIN_BATT_PERIOD    430 /* 4.67ms period, 10 volts */
#define DEAD_MAN_TIMEOUT   2712 /* 45 minutes */
#define MAINTENANCE_PERIOD 500 /* 1 pulse in 500 Seconds */
#define INITIALIZATION_TIME 20

/* errors */
#define BATTERY_VOLTAGE_OUT_OF_RANGE 0x01
#define BATTERY_CHARGED              0x02
#define DM_TIMEOUT                    0x03

/* states */
#define FAULT      0x01
#define INITIALIZING 0x02
#define CHARGING   0x03
#define DONE      0x04

/* global variables */
byte    tic;
byte    state;
word    seconds;
word    this_period;
word    last1;
word    last2;
word    last3;
word    last4;
word    last5;
word    last6;
word    last7;
word    valley;

sbit    comp_out    = 0x90; /* P1.0 */
sbit    clear_cap   = 0x93; /* P1.3 */
sbit    charge      = 0x97; /* P1.7 */
sbit    fault_led   = 0x80; /* P0.0 */
sbit    charge_led  = 0x82; /* P0.2 */

```

87C751 fast NiCad charger

AN439

```

word
measure_batt(void) {
    byte    tic_now;
    word    interval;

    tic_now = tic;
    interval = 0;
    clear_cap = FALSE;
    while(!comp_out && tic==tic_now)
        interval++;
    clear_cap = TRUE;
    return (interval);
}

byte
check_limits(
    word    batt_period
) {
    if ((batt_period > MIN_BATT_PERIOD)&&(batt_period < MAX_BATT_PERIOD)) {
        return(FALSE);
    }
    else {
        return(BATTERY_VOLTAGE_OUT_OF_RANGE);
    }
}

word
filter (
    word    last0
) {
    word temp1, temp2, temp3, temp4;
    word result1, result2;

    temp1 = ((last0 / 2) + (last1 / 2));
    temp2 = ((last2 / 2) + (last3 / 2));
    temp3 = ((last4 / 2) + (last5 / 2));
    temp4 = ((last6 / 2) + (last7 / 2));

    result1 = ((temp1 / 2) + (temp2 / 2));
    result2 = ((temp3 / 2) + (temp4 / 2));

    last7 = last6;
    last6 = last5;
    last5 = last4;
    last4 = last3;
    last3 = last2;
    last2 = last1;
    last1 = last0;
    return((result1 / 2) + (result2 / 2));
}

byte
delta_peak (
    word    period
) {
    if (period < valley)
        valley = period;
    if (period > (valley + (valley/ONE_OVER_DELTA)))
        return (BATTERY_CHARGED);
    else
        return (FALSE);
}

byte
watchdog (
    word    now
)

```

87C751 fast NiCad charger

AN439

```

(
    if (now < DEAD_MAN_TIMEOUT)
        return (FALSE);
    else
        return (DM_TIMEOUT);
)
void
charger ( void ) {
    this_period = measure_batt();
    this_period = filter(this_period);
    switch (state) {
        case FAULT: {
            if (!check_limits(this_period)) {
                seconds = 0;
                state = INITIALIZING;
            }
            else
                state = FAULT;
            break;
        }
        case INITIALIZING: {
            if (check_limits(this_period))
                state = FAULT;
            else {
                charge = TRUE;
                if (seconds < INITIALIZATION_TIME)
                    state = INITIALIZING;
                else {
                    valley = 0xffff;
                    state = CHARGING;
                }
            }
            break;
        }
        case CHARGING: {
            if (check_limits(this_period))
                state = FAULT;
            else {
                if (!watchdog(seconds)) {
                    if (delta_peak(this_period)) {
                        state = DONE;
                        seconds = 0;
                    }
                    else {
                        state = CHARGING;
                        charge = TRUE;
                    }
                }
                else {
                    state = DONE;
                    seconds = 0;
                }
            }
            break;
        }
        case DONE: {
            if (check_limits(this_period))
                state = FAULT;
        }
    }
}

```

87C751 fast NiCad charger

AN439

```
        else {
            if(seconds < MAINTENANCE_PERIOD) {
                charge = TRUE;
                seconds = 0;
            }
            state = DONE;
        }
        break;
    }
}

void
leds ( void ) {
    switch (state) {
        case FAULT: {
            charge_led = OFF;
            fault_led = ON;
            break;
        }
        case INITIALIZING: {
            charge_led = ON;
            fault_led = OFF;
            break;
        }
        case CHARGING: {
            charge_led = ON;
            fault_led = OFF;
            break;
        }
        case DONE: {
            charge_led = !charge_led;
            fault_led = OFF;
            break;
        }
    }
}

/* Timer 0 interrupt */
void
timer0(void) interrupt 1 {
    tic++;
}

void
main()
{
    /* initialize pins */
    charge=FALSE;
    charge_led = OFF;
    fault_led = OFF;

    /* initialize timer */
    TR=1;
    CT=0;
    GATE=0;
    RTH=0;
    RTL=0;
    IT0 = TRUE;
    ET0=1;
    EA=TRUE;

    /* initialize globals */
    tic=0;
    seconds = 0;
    valley = 0xffff;
    state = FAULT;
}
```


87C751 fast NiCad charger

AN439

```
/* main program scheduler */
while(1) {
    switch (tic) {
        case 0: {
            charger();
            while (tic < 1);
            break;
        }
        case 7: {
            leds();
            while (tic < 8);
            break;
        }
        case ONE_SECOND: {
            tic = 0;
            seconds++;
            break;
        }
    }
}
}
```

(BCM) 87C751 Specification for a bus-controlled monitor

AN442

SUMMARY

BCM87C751 is a powerful, flexible and low cost Digital Controlled Monitor System, based on the 87C751 microcontroller. It employs I²C bus control with various I²C bus controlled peripherals (PCF8582EP-EEPROM and TDA8444 D/A converter). The control function is implemented via 8 6-bit DC voltage output from TDA8444.

Some features of the system:

- Flexible approach, especially for multisync or auto sync operation
- Mode detection and frequency measurements by microprocessor
- Mode switching under software control
- Elimination of potentiometers
- Quick factory alignment (DACs can be preset)
- Automatic factory alignment possible

This document describes the operation and the use of the system. It provides necessary information concerning operation, required hardware, flow charts and their effect on the performance.

INTRODUCTION

Figure 1 shows the block diagram of a high-performance color monitor with microcontroller and several parts that communicate via the two-wire I²C-bus.

The system can perform the following:

- Determine the mode and standard of incoming signals with the stored values in memory (e.g., multisync modes).
- Enter parameters of user defined modes into memory via a keyboard.
- Control analog parameters such as contrast and brightness via the bus from keyboard inputs.
- Control mode and standard parameters (such as picture geometry parameters and the free-running oscillator frequency).

Features

- Multisync or Autosync operation
 - FH = 31kHz-95kHz
 - FV = 50Hz-114Hz
- Selectable four or five function control configurations
- Selectable one digit, or one and one-half digit, or null seven segment mode display.
- Selectable Horizontal direct ratio or indirect ratio F to V converter DAC output configurations
- Selectable ten user modes plus ten factory modes, or, one user mode plus 19 factory modes configurations
- Selectable multiple up/down keys or minimum keyboard configuration
- Change function without save key, all keys but the reload key have a repeat function
- Both Horizontal and Vertical outputs have F-to-V converter DAC outputs

- Four outputs of Horizontal PLL capacitor selection signal. This can be easily adapted for further extension.

IC DESCRIPTIONS

87C751

A derivative of the 8051 family of microcontrollers, the 87C751 has an 8-bit CPU, 2k bytes EPROM, 64 bytes RAM, 19 I/O lines, a bi-directional inter-integrated circuit (I²C) serial bus interface, and an on-chip oscillator.

PCF8581/2

A 1k- or 2k-bit, 5V electrically erasable programmable read only memory (EEPROM) organized as 128 or 256 × 8 bits. The stored information is electrically alterable on a word-by-word basis, I²C-bus controlled.

TDA8444

Comprises eight DACs, each controlled via the I²C-bus. The DACs are individually programmed using a 6-bit word to select an output from one of 64 voltage steps. The maximum output voltage of all DACs is set by the input VMAX and the resolution is approximately VMAX/64.

For detailed information on these devices, please refer to their respective data sheets.

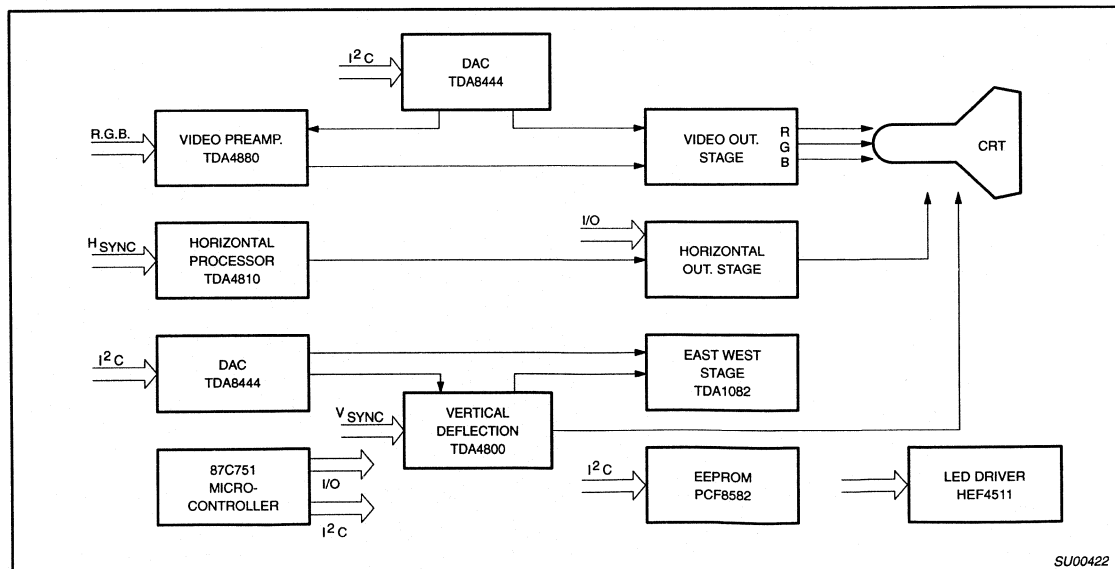


Figure 1. Block Diagram of Bus Controlled Monitor System BCM 87C751

SU00422

(BCM) 87C751 Specification for a bus-controlled monitor

AN442

OPERATION INSTRUCTION

Function Selection and Change

When the monitor is powered on it will automatically enter the corresponding mode depending on the input signal. Different configurations and functions can be defined by users with the use of different features. The following configuration's functions are examples for user's reference. See Figure 2 for Configurations 1 through 4.

To adjust functions such as V-size, V-shift, H-center, H-shift, and PCC (Pincushion Correction Circuitry):

1. The "Function" key should be pressed until the required function LED is lit.
2. If the V-shift LED is on, the user can then adjust V-shift by pressing Up or Down key. If the Up key is pressed, the V-shift DAC output will increase one step. While the Down key is pressed, the V-shift DAC output will decrease one step. The user can repeat the Up or Down key simply by pressing it longer than 0.5 second. It will then automatically repeat approximately 2 times per second until the key is released.

Mode Selection and Change

(See Figure 4)

In Figures 3 and 4, the mode number is displayed on the two seven-segment LEDs.

Each number denotes one mode. Modes 10 through 19 are user modes, which can be defined by the user. When there is a new mode entering the monitor that does not belong to any mode stored in the EEPROM, the mode display will show "19". If the user presses the Reload key while the mode display is "19", the display will flash. When the mode display is flashing, the user can select the destination mode by pressing the Up/Down keys. The destination mode is between 10 and 19.

Every press of the Up key causes the flashing display to add one, unless it already reached 19. Every press of the Down key causes the flashing display to subtract one, unless it has reached 10. When the user lets the destination mode flash on the display, the user can press the Reload key to store the new mode to destination mode. When the mode display stops flashing, the new mode is stored. The newly stored destination mode is permanent, unless the user repeats the entire procedure.

To change an old user mode, already stored in EEPROM, to a different user mode, press the Reload key for longer than 8 seconds while the monitor is working in the old mode. The mode display will flash the old mode, then the user can use the Up/Down key to select the new mode. Press the Reload key again to copy the old to new destination user

mode. If the user forgets to press the Reload key again, the flashing of the mode display will last for 2 minutes, then the program will cancel the copy old mode to new user mode command.

During the flash period, the program still monitors the Horizontal and Vertical sync signal to adapt the DAC to the proper mode. For example, while the user tries to copy new mode 13 to mode 15, and the mode display 13 (15) is flashing, if the PC sends out a signal for mode 3, the program will change the DAC output to adjust the monitor to work in mode 3, but the mode display is still flashing on user mode 13 (15) and the store procedure is still going on until the user presses the Reload key again, or terminates the copy procedure after the 2 minutes time out. The mode display then shows 3.

NOTE: Upon request, we can also program in advance those 10 user-defined modes that can still be changed by the user if necessary for further extension.

For Configuration 5 to Configuration 8, see Figure 3.

To adjust functions, the user can simply press the corresponding push button. The upper push button will increase the DAC output. The lower push button will decrease the DAC output. (A total of 64 steps can be programmed in advance.)

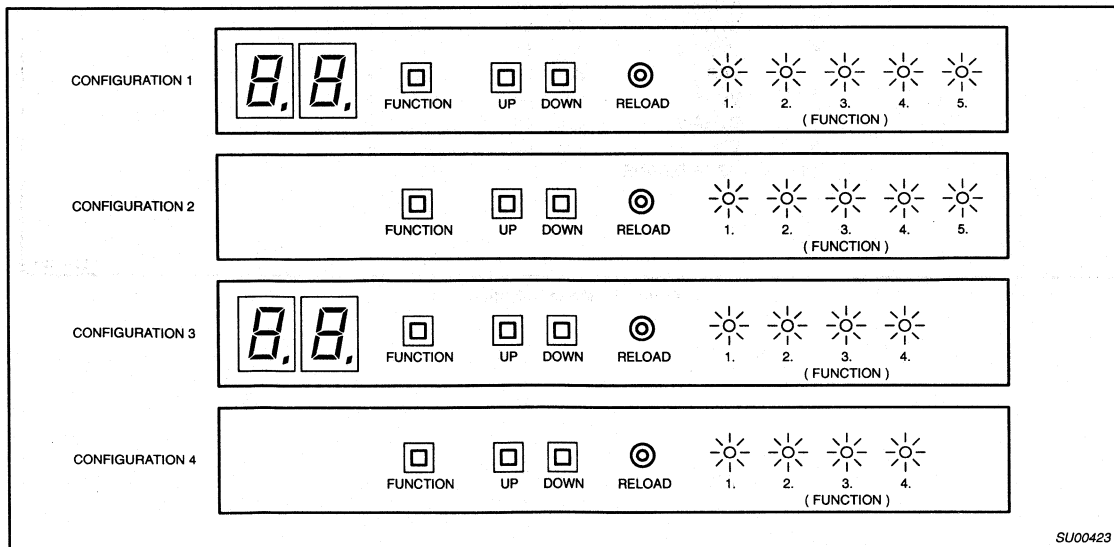
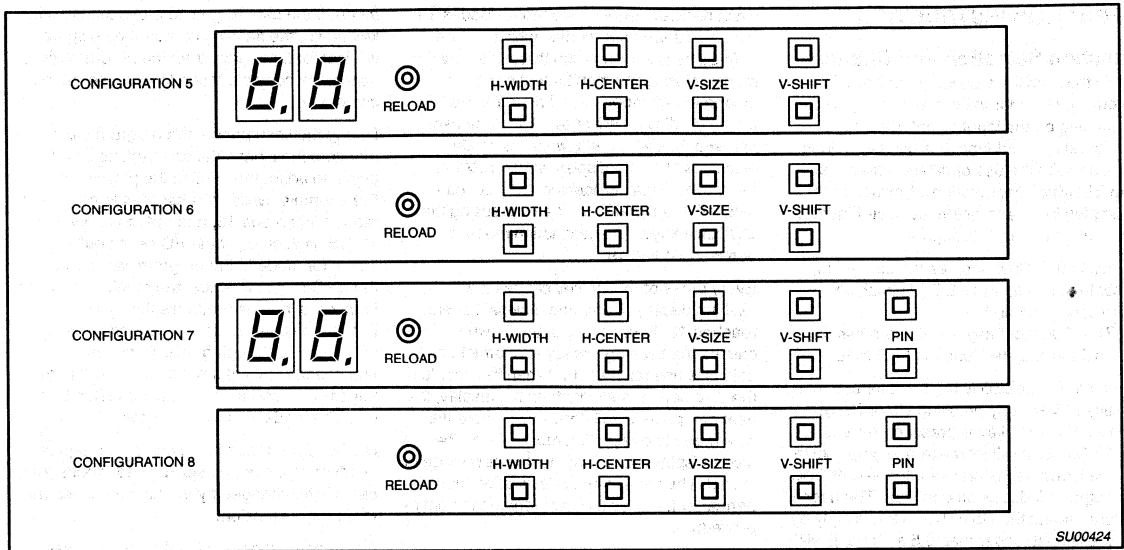


Figure 2. Control Panel

SU00423

(BCM) 87C751 Specification for a bus-controlled monitor

AN442



SU00424

Figure 3. Control Panel

MODE	NAME	H.F.	V.F.	H.P.	V.P.
0	VGA-1	31k	70	+	-
1	VGA-2	31k	70	-	+
2	VGA-3	31k	60	-	-
3	8514A	35k	87	+	+
4	SVGA-1	35k	56	+	+
5	UVGA-1	48k	60	+	+
6	VESA	56k	70	-	-
7	V64-1	64k	60	+	+
8	SVGA-2	37k	60	+	+
9	V78	78k	60	+	+
10	USER DEFINE
.
18
19

SU00425

Figure 4. Mode Selection

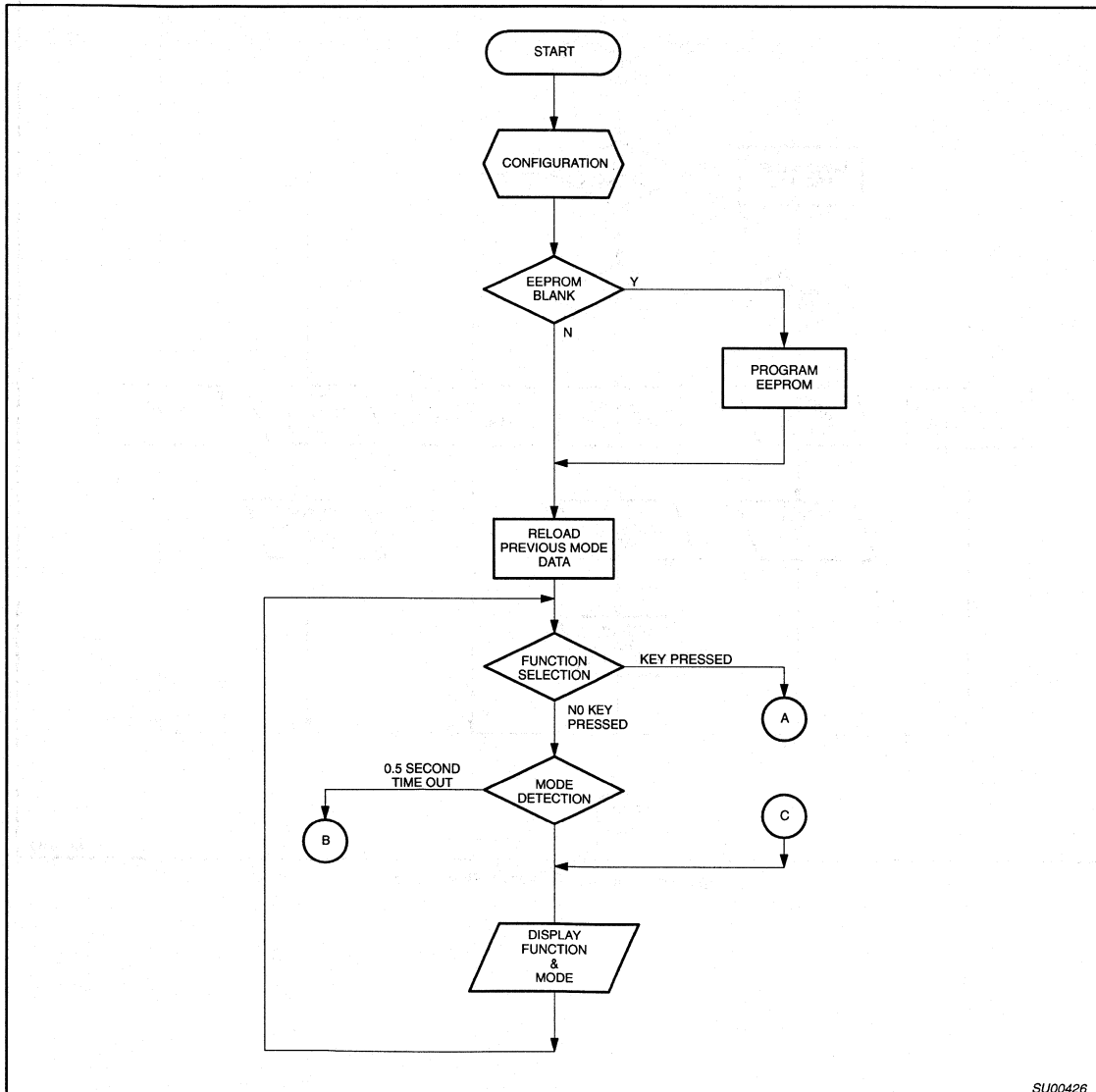
(BCM) 87C751 Specification for a bus-controlled monitor

AN442

SOFTWARE FLOW CHART DESCRIPTION

(See Figures 5 through 7)

When power is on, software initializes the hardware first. The microcontroller waits 100µs for the settlement of the hardware, then initializes itself by specifying stack, setting timer, clearing RAM, arranging interrupt, . . . , etc.



SU00426

Figure 5. Main Flow Chart

(BCM) 87C751
Specification for a bus-controlled monitor

AN442

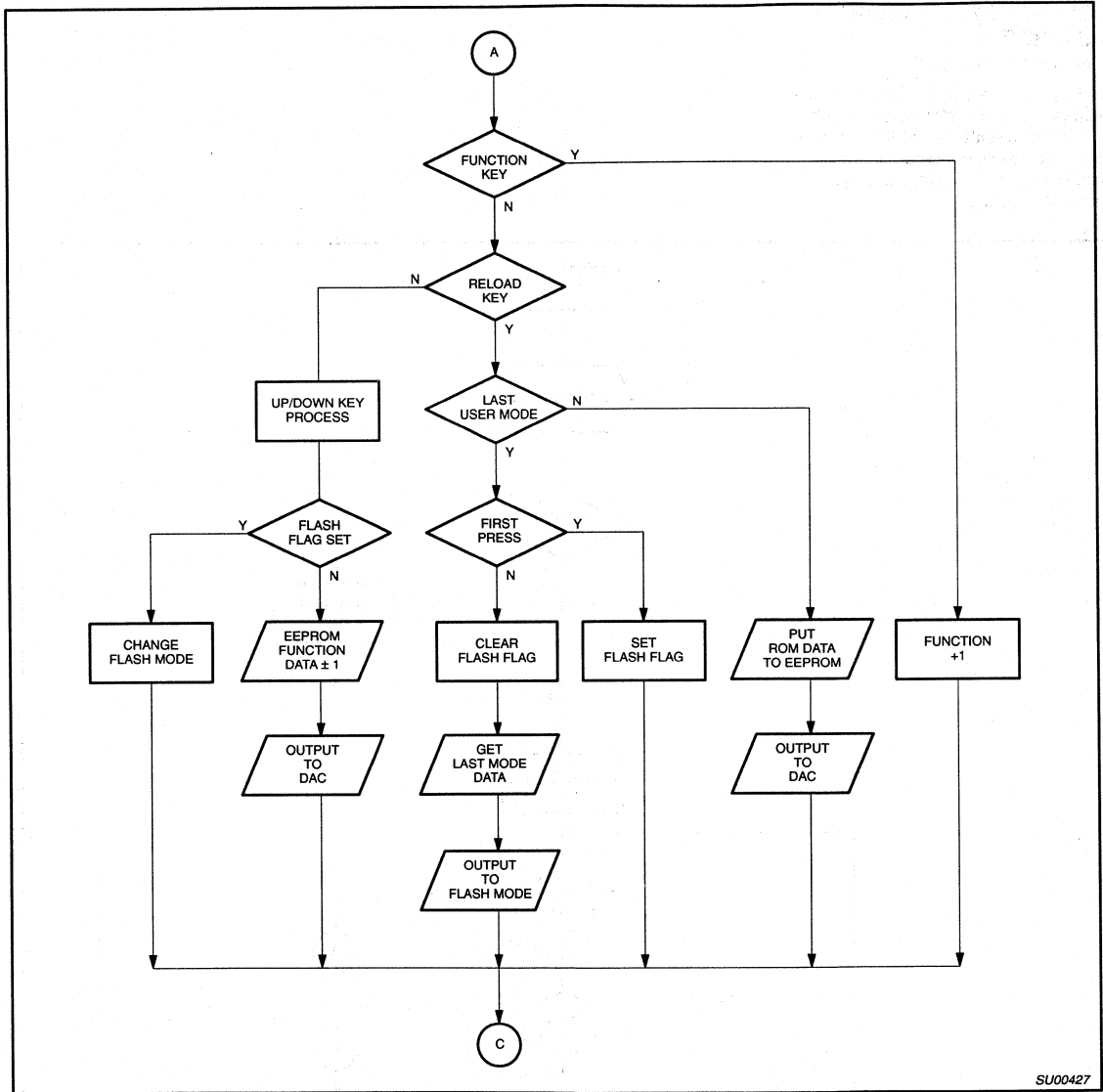
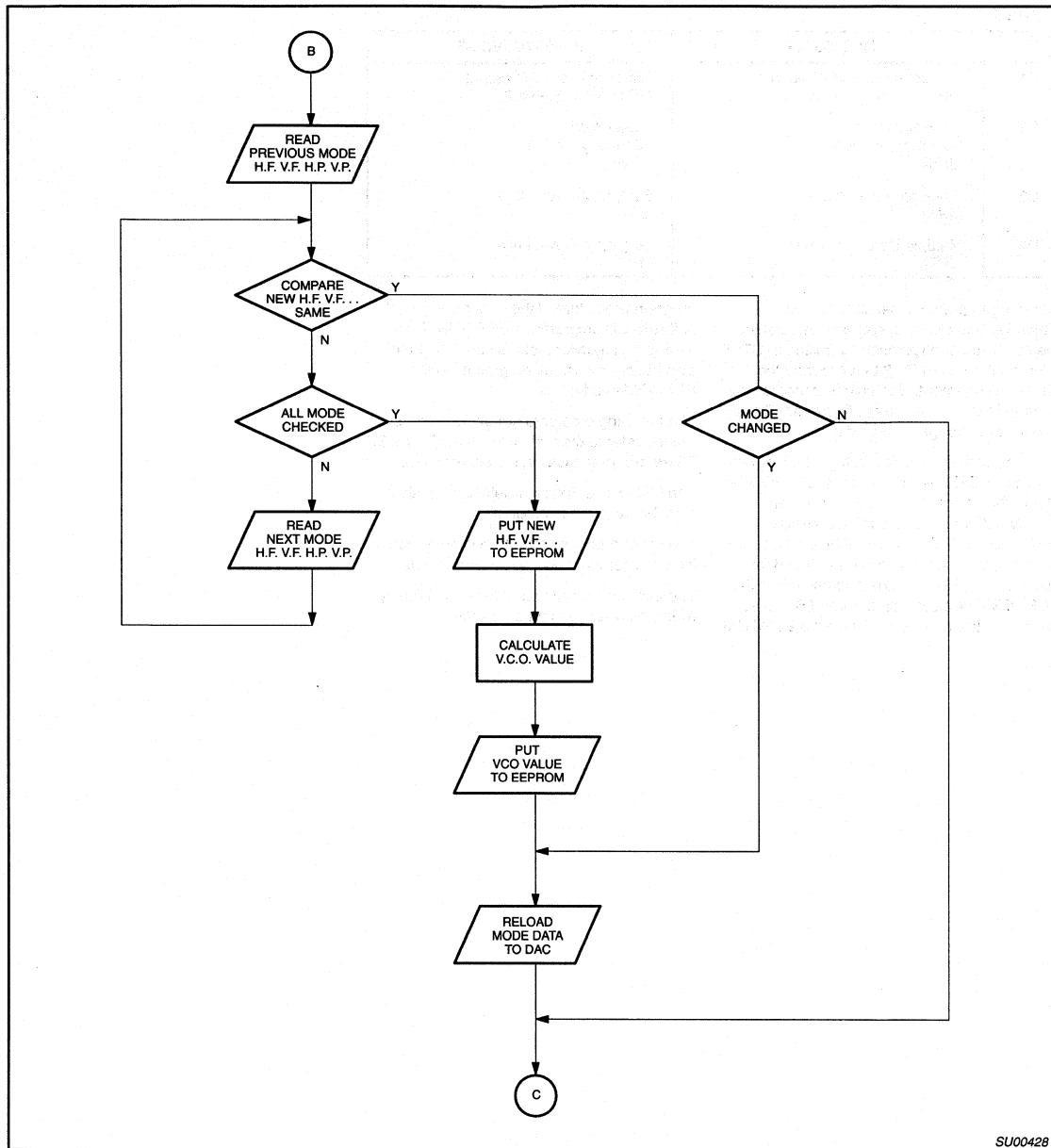


Figure 6. Function Selection Flow Chart

SU00427

(BCM) 87C751
 Specification for a bus-controlled monitor

AN442



SU00428

Figure 7. Mode Detection Flow Chart

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

Table 1.

	ADD DIODE	REMOVE DIODE
D1	Indirect ratio of VCO output (JP4)—See Figure 8.	Direct ratio of VCO output (JP4)—See Figure 8.
D2	10 user modes 10 factory modes (JP5)	1 user mode 19 factory modes (JP5)
D3	4 adjustable functions (JP6)	5 adjustable functions (JP6)
D4	Multiple Up/Down keys (JP7)	Single Up/Down keys (JP7)

Referring to previous section, *Function Selection and Change*, software can detect the hardware configuration by pulling 87C751 microcontroller pin P0.2 LOW to read the diode arrangement. Each diode denotes one change in the hardware configuration. Table 1 explains the usage of each diode.

After the configuration detection, then it goes to check EEPROM status. If the EEPROM is blank, the program will start to move all factory set data from the microcontroller's PROM to EEPROM. The last byte datum in the microcontroller's PROM is 66 used for blank check. First it reads address DD in the EEPROM, then compares it with 66. If they are equal, the software will skip the EEPROM

program procedure. If they are not equal, the software will program the EEPROM. This means that monitor makers can define their own factory modes by programming the EEPROM in advance.

The following programs are endless loops. Please refer to the main flow chart (Figure 5). There are three tasks in the endless loop.

The first task is Function Selection, basically a keyboard process program.

The second task is Mode Detection, which includes search mode and change mode.

The third task is Mode and Function Display, which includes flash mode display.

(BCM) 87C751 Specification for a bus-controlled monitor

AN442

Mode Detection

Beginning with branch "B", Mode Detection Flow Chart (Figure 7), the block at the top of the flow chart is "Read Previous Mode" (the time before 0.5 second ago) and includes Horizontal Sync Frequency, Vertical Sync Frequency, Horizontal Sync Polarity, and Vertical Sync Polarity. The second block is a comparison test block. When current mode (from 0.5 second ago until now) parameters are the same as those in previous mode, the program will branch to the right test block. Since the mode is not changed, the second test block in the right part of the flow chart will branch to leave the Mode Detection section.

If the current mode (from 0.5 second ago until now) parameters are not the same as the previous mode (the time before 0.5 second ago), the first test block from the top of the flow chart will branch to search all mode parameters in the EEPROM to find out what the current mode should be. The left loop of the flow chart checks for the end of the search procedure, i.e., if all modes in the EEPROM are searched and checked, and

the outcome is the same, then this test block will branch to set up a new user mode (19), as per the 4 steps indicated in the central flow chart line.

The first step in setting up a new user mode is to "Put New Parameters" (such as Horizontal Sync Frequency, Vertical Sync Frequency, Horizontal Sync Polarity, and Vertical Sync Polarity) into the EEPROM. The new mode parameters are always saved in the last mode address. If the configuration allowing 10 user modes is selected, then diode 2 is added. If one was found to be the same, the program will branch to the right test block. If it then finds that there is a mode change, it will branch to Reload Mode Data to DAC to complete the mode change procedure.

When the mode change procedure is completed, the monitor will be working in a new mode. Since the program enters the Mode Detection task every 0.5 second, it takes from 0.5 to 1.0 second to finish the change of mode. To save the new user mode (mode 19) to other user mode (10-18), the

user can use the RELOAD key to save the new user mode to other user mode (modes 10 through 18).

If the configuration for 10 user modes is selected, it is highly recommended that you save new user mode to other user mode (10-18) because the last mode "mode 19", will be overwritten by any new user mode whenever a new user mode is detected, after new user mode parameters are detected.

The second step in setting up a new user mode is "Calculating VCO Output Value" (see Figure 8). There are two different curves for the designer to select. If diode 1 is removed, the VCO Output Voltage will be in direct ratio to the Horizontal Sync Frequency. If diode 1 is added, the VCO Output Voltage will have an indirect ratio.

The third step in setting up a new user mode is "Put VCO Value to EEPROM".

The last step is "Reload Mode Data to DAC". After reloading the DAC, the monitor is changed to the new user mode.

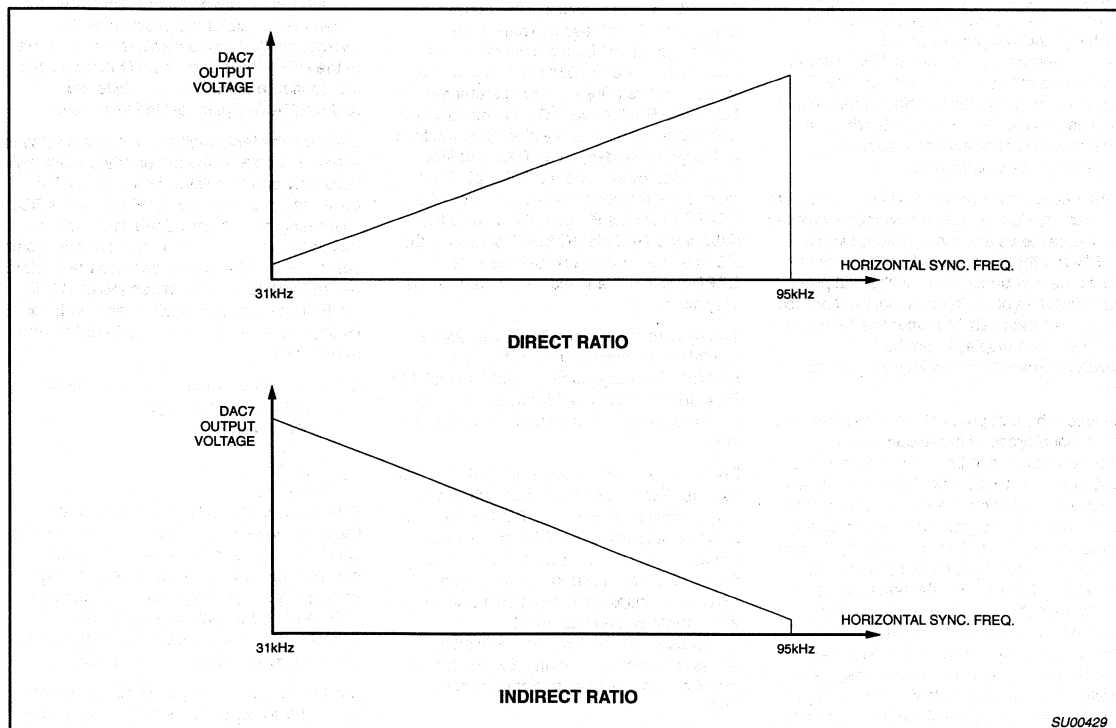


Figure 8. Horizontal VCO Output

SU00429

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

Key Function Selection

The first task is Function Selection (see Figure 6). When the program scans the keyboard, it first checks for the function key. If it is depressed, the program will branch right to change function VALUE(+1), the function LEDs are lit in sequence, i.e., when the second LED is lighting and the program detects a press in function key, the program will light the third LED and turn off the second LED. If the program detects that the last LED is lighting, it will turn on the first LED and turn off the last LED.

If the key pressed by the user is not a function key, the program will check if it is the reload key. If it is not, then the second test block will branch left to check Up/Down keys, both the Up and Down keys have two different definitions. When the user is updating function contents, Up and Down keys are used to change the function contents stored in the EEPROM. In that situation, the flash flag is not set; after the program branch left from the reload key test block, the program will test flash flag, then the program will change the output of DAC.

If a flash flag is set (see next paragraph, also), Up and Down keys will change the flashing mode displayed on two seven-segment LED displays. The flashing mode is valid between 10 and 19. The Up key will add one to the flashing mode unless the flashing mode is already 19, while the Down key will subtract one unless the flashing mode is already 10.

If the reload key is pressed while the program is in last mode (19), i.e., a new mode which is not the same as any mode entered in the EEPROM, the program will check to see if this is the first press. If this is the first press, the fourth test block in the middle will branch right to set the flash flag, after the flash flag is set, the mode displayed on the two seven-segment LED displays will start to flash.

Between the first press of the reload key and the second press of the reload key, the Up and Down keys can be used to change the flashing mode to a destination mode. When the program detects the second press of the reload key, the program will execute all the center blocks in the flow chart. First, starting to clear the flash flag; second, to get last mode data from EEPROM; and third, to put the mode data into a new address. The destination is decided by the user via Up and Down keys to select flashing mode, then pressing the reload key to make the flashing mode be a still mode. After mode display is still, the user finishes defining the destination user mode. This destination user mode will last forever unless the reload key is used to redefine it.

If the user presses the reload key, and the program is not in the last mode, it will branch right to reset mode data in EEPROM, starts to read original mode data in microcontroller's PROM, then puts these data to EEPROM, the outputs to DAC. This function is to help the user to restore the monitor when the monitor display is out of control. For example, if the user adjusts the Horizontal phase too broad, then monitor may become out of sync. As a consequence the screen will be a mess, and it is not easy for the user to re-adjust for correction. This feature will minimize possible complaints from customers.

CIRCUIT DESCRIPTION

U1-87C751 is an 8-bit microcontroller and the heart of the Bus-Controlled Monitor. The 87C751 receives Vertical Sync and Horizontal Sync signals from pins P1.5 and P1.6. The R3,C5 in pin P1.5 is a low pass protection circuit. It can prevent the Vertical Sync signals, including Horizontal Sync Pulse, from interfering with the counting of the Vertical Sync Frequency. The R2 in pin P1.6 is only used for protection of 87C751.

The 87C751 automatically checks the mode parameters from these two pins, then switches the DAC from the old mode to a new mode. When 87C751 is checking the mode, it reads different mode parameters from the EEPROM via I²C bus, then decides whether there has been a change in mode. If a change is needed, the 87C751 will first mute video by sending a LOW to pin P0.2, then reads the correct mode data from the EEPROM via the I²C bus. It then puts the data to the DAC via I²C bus. Because of the I²C bus, the connections between the EEPROM, DAC, and the microcontroller are very simple.

The system clock is provided by a 12MHz crystal connected to Pins 10, 11 of the 87C751. Basically, port1 is used for input, but P1.4 and P1.7 are used for output. P1.0 to P1.3 are used for keyboard and configuration input.

Port0 has only three pins; P0.0 and P0.1 are used for I²C control. P0.2 is an output pin used to test configurations. Port3 is basically used for output; P3.0 to P3.4 are used for mode seven-segment LED display output; P3.5 to P3.7 are used for function display. Both mode display and function display need extra decoders, an HEF4511B seven-segment display driver is used to display the mode, while an HEF4556B dual 2-to-4 decoder is used to display function LEDs.

PCF8582 is a 2k-bit EEPROM. R4,C1 constructs an external R-C time to program the EEPROM. In normal operation the

EEPROM needs 30ms to program one byte. C2 is a decoupling capacitor to stabilize the DC supply voltage for PCF8582.

TDA8444 is an octal 6-bit DAC. R7,VR5,C# constructs a reference voltage to define the DAC's maximum output voltage. In practice, the reference voltage must be below 10.5 volts, so R7 is added to prevent the reference voltage from exceeding that limit. C4 is also a decoupling capacitor to stabilize the DC supply voltage for TDA8444.

The upper half of the HEF4556 is used to provide a switch signal to select Horizontal OSC time constant capacitors. The four outputs are Active-LOW. When the Horizontal Sync Frequency (H.S.F.) falls in one of the four ranges, the corresponding output pin will go low. The four ranges are:

(H.S.F. < 35kHz),
(35kHz < H.S.F. < 40kHz),
(40kHz < H.S.F. < 50kHz),
(H.S.F. > 50kHz).

The enable input in the upper part of the HEF4556B can be used to extend the upper limit of the switch signal. The lower part of the HEF4556B is used to display function LEDs, the fifth LED is driven by an NPN transistor. When the transistor is turned on by the microcontroller, it also disables the lower part of the HEF4556B. If multiple Up/Down keys are configured, the function LEDs are replaced by five pairs of Up/Down keys.

Because the tenth digit of the mode display is either "1" or blank, the driver of the tenth digit uses only one transistor. The driver of the base digit employs an HEF4511B, it is a BCD seven-segment display driver with output Active-HIGH, so only common cathode types can be used. The factory can use one LED to replace a BCD display. When the LED is lit, the BCD display can display user mode, or factory mode. This is one way to reduce the system cost.

If multiple Up/Down keys are not configured, the keyboard has four keys:

Function key,
Up key
Down key
Reload key.

If multiple Up/Down keys are configured, there will be no function key, but five pairs of Up/Down keys and one Reload key. The multiple Up/Down keys are configured by adding a diode in JP7. If adding a diode in JP6, there will be only four functions available, i.e., there will be only four pairs of Up/Down keys or four function LEDs.

LM7805 is a power regulator IC, it changes 12V to 5V to supply the whole circuit except TDA8444, which uses a 12V power supply to provide a wider range of DAC output.

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

There is a table to explain the usage of each pin in the JP1 socket. JP2 and JP8 are connected together, JP3 is only used for future automatic alignment (including production line) if necessary. JP4 to JP7 are used to select hardware configurations as previously mentioned.

SPECIFICATION OF THE SYSTEM

- The input signals to the system are Horizontal Sync and Vertical Sync. The system accepts standard TTL level signals, i.e., $V_{IH} > 2V$, and $V_{IL} < 0.4V$. Horizontal Sync tolerance is $\pm 0.5kHz$, and Vertical Sync tolerance is $\pm 2-2$ Hz.
- There are eight DAC output signals. Their maximum output voltage can be preset by setting a voltage on the TDA8444's V_{MAX} pin. The voltage on the V_{MAX} pin must be below 10.5V and also below the voltage on the TDA8444's V_p pin. For other detailed output current characteristics, please refer to Philips data books IC02a, IC02b and *80C51 and Derivative Microcontrollers*.
- The Horizontal switch outputs have four pins. they are standard CMOS B-type buffered outputs. They are Active-LOW, i.e., there will be only one output active at any time. If the designers wants to add ranges in higher Horizontal Sync Frequency, the designer can put extra circuits onto the demo board. For example, an OPA can be added as a comparator to detect the VCO output. If the VCO output is higher than a certain voltage (VCO's 60kHz output voltage), the OPA will be triggered and the upper half of the HEF 4556 can be disabled by the OPA via HEF4556's Pin 1, when HEF4556 is disabled, the four Horizontal switch outputs will remain HIGH, then the OPA's output can be used as another switch output.
- Total current consumption is around 25-90mA, depending on the number of LED and seven-segment displays being lit.

PARTS LIST

87C751 BUS CONTROLLED MONITOR
TH-9102/4

Bill of Materials

November 7, 1991

Revised: November 7, 1991

Revision:

12:08:46

Page 1

ITEM	QUANTITY	REFERENCE	PART
1	1	C1	2700pF
2	6	C2, C3, C4, C8, C12, C13	0.1 μ F
3	1	C5	0.01 μ F
4	1	C6	100 μ F
5	2	C7, C11	1 μ F
6	2	C9, C10	33pF
7	5	D1, D2, D3, D4, D5	LED
8	3	JP1, JP2, JP8	Header 16
9	1	JP3	Header 4
10	4	JP4, JP5, JP6, JP7	Jumper (add diode)
11	2	Q1, Q2	BC548
12	1	R1	470R*7
13	11	R2, R3, R8, R9, R10, R11, R14, R15, R17, R19, R20	470R
14	6	R4, R6, R12, R13, R16, R18	22k
15	1	R5	VR10k
16	1	R7	2k
17	2	R21, R22	56R
18	5	S1, S2, S3, S4, S5	SW Pushbutton
19	1	U1	87C751
20	1	U2	HEF4556B
21	1	U3	PCF8582
22	1	U4	TDA8444
23	1	U5	HEF4511B
24	1	U6	LM7805
25	2	U7, U8	DISP-7
26	1	Y1	12MHz

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

PARTS LIST

87C751 BUS CONTROLLED MONITOR

TH-9102/5

Bill of Materials

November 13, 1991

Revised: November 13, 1991

Revision:

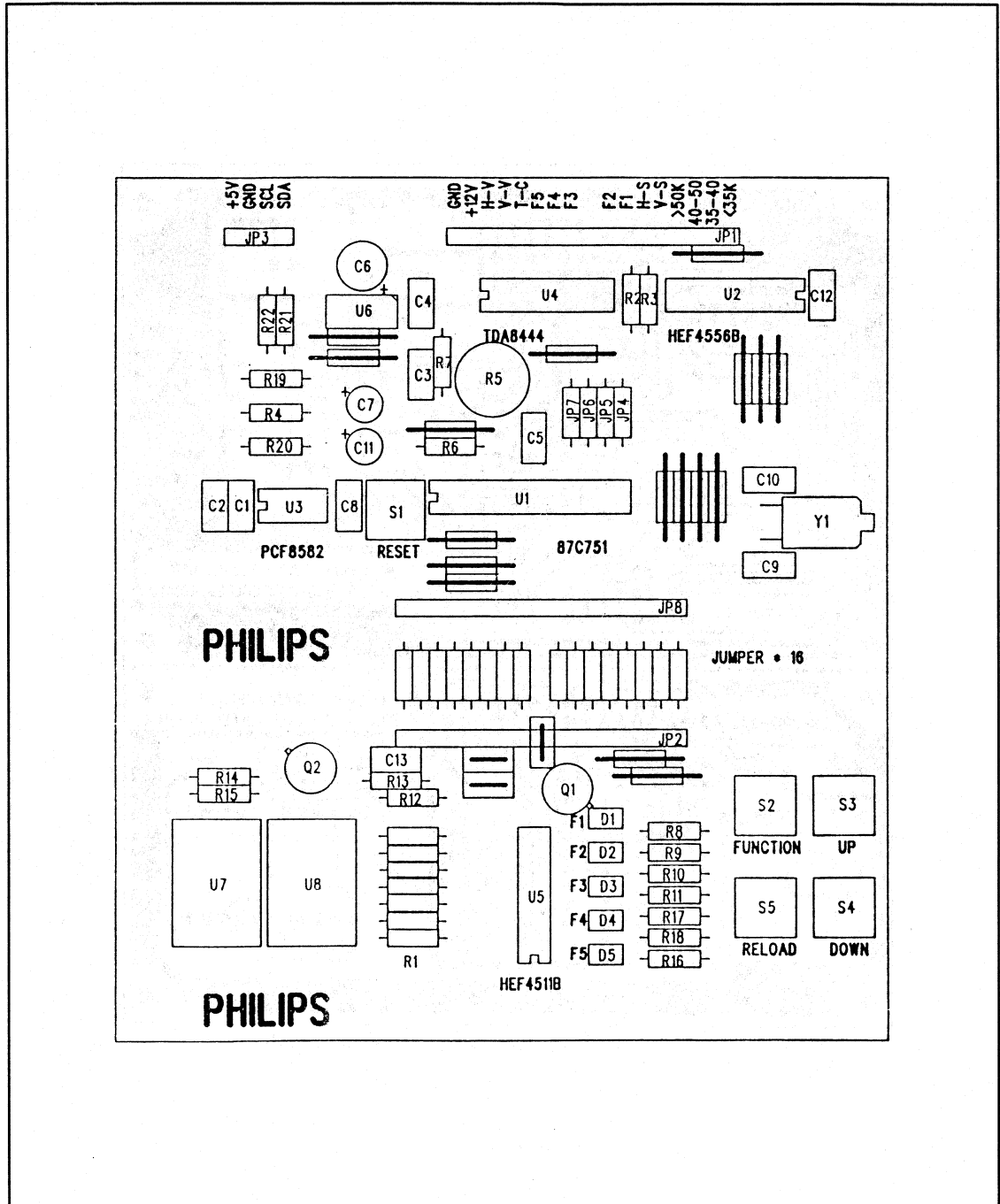
15:42:31 Page 1

ITEM	QUANTITY	REFERENCE	PART
1	1	C1	2700pF
2	6	C2, C3, C4, C8, C12, C13	0.1 μ F
3	1	C5	0.01 μ F
4	1	C6	100 μ F
5	2	C7, C11	1 μ F
6	2	C9, C10	33pF
7	3	JP1, JP2, JP8	Header 16
8	1	JP3	Header 4
9	4	JP4, JP5, JP6, JP7	Jumper (add diode)
10	2	Q1, Q2	BC548
11	1	R1	470R*7
12	10	R2, R3, R8, R9, R10, R11, R14, R15, R19, R20	470R
13	6	R4, R6, R12, R13, R16, R18	22k
14	1	R5	VR10k
15	1	R7	2k
16	2	R21, R22	56R
17	12	S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12	SW Pushbutton
18	1	U1	87C751
19	1	U2	HEF4556B
20	1	U3	PCF8582
21	1	U4	TDA8444
22	1	U5	HEF4511B
23	1	U6	LM7805
24	2	U7, U8	DISP-7
25	1	Y1	12MHz

(BCM) 87C751
 Specification for a bus-controlled monitor

AN442

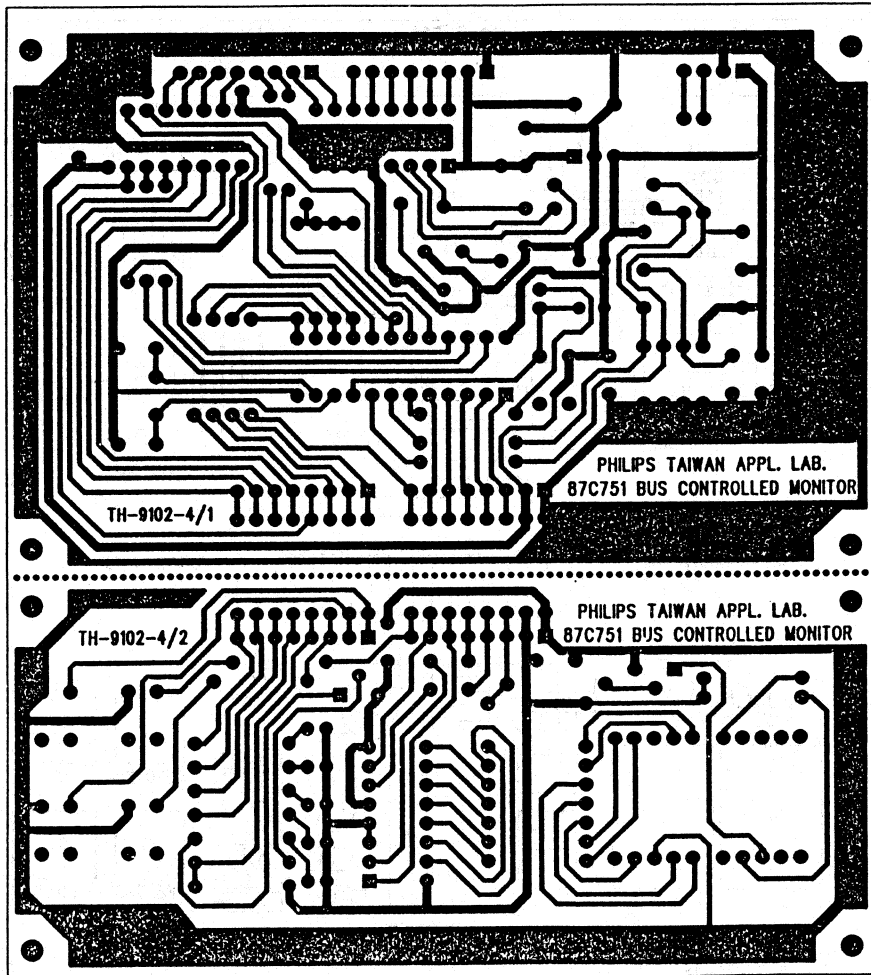
CIRCUIT DIAGRAM



(BCM) 87C751
Specification for a bus-controlled monitor

AN442

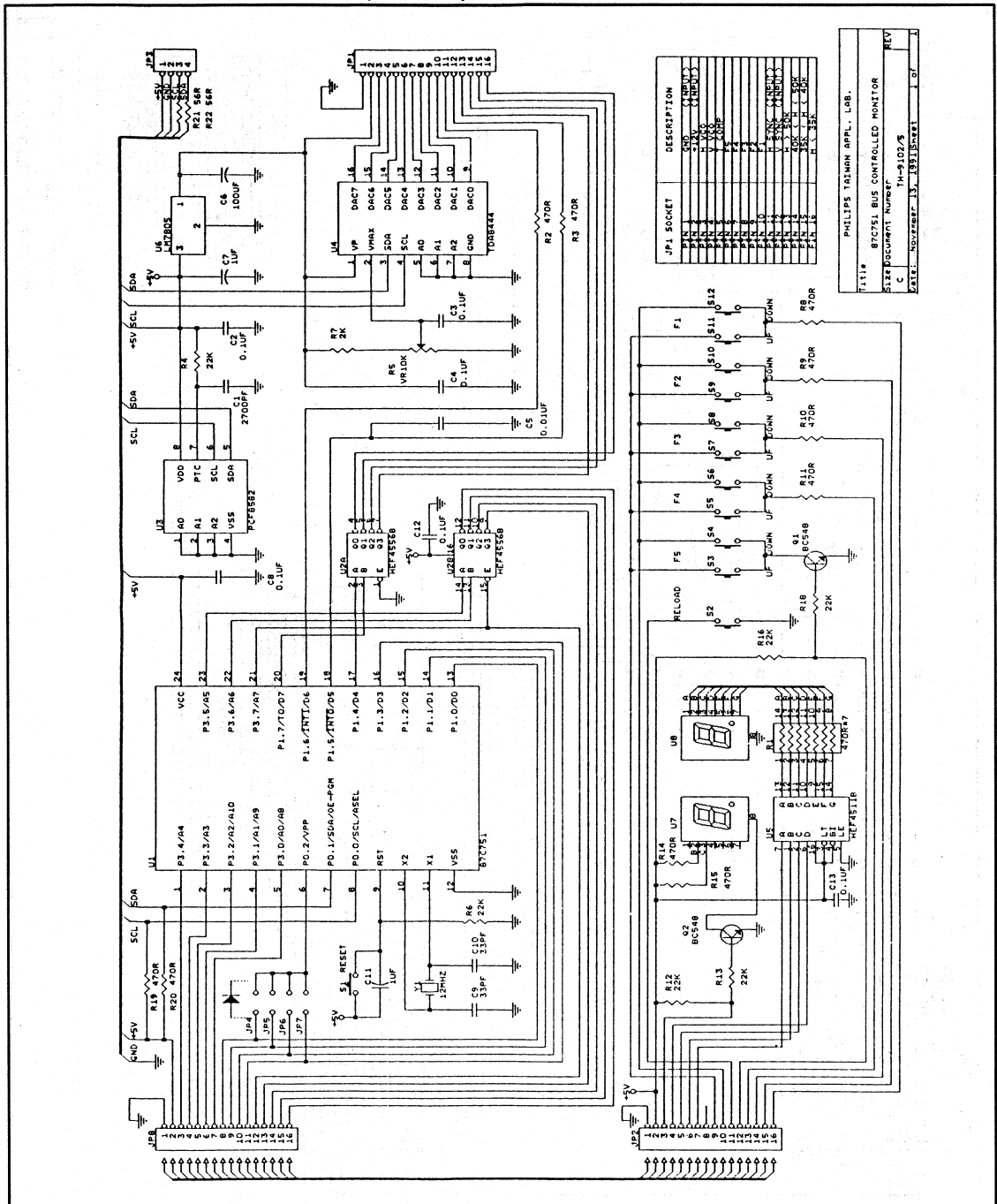
PRINTED CIRCUIT BOARD



(BCM) 87C751 Specification for a bus-controlled monitor

AN442

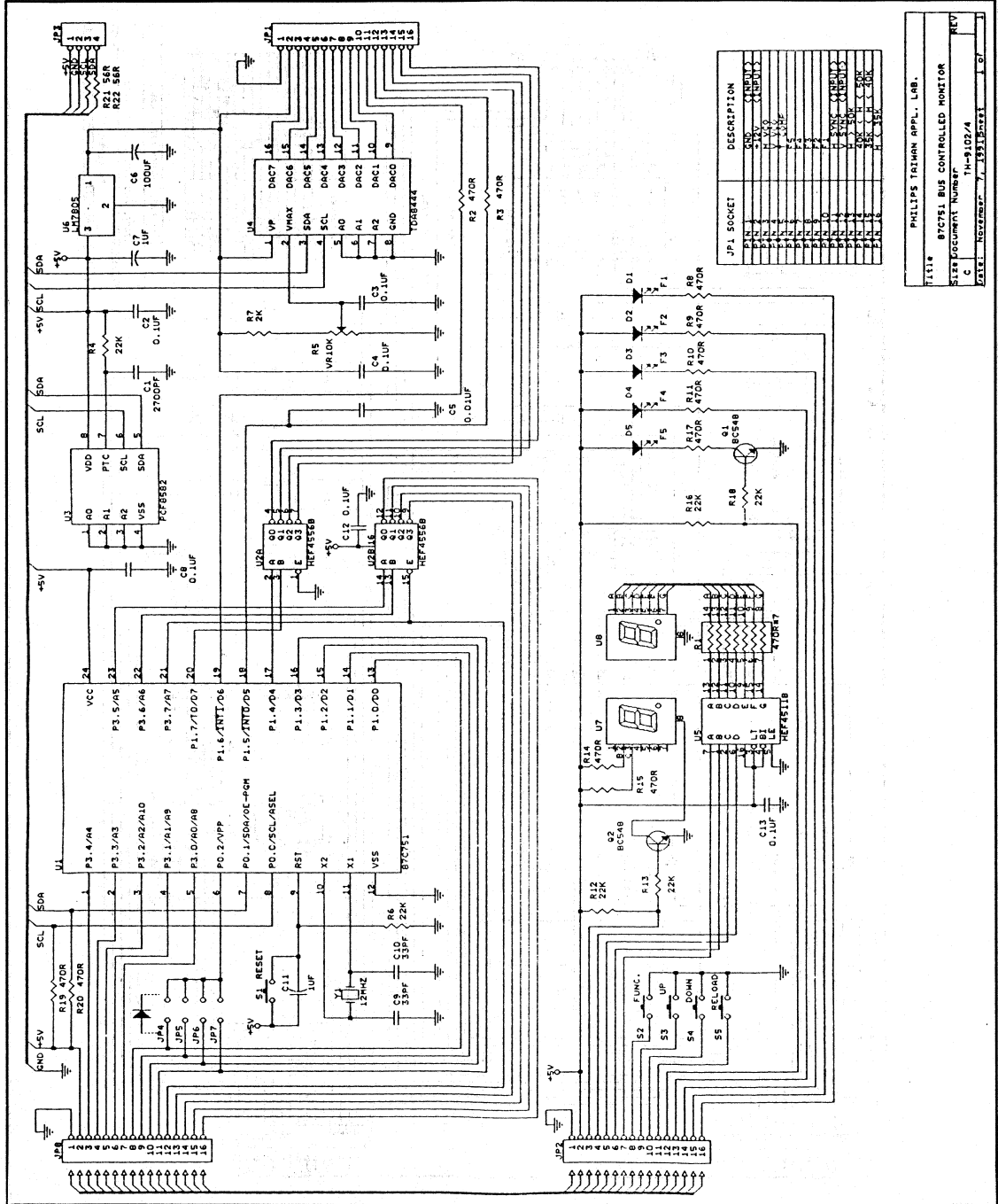
87C751 BUS CONTROLLED MONITOR (TH-9102/4)



(BCM) 87C751 Specification for a bus-controlled monitor

AN442

87C751 BUS CONTROLLED MONITOR (TH-9102/5)



(BCM) 87C751

Specification for a bus-controlled monitor

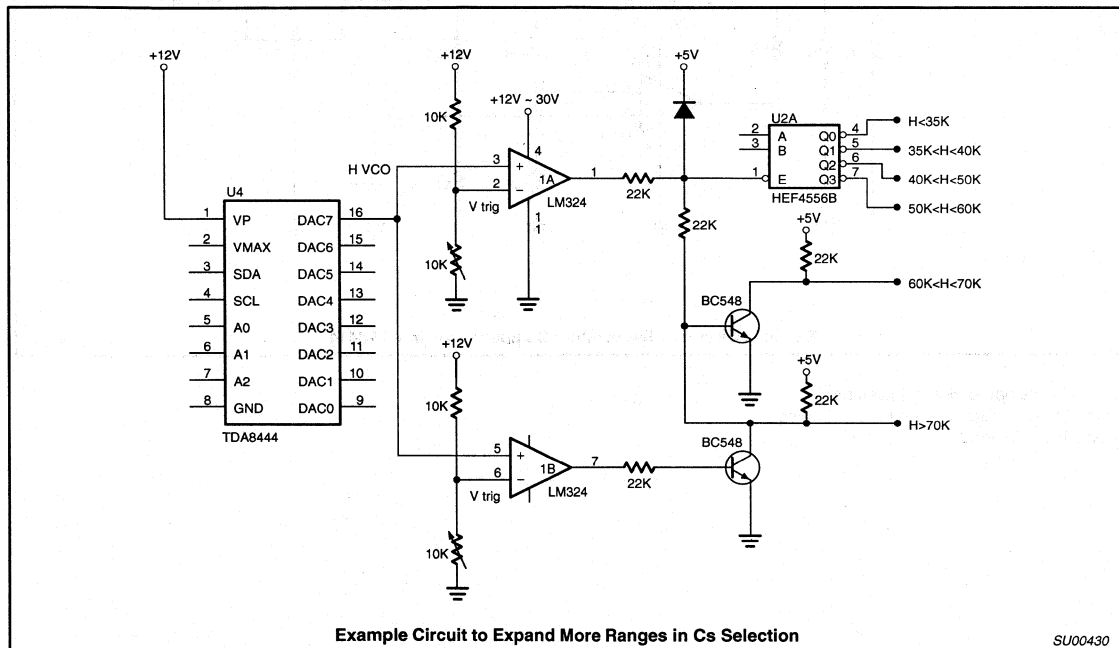
AN442

APPENDIX A

References

- Philips Data Book IC02a, IC02b
Video and Associated Systems
- Philips Data Book IC20
80C51 and Derivative Microcontrollers
Title: "AN422: Using the 8XC751
Microcontroller as an I²C bus Master"
- Philips Data Book
RF Communications
Title: "AN168: The Inter-Integrated (I²C)
Serial Bus: Theory and Practical
Consideration",
Author: Carl Fenger
- Title: "ETV8831: Deflection Processor
TDA8433 with I²C Control"
Author: DJA Teuling
- Title: "ETV89008: VGA Monitor with the
High Resolution Colour Tube
M34ECL10X36"
Author: H. Nerhees

APPENDIX B



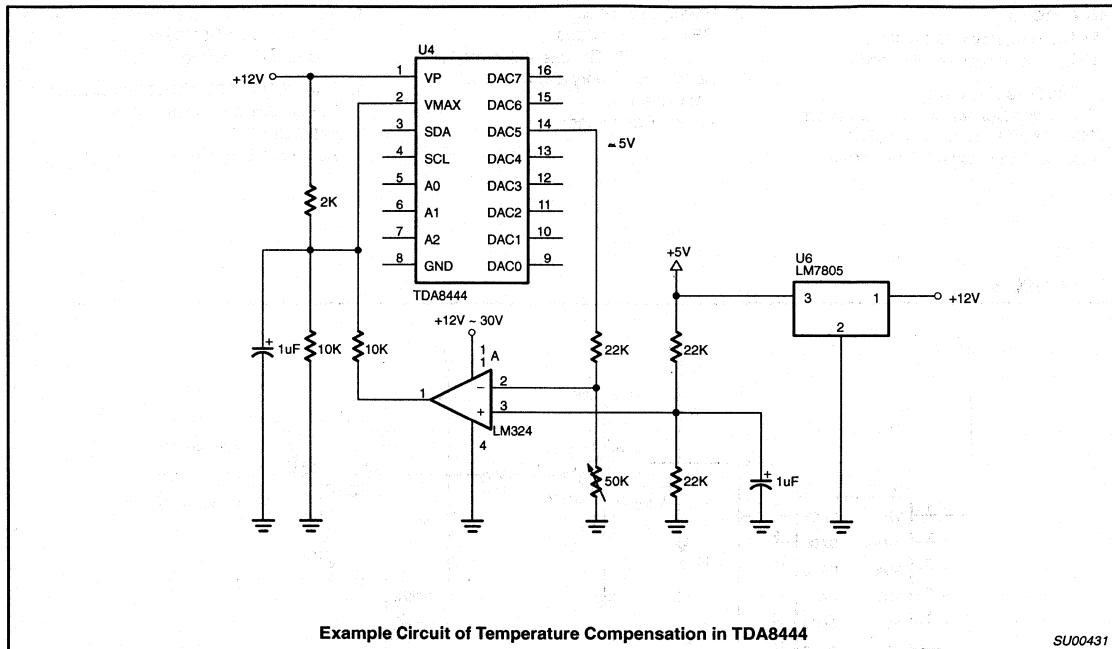
When you want to determine Vtrig signal, please disconnect Pulse Signal Generator (P.S.G.) Output to H. Sync demoboard from monitor and connect, and use input, set P.S.G. to 60kHz, TTL level output, then power on the demoboard, the mode display should display "19", the the voltage in DAC H-V output pin is the trigger level voltage of 60kHz.

You can set P.S.G. to 70kHz to measure trigger level voltage of 70kHz.

(BCM) 87C751
Specification for a bus-controlled monitor

AN442

APPENDIX C



SU00431

All VRs in this application note can be charged to fix resistors, when proper dividing voltage is determined.

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

DESCRIPTION

ACCESS.bus is an open standard, defining a system for connecting a number of relatively low speed peripheral devices to a host computer, typically a desktop system. The ACCESS.bus (A.b) standard is driven by the increasing demand of workstation and PC users for more peripherals on the desktop than ever before. Devices range from keyboards, mice and trackballs to hand held scanners, card readers and 'virtual reality' gloves. Some of the problems the A.b standard addresses are: difficulty of linking peripherals by non-expert users, desktop wiring clutter, limited number of I/O ports on a workstation, peripheral compatibility with different platforms and the high cost of software driver development associated with adding new peripherals to a system.

At the hardware level, the A.b is based on the I²C serial bus developed by Philips. The I²C protocol is supported by standard IC components, including a range of microcontrollers of the 80C51 family. These microcontrollers provide the intelligence for executing the A.b protocol in both peripheral devices and host systems. Many desktop peripherals can be implemented with a single, low cost 8XC751 microcontroller where the firmware supports both the I/O activity and the A.b protocol implementation.

This application note shows the 8XC751 firmware of Digital Equipment Corporation's A.b mouse implementation. Many A.b desktop devices could be implemented with a very similar code. After some discussion of mouse operation we shall give a short overview of the A.b protocol. Our discussion of the A.b is by no means complete—please refer to the specifications for more detailed information.

MOUSE OPERATION

The mouse is the most popular pointing device for interactive operation with a workstation, personal computer or Windows terminal. It reports to the host two dimensional planar movement, and user's activation of two or three buttons.

Many of the mice available today are opto-mechanical, using shaft encoders. As the mouse is moved over its pad, a lightweight rubber ball turns two perpendicular shafts. When the mouse is held with its cable at the top (away from the user), a left-right movement will rotate the 'X' shaft and an up-down movement will rotate the 'Y' shaft. Any diagonal movement will affect both. The shafts rotate slotted encoder disks which intercept light emitted by an LED. For each shaft there are two phototransistors

detecting the light, producing two signals which are out of phase by 90 degrees. Figure 1 shows the waveforms produced for one of the shafts when it rotates. The changes in these quadrature signals can be detected to determine the direction of the mouse movement, and its magnitude. The "positive movement" waveforms relate, for example, to a left to right movement in the X direction. Denoting channel samples as 'AB', a transition from a '00' state to '10' shows a positive movement, while a transition from '00' to '01' shows a negative movement.

The resolution of a mouse is determined by the number of changes to the quadrature waveforms produced in a unit length of planar movement. This is determined by the mechanics of the mouse, regardless of the speed in which the mouse is being moved. The mouse is an incremental pointing device, giving the host periodical position reports which show the displacement change relative to the last report. The microcontroller in the mouse takes the burden of keeping track of the rapid quadrature waveform changes and computing the relative displacement accumulated for each new position report. The quadrature waveforms are sampled, the changes are determined to be positive or negative, and X and Y relative displacement accumulators are being incremented or decremented accordingly.

The average rate of change is determined by the speed of mouse movement. For accurate position reports the encoder waveforms should be sampled frequently enough in order not to miss changes. The DEC mouse produces 200 changes for one inch of movement. Mouse movement at 10 inches per second will yield event rate of 2000 per second, and the microcontroller attempts to sample the encoder waveforms with at least twice that rate—no more than 250 μ S between samples. The MAIN routine of the example program performs this sampling in an infinite loop. It reads the position detectors at port 3, compares it to prior readings and if there was a change computes the new value of the relative displacement accumulators YCOUNT and XCOUNT.

Position reports are sent to the host at a much slower rate. In this example, Timer0 interrupts the code at the reporting intervals, and its interrupt routine ("Timer0") initiates a message transmission to the host with the latest information if there was some change in the mouse position or the buttons. The Timer0 service routine samples the position of the three mouse buttons sensed on port 1. Button changes are reported to the host in the same message as the position reports.

ACCESS.BUS PROTOCOL OVERVIEW

The A.b communications protocol is layered in three levels. The lowest level is a subset of the Philips Inter-integrated Circuit (I²C) bus protocol, above it the A.b Base Protocol common to all types of A.b devices, and on top are the Application Protocols which define message semantics that are specific to particular functional types of devices.

The I²C protocol defines the low level transaction over the I²C serial bus, using a single data line (SDA) and a clock line (SCL). The hardware definition for the A.b includes a four wire cable comprised of SDA, SCL and two voltage supply lines. The I²C provides for cooperative synchronization of the serial clock, bus arbitration, addressing, byte framing and byte acknowledgement by the receiver. The I²C is a multimaster protocol, and in ACCESS.bus subset the transmitter is always a master. The I²C allows 128 7 bit addresses, of which 125 may be used in A.b for peripheral microcontrollers. The I²C protocol burden is typically handled by microcontrollers both at the peripherals and at the host.

The Base Protocol establishes the A.b characteristics including message envelope format, predefined control and status messages, configuration process and the special role of the host. The host acts as a manager of the bus, and all data communication is between the host and peripheral devices—there are no message transactions between peripherals. In A.b, masters are exclusively senders and slaves are exclusively receivers. The host and the attached devices assume master or slave roles at the proper time.

An A.b message is an I²C bus transaction—a string of bytes sent by a master transmitter where each byte is acknowledged by the slave receiver, and the whole transaction is delimited by Start and Stop conditions. The minimum length of a message is four bytes, and the format definition includes specific locations for source address, destination address, message length and checksum. A protocol flag bit specifies whether the message is a device data stream or a control/status message.

The configuration process is designed to permit auto addressing and hot-plugging of devices. This process detects what devices are present on the bus, assigns unique bus addresses to the attached devices and connects them with the appropriate bus drivers. The configuration process is supported by eight pre-defined control/status

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

messages. In any A.b system the host address is always the same (50H). When the system is powered up all the peripherals perform self testing, assume a default address (6EH) and send to the host an Attention message announcing their presence. The host sends to each device an identification request message, to which the devices respond with a unique 28 byte ID string. Having received the ID string, the host assigns to each device a unique address. In the case of hot-plugging, the peripheral device and the host will interact in a manner similar to the message exchange during system power up.

In the last phase of the configuration process the host interrogates each device for its "capabilities string"—which describes the functional characteristics and the potential operating modes of the A.b peripheral. Capabilities information allows the software to recognize and use bus devices without additional knowledge about their specific implementation. Using the capabilities information enables writing 'generic' software drivers that can support a range of similar devices, providing some level of device independence and modularity. The capabilities information is transferred to the host as a readable ASCII string with a simple syntax.

A.b application protocols are specific to particular types of devices. The initial A.b specifications define Application protocols for three classes of peripherals: keyboards, locators and text devices. Each class is relatively broadly defined, leaving room for a variety of different devices. When drivers in the operating software of the host fully support a certain class, all devices conforming to the relevant Application Protocol will be supported, without any need for a special software driver.

The Application protocols already defined can support many standard desktop peripherals. The Keyboards protocol supports up to 255 keys. Locator devices can have up to 15 degrees of freedom and up to 16 binary keys or buttons. This can cover devices like mouse, tablet, trackball, 'virtual reality' pointing gloves, dial boxes and function key boxes. The Text Device protocol supports devices that transmit or receive messages consisting of strings of characters in some fixed character set. The simple protocol allows high level flow control, and is appropriate for devices like barcode readers, printers and magnetic card readers.

Each of the Application Protocols has its own set of control/status messages in addition to the predefined messages of the Base Protocol.

I²C Protocol Handling

The I²C hardware interface on the 8XC751 operates on a bit by bit basis, and the full I²C protocol is supported by a combination of hardware and firmware. This arrangement results in a very compact hardware circuitry necessary for a low cost integrated circuit. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing errors checks, and takes care of clock stretching and synchronization. The hardware is synchronized to the software either through polled loops or interrupts. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates a new data on the bus (DRDY), or when a special condition occurs: a frame Start (STR), Stop (STP) or an arbitration loss (ARL). The interrupt is caused by the ATN flag, which is turned on by any of the interrupt inducing conditions. The ATN flag can be polled in a software loop as well.

The example code handles the I²C protocol from an interrupt service routine (ISR). Typically, processing of a frame will be started with an interrupt (at the I2CINT label). If the bus operates at full speed, firmware processing inside a frame will be synchronized to the hardware bit by bit by a polling loop. The firmware polls the ATN flag in a loop limited to about 50 us (WaitATN) whenever it expects something to happen on the bus. If nothing happens during this period of time, the ISR is exited with the I²C interrupt re-enabled. When some bus event will occur later on, processing will resume with a new interrupt.

Processing of bus events monitored by the polling loop is identical to processing events detected by an interrupt. The context from which the mouse was sending or receiving a message is maintained between events (ATN flag activations), and is not lost when exiting the interrupt service routine. The I2CCxt byte stores the event that is expected, like waiting to send a bit or waiting for an acknowledge. Other I²C context elements are the data byte currently in the send or receive process (I2Cdat), a bit counter (BitCnt) keeping track of the location within that data byte and a message byte counter (ByteCnt).

In addition to the parameters that maintain the context of the very 'generic' I²C communications, the code maintains some additional context elements that are relevant to the higher level A.b protocol. These are the computed checksum (Check), the type of message or command being received (RcvType), the type of message being sent or pending (SndType) and a flag indicating that a Position Report transmission is pending (SendRpt).

The Interrupt service routine proceeds in handling the low level details of the I²C protocol as a Slave receiver or a Master transmitter. The routines for Slave or Master processing are separate, and the jump to either one from DISPATCH in I2CDONE routine is determined by the MST bit of the I2CON hardware register. The code examines the flags determining which event caused the ATN and then handles the low level hardware according to the context, performing actions like reading a new bit, acknowledging, sending a bit, issuing a Stop and so forth.

When the low level slave receiver code completes reception of a byte, it calls the DORXB routine which deals with the contents of the byte—"application level" routine. Upon return from DORXB there is a call to the Sample subroutines. We effectively sample the quadrature waveforms in between I²C words in order to comply with the requirement for minimum sampling interval. It is interesting to note that code design does not completely separate application code from the I²C low level code—we call Sample from an I²C reception routine (and we do the same in the Master transmission routine). This is because in the 8XC751 the I²C bit processing cannot be done in parallel to other firmware activities and we have to make sure that the application's timing requirements are not being violated.

The Master code will start sending a message if the processing routine was entered due to a Start condition. The routine, in fact, fulfills a request that was issued somewhere else in the code. For example, Timer 0 ISR sets the MASTRQ bit of the I2CON register, and sets the SendRpt flag. MASTRQ causes the processor to seize the bus when it is free and issue a Start. The Master processing routine examines the SendRpt flag, and if it is set the routine will start sending a Position Report.

In a structure similar to the Slave code bit level details are handled in the MASTER routine. Byte transmissions are set up in the DOTXB routine.

Processing At The ACCESS.bus Protocol Level

A control/status message from the host is identified by the Protocol Flag, the most significant bit of the third message byte. The message body is a code for the command. When such messages are received, they are processed by the DORXCMD routine. Control/Status messages can be of either the Base Protocol or the Application Protocol. In the listing, base protocol codes have the prefix 'l_', and application level protocol

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

commands has the prefix 'App_' (the definitions are in the include file 'ab.inc').

The Base Protocol commands from the host are `I_Reset`, `I_IdReq`, `I_ASgnAdr` and `I_CapReq`. During the configuration process the host responds to the host with device to host control/status messages: `I_Attn`, `I_IdReply`, `I_CapReply` and `I_Error`.

The string for the `I_IdReply` message is defined in `GET_ID`. The module revision and vendor name are padded with space characters in order to fit the fixed string length. The last four bytes of the ID string are a device number that can distinguish otherwise like devices with the same firmware. The protocol allows it to be a serial number or a pseudo random number. Our mouse uses a pseudo random number, produced by reading the 16 bit contents of `Timer0` that is active since power-up (the number is extended to 32 bits by appending `FFFF`). The protocol includes 'protection' against the rare event in which two like devices report the same pseudo random number or are mistakenly assigned to the same address. Just prior to sending the first data message to the host, each interactive device transmits a Reset message to its own assigned address (see `PosMsg` label in the example code). Any other device with the same address will be forced to the power-up

default address and will undergo configuration again, as it was hot-plugged onto the bus.

The Capabilities String for the `I_CapReply` message is defined in `GET_CAP`. The string identifies the device as a mouse with specific characteristics: three buttons, two dimensions, relative location reports with 200 dpi resolution etc. The 'prot(locator)' element tells the A.b software driver to use the Locator Device Protocol.

The Locator Device Protocol is one of three application protocols already defined for the highest layer of the A.b protocol. This protocol defines a "Locator Event Report" which is used for the Position Reports of the mouse.

A Locator Event Report is sent in the format of the device data stream Message defined in the base A.b protocol. The message body includes the current state of the buttons and the location difference from the last report. This data is coded as a sequence of two byte integers. For the mouse which is a two dimensional device, the message data stream length is six bytes, or three integers. The first integer contains the state of 0-15 locator key switches. For the three button mouse, only three of these sixteen bits carry

meaningful information. The remaining integers represent the position of the locator dimensions—the contents of the X and Y displacement accumulators.

Three control messages specific to the Locator Protocol are processed at `DORXCMD`. The host initiates a self test by `App_Test`. `App_Poll` initiates one time transmission of a position report, and `App_Setinterval` modifies the default reporting interval controlled by the reload value of `Timer0`.

This note highlights some of the implementation details—the commented listing covers the rest. As one can see, the A.b protocols are relatively simple to program in firmware. The low-level I²C implementation on the 8XC751 is somewhat involved, but the same low level routines can be re-used for different devices.

The source code files for this program are available for download from the Philips Semiconductors computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.

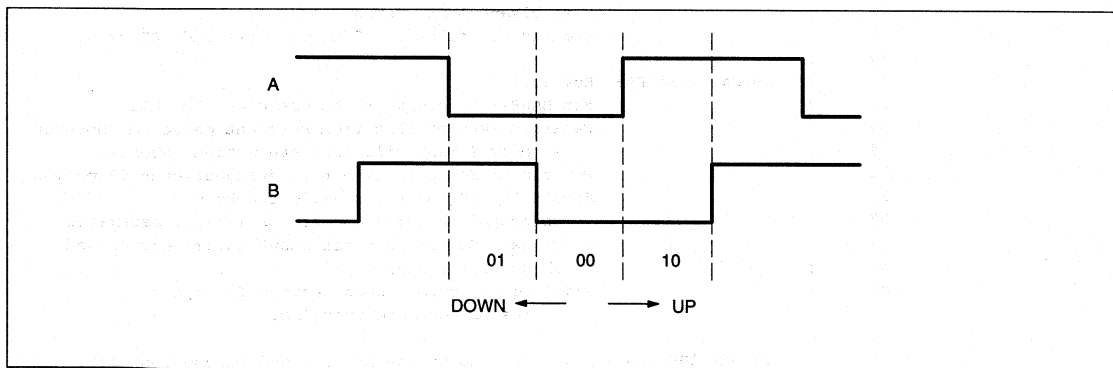


Figure 1. Example of Quadrature Encoding Waveforms

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

MS-DOS MACRO ASSEMBLER A51 V4.4

OBJECT MODULE PLACED IN MOUSE.OBJ

ASSEMBLER INVOKED BY: A51 MOUSE.A51

```

LOC  OBJ          LINE      SOURCE
1      ;*****
2      ; Module: mouse.a51
3      ;
4      ; Firmware design and code for I2C desktop bus Mouse
5      ; Environment: 83C751 Assembler
6      ; Author:      Robert Clemens 10-Jul-1990
7      ;              (I2C I/O adapted from P.Sichel's "Keyboard" code)
8      ; Revision:   6-Mar-1991
9      ;
10     ; 31-Jan-1991 PAS Add numerous keyboard fixes.
11     ;              Streamline input sample and I2C code.
12     ;              Separate HW dependent constants.
13     ;              Fix RxEnable after bus time out.
14     ;
15     ; 06-Feb-1991 PAS Rev X0.6
16     ;              Fix sample timer initialization, use 14ms as default.
17     ;              Fix length checking to allow commands with
18     ;              more parameters than required.
19     ;              Implement Set Interval command.
20     ;              Handle LLLLL=0 to mean 32.
21     ;              Fix ARL during self-addressed reset message.
22     ;              Fix to handle DRDY and ARL together.
23     ;              Re-order mouse buttons as MRL, update Capabilities.
24     ;              Do not allow other interrupts during TimerI svc.
25     ;              Document sampling requirements.
26     ;              Document hardware details.
27     ;              Add check to skip waiting after DNRXB.
28     ;              Misc clean-up in: BeMast, Assign,...
29     ;              Use include files for 751 registers and ODB msgs.
30     ;
31     ; 6-Mar-1991 PAS Rev X0.7
32     ;              Fix MN8Bit to check ARL before clearing DRDY.
33     ;              Separate SendRpt flag from movement detected (Movement)
34     ;              so only Timer0 will initiate motion reports.
35     ;              Don't send Position report to def_addr even if polled.
36     ;              Report InputError for invalid checksum,
37     ;              unrecognized command code, or illegal parameter
38     ;              value. Do not complain about parameters beyond
39     ;              those anticipated.
40     ;              Sample quadrature inputs between I2C bytes
41     ;              to insure accurate tracking.
42     ;
43     ; 13-Mar-1991 PAS Fix DoStp4 to borrow Rx code and become IDLE rcv.
44     ;
45     ; 26-Mar-1991 PAS Rev X0.8
46     ;              Update to use 4-byte device number.
47     ;              Get new device number only after Reset.
48     ;
49     ; 8-Apr-1991 PAS Add error checking to CapReq.
50     ;
51     ; 9-Jul-1991 PAS Add protocol_revision as part of module revision.
52     ;
53     ; 29-Jul-1991 PAS Protocol_revision in 1-byte, fix ARL and MASTER bug.
54     ;              Ignore unrecognized commands. Add I_MsgCheck.
55     ;
56     ; 11-Sep-1991 PAS Rev X1.2. Identify button positions as 1,2,3.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
      57          ;          Use new ab.inc file.  Change I_MsgCheck to
      58          ;          I_Error and do not overwrite pending SndType.
      59          ;          Retry message after Negative Ack (NACnt).
      60          ;
      61          ;    7-Oct-1991 PAS  Rev X1.3.  Fix spurious large count problem.
      62          ;          Fix STOP detected while MASTER without ARL.
      63          ;
      64          ;    25-Oct-1991 PAS  Rev X1.4.  Improve TimerI handler to avoid
      65          ;          lockup when MASTRQ with SCL low.
      66          ;
      67          ;    4-Nov-1991 PAS  V1.0 release for Boston mfg.
      68          ;    22-Dec-1991 PAS  V1.1 align data with tx bit that lost arbitration
      69          ;
      70          ;
      74          ;
      75          ;*****
      76          $ TITLE (Digital ACCESS.bus Mouse, V1.1)
      77          $ DATE (12/22/91)
      78          $ DEBUG
      79          $ NOMOD51          ;83C751 is not model 51
      80          ;Define SFRs explicitly
      81
      82
      83          ; Symbolic addresses and masks
      84
      85          $ INCLUDE( /dskbus/include/arch/reg751.inc )
=1    86          ;*****
=1    87          ; Module: /dskbus/include/arch/reg751.inc
=1    88          ;
=1    89          ; 83c751 SFR declarations
=1    90          ; Environment: 83C751 Assembler
=1    91          ;
=1    95          ; Date          Revision          Perpetrator
=1    96          ;
=1    97          ; 30-Jan-91   X0.1              Mark Shepard
=1    98          ;          Created (from previous keyboard module)
=1    99          ;
=1   100         ;
=1   101         ;          =1   102   $EJ
=1   103         ;*****
=1   104
=1   105         ; Interrupt Enable Register
00A8  =1   106         IE      EQU      0A8h
0000  =1   107         EX0     EQU      0          ;External interrupt 0
0001  =1   108         ET0     EQU      1          ;Timer0 interrupt
0002  =1   109         EX1     EQU      2          ;External interrupt 1
0003  =1   110         ET1     EQU      3          ;TimerI interrupt
0004  =1   111         EI2     EQU      4          ;I2C interrupt
0007  =1   112         EA      EQU      7          ;All interrupt enable/disable bit
=1   113
=1   114         ; I2C Control Register
0098  =1   115         I2CON   EQU      098h
=1   116         ; Input (read) is bit #s for JB etc...
0007  =1   117         RDAT     EQU      7          ;receive data
0006  =1   118         ATN      EQU      6          ;attention
0005  =1   119         DRDY     EQU      5          ;data ready
0004  =1   120         ARL      EQU      4          ;arbitration loss
0003  =1   121         STR      EQU      3          ;start
0002  =1   122         STP      EQU      2          ;stop

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC OBJ          LINE    SOURCE
0001             =1 123   MST      EQU      1           ;master
                  =1 124   ; Output (write) is binary values for MOV I2CON,#...
0080             =1 125   CXA      EQU     80h         ;clear xmit active
0040             =1 126   IDLE    EQU     40h         ;set to idle slave
0020             =1 127   CDR      EQU     20h         ;clear data ready
0010             =1 128   CARL    EQU     10h         ;clear arbitration loss
0008             =1 129   CSTR    EQU     08h         ;clear start
0004             =1 130   CSTP    EQU     04h         ;clear stop
0002             =1 131   XSTR    EQU     02h         ;transmit start
0001             =1 132   XSTP    EQU     01h         ;transmit stop
                  =1 133
                  =1 134   ; I2C Data Register
0099             =1 135   I2DAT   EQU     099h
0080             =1 136   XDAT    EQU     80h         ;transmit data
                  =1 137
                  =1 138   ; I2C Configuration Register
00D8             =1 139   I2CFG   EQU     0D8h
0080             =1 140   SLAVEN  EQU     80h         ;enable slave mode
0007             =1 141   SLAVENB EQU     7
0040             =1 142   MASTRQ  EQU     40h         ;master request
0006             =1 143   MASTRQB EQU     6
0020             =1 144   CLRTI   EQU     20h         ;clear timerI
0005             =1 145   CLRTIB  EQU     5
0010             =1 146   TIRUN   EQU     10h         ;timerI run
0004             =1 147   TIRUNB  EQU     4
                  =1 148
0088             =1 149   TCON    EQU     088h         ;Timer Control
                  =1 150
                  =1 151   ; 83C751 SFRs
0080             =1 152   P0      EQU     080h
0080             =1 153   SCL     BIT     P0.0
0081             =1 154   SDA     BIT     P0.1
0081             =1 155   SP      EQU     081h
0082             =1 156   DPL     EQU     082h
0083             =1 157   DPH     EQU     083h
008A             =1 158   TL      EQU     08Ah         ;Timer Lo
008B             =1 159   RTL     EQU     08Bh         ;Reload TL
008C             =1 160   TH      EQU     08Ch         ;Timer Hi
008D             =1 161   RTH     EQU     08Dh         ;Reload TH
0090             =1 162   P1      EQU     090h
00B0             =1 163   P3      EQU     0B0h
00D0             =1 164   PSW     EQU     0D0h
00E0             =1 165   ACC     EQU     0E0h
00F0             =1 166   B       EQU     0F0h
                  =1 167
                  =1 168   ;***** End of include file
169
170   $ INCLUDE( /dskbus/include/ab.inc )
=1 171   ;*****module ab.inc*****
=1 172   ;
=1 173   ; Ab Base Protocol definitions
=1 174   ; Environment: 83C751 Assembler
=1 175   ;
=1 179   ; Date      Revision      Perpetrator
=1 180   ;
=1 181   ; 04-Sep-91  ----          Mark Shepard
=1 182   ; Changed I_MsgCheck to I_Error, kept I_MsgCheck for backpatibility.
=1 183   ; Added App_Error (0xb4 to correspond to I_Error).
=1 184   ; Changed Sig_Attn from 0 to 3 to make KB code easier.

```


ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
=1    185          ;
=1    186          ;   04-Aug-91   ----           Mark Shepard
=1    187          ;           Renamed to generic "ab.inc"
=1    188          ;           Added defines for header structure (mainly for documentation purposes).
=1    189          ;           Added defines for bus signal codes (RESET, HALT, ATTN, etc.)
=1    190          ;
=1    191          ;   22-May-91   X0.2           Peter Sichel
=1    192          ;           Added Vendor command codes, and I_MsgCheck.
=1    193          ;
=1    194          ;   30-Jan-91   X0.1           Mark Shepard
=1    195          ;           Created from ODB base protocol spec, X0.7.
=1    196          ;
=1    197          ;
=1    198          $EJ
=1    199          ;*****
=1    200          ; Desktop bus command codes for all Interface-part and Application-part
=1    201          ; commands in the Base Protocol spec.
=1    202          ;
=1    203          ; Naming Convention -
=1    204          ; Interface-Part codes are prefixed with "I_",
=1    205          ; Application-Part with "App_". Codes specific to a particular
=1    206          ; sub-protocol (e.g. Keyboard Protocol) could be prefixed with
=1    207          ; "Key_", "Kb_", or "Kbp_".
=1    208          ;
=1    209          ; Definitions for sub-protocols should go in separate include files.
=1    210          ; (keyp.inc, locp.inc, textp.inc, etc.).
=1    211          ;
0000  =1    212          I_NoMsg      EQU      000h   ; special value, means not a valid message
=1    213
00F0  =1    214          I_Reset      EQU      0f0h   ; reset device
00F1  =1    215          I_IdReq      EQU      0f1h   ; Identify request
00F2  =1    216          I_AsgnAdr    EQU      0f2h   ; assign address
00F3  =1    217          I_CapReq     EQU      0f3h   ; capabilities (fragment) request
=1    218
00E0  =1    219          I_Attn       EQU      0e0h   ; power-on/reset attention
00E1  =1    220          I_IdReply    EQU      0e1h   ; identify reply
00E3  =1    221          I_CapReply   EQU      0e3h   ; capabilities (fragment) reply
00E4  =1    222          I_Error     EQU      0e4h   ; (the future) interface/bus std error report
00E4  =1    223          I_MsgCheck   EQU      0e4h   ; *** obsolete, don't use in new code ***
=1    224
00C0  =1    225          I_Vendor0    EQU      0c0h   ; vendor reserved command
00C1  =1    226          I_Vendor1    EQU      0c1h   ; vendor reserved command
00C2  =1    227          I_Vendor2    EQU      0c2h   ; vendor reserved command
00C3  =1    228          I_Vendor3    EQU      0c3h   ; vendor reserved command
=1    229
00A0  =1    230          App_Sig     EQU      0a0h   ; hardware signal
00A1  =1    231          App_TestReply EQU      0a1h   ; test reply (for a specific application-part)
=1    232
00B1  =1    233          App_Test     EQU      0b1h   ; self-test result request
00B4  =1    234          App_Error   EQU      0b4h   ; (the future) std application error report
=1    235
=1    236          ;*****
=1    237          ; Well-known I2C addresses used by desktop bus peers
=1    238          ;
=1    239          ; I2C Address Allocation
=1    240          ; row          column
=1    241          ; A6,A5,A4 / A3,A2,A1,A0          R/W=0 (write)
=1    242          ; Host: 2/8          50h
=1    243          ; Devices: 2/9-2/15, 3/0-3/7          52-6Eh (even numbers only)

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
                                =1  244      ; Device default: 3/7                6Eh
                                =1  245
0050  =1  246      Adr_Host      EQU    050h    ; standard Host address
006E  =1  247      Adr_Default   EQU    06eh    ; default (pwr-up) address for peripherals
                                =1  248
                                =1  249      ;*****
                                =1  250      ; Bus-Signal codes defined at the Base Protocol level
                                =1  251      ;
0001  =1  252      Sig_Reset     EQU    1      ; hardware reset!
0002  =1  253      Sig_Halt      EQU    2      ; debugging interrupt
0003  =1  254      Sig_Attn     EQU    3      ; general-purpose interrupt from User
                                =1  255
                                =1  256      ;*****
                                =1  257      ; Defines related to message structure
                                =1  258      ;
                                =1  259      ; Declarations prefixed with "AbWire_" refer to Ab objects, fields, etc.
                                =1  260      ; as transmitted across the i2c "wire" (as opposed to the "optimized frame
                                =1  261      ; format used between the host and 83c751 host-ctrlr).
                                =1  262      ;
                                =1  263      ;
0003  =1  264      AbWire_HdrSiz EQU    3      ; Ab Header Size on the wire: dst + src + PLen
007F  =1  265      AbWire_LenMask EQU    7fh    ; Ab mask for length field
007F  =1  266      AbWire_MaxLen EQU    127    ; Ab maximum data bytes in message
                                =1  267
                                =1  268      ;***** End of include file
                                269
                                270
                                271      $EJ
                                272      ;*****
                                273      ;* Hardware/timing-dependent constants *
                                274      ;*****
                                275
                                276      ;CT      EQU    02h    ;CT1, CT0 fmax = 16.8 MHz
                                277      ;CT      EQU    01h    ;CT1, CT0 fmax = 14.25 MHz
                                278      ;CT      EQU    00h    ;CT1, CT0 fmax = 11.7 MHz
0003  =1  279      CT      EQU    03h    ;CT1, CT0 fmax = 9.14 MHz
                                280
009A  =1  281      IntEnab     EQU    09Ah    ;enable EA+EI2+ET1+ET0.
0010  =1  282      INIT_TCON    EQU    010h    ;Timer0 init for internal operation.
                                283
00DB  =1  284      DEF_RTH     EQU    0DBh    ;Sampling interval (14ms with 8 MHz clock).
008B  =1  285      DEF_RTL     EQU    08Bh    ; Used as default Timer0 reload value.
0002  =1  286      MSECH      EQU    002h    ;Timer offset for 1 ms with 8 MHz clock.
009A  =1  287      MSECL      EQU    09Ah    ; 029Ah=666 * 3/2 mms/tick = 1000 ticks = 1ms.
                                288
0008  =1  289      DelayATN    EQU    8      ;about 50µS (wait for I2C activity)
                                290
0010  =1  291      CapFragLen  EQU    16     ;Capabilities fragment length.
                                292
                                293      ; *** Hardware Interface Notes ***
                                294      ; P3 is used to read the X-Y quadrature inputs.
                                295      ;   P3.0 = XB
                                296      ;   P3.1 = XA
                                297      ;   P3.2 = YA
                                298      ;   P3.3 = YB
                                299      ;
                                300      ; Notice P3.0 is connected to the B side of the X encoder
                                301      ; while P3.2 is connected to the A side of the Y encoder.
                                302      ; This is to compensate for the orientation of the inclined cylinders.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
                                303    ; Positive X movement causes clockwise rotation of the encoder shaft.
                                304    ; Positive Y movement causes counter clockwise rotation of the encoder shaft.
                                305    ;
                                306    ; P1 is used to read the switch inputs.
                                307    ;   P1.0 = middle mouse button.
                                308    ;   P1.1 = left mouse button.
                                309    ;   P1.2 = right mouse button.
                                310    ;
                                311    ; 6-Feb-1991 This order of buttons reflects the current PCB layout
                                312    ;         and is subject to change.
                                313
                                314    $EJ
                                315    ; Locator messages (device defined)
                                316    ;   send types
0003   317    LD_Position    EQU    3        ;Device state report
                                318
                                319    ;   receive types
00B0   320    LD_Poll        EQU    0B0h
0082   321    LD_SetInterval EQU    082h
                                322
                                323
                                324    ; SelfTest errors (0=success)
                                325    NO_ERROR      EQU    0
0000   326    ROM_ERROR     EQU    1
0001   327    RAM_ERROR    EQU    2
0002   328
                                329
                                330    $EJ
                                331    ;*****
                                332    ; RAM usage *
                                333    ;*****
                                334    ; I2C variables
                                335    ; Register bank 0
                                336
                                337                                ;R0 - command parameter
                                338                                ;R1 - index pointer
0002   338    I2CDat    DATA  02h        ;The byte being sent or received.
0003   339    BitCnt    DATA  03h        ;I2C bit counter
0004   340    ByteCnt   DATA  04h        ;I2C message byte counter
0005   341    ATNCnt   DATA  05h        ;ATN Retry counter
0006   342    I2CCxt   DATA  06h        ;I2C context, the event the CPU is waiting for.
0007   343    Temp     DATA  07h        ;All purpose temp
                                344
                                345    ; Desktop Bus Protocol
0008   346    MyAddr   DATA  08h        ;I2C address assigned this device.
0009   347    NACnt   DATA  09h        ;Negative Ack retry counter.
000A   348    RcvType  DATA  0Ah        ;Message or command type being received.
000B   349    SndType  DATA  0Bh        ;Message type being sent, or pending.
000C   350    MsgLen   DATA  0Ch        ;Message length field.
000D   351    Check   DATA  0Dh        ;Message checksum.
000E   352    RandH    DATA  0Eh        ;Random number (2 bytes)
000F   353    RandL    DATA  0Fh
                                354
                                355    ; Locator report buffer
0010   356    ReportBuf EQU    10h      ;Beginning of position report buffer.
0010   357    Switch2   DATA  10h      ;Switch data (Buttons 9 to 16)
0011   358    Switch1   DATA  11h      ;Switch data (Buttons 1 to 8 )
0012   359    XBUF2    DATA  12h      ;XData transmission buffer (MSB)
0013   360    XBUF1    DATA  13h      ;XData transmission buffer (LSB)
0014   361    YBUF2    DATA  14h      ;YData transmission buffer (MSB)

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
0015          362      YBUF1      DATA    15h      ;YData transmission buffer (LSB)
          363
0016          364      XCOUNT   DATA    16h      ;XData
0017          365      YCOUNT   DATA    17h      ;YData
0018          366      MARKER    EQU      YCOUNT+1
          367
0018          368      CapOffset DATA    18h      ;Capabilities fragment offset
0019          369      CapLen    DATA    19h      ;Capabilities fragment length
001A          370      SelfTest  DATA    1Ah      ;Self Test result variable location
001B          371      RomSum    DATA    1Bh      ;Hold ROM checksum
001C          372      SampleClock DATA    1Ch      ;Time stamp of last sample
          373      ;3 spare.
          374
          375      ; Bit addressable area begins at 20h
          376      ; Location and switch compare variables
0020          377      LastXY    DATA    20h      ;Last X/Y Position (from P3)
0021          378      LastSW    DATA    21h      ;Last Switch Status (from P1)
0022          379      TranXY    DATA    22h      ;Port 3 transition register (bit addressable)
          380
          381      $EJ
          382      ; I2C status and position scanning flags.
0023          383      Flags    DATA    23h
0018          384      Prot     BIT     Flags.0    ;1=C/S message; 0=device data stream.
0019          385      SendRpt  BIT     Flags.1    ;New Position Report flag.
001A          386      Movement BIT     Flags.2    ;Movement detected flag.
001B          387      RxEnable  BIT     Flags.3    ;I2C receive enable.
001C          388      TxSelfRst BIT     Flags.4    ;Indicates send Self-Reset after Assign (0).
001D          389      KeepID    BIT     Flags.5    ;Set means keep same device number.
001E          390      NotMyID  BIT     Flags.6
001F          391      AA        BIT     Flags.7    ;Assert Acknowledge.
          392
0024          393      FlagsA    DATA    24h
0020          394      MsgCheck  BIT     FlagsA.0    ;I2C message checksum or framing error.
          395
          396      ;11 spare
          397
0030          398      StackBase DATA    30h      ;16 byte stack
          399      ; need 2 bytes per subroutine
          400      ; 4 bytes per interrupt (max 2)
          401      ;RAM ends at 3Fh
          402      ; 50 bytes RAM used, 64 maximum.
          403
          404
          405      $EJ
          406      ;*****
          407      ; Code begins
          408      ;*****
0000          409      ORG      0          ;Power up reset vector.
0000 01B4          410      AJMP     PwrUp
          411
0003          412      ORG      003h      ;INT0 is not used.
0003 01B4          413      AJMP     PwrUp
          414      ;
          415      ; Use this spare byte to hold the ROM checksum complement.
0005 51          416      DB      051h
          417
000B          418      ORG      00Bh      ;Timer0 interrupt vector
000B 0125          419      AJMP     Timer0    ;16 bit system tick generator
          420

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
0013          421      ORG    013h    ;INT1 is not used.
0013 01B4      422      AJMP   PwrUp
          423
001B          424      ORG    01Bh    ;TimerI interrupt vector
001B 014F      425      AJMP   TimerI ;I2C time out timer
          426
0023          427      ORG    023h    ;I2C interrupt vector
0023 21D8      428      AJMP   I2CINT
          429
          430
          431      $EJ
          432      ;*****
          433      ; Timer0 Interrupt
          434      ; Position sampling interval time out.
          435      ;*****
          436
0025 C0D0      437      Timer0: PUSH  PSW          ;Save registers we need.
0027 C0E0      438          PUSH  ACC
0029 E508      439          MOV   A,MyAddr          ;Check for default address.
002B B46E02    440          CJNE  A,#Adr_Default,AddrOK
002E 801A      441          SJMP  T0Exit          ;Don't send Position reports
          442          ; to default address.
0030 E590      443      AddrOK: MOV   A,P1          ;Get switch info from P1
          444          ; (0=button depressed).
0032 F4        445          CPL   A                ;Complement
0033 5407      446          ANL  A,#00000111b    ;We need low 3 bits of P1
0035 B52105    447          CJNE  A,LastSW,CHNSWI ;if new, switches did change
          448          ;Switches did not change, check movement.
0038 301A0F    449          JNB  Movement,T0Exit
003B 8009      450          SJMP  T0Send          ;Yes, go send report.
          451
003D F521      452      CHNSWI: MOV   LastSW,A          ;Save LastSW for next compare.
          453          ;Re-order switches from RLM to MRL until PCB is fixed.
003F 13        454          RRC   A
0040 92E2      455          MOV   ACC.2,C
0042 5407      456          ANL  A,#00000111b
0044 F511      457          MOV   Switch1,A          ;Move to output buffer.
          458
0046 D219      459      T0Send: SETB  SendRpt          ;Set to send Position report.
0048 D2DE      460          SETB  I2CFG.MASTRQB      ;Request to be master.
          461
004A D0E0      462      T0Exit: POP   ACC
004C D0D0      463          POP   PSW
004E 32        464          RETI
          465
          466      $EJ
          467      ;*****
          468      ; TimerI interrupt
          469      ; The I2C bus has timed out,
          470      ; no SCL for at least 1020 machine cycles during an active frame.
          471      ; Since SCL is stuck, we can't wait for DRDY.
          472      ; Try to fix it manually.
          473      ;*****
          474
004F C2AF      475      TimerI: CLR   IE.EA          ;Disable interrupts.
0051 75D823    476          MOV   I2CFG,#CLRTI+CT ;Clear interrupt and turn off TimerI.
          477          ; manually clear SLAVEN & MASTRQ.
0054 7598BC    478          MOV   I2CON,#CXA+CARL+CDR+CSTR+CSTP ;Clear I2C flags.
0057 758130    479          MOV   SP,#StackBase ;Reset SP.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
                                480
                                481      ; Attempt to regain control of the I2C bus after a bus fault.
005A D280        482      FixBus:  SETB  SCL          ;Insure I/O port is not locking I2C.
005C D281        483          SETB  SDA
005E 308020      484          JNB   SCL,ResetBus    ;If SCL is low, bus cannot be fixed.
0061 208113      485          JB   SDA,RStop      ;If SCL & SDA are high, force a stop.
                                486          ;SDA is low, attempt to release SDA by clocking SCL.
0064 750309      487          MOV   BitCnt,#9      ;Set max # of tries to clear bus.
0067 C280        488      ClockBus: CLR   SCL          ;Force an I2C clock.
0069 11AF        489          ACALL SDelay
006B 208109      490          JB   SDA,RStop      ;Did it work?
006E D280        491          SETB  SCL
0070 11AF        492          ACALL SDelay
0072 D503F2      493          DJNZ  BitCnt,ClockBus ;Repeat clocks until SDA clears or retry limit.
0075 800A        494          SJMP  ResetBus      ;Failed to fix bus by this method.
                                495
0077 C281        496      RStop:   CLR   SDA          ;Try forcing a stop since
0079 11AF        497          ACALL SDelay          ; SCL & SDA are both high.
007B D280        498          SETB  SCL
007D 11AF        499          ACALL SDelay
007F D281        500          SETB  SDA
                                501
                                502      ResetBus: ;Wait for bus to clear.
0081 3080FD      503          JNB   SCL,$
0084 3081FD      504          JNB   SDA,$
0087 7401        505          MOV   A,#1          ;Pause for bus to stabilize.
0089 11A7        506          ACALL LDelay
                                507          ;Re-enable I2C functions.
008B 750B00      508          MOV   SndType,#I_NoMsg ;Cancel message if any.
008E D21B        509          SETB  RxEnable      ;Enable receiving.
0090 1194        510          ACALL InitI2C     ;Initialize I2C.
0092 2154        511          AJMP  MAIN        ;Restart MAIN.
                                512
                                513
                                514      ; Initialize I2C functions
0094 75D893      515      InitI2C: MOV   I2CFG,#SLAVEN+TIRUN+CT ;Enable I2C
                                516          ;Set I2C to be idle receiver & clear all flags.
0097 7598FC      517          MOV   I2CON,#CXA+IDLE+CDR+CRL+CSTR+CSTP
009A 750601      518          MOV   I2CCxt,#RXIDLE ;Context idle receiver
009D 31F3        519          ACALL XRETI      ;Clear pending interrupt if any.
009F 75A89A      520          MOV   IE,#IntEnab  ;Enable interrupts (EA+EI2+ET1+ET0)
00A2 7410        521          MOV   A,#16
00A4 11A7        522          ACALL LDelay
00A6 22          523          RET
                                524
                                525      ; Long Delay, A/2 milliseconds.
00A7 7FA6        526      LDelay: MOV   R7,#166
00A9 DFFE        527          DJNZ  R7,$
00AB D5E0F9      528          DJNZ  ACC,LDelay
00AE 22          529          RET
                                530
                                531      ; Short delay routine (10 machine cycles).
00AF 11B1        532      SDelay: ACALL SD1
00B1 00          533      SD1:   NOP
00B2 00          534          NOP
00B3 22          535          RET
                                536
                                537      $EJ

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
                                538      ;*****
                                539      ; Power up initialization starts here
                                540      ;*****
                                541      ; Reset command branches to here.
00B4  75086E      542      PwrUp:  MOV    MyAddr,#Adr_Default      ;Re-initialize default address.
00B7  E4          543          CLR    A
00B8  F5A8        544          MOV    IE,A          ;Disable interrupts.
00BA  75D803      545          MOV    I2CFG,#CT     ;Disable I2C
00BD  53D0E7      546          ANL   PSW,#0E7h     ;Select RB0.
00C0  758130      547          MOV    SP,#StackBase
                                548          ; Initialize I/O pins
00C3  758003      549          MOV    P0,#03h      ;Initialize I2C I/O pins, SCL & SDA high
00C6  7590FF      550          MOV    P1,#0FFh     ;P1 set for input to read switches.
00C9  75B0FF      551          MOV    P3,#0FFh     ;P3 set for input to read X-Y.
                                552          ; Initialize Timer0
00CC  758DDB      553          MOV    RTH,#DEF_RTH ;14 ms at 8 MHz
00CF  758B8B      554          MOV    RTL,#DEF_RTL
00D2  758810      555          MOV    TCON,#INIT_TCON ;Running, internal mode clock/12.
                                556
                                557      ;*****
                                558      ; Perform ROM Test (0-7FFh, 2K bytes)
                                559      ;*****
00D5  75F000      560      TestROM:MOV  B,#0          ;Initialize sum
00D8  900000      561          MOV    DPTR,#0000h     ;Set pointer to start of ROM
00DB  C3          562          CLR    C
00DC  E4          563      SumLp: CLR  A
00DD  93          564          MOVC  A,@A+DPTR       ;Get byte from ROM
00DE  35F0        565          ADDC  A,B             ;Add sum
00E0  F5F0        566          MOV    B,A           ;Save sum in B
00E2  A3          567          INC  DPTR
00E3  E583        568          MOV    A,DPH         ;Check if ROM complete
00E5  B408F4      569          CJNE  A,#08h,SumLp
00E8  5002        570          JNC   TestSum        ;Add carry if set
00EA  05F0        571          INC  B
00EC  E5F0        572      TestSum:MOV  A,B
00EE  6007        573          JZ    TestRAM        ;If zero, ROM is Okay
00F0  751A01      574          MOV    SelfTest,#ROM_ERROR
00F3  F51B        575          MOV    RomSum,A      ;Save bad checksum.
00F5  8023        576          SJMP  BadMem1
                                577
                                578      $EJ
                                579      ;*****
                                580      ; Perform RAM test (0-3Fh, 64 bytes)
                                581      ; Does not test special function registers.
                                582      ; A, B, and R0 are not preserved.
                                583      ;*****
00F7  78AA        584      TestRAM:MOV  R0,#0AAh     ;Test RAM location 0.
00F9  B8AA1B      585          CJNE  R0,#0AAh,BadMem
00FC  7855        586          MOV    R0,#055h
00FE  B85516      587          CJNE  R0,#055h,BadMem
0101  783F        588          MOV    R0,#3Fh       ;Init R0 to top of RAM.
0103  74AA        589          MOV    A,#0AAh       ;Test alternate bits.
0105  86F0        590      ChkRAM:MOV  B,@R0       ;Save previous contents.
0107  F6          591          MOV    @R0,A
0108  B6AA0C      592          CJNE  @R0,#0AAh,BadMem
010B  23          593          RL   A              ;Test other bits.
010C  F6          594          MOV    @R0,A
010D  B65507      595          CJNE  @R0,#055h,BadMem
0110  23          596          RL   A

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
0111 A6F0         597          MOV      @R0,B           ;Restore contents.
0113 D8F0         598          DJNZ    R0,ChkRAM
0115 8005         599          SJMP    MemOK
0117 751A02       600          BadMem: MOV     SelfTest,#RAM_ERROR
                                601
                                602          ; Report bad memory. Since a memory problem was detected, the
                                603          ; normal I2C transmit code may be unreliable. Hope it isn't
                                604          ; a fatal problem and use it anyway. There's only so much
                                605          ; we can do. Could add special code here.
011A 8005         606          BadMem1: SJMP   InitRAM
                                607
011C 751A00       608          MemOK:  MOV     SelfTest,#0
011F 8000         609          SJMP    InitRAM
                                610
                                611
                                612          $EJ
                                613          ;*****
                                614          ;Initialize RAM
                                615          ;*****
0121 E4           616          InitRAM:CLR    A           ;Init Acc to Zero
0122 F50A         617          MOV     RcvType,A         ;Clr RcvType
0124 F50B         618          MOV     SndType,A         ;Clr SndType
0126 7910         619          MOV     R1,#ReportBuf     ;Clear report buffer & compare vars
0128 F7           620          ClrBuf: MOV    @R1,A       ;Clear location
0129 09           621          INC     R1                 ;Go to next location
012A B918FB       622          CJNE   R1,#Marker,ClrBuf  ;Check for end
012D F523         623          MOV     Flags,A           ;Init Flags
012F F521         624          MOV     LastSW,A          ;Init last switch image.
0131 750905       625          MOV     NACnt,#5          ;Negative Ack retry count.
                                626
                                627          ;Init LastXY
0134 E5B0         628          MOV     A,P3              ;Read X-Y quad inputs to init LastXY.
0136 540F         629          ANL    A,#00Fh           ;Low 4 bits only.
0138 F520         630          MOV     LastXY,A          ;Init LastXY
                                631
                                632          ;*****
                                633          ;Set up to transmit self test report.
                                634          ;*****
013A C21B         635          CLR     RxEnable          ;Disable receiving.
013C 1194         636          SetUp: ACALL  InitI2C
                                637
013E 750601       638          Report: MOV    I2CCxt,#RXIDLE ;Set context idle receiver. ;*RC*
0141 750BE0       639          MOV     SndType,#I_Attn
0144 D2DE         640          SETB   I2CFG.MASTRQB      ;Request to be master.
0146 20DEFD       641          JB     I2CFG.MASTRQB,$     ;Wait for message sent. ;*PAS*
0149 E51A         642          MOV     A,SelfTest
014B 6005         643          JZ     RepDn              ;Go if Selftest OK.
                                644
                                645          ;Selftest failed.
                                646          ;Send 2nd report and try to start anyway.
014D 751A00       647          MOV     SelfTest,#NO_ERROR
0150 80EC         648          SJMP    Report
                                649
0152 D21B         650          RepDn: SETB    RxEnable     ;Enable Receiver and
                                651          ; fall through to MAIN.
                                652
                                653          $EJ

```


ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
654          ;*****
655          ; Main Routine
656          ;
657          ; Sample X-Y quadrature inputs and compute mouse movement.
658          ; Accuracy requirement is: +/- 3% 0-10 inches per second.
659          ; 10 inches per second @ 200 dpi means up to 2000 input changes/second.
660          ; Minimum (Nyquist) sampling rate is 4000/sec or sample every 250µs.
661          ; This is only two character times at 80k bps.
662          ; For best accuracy, should sample between every I2C character.
663          ;
664          ; Sample timing with 8 MHz crystal:
665          ;   no transition:      6 cycles      9µs
666          ;   X or Y transition: 25-31 cycles 38-47µs
667          ;   X and Y transition: 42-46 cycles 63-69µs
668          ;
669          ;*****
670
0154          671      MAIN:
0154 E58A      672          MOV     A,TL           ;Read timer to wait at least
0156 951C      673          SUBB    A,SampleClock ; 64 cycles (96 µsec) between samples.
0158 30E6F9    674          JNB     ACC.6,MAIN
675          ;Take a sample
676          ;CLR     IE.EA         ;Protect sample code from interrupts.
015B 858A1C   677          MOV     SampleClock,TL
015E 3162      678          ACALL   Sample         ;Sample X-Y quadrature inputs.
679          ;SETB    IE.EA
0160 80F2      680          SJMP   MAIN
681
682          ; Sample X-Y quadrature inputs and update X-Y counters.
683          ;   b3-b0 = YB YA XB XA.
684          ;
685          ;   Channel A:  0 1 1 0 0 1 1 0 0    --->positive movement
686          ;   Channel B:  0 0 1 1 0 0 1 1 0    <---negative movement
687
688          ;   A and C are not preserved.
689          ;
0162 E5B0      690      Sample: MOV     A,P3           ;Read X & Y position detectors.
0164 540F      691          ANL     A,#00001111B ;We only need the low 4 bits
0166 B52001    692          CJNE   A,LastXY,TRAN ;Compare to last image.
0169 22        693          RET                      ; If no change, return.
694
016A 852022   695      TRAN:  MOV     TranXY,LastXY ;Set up to calculate XY transition.
016D F520      696          MOV     LastXY,A         ;Save new P3 image.
016F 6222      697          XRL     TranXY,A         ;Mark transition bits (1=changed).
698
0171 20100A   699          XPULSE: JB     TranXY.0,XA ;Branch on bit transitions.
0174 201113   700          JB     TranXY.1,XB
0177 201232   701          YPULSE: JB     TranXY.2,YA
017A 20133B   702          JB     TranXY.3,YB
017D 22        703          RET
704
705          ;Decode direction from quadrature.
706          ;Change in XA
017E E520      707          XA:   MOV     A,LastXY
0180 5403      708          ANL     A,#00000011B ;Get X' X
0182 6010      709          JZ     XDEC         ;If 00, backward
0184 6403      710          XRL     A,#00000011B
0186 600C      711          JZ     XDEC         ;If 11, backward
0188 8016      712          SJMP   XINC         ;01 or 10, forward.
    
```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE  SOURCE
                                713          ;Change in XB
018A E520        714  XB:  MOV    A,LastXY
018C 5403        715          ANL    A,#00000011B    ;Get X' X
018E 6010        716          JZ     XINC           ;If 00, forward
0190 6403        717          XRL   A,#00000011B
0192 600C        718          JZ     XINC           ;If 11, forward
                                719          ;01 or 10, backward, fall through to decrement
                                720
0194 7480        721  XDEC: MOV    A,#080h    ;Do not decrement if
0196 6516        722          XRL   A,XCOUNT    ; count already at minimum -127.
0198 6004        723          JZ     XDEC2
019A 1516        724          DEC   XCOUNT
019C D21A        725          SETB  Movement    ;Note position has changed.
019E 80D7        726  XDEC2: SJMP  YPULSE    ;Check for possible Y pulse.
                                727
01A0 747F        728  XINC: MOV    A,#07Fh    ;Do not increment if
01A2 6516        729          XRL   A,XCOUNT    ; count already at maximum 127.
01A4 6004        730          JZ     XINC2
01A6 0516        731          INC   XCOUNT
01A8 D21A        732          SETB  Movement    ;Note position has changed.
01AA 80CB        733  XINC2: SJMP  YPULSE    ;Check for possible Y pulse.
                                734
                                735          ;Change in YA
01AC E520        736  YA:  MOV    A,LastXY
01AE 540C        737          ANL   A,#00001100B    ;Get Y' Y
01B0 6010        738          JZ     YDEC           ;If 00, backward
01B2 640C        739          XRL   A,#00001100B
01B4 600C        740          JZ     YDEC           ;If 11, backward
01B6 8015        741          SJMP  YINC           ;01 or 10, forward.
                                742          ;Change in YB
01B8 E520        743  YB:  MOV    A,LastXY
01BA 540C        744          ANL   A,#00001100B    ;Get Y' Y
01BC 600F        745          JZ     YINC           ;If 00, forward
01BE 640C        746          XRL   A,#00001100B
01C0 600B        747          JZ     YINC           ;If 11, forward
                                748          ;01 or 10, backward, fall through to decrement.
                                749
01C2 7480        750  YDEC: MOV    A,#080h    ;Do not decrement if
01C4 6517        751          XRL   A,YCOUNT    ; count already at minimum -127.
01C6 6004        752          JZ     YDEC2
01C8 1517        753          DEC   YCOUNT
01CA D21A        754          SETB  Movement    ;Note position has changed.
01CC 22          755  YDEC2: RET
                                756
01CD 747F        757  YINC: MOV    A,#07Fh    ;Do not increment if
01CF 6517        758          XRL   A,YCOUNT    ; count already at maximum 127.
01D1 6004        759          JZ     YINC2
01D3 0517        760          INC   YCOUNT
01D5 D21A        761          SETB  Movement    ;Note position has changed.
01D7 22          762  YINC2: RET
                                763
                                764  $EJ
                                765          ; I2C message processing contexts:
0001          766  RXIDLE EQU    1          ;Idle receiver waiting for start.
0002          767  RXBIT  EQU    2          ;Waiting to receive a bit.
0003          768  RXACK  EQU    3          ;Waiting for ACK to complete.
                                769
0004          770  TXBIT  EQU    4          ;Waiting to send a bit.
0005          771  TXREAD EQU    5          ;Waiting to read ACK.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ      LINE      SOURCE
0006                772      TXACK  EQU    6                ;Waiting for ACK.
                773
                774
                775
                776      ; I2C Interrupt Reasons
                777      ;   Events CPU might be waiting for
                778      ;   Receive
                779      ;   (1) Start signal detected by idle slave (DRDY)
                780      ;   (2) Next bit received (DRDY)
                781      ;   (3) Acknowledge has been sent (DRDY)
                782      ;   Transmit
                783      ;   (4) Bus mastership granted (START and MASTER)
                784      ;   (5) Ready to transmit next bit (DRDY)
                785      ;   (6) Acknowledge received (DRDY)
                786      ;   Unsolicited
                787      ;   (7) Arbitration loss (ARL)
                788      ;   (8) Sender aborted message (STOP)
                789      ;   (9) Sender started new message before slave became idle (START)
                790      ;
                791      ; Only some of these events can occur at any time depending on
                792      ; the state of sending or receiving a message.
                793      ;
                794      ; When the interrupt occurs, the keyboard needs to recover
                795      ; the context from which it was sending or receiving a
                796      ; message. This context is maintained as follows:
                797      ;
                798      ; I2Ccxt  I2C context, what event is expected.
                799      ; I2CDat  The byte being sent or received.
                800      ; BitCnt  Where we are in the sending or receiving the byte.
                801      ; ByteCnt  Where we are in sending or receiving a message
                802      ; Check   Computed checksum.
                803      ; RcvType  The message or command type being received.
                804      ;           This may determine how successive bytes
                805      ;           are to be processed.
                806      ; SndType  Type of message being sent or pending.
                807      ;           This will determine how bytes are transmitted.
                808      ; SendRpt  Flag indicating CPU is waiting to send a Position
                809      ;           report (and requested to become master).
                810
                811
                812
                813      $EJ
                814      ;*****
                815      ; Enter I2C Interrupt
                816      ;*****
01D8  C2AC      817      I2CINT: CLR    IE.EI2                ;disable the I2C interrupt
01DA  31F3      818                ACALL  XRETI                ;then re-enable others
01DC  C0D0      819                PUSH   PSW                ;save registers
01DE  C0E0      820                PUSH   ACC
                821
                822      ; Dispatch interrupt
01E0  309911    823      DISPATCH: JNB   I2CON.MST,SLAVE
01E3  41B1      824                AJMP   MASTER                ;go if we're master
                825
                826      ; Wait for ATN
01E5  7D08      827      WaitATN: MOV    R5,#DelayATN        ;Load ATN count (about 50ms)
01E7  209EF6    828      Wait1:  JB     I2CON.ATN,DISPAT      ;If ATN, dispatch next event,
01EA  DDFB      829                DJNZ  R5,Wait1                ; else loop to try again.
                830                ;If not seen after count tries,

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE  SOURCE
      831          ; return from I2C interrupt.
      832
      833          ; Exit I2C interrupt
      834          ; restore registers and return from interrupt
01EC  D0E0        835  I2CRTI: POP     ACC
01EE  D0D0        836          POP     PSW
01F0  D2AC        837          SETB   IE.EI2          ;re-enable I2C interrupt
01F2  22          838          RET              ;return to interrupted process
      839
01F3  32          840  XRETI: RETI          ;used at start of service routine
      841
      842
      843          $EJ
      844          ;*****
      845          ; SLAVE RECEIVER
      846          ;   R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
      847          ;*****
      848          ; Handle DRDY
      849          ;*****
01F4  309D5A      850  SLAVE: JNB     I2CON.DRDY,NDRDY          ;Is it DRDY?
      851          ; - context waiting for bit
01F7  BE0235      852          CJNE   R6,#RXBIT,NRxBit          ;Context waiting for bit?
01FA  EA          853  Rx0:  MOV     A,R2              ;Yes, get data in A
01FB  DB21        854  Rx1:  DJNZ   R3,N8Bit          ;8th bit?
      855
      856          ; Read 8th bit
01FD  A29F        857  MOV     C,I2CON.RDAT          ;Get 8th bit, don't clear ATN
01FF  33          858  RLC     A                    ;Include the 8th bit
0200  FA          859  MOV     R2,A                ;Put data in R2.
0201  620D        860  XRL    Check,A              ;XOR it to check
      861
      862          ;Send Acknowledge as appropriate.
0203  BC0007      863  CJNE   R4,#0,DoAck1          ;Address byte? (ByteCnt=0)
0206  B5080E      864  CJNE   A,MyAddr,NotMe        ;Is it my address?
0209  F50D        865  MOV     Check,A              ;Yes, initialize check
020B  C21F        866  CLR     AA                    ;AA=Flags.7
020D  852399      867  DoAck1: MOV    I2DAT,Flags      ;Assert Acknowledge (AA=Flags.7)
0210  7E03        868  MOV     R6,#RXACK            ;Set context waiting for ACK
0212  209D1D      869  JB     I2CON.DRDY,AckCmp      ;Can we skip waiting?
0215  21E5        870  AJMP   WaitATN              ;Wait for ATN.
      871          ; Not addressed to me
0217  7E01        872  NotMe: MOV    R6,#RXIDLE       ;Set context to be idle receiver
0219  75987C      873  MOV     I2CON,#CDR+CSTR+CSTP+CARL+IDLE
021C  21EC        874  AJMP   I2CRTI              ;Resume interrupted activity
      875
      876          ; Read bits 1-7
021E  C2E7        877  N8Bit: CLR    ACC.7
0220  4599        878  ORL    A,I2DAT              ;Include the bit, clear ATN.
0222  23          879  RL     A                    ;Data comes in at MSB.
0223  209DD5      880  JB     I2CON.DRDY,Rx1        ;If DRDY, short cut.
0226  209DD2      881  JB     I2CON.DRDY,Rx1
0229  209DCF      882  JB     I2CON.DRDY,Rx1        ;One more try.
022C  FA          883  MOV     R2,A                ;Put data back.
022D  21E5        884  AJMP   WaitATN
      885
      886          ; - context waiting for ACK to complete
022F  BE0311      887  NRxBit: CJNE   R6,#RXACK,NRxAck      ;Context waiting for ACK?
0232  7598A0      888  AckCmp: MOV    I2CON,#CDR+CXA        ;ACK complete, clr xmt.
      889          ;Process complete byte.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ      LINE      SOURCE
0235  618F      890          AJMP      DORXB
          891          ; Return is AJMP DNRXB.
0237  0C        892  DNRXB:  INC      R4              ;Increment ByteCnt.
          893          ;ACALL  Sample
0238  7E02      894  SetRXB:  MOV      R6,#RXBIT      ;Set context for next byte.
023A  7A00      895          MOV      R2,#0              ;Clear receive buffer
023C  7B08      896          MOV      R3,#8              ;BitCnt=8
023E  209DB9    897          JB       I2CON.DRDY,Rx0     ;If DRDY, short cut.
0241  21E5      898          AJMP      WaitATN          ;Wait for ATN
          899
          900          ; - context idle slave
0243  BE010B    901  NRxAck:  CJNE     R6,#RXIDLE,NRxBld ;Context idle slave?
0246  301BCE    902          JNB      RxEnable,NotMe     ;Am I enabled?
          903          ;Yes, initialize to receive first byte
0249  7B07      904          MOV      R3,#7              ;BitCnt=7 (remaining)
024B  E4        905          CLR      A                  ;Data in A
024C  FC        906          MOV      R4,A              ;ByteCnt=0
024D  7E02      907          MOV      R6,#RXBIT        ;I2CCxt=receive next bit
024F  411E      908          AJMP      N8Bit
          909          ; Context was not waiting for Next bit, ACK, or Idle slave.
          910          ; Could be ARL. Dispatch other flags.
          911          ; Do not clear DRDY, we'll come back after ARL if necessary.
0251  912  NRxBld:
          913
          914
          915          $EJ
          916          ;*****
          917          ; It wasn't DRDY, could be ARL, START, or STOP
          918          ; Handle ARL
          919          ;*****
0251  309C38    920  NDRDY:  JNB      I2CON.ARL,RxStop   ;Is it ARL?
          921          ;Yes, note MASTRQ is still on unless we were sending STOP.
          922          ;SndType is still pending. If it was a position report,
          923          ;indicate pending report in SendRpt in case SndType is needed.
0254  30DE2E    924  ARL0:   JNB      I2CFG.MASTRQB,Naddr  ;Was I sending STOP?
0257  E50B      925          MOV      A,SndType
0259  B40305    926          CJNE     A,#LD_Position,ARL01 ;Was I sending position?
025C  750B00    927          MOV      SndType,#I_NoMsg
025F  D219      928          SETB     SendRpt
0261  B4E405    929  ARL01:  CJNE     A,#I_Error,ARL1     ;Was I sending Error?
0264  750B00    930          MOV      SndType,#I_NoMsg
0267  D220      931          SETB     MsgCheck
0269  301B19    932  ARL1:   JNB      RxEnable,NAddr     ;Am I enabled?
026C  209B39    933          JB       I2CON.STR,SlvStart  ;Handle start.
026F  BC000A    934          CJNE     R4,#0,ARL3         ;Did we ARL in Address (ByteCnt=0)?
          935          ; Lost arbitration in address, set context to read rest
          936          ; of address in case message is to me.
0272  759810    937  ARL2:   MOV      I2CON,#CARL      ;Clear ARL.
0275  7E02      938          MOV      R6,#RXBIT        ;Set context waiting to read bit.
0277  209D80    939          JB       I2CON.DRDY,Rx0     ;If DRDY, short cut.
027A  21E5      940          AJMP      WaitATN
          941          ; Lost arbitration outside dst addr
027C  B4F006    942  ARL3:   CJNE     A,#I_Reset,NAddr  ;Was I sending I_Reset to my Addr?
027F  C21C      943          CLR      TxSelfRst         ;Yes, reset flag since I lost.
0281  C21F      944          CLR      AA                ;Acknowledge received bytes.
0283  80ED      945          SJMP     ARL2              ;Message must be for me.
          946          ; Message not for me, go back to idle receive.
0285  947  NAddr:
0285  7598FC    948  IdleS:  MOV      I2CON,#CXA+IDLE+CARL+CDR+CSTR+CSTP

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
0288 7E01         949          MOV     R6,#RXIDLE
028A 21EC         950          AJMP    I2CRTI
          951          $EJ
          952          ;*****
          953          ; Handle STOP                      ;*RC*  Check for STOP first
          954          ;*****

028C 309A16       955          RxStop: JNB    I2CON.STP,RxStart      ;Confirm it was Stop.
028F 759804       956          MOV     I2CON,#CSTP                ;Clear it.
0292 201F04       957          JB     AA,RxStop0                  ;Check end of message reached,
          958          ; Assert Acknowledge=1.
          959          ; Received STOP before end of message.

0295 D220         960          SETB   MsgCheck                    ;Signal interface error.
0297 D2DE         961          SETB   I2CFG.MASTRQB
0299 7E01         962          RxStop0: MOV   R6,#RXIDLE          ;Set context idle receiver.
029B 309E02       963          JNB    I2CON.ATN,RxStop1          ;If ATN, dispatch next event
029E 21E0         964          AJMP   DISPAT
02A0 759840       965          RxStop1: MOV   I2CON,#IDLE        ;Become idle.
02A3 21EC         966          AJMP   I2CRTI                    ;Resume interrupted activity
          967
          968          ;*****
          969          ; Handle START
          970          ;*****

02A5 309B07       971          RxStart: JNB   I2CON.STR,RxFault    ;Was it start?
02A8                                     972          SlvStart:
02A8 759818       973          MOV     I2CON,#CSTR+CARL          ;Yes, clear it.
02AB 7C00         974          MOV     R4,#0                     ;ByteCnt=0
02AD 4138         975          AJMP   SetRXB                    ;Set up to receive byte.
          976
          977          ; It wasn't DRDY, ARL, START, or STOP. Inconsistency error.

02AF 01B4         978          RxFault: AJMP  PwrUp
          979
          980
          981          $EJ
          982          ;*****
          983          ; MASTER TRANSMITTER
          984          ; R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
          985          ;*****
          986          ; Handle DRDY
          987          ;*****

02B1 EA          988          MASTER: MOV   A,R2                ;Get data in A.
02B2 309D48       989          JNB    I2CON.DRDY,MNDRDY          ;Is it DRDY?
          990          ; - context waiting to send bit

02B5 BE0418       991          CJNE   R6,#TXBIT,NTxBit          ;Context waiting to send bit?
02B8 F599         992          Tx1:  MOV    I2DAT,A              ;Send bit
02BA 23          993          Tx2:  RL     A                    ;Rotate the byte.
02BB DB07         994          DJNZ   R3,MN8Bit                 ;Was it 8th bit?
02BD 7E05         995          MOV    R6,#TXREAD                ;Set context waiting to read ACK
02BF 209D11       996          JB     I2CON.DRDY,TxAck1          ;If DRDY, short cut.
02C2 21E5         997          AJMP   WaitATN
          998
          999          ; prepare next bit 1-7

02C4 FA          1000         MN8Bit: MOV   R2,A                 ;Put the data back.
02C5 209DF0       1001          JB     I2CON.DRDY,Tx1             ;If DRDY, short cut.
02C8 209DED       1002          JB     I2CON.DRDY,Tx1
02CB 209DEA       1003          JB     I2CON.DRDY,Tx1            ;One more try.
02CE 21E5         1004          AJMP   WaitATN
          1005
          1006          ; - context waiting to read ACK

02D0 BE050D       1007          NTxBit: CJNE  R6,#TXREAD,NTxAck1  ;Context waiting to read ACK?

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
02D3  7598A0      1008    TxAck1: MOV     I2CON,#CDR+CXA      ;Switch to receive mode.
02D6  7E06        1009          MOV     R6,#TXACK            ;Set context waiting for ACK.
02D8  209D08      1010          JB      I2CON.DRDY,NTxAck2      ;If DRDY, short cut.
02DB  23            1011          RL      A                      ;Align data in case ARL.
02DC  1B           1012          DEC     R3
02DD  FA           1013          MOV     R2,A
02DE  21E5         1014          AJMP   WaitATN
                                1015
                                1016          ; - context waiting for ACK
02E0  BE0629       1017    NTxAck1: CJNE   R6,#TXACK,BeMast ;Context waiting for ACK?
02E3  E598         1018    NTxAck2: MOV     A,I2CON          ;Read from I2CON
02E5  5480         1019          ANL    A,#80h                  ;Only need 7th bit
02E7  6005         1020          JZ      AckOK
02E9  1021          1021    BadAck: ;Stop if negative ACK.
02E9  D5097D       1022          DJNZ   NACnt,DoStp1            ;Keep pending msg till retry expires.
02EC  6166         1023          AJMP   DoStp
                                1024          ;Ack Okay, prepare to send next byte.
02EE  0C           1025    AckOK:  INC     R4                ;Point to byte we want to send.
                                1026          ;ACALL Sample
02EF  81B1         1027          AJMP   DOTXB
                                1028          ;return should AJMP DNTXB
02F1  EA           1029    DNTXB:  MOV     A,R2                ;Next byte in A.
02F2  620D         1030          XRL    Check,A                 ;XOR with message check.
02F4  7B08         1031          MOV     R3,#8                   ;BitCnt=8
02F6  7E04         1032          MOV     R6,#TXBIT               ;Set context to send bit.
02F8  209DBD       1033          JB      I2CON.DRDY,Tx1          ;If DRDY, short cut.
02FB  21E5         1034          AJMP   WaitATN
                                1035
                                1036          $EJ
02FD  309C07       1045    MNRDY:  JNB     I2CON.ARL,MNARL        ;ARL?
0300  03           1046          RR      A                       ;Rotate data back.
0301  0B           1047          INC     R3                       ;Increment bit count.
0302  03           1048          RR      A
0303  0B           1049          INC     R3
0304  FA           1050          MOV     R2,A
0305  4154         1051          AJMP   ARL0
0307  209B02       1052    MNARL:  JB      I2CON.STR,BeMast1
030A  4185         1053          AJMP   Idles
                                1054
                                1055          ; - context START or not waiting for bit or ACK (from above).
                                1056          ; Must have just become master. Start new message.
                                1057          ; If message is pending in send type, do it.
                                1058          ; Otherwise, check to send Position Report if needed.
030C  A90B         1059    BEMAST: MOV     R1,SndType          ;Get message type.
030E  B90043       1060          CJNE   R1,#I_NoMsg,SndMsg       ;No message pending?
0311  302007       1061          JNB    MsgCheck,BeMast1         ;Send error?
0314  750BE4       1062          MOV     SndType,#I_Error
0317  C220         1063          CLR    MsgCheck
0319  6154         1064          AJMP   SndMsg
031B  201902       1065    BeMast1: JB     SendRpt,PosMsg     ;Send position report?
031E  6166         1066          AJMP   DoStp                    ; then stop.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
1067
1068      ; Send Position report
0320 201C0C      1069      PosMsg: JB      TxSelfRst,PosMsg1      ;If first user data,
0323 D21C        1070          SETB      TxSelfRst
0325 750BF0      1071          MOV       SndType,#I_Reset      ;Send a I_Reset to my address.
0328 E508        1072          MOV       A,MyAddr
032A 750901      1073          MOV       NACnt,#1      ;No retry if not acknowledged.
032D 8027        1074          SJMP      SndMsg1
032F 750B03      1075      PosMsg1:MOV    SndType,#LD_Position      ;Send position report.
0332 C219        1076          CLR       SendRpt      ;Clear pending report state.
0334 E4          1077          CLR       A      ;Copy X-Y counts to xmt buffer.
0335 F512        1078          MOV       XBUF2,A      ;Hi byte=0.
0337 F514        1079          MOV       YBUF2,A
0339 E516        1080          MOV       A,XCOUNT      ;Copy X.
033B F513        1081          MOV       XBUF1,A
033D 30E703      1082          JNB      ACC.7,Copy2      ;If negative,
0340 7512FF      1083          MOV       XBUF2,#0FFh      ; extend sign.
0343 E517        1084      Copy2: MOV     A,YCOUNT      ;Copy Y.
0345 F515        1085          MOV       YBUF1,A
0347 30E703      1086          JNB      ACC.7,Copy3      ;If negative,
034A 7514FF      1087          MOV       YBUF2,#0FFh      ; extend sign.
034D E4          1088      Copy3: CLR     A
034E F516        1089          MOV       XCOUNT,A      ;Reset X-Y counters.
0350 F517        1090          MOV       YCOUNT,A
0352 C21A        1091          CLR       Movement      ;Clear movement flag.
0354 7450        1092      SndMsg: MOV    A,#Adr_Host
0356 F599        1093      SndMsg1:MOV   I2DAT,A      ;Send first bit by hand
0358 75981C      1094          MOV       I2CON,#CARL+CSTR+CSTP      ;Clear start, release SCL
1095          ;Set context for rest of address
035B F50D        1096          MOV       Check,A      ;Init checksum
035D FA          1097          MOV       R2,A      ;I2Dat=A
035E 7B08        1098          MOV       R3,#8      ;BitCnt=8
0360 7C00        1099          MOV       R4,#0      ;ByteCnt=0
0362 7E04        1100          MOV       R6,#TXBIT      ;Set context waiting to send bit
0364 41BA        1101          AJMP      Tx2
1102      $EJ
1103      ; Completed sending message, do STOP
0366 750B00      1104      DoStp: MOV     SndType,#I_NoMsg      ;Indicate cmd no longer pending.
0369 717B        1105      DoStp1: ACALL  SndStop      ;Send STOP.
036B E50B        1106          MOV       A,SndType      ;Is there a pending message?
036D 7006        1107          JNZ      DoStp3
036F 201903      1108          JB       SendRpt,DoStp3      ;Is there a Position Report?
0372 302004      1109          JNB      MsgCheck,DoStp4      ;Is there an error message?
0375 11AF        1110      DoStp3: ACALL  SDelay      ;Yes, delay to give others
1111          ; a chance to become master without contention.
0377 D2DE        1112          SETB     I2CFG.MASTRQB      ;Request to be master again.
0379 4199        1113      DoStp4: AJMP   RxStop0      ;Borrow code from receiver.
1114
1115      ; Send I2C STOP signal
037B C2DE        1116      SndStop:CLR   I2CFG.MASTRQB      ;Release Master request
037D 759821      1117          MOV       I2CON,#CDR+XSTP      ;Set to send stop
0380 309EFD      1118          JNB      I2CON.ATN,$      ;Wait for ATN
0383 759820      1119          MOV       I2CON,#CDR      ;Clear useless DRDY (rising SCL)
0386 309EFD      1120          JNB      I2CON.ATN,$      ;Wait for stop sent
0389 759894      1121          MOV       I2CON,#CARL+CSTP+CXA      ;Clear I2C bus
038C 7E01        1122          MOV       R6,#RXIDLE      ;Set context idle receiver.
038E 22          1123          RET
1124
1125      $EJ

```


ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
1126          1126      ;*****
1127          1127      ; DO_RX_BYTE
1128          1128      ; Received a complete byte, already acknowledged.
1129          1129      ; Examine context to decide what to do with it.
1130          1130      ;
1131          1131      ; Enter: R2 (I2CDat) is byte received.
1132          1132      ;       R4 is offset to byte just received.
1133          1133      ; Exit:  Command parameters saved as needed.
1134          1134      ;       Checksum verified.  Valid commands executed.
1135          1135      ;       Return by AJMP DNRXB
1136          1136      ;
1137          1137      ;       R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
1138          1138      ;*****
038F BC0002    1139      DORXB:  CJNE    R4,#0,DoRx1    ;Is it Address? (ByteCnt=0)?
0392 4137      1140          AJMP    DNRXB
0394 BC0102    1141      DoRx1:  CJNE    R4,#1,DoRx2    ;Is it source Addr? (ByteCnt=1)
0397 4137      1142          AJMP    DNRXB
1143          1143      ;
0399 EA        1144      DoRx2:  MOV     A,R2           ;Get byte.
039A BC020C    1145          CJNE    R4,#2,DoRx3    ;Is it P+len?
039D 33        1146          RLC     A               ;Rotate Prot bit into C.
039E 9218      1147          MOV     Prot,C           ;Save it.
03A0 EA        1148          MOV     A,R2           ;Get "len".
03A1 547F      1149          ANL    A,#07Fh
03A3 2403      1150          ADD    A,#3             ;Add overhead.
03A5 F50C      1151          MOV     MsgLen,A         ;Save message length.
03A7 4137      1152          AJMP    DNRXB
1153          1153      ;
03A9 BC0304    1154      DoRx3:  CJNE    R4,#3,DoRx4    ;Is it Command byte?
03AC F50A      1155          MOV     RcvType,A        ;Save it
03AE 4137      1156          AJMP    DNRXB
1157          1157      $EJ
1158          1158      ;
1159          1159      ; Test for end of command
1160          1160      ;   If command has no data, byte offset 4 will be the checksum.
03B0 E50C      1161      DoRx4:  MOV     A,MsgLen      ;Get message length.
03B2 B5040A    1162          CJNE    A,ByteCnt,ToMny ;End of command?
1163          1163      ; sets carry if MsgLen<ByteCnt
03B5 D21F      1164          SETB   AA               ;Yes, do not Acknowledge more bytes.
03B7 E50D      1165          MOV     A,Check         ;Check in A
03B9 6002      1166          JZ     CheckOk           ;Bad check?
03BB 811F      1167          AJMP   RxErr
1168          1168      ; Message check is Ok, dispatch valid commands
03BD 8125      1169      CheckOk: AJMP   DORXCMD      ;Return AJMP DNRXB
1170          1170      ;
1171          1171      ; Test for ByteCnt beyond message length
03BF 5002      1172      ToMny:  JNC     DoDat        ;Too many bytes?
1173          1173      ; Yes, just exit, negative acknowledge already sent.
03C1 4137      1174      DoRx9:  AJMP   DNRXB
1175          1175      ;
1176          1176      ;
1177          1177      ; Receive message data bytes
1178          1178      ;   ByteCnt from 4 to (MsgLen-1)
1179          1179      ;
1180          1180      ; Branch on RcvType to decide what to do with each byte.
1181          1181      ; Notice Reset, Identify, and Poll have no data.
1182          1182      ;   Protocol: Assign New Address, Capabilities request
1183          1183      ;
03C3 EA        1184      DoDat:  MOV     A,R2           ;Get data again since DoRx4 wiped it.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
03C4  AF0A        1185      MOV     R7,RcvType      ;Put RcvType in R7 so we can CJNE
03C6  3018F8     1186      JNB     Prot,DoRx9      ;Ignore device data stream.
                                1187      ;Process C/S command bytes.
                                1188
                                1189      $EJ
                                1190      ;*****
                                1191      ; DORXB: Assign
                                1192      ; Compare incoming bytes with ID string (ByteCnt=4-29).
                                1193      ; If not equal, set not equal bit and ignore.
                                1194      ; Compare ByteCnt 30-31 with random number.
                                1195      ; If not equal, ignore (become idle receiver).
                                1196      ; Save ByteCnt 32, the address as parameter (R0).
                                1197      ;*****
                                1198
03C9  BFF225     1199      Do5:    CJNE    R7,#I_AsgnAdr,Do6 ;Assign command?
03CC  BC040B     1200      CJNE    R4,#4,Asn2      ;Start of ID string (ByteCnt=4)?
03CF  7900        1201      MOV     R1,#0           ;Initialize R1 is index
                                1202
                                1203      ; ByteCnt <= 29
03D1  E9          1204      Asn4:   MOV     A,R1         ;Get offset
03D2  B18C        1205      ACALL  GET_ID
03D4  09          1206      Asn5:   INC     R1             ;Increment for next
03D5  B50215     1207      CJNE    A,I2CDAT,AsnIg  ;Compare to received byte
03D8  4137        1208      AJMP   DNRXB           ;Ok so far
                                1209
                                1210      Asn2:   CJNE    R4,#30,Asn3  ;ByteCnt=30?
03DA  BC1E02     1211      MOV     R1,#RandH      ;Yes, set to read random#
03DD  790E        1212      Asn3:   JC     ASN4         ;Jump if less than 30.
                                1213
                                1214      ; ByteCnt >= 30
03E1  BC2004     1215      CJNE    R4,#32,Asn6    ;ByteCnt=32?
03E4  A802        1216      MOV     R0,I2CDat      ;Save new address in R0.
03E6  4137        1217      AJMP   DNRXB
03E8  5003        1218      Asn6:   JNC     AsnIg       ;Jump if ByteCnt>32.
03EA  E7          1219      MOV     A,@R1          ;Get byte of random #.
03EB  80E7        1220      SJMP   Asn5           ;Steal code from above.
                                1221
                                1222      AsnIg:  SETB   NotMyID      ;It's not for me
03ED  D21E        1223      AJMP   DNRXB
                                1224
                                1225      ; Application Capabilities Request
03F1  BFF321     1226      Do6:    CJNE    R7,#I_CapReq,Do7 ;CapRequest?
03F4  BC0403     1227      CJNE    R4,#4,Cpr1     ;ByteCnt=4?
03F7  EA          1228      MOV     A,R2           ;Check Cap hi pointer=0?
03F8  7014        1229      JNZ    Cpr4           ;No, reset length and offset to zero.
03FA  BC0516     1230      Cpr1:   CJNE    R4,#5,Cpr9  ;ByteCnt=5?
                                1231      ;Yes, check for valid offset.
03FD  E518        1232      MOV     A,CapOffset    ;Compute next offset.
03FF  2519        1233      ADD    A,CapLen
0401  B50202     1234      CJNE    A,I2CDat,Cpr2  ;Send next?
0404  8111        1235      AJMP   Cpr8           ; Yes, jump to set offset.
0406  EA          1236      Cpr2:   MOV     A,R2         ;A=received offset.
0407  6008        1237      JZ     Cpr8           ;Send first?
0409  B51802     1238      CJNE    A,CapOffset,Cpr4 ;Send previous?
040C  4137        1239      AJMP   DNRXB         ; Yes, use current offset.
040E  E4          1240      Cpr4:   CLR    A             ;Reset Length and offset to zero.
040F  F519        1241      MOV     CapLen,A
0411  F518        1242      Cpr8:   MOV     CapOffset,A   ;Load Cap offset.
0413  4137        1243      Cpr9:   AJMP   DNRXB

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
                                1244
                                1245      ; Set reporting Interval
0415  BF8205      1246      Do7:   CJNE   R7,#LD_SetInterval,Do7a
0418  BC0402      1247          CJNE   R4,#4,Do7a          ;ByteCnt=4?
041B  A802        1248          MOV    R0,I2CDat          ;Save parameter in R0.
041D  4137        1249      Do7a:  AJMP   DNRXB
                                1250
                                1251      ; Command error (I_Error)
041F  D220        1252      RxErr: SETB   MsgCheck          ;Set to report error.
0421  D2DE        1253          SETB   I2CFG.MASTRQB
0423  4137        1254          AJMP   DNRXB
                                1255      $EJ
                                1256      ;*****
                                1257      ; DO_RX_CMD
                                1258      ; Valid command received, do it.
                                1259      ; Dispatch commands based on RcvType
                                1260      ; Parameter value is in R0.
                                1261      ;
                                1262      ; Commands Recognized:  I_Reset, I_IdReq, I_AsgnAdr, I_CapReq,
                                1263      ;                      App_Test, App_Poll, App_SetInterval
                                1264      ;
                                1265      ; Return is:  AJMP DNRXB.  A is not preserved.
                                1266      ;*****
0425  AF0A        1267      DORXCMD:MOV  R7,RcvType          ;Put RcvType in R7 so we can CJNE.
0427  201802      1268          JB    Prot,DoC4          ;Go for Control/Status commands.
042A  4137        1269          AJMP   DNRXB          ;Ignore device data stream msgs.
                                1270
                                1271      ;Bus protocol commands
                                1272      ; Reset
042C  BFF002      1273      DoC4:  CJNE   R7,#I_Reset,DoC5
042F  01B4        1274          AJMP   PwrUp          ;Do power up reset.
                                1275      ; Identify
0431  BFF112      1276      DoC5:  CJNE   R7,#I_IdReq,DoC6
0434  750BE1      1277          MOV    SndType,#I_IdReply      ;Message type is identify
0437  201D08      1278          JB    KeepID,RTBM          ;Keep same device number?
043A  D21D        1279          SETB   KeepID
043C  858A0F      1280          MOV    RandL,TL          ;Random number <- T0
043F  858C0E      1281          MOV    RandH,TH
0442  D2DE        1282      RTBM:  SETB   I2CFG.MASTRQB      ;Request to be master
0444  4137        1283          AJMP   DNRXB
                                1284      ; Assign
0446  BFF20C      1285      DoC6:  CJNE   R7,#I_AsgnAdr,DoC7
0449  101E07      1286          JBC   NotMyID,ND01          ;Was it a complete match?
044C  BC21D0      1287          CJNE   R4,#33,RxErr        ;Check len=30+3
044F  C21C        1288          CLR    TxSelfRst          ;Anticipate first user data.
0451  8808        1289          MOV    MyAddr,R0          ;Load new address
0453  4137        1290      ND01:  AJMP   DNRXB
                                1291      ; Capabilities Request
0455  BFF30C      1292      DoC7:  CJNE   R7,#I_CapReq,DoC8
0458  BC0600      1293          CJNE   R4,#6,$+3          ;Check len>=3+3
045B  40C2        1294          JC    RxErr
045D  750BE3      1295          MOV    SndType,#I_CapReply      ;Message type is Cap Report
0460  D2DE        1296          SETB   I2CFG.MASTRQB      ;Request to be master
0462  4137        1297          AJMP   DNRXB
                                1298      ; App Test
0464  BFB107      1299      DoC8:  CJNE   R7,#App_Test,DoC9
0467  750BA1      1300          MOV    SndType,#App_TestReply  ;Send a test report
046A  D2DE        1301          SETB   I2CFG.MASTRQB      ;Request to be master.
046C  4137        1302          AJMP   DNRXB

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
                                1303      ; App Poll
046E  BFB00D      1304      DoC9:  CJNE    R7,#LD_Poll,DoC10
0471  E508        1305          MOV     A,MyAddr                ;Check for default address.
0473  B46E02      1306          CJNE    A,#Adr_Default,DoC9a
0476  8004        1307          SJMP   DoC9b                    ;Don't send Position reports
                                1308          ; to default address.
0478  D219        1309          DoC9a: SETB   SendRpt              ;Flag to send Position report
047A  D2DE        1310          SETB   I2CFG.MASTRQB            ;Request to be master.
047C  4137        1311          DoC9b: AJMP   DNRXB
                                1312
                                1313      ; Set Locator report interval
047E  BF822E      1314          DoC10: CJNE    R7,#LD_SetInterval,DoC11
0481  BC0500      1315          CJNE    R4,#5,$+3              ;Check len>=2+3
0484  4099        1316          JC     RxErr
0486  B80005      1317          CJNE    R0,#0,DoC10a          ;Check parameter range.
                                1318          ; parameter=0, polling only.
0489  758800      1319          MOV     TCON,#0                ;Turn off Timer0.
048C  4137        1320          AJMP   DNRXB
048E  B80800      1321          DoC10a: CJNE   R0,#8,$+3
0491  401A        1322          JC     DoC10c                  ;Jump if R0<8.
0493  B81A00      1323          CJNE    R0,#26,$+3
0496  5015        1324          JNC    DoC10c                  ;Jump if R0>=26.
                                1325          ; 8 <= param <= 25, compute Timer0 reload value.
0498  74FF        1326          MOV     A,#0FFh                ;Start at FFFFh
049A  F9          1327          MOV     R1,A
049B  C3          1328          DoC10b: CLR     C                ;Loop to subtract
049C  949A        1329          SUBB   A,#MSECL                ; R0 milliseconds.
049E  C9          1330          XCH    A,R1
049F  9402        1331          SUBB   A,#MSECH
04A1  C9          1332          XCH    A,R1
04A2  D8F7        1333          DJNZ   R0,DoC10b
04A4  F58B        1334          MOV     RTL,A                  ;Set Timer0 reload.
04A6  898D        1335          MOV     RTH,R1
04A8  758810      1336          MOV     TCON,#INIT_TCON        ;Turn on Timer0.
04AB  4137        1337          AJMP   DNRXB
04AD  811F        1338          DoC10c: AJMP  RxErr
                                1339
                                1340          ; Unrecognized command (ignore it)
04AF  4137        1341          DoC11: AJMP  DNRXB
                                1342
                                1343
                                1344          $EJ
                                1345          ;*****
                                1346          ; DOTXB
                                1347          ; Transmitted a complete byte, has been acknowledged.
                                1348          ; Get next byte based on context.
                                1349          ; Enter:
                                1350          ;   R4 (ByteCnt) is the offset of the byte we wish to send.
                                1351          ; Exit:
                                1352          ;   R2 (I2CDat) is the next byte to transmit (if any).
                                1353          ;   A is not preserved.
                                1354          ;   Return via AJMP DNTXB.
                                1355          ;
                                1356          ;   R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
                                1357          ;*****
                                1358          ; - Source address
04B1  BC0104      1359          DOTXB: CJNE   R4,#1,DoTx1      ;ByteCnt=1?
04B4  AA08        1360          MOV     R2,MyAddr              ;Send source addr
04B6  41F1        1361          AJMP   DNTXB

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
                                1362
                                1363 ; - Message length
04B8  BC0258      1364 DoTx1: CJNE   R4,#2,DoTx2      ;ByteCnt=2?
04BB  7A82        1365         MOV    R2,#082h           ;Use 2 as default length,
                                1366         ; P=1, Control/Status msg.
04BD  750C05      1367         MOV    MsgLen,#5           ;Include overhead
                                1368         ; Compute length based on message type.
                                1369         ; If not Position Report, Identify, Output Error,
                                1370         ; or Reset, the length is 2.
04C0  AF0B        1371         MOV    R7,SndType
                                1372 ; Position report
04C2  BF0307      1373         CJNE   R7,#LD_Position,TxA ;Position report?
04C5  7A06        1374         MOV    R2,#6               ;Data length is 6 (P=0, data stream).
04C7  750C09      1375         MOV    MsgLen,#9           ;6 plus 3 overhead.
04CA  41F1        1376         AJMP  DNTXB
                                1377 ; Attention
04CC  BFE00C      1378 TxA:   CJNE   R7,#I_Attn,TxI
04CF  E51A        1379         MOV    A,SelfTest         ;Check for ROM error
04D1  B40105      1380         CJNE   A,#ROM_ERROR,TxA9
04D4  7A83        1381 Len3:  MOV    R2,#083h       ;Use Len=3 to include checksum.
04D6  750C06      1382         MOV    MsgLen,#6
04D9  41F1        1383 TxA9:  AJMP  DNTXB
                                1384 ; Identify Reply
04DB  BFE107      1385 TxI:   CJNE   R7,#I_IdReply,TxC
04DE  7A9D        1386         MOV    R2,#(80h+29)       ;Length for Identify.
04E0  750C20      1387         MOV    MsgLen,#(29+3)     ;Add overhead for MsgLen
04E3  41F1        1388         AJMP  DNTXB
                                1389 ; Capabilities Report
04E5  BFE31C      1390 TxC:   CJNE   R7,#I_CapReply,TxE
04E8  7410        1391         MOV    A,#CapFragLen      ;Get default fragment length.
04EA  F519        1392         MOV    CapLen,A           ;Save it.
04EC  2518        1393         ADD   A,CapOffset         ;Find end of fragment.
04EE  C3          1394         CLR   C
04EF  9475        1395         SUBB  A,#(CAP_END-CAP_START) ;Is it beyond end of Cap String?
04F1  4006        1396         JC    TxC3                ;No, use default length.
04F3  F4          1397 TxC1:  CPL    A              ;Yes, shorten as needed.
04F4  04          1398         INC   A
04F5  2519        1399         ADD   A,CapLen
04F7  F519        1400         MOV    CapLen,A
04F9  E519        1401 TxC3:  MOV    A,CapLen         ;Get fragment length.
04FB  2483        1402         ADD   A,#083h             ;Compute data length.
04FD  FA          1403         MOV    R2,A               ;Prepare to send it.
04FE  2483        1404         ADD   A,#083h             ;Add 3 overhead for MsgLen (clear C/S).
0500  F50C        1405         MOV    MsgLen,A
0502  41F1        1406         AJMP  DNTXB
                                1407 ; Checksum or message framing error
0504  BFE402      1408 TxE:   CJNE   R7,#I_Error,TxR
0507  8003        1409         SJMP  TxR1
                                1410 ; Reset
0509  BFF005      1411 TxR:   CJNE   R7,#I_Reset,TxU
050C  7A81        1412 TxR1:  MOV    R2,#081h         ;Length for Reset (80+1).
050E  750C04      1413         MOV    MsgLen,#4          ;1 plus 3 overhead.
0511  41F1        1414 TxU:   AJMP  DNTXB
                                1415
                                1416 ; - Command code
0513  BC0309      1417 DoTx2: CJNE   R4,#3,DoTxLast   ;ByteCnt=3?
0516  AA0B        1418         MOV    R2,SndType         ;Send command code
0518  BA0302      1419         CJNE   R2,#LD_Position,TCCL1 ; unless it is position report.
051B  AA10        1420         MOV    R2,ReportBuf       ;In that case send 1st byte of report.

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
051D  41F1        1421    TCC1:  AJMP    DNTXB
          1422    ;
          1423    ; - Test for last byte of command (message check)
051F  E50C        1424    DoTxLast: MOV   A,MsgLen
0521  B50404      1425    CJNE   A,ByteCnt,DoTxEnd ;Last byte of command?
          1426    ; sets Carry if A<ByteCnt
0524  AA0D        1427    MOV    R2,Check          ;Yes, send check.
0526  41F1        1428    AJMP   DNTXB
          1429    ;
          1430    ; - Test for beyond last byte of command
0528  5005        1431    DoTxEnd: JNC   DoTx3      ;Beyond last byte (check)?
052A  750905      1432    MOV    NACnt,#5          ;Reset Negative Ack retry count.
052D  6166        1433    AJMP   DoStp            ;Send STOP.
          1434
          1435
          1436    ; Transmit message data bytes
          1437    ; ByteCnt from 3 to (length+2)
          1438    ; Dispatch based on command type
          1439
052F  AF0B        1440    DoTx3:  MOV    R7,SndType
0531  BF0309      1441    CJNE   R7,#LD_Position,DoT3 ;Position report?
0534  E504        1442    MOV    A,ByteCnt        ;Yes, send next byte.
0536  240D        1443    ADD    A,#ReportBuf-3
0538  F9         1444    MOV    R1,A
0539  8702        1445    MOV    I2CDat,@R1       ;R2=I2CDat=@R1
053B  41F1        1446    AJMP   DNTXB
          1447
          1448    ;Attention report
053D  BFE009      1449    DoT3:  CJNE   R7,#I_Attn,DoT4
0540  AA1A        1450    MOV    R2,SelfTest
0542  BC0502      1451    CJNE   R4,#5,AT4        ;Send Power-up selftest and attention
0545  AA1B        1452    MOV    R2,RomSum        ;ByteCnt=5?
0547  41F1        1453    AT4:  AJMP   DNTXB        ;Yes, send checksum byte.
          1454
          1455    ;Application test report
0549  BFA104      1456    DoT4:  CJNE   R7,#App_TestReply,DoT5
054C  AA1A        1457    MOV    R2,SelfTest
054E  41F1        1458    AJMP   DNTXB           ;Send Selftest result
          1459
          1460    ;*****
          1461    ; Identify report
          1462    ; Send ID string for ByteCnt 4-29 (26 bytes, last two are FFh).
          1463    ; Send random number for ByteCnt 30 and 31.
          1464    ;*****
0550  BFE11D      1465    DoT5:  CJNE   R7,#I_IdReply,DoT6
0553  BC0409      1466    CJNE   R4,#4,IDR2        ;First byte (ByteCnt=4)?
          1467    ;yes, set up to send ID string
0556  7900        1468    MOV    R1,#0            ;R1 is index
          1469
0558  E9         1470    IDR4:  MOV    A,R1          ;Get offset
0559  B18C        1471    ACALL GET_ID
055B  FA         1472    IDR5:  MOV    R2,A          ;Prepare to send it
055C  09         1473    INC    R1                ;Increment for next
055D  41F1        1474    AJMP   DNTXB
          1475
          1476    ;
055F  BC1E02      1476    IDR2:  CJNE   R4,#30,IDR3  ;ByteCnt=30?
0562  790E        1477    MOV    R1,#RandH        ;Set to send random #.
0564  40F2        1478    IDR3:  JC     IDR4          ;Jump if less than 30.
          1479    ; BytCnt >= 30

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC OBJ          LINE          SOURCE
0566 BC2000      1480          CJNE    R4,#32,$+3
0569 5003        1481          JNC     IDR7          ;Jump if >= 32.
056B E7          1482          MOV     A,@R1        ;Get byte of random #.
056C 80ED        1483          SJMP   IDR5
056E 6166        1484          IDR7:  AJMP   DoStp          ;Error! Beyond last byte,
                                1485          ; STOP now.
                                1486          ;
                                1487          ;*****
                                1488          ; Capability report
                                1489          ; Send next byte of capability string.
                                1490          ; Uses R1 as index.
                                1491          ;*****
0570 BFE317      1492          DoT6:  CJNE   R7,#I_CapReply,DoT7
0573 BC0404      1493          CJNE   R4,#4,Cap1      ;First byte (ByteCnt=4)?
0576 7A00        1494          MOV     R2,#0          ;Send High byte of Offset (always 0).
0578 41F1        1495          AJMP   DNTXB
                                1496
057A BC0506      1497          Cap1:  CJNE   R4,#5,CAP2      ;Second byte (Offsetlo)?
057D AA18        1498          MOV     R2,CapOffset   ;Send Low byte of Offset.
057F A918        1499          MOV     R1,CapOffset   ;Initialize R1 to use as index.
0581 41F1        1500          AJMP   DNTXB
                                1501
0583 E9          1502          Cap2:  MOV     A,R1          ;Get Capabilities Character.
0584 B1A9        1503          ACALL  GET_CAP
0586 FA          1504          MOV     R2,A            ;Prepare to send it.
0587 09          1505          INC     R1              ;Increment for next Character.
0588 41F1        1506          Cap3:  AJMP   DNTXB
                                1507          ;
                                1508          ; Unknown: How can we not know what we're sending?
058A 41F1        1509          DoT7:  AJMP   DNTXB
                                1510
                                1511
                                1512          $EJ
                                1513          ;*****
                                1514          ; GET_ID
                                1515          ; Get byte of ID string
                                1516          ; Enter: offset of desired byte in A.
                                1517          ; Exit: A is the desired byte.
                                1518          ;*****
058C 04          1519          GET_ID: INC     A            ;Skip RET
058D 83          1520          MOVCA  A,@A+PC        ;Get the byte
058E 22          1521          RET
                                1522          ;ID string is defined here, length is 25
058F 41          1523          DB     'A'            ;Protocol revision
0590 56312E31    1524          DB     'V1.1 '         ;Module revision
0594 202020      1525          DB     'DEC '          ;Vendor name
059B 20202020    1526          DB     'VSXXX-BB'     ;Module name
059F 56535858    1527          DB     0FFh           ;1st byte of device #
05A3 582D4242    1528          DB     0FFh           ;2nd byte of device #
05A7 FF          1529
05A8 FF          1530          ;*****
                                1531          ; GET_CAP
                                1532          ; Get byte of Capabilities string.
                                1533          ; This implementation supports up to 254 bytes only!
                                1534          ; Enter: offset of desired byte in A.
                                1535          ; Exit: A is the desired byte.
                                1536          ;*****

```

ACCESS.bus mouse application code for the 8XC751 microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
05A9  04          1537      GET_CAP: INC    A           ;Skip RET
05AA  83          1538      MOVCL         A,@A+PC      ;Get the byte
05AB  22          1539      RET
                                1540      ;Capabilities string is defined here.
05AC  28          1541      CAP_START: DB  '('
05AD  2070726F    1542      DB            ' prot(locator)'
05B1  74286C6F
05B5  6361746F
05B9  7229
05BB  20747970    1543      DB            ' type(mouse)'
05BF  65286D6F
05C3  75736529
05C7  20627574    1544      DB            ' buttons(1(L)2(R)3(M))'
05CB  746F6E73
05CF  2831284C
05D3  29322852
05D7  2933284D
05DB  2929
05DD  2064696D    1545      DB            ' dim(2)'
05E1  283229
05E4  2072656C    1546      DB            ' rel'
05E8  20726573    1547      DB            ' res(200 inch)'
05EC  28323030
05F0  20696E63
05F4  6829
05F6  2072616E    1548      DB            ' range(-127 127)'
05FA  6765282D
05FE  31323720
0602  31323729
0606  20643028    1549      DB            ' d0(dname(X))'
060A  646E616D
060E  65285829
0612  29
0613  20643128    1550      DB            ' d1(dname(Y))'
0617  646E616D
061B  65285929
061F  29
0620  29          1551      DB            ')'
0621  00          1552      CAP_END: DB    0           ;Null terminator (not used).
                                1553      ; Capabilities length is 121 bytes
                                1554
                                1555      END

```


A software duplex UART for the 751/752

AN446

Author: Greg Goodhue

The following program contains routines that will allow an 8xC751 or 8xC752 to implement a software UART that can send and receive serial data simultaneously. Other published software UARTs only allow either transmit or receive to occur at any one time. The demo application shown in the code listing waits for data to be received, then echoes it and follows this with a hexadecimal interpretation of the data plus a space. For instance, if the program receives the character "\$", it echoes back the string "\$24". The reason for echoing these additional characters is to make it easy to force the receiver buffer to fill up in order to test the handshaking. If the program simply echoed what was received, it would likely never use more than the very first receiver buffer location since it can normally transmit just as fast as it can receive.

CHIP RESOURCES

The UART routines use about 400 bytes of code space and use the timer to provide a constant time interrupt to synchronize both transmit and receive operations. The hardware connections require four device pins to accomplish serial I/O with RTS/CTS handshaking. Only two pins would be needed if handshaking is not required. Three of the four pin functions may be assigned to any port pin. The serial input pin must be assigned as one of the external interrupt pins. Another two pins are used in the demo application to input a selection of one of four baud rates (1200, 2400, 4800, or 9600).

LIMITATIONS

To obtain duplex operation, a fairly large portion of the chip's time is used. The routines were tested up to 9600 baud running on a 16 MHz 87C751. When serial input and output were both occurring at the same time, the routines could not support continuous operation with no pauses between characters. At 4800 baud, full speed tight reception and transmission worked flawlessly. In other words, 4800 baud should work with all applications, while 9600 baud may not work with all applications.

THEORY OF OPERATION

There are three possible sequences of events when serial transmit and receive may both be operating at once: transmit and receive begin simultaneously; transmit is requested while the receiver is busy; and receive starts while the transmitter is busy. The first 2 cases could be handled fairly simply with only one interrupt for each bit time. In the first case, everything is already in

synch and only one timer and one interrupt per bit is needed to do both operations. In the second case (transmit is requested while the receiver is busy) the program could just wait for the next bit time to start transmitting. Unfortunately, the third case presents a problem. If the program is already transmitting, it cannot always wait for the next bit time to start sampling the serial data if the application is not to lose bits. Also, the timer cannot be adjusted to the incoming data since this would distort the duration of one of the transmitted bits.

The method used here to deal with this problem is to always divide all bit times into 4 sub-bit times. When transmission and/or reception is in progress, the timer runs at 4X the bit rate for the selected baud rate. The variables TxTime and RxTime are used to count sub-bit times for the transmitter and the receiver, respectively. Both are initialized to a negative value and count up to simplify testing for an active sub-bit time. The maximum baud rate that can be supported is essentially determined by the maximum amount of time that it might take the microcontroller to do all of the operations associated with transmitting one bit and receiving one bit. This must be done within the time between timer interrupts.

When both transmit and receive operations are scheduled for the same timer interrupt, priority is given to the transmitter routine. The reason for this is that a great deal of jitter can be tolerated in the timing of the received bit sampling, but the transmitted data must "look" good to the outside world.

The actual bit times for transmit and receive are counted by the variables TxCnt and RxCnt, respectively. When an active sub-bit time slice occurs, these variables tell the transmit and receive routines what to do in the current time slice. The value 11 hex indicates a start bit, 10 hex indicates a stop bit, and the values 8 through F hex indicate a data bit. The values were chosen to allow quick determination of the appropriate action by the code.

The routines provide for a small amount of data buffering for both the transmitter and the receiver. As implemented here, the transmitter buffer is only one byte deep, allowing one data byte to be held while another is being transmitted. The receiver buffer is larger, allowing three bytes to be held while a fourth is being received. If the receiver buffer fills up (indicated by the flag RxFull), the application code must retrieve one byte before a fourth one finishes, or data will be lost. If this happens, a flag will be set (OverrunErr) to indicate that the receiver buffer has been overrun. There is no similar

flag for the transmitter, since the transmit request routine waits for the transmitter buffer to be available (indicated by the TxFull flag) before taking action. It is up to the application code to check this flag in advance if it does not want to stall execution while waiting to transmit data.

As each routine finishes a whole data byte by completing the send or receive of a stop bit, it checks to see if there is something still happening to warrant having the time slice interrupt running. In the case of a received stop, the transmit activity flag (TxOn) is examined. If it is not set, the timer is turned off. The timer will be turned back on if an interrupt from a serial start bit is received or the main code requests data to be transmitted. In the case of a transmitted stop, both the receiver activity flag (RxOn) and the transmit buffer flag (TxFull) are examined. If the receiver is active or there is more data to transmit, the timer is left running.

All of the status flags are in the "Flags" register. Other status flags found there are: RxAvail, which indicates that the receiver buffer contains unprocessed data; and FramingErr which is set when the receiver routines find an improper start or stop bit, usually caused by mismatched baud rates.

Flow control handshaking is provided by the RTS/CTS scheme. The transmit routine looks at the incoming CTS line before beginning each start bit transmission, and simply exits, waiting for the next time slice, if CTS is not asserted. The receive routine checks the buffer status whenever a start bit interrupt occurs and de-asserts the outgoing RTS line if the buffer already contains two bytes (i.e., it will be full when the current byte finishes). If the device at the other end of the communication line follows the same rules (which may very well NOT be the case) the program should be able to communicate without buffer overflows in either direction.

Baud rates in both the send and receive routines are determined by two things: the timer interrupt rate; and the number of time slices per bit. The method of calculating the timer value for various baud rates is discussed in the code listing at the BaudRate routine. This discussion has centered on there being four time slices per bit, but if the user wants, either the transmitter or the receiver can be set to run at a baud rate that is a multiple of the other by adjusting the value of the constant TxBitLen or RxBitLen. The baud rate would be calculated as indicated for the faster channel, and TxBitLen or RxBitLen would be changed for the slower channel. For example, the transmitter can be set to run at half of the receiver baud rate by setting TxBitLen to -8 + 1.

A software duplex UART for the 751/752

AN446

The routines shown also make provision for changing the baud rate "on the fly", although the application code given does not implement this feature. If the application code changes the baud rate for some reason, the change will be effected when the next data transmission or reception begins, if both the transmitter and receiver were already idle. This prevents the timer value from being changed in the middle of a data byte.

THE CODE

There are a number routines in the code of which the user should be aware:

- Intrl—Called (by interrupt) when a serial start bit is received.
 - Timer0—Called (by interrupt) for every sub-bit time slice.
 - RS232TX—Called by Timer0 when the transmitter has business to conduct in the current time slice.
 - RS232RX—Called by Timer0 when the receiver has business to conduct in the current time slice.
 - BaudRate—Sets the baud rate variables
- BaudHigh and BaudLow based on the accumulator value.
- TxSend—Called by the application code to request that a data byte be transmitted. The data to be transmitted is in the accumulator.
 - GetRx—Called by the application code to request return of a received data byte from the buffer. Data is returned in the accumulator. This routine should not be called unless the receiver buffer has data available.
 - Reset—Start of the initialization code to set up the UART.
 - MainLoop—Start of the mainline code of the demo application.

A software duplex UART for the 751/752

AN446

```

;*****
;           Duplex UART Routines for the 8xC751 and 8xC752 Microcontrollers
;*****
; This is a demo program showing a way to perform simultaneous RS-232
; transmit and receive using only one hardware timer.
;
; The transmit and receive routines divide each bit time into 4 slices to
; allow synchronizing to incoming data that may be out of synch with outgoing
; data.
;
; The main program loop in this demo processes received data and sends it
; back to the transmitter in hexadecimal format. This insures that we can
; always fill up the receiver buffer (since the returned data is longer than
; the received data) for testing purposes. Example: if the letter "A" is
; received, we will echo "A41 ".
;*****
$Title(Duplex UART Routines for the 751/752)
$Date(8/20/92)
$MOD751
;*****
;                               Definitions
;*****
; Miscellaneous
TxBitLen EQU -4 + 1           ; Timer slices per serial bit transmit.
RxBitLen EQU -4 + 1           ; Timer slices per serial bit receive.
RxHalfBit EQU (RxBitLen / 4) + 1 ; Timer slices for a partial bit time.
; Used to adjust the input sampling
; time point.
; Note: TxBitLen and RxBitLen are kept separate in order to facilitate the
; possibility of having different transmit and receive baud rates. The timer
; would be set up to give four slices for the fastest baud rate, and the
; BitLen for the slower channel would be set longer for the slower baud rate.
; BitLen = -4 + 1 gives four timer interrupts per bit. BitLen = -8 + 1 would
; give 8 slices, BitLen = -16 + 1 would give 16 slices, etc.
TxPin     BIT    P1.0           ; RS-232 transmit pin (output).
RxPin     BIT    P1.5           ; RS-232 receive pin (input).
RTS       BIT    P1.3           ; RS-232 request to send pin (output).
CTS       BIT    P1.6           ; RS-232 clear to send pin (input).
; Note: P1.1 and P1.2 are used to input the baud rate selection.
; RAM Locations
Flags     DATA  20h           ; Miscellaneous bit flags (see below).
TxOn      BIT    Flags.0       ; Indicates transmitter is on (busy).
RxOn      BIT    Flags.1       ; Indicates receiver is on (busy).
TxFull    BIT    Flags.2       ; Transmit buffer (1 byte only) is full.
RxFull    BIT    Flags.3       ; Receiver buffer is full.
RxAvail   BIT    Flags.4       ; RX buffer is not empty.
OverrunErr BIT  Flags.6       ; Overrun error flag.
FramingErr BIT  Flags.7       ; Framing error flag.
BaudHigh  DATA  21h           ; High byte timer value for baud rate.
BaudLow   DATA  22h           ; Low byte timer value for baud rate.
TxCnt     DATA  23h           ; RS-232 byte transmit bit counter.
TxTime    DATA  24h           ; RS-232 transmit time slice count.
TxShift   DATA  25h           ; Transmitter shift register.
TxDat     DATA  26h           ; Transmitter holding register.

```

A software duplex UART for the 751/752

AN446

```

RxCnt      DATA 27h          ; RS-232 byte receive bit counter.
RxTime     DATA 28h          ; RS-232 receive time slice count.
RxShift    DATA 29h          ; Receiver shift register.
RxDatCnt   DATA 2Ah          ; Received byte count.
RxBuf      DATA 2Bh          ; Receive buffer (3 bytes long).

Temp       DATA 2Fh          ; Temporary holding register.

;*****
;
;                               Interrupt Vectors
;*****

ORG 00h          ; Reset vector.
AJMP RESET

ORG 03h          ; External interrupt 0
AJMP Intr0       ; (received RS-232 start bit).

ORG 0Bh          ; Timer 0 overflow interrupt.
AJMP Timer0      ; (4X the RS-232 bit rate).

ORG 13h          ; External interrupt 1.
RETI             ; (not used).

ORG 1Bh          ; Timer I interrupt.
RETI             ; (not used).

ORG 23h          ; I2C interrupt.
RETI             ; (not used).

;*****
;
;                               Interrupt Handlers
;*****

; External Interrupt Int0.
; RS-232 start bit transition.

Intr0:        PUSH ACC          ; Save accumulator,
              PUSH PSW          ; and status.
              CLR IE.0          ; Disable more RX interrupts.

              SETB RxOn         ; Set receive active flag.
              MOV RxCnt,#11h     ; Set bit counter to expect a start.
              MOV RxTime,#RxHalfBit ; First sample is at a partial bit time.
              JB TxOn,IOTimerOn  ; If TX active then timer is on.

              MOV RTH,BaudHigh   ; Set up timer for selected baud rate.
              MOV RTL,BaudLow
              MOV TH,BaudHigh
              MOV TL,BaudLow
              SETB TR             ; Start timer 0.

IOTimerOn:    MOV A,RxDatCnt     ; Check for buffer about to be full:
              CJNE A,#2,Int0Ex   ; one space left and a byte starting.
              SETB RTS           ; If so, tell whoever is on the
              ; other end to wait.

Int0Ex:       POP PSW           ; Restore status,
              POP ACC           ; and accumulator.
              RETI

; Timer 0 Interrupt
; This is used to generate time slices for both serial transmit and receive
; functions.

```

A software duplex UART for the 751/752

AN446

```

Timer0:    PUSH  ACC           ; Save accumulator,
           PUSH  PSW           ; and status.
           JNB   TxTime.7,RS232TX ; Is this an active time slice
                                           ; for an RS-232 transmit?
           JNB   TxOn,CheckRx  ; If transmit is active,
           INC   TxTime        ; increment the time slice count.
CheckRx:   JNB   RxTime.7,RS232RX ; Is this an active time slice
                                           ; for an RS-232 receive?
           JNB   RxOn,T0Ex     ; If receive is active, increment
           INC   RxTime        ; the time slice count.

T0Ex:     POP   PSW           ; Restore status,
           POP   ACC           ; and accumulator.
           MOV   P3,Flags     ; For demo purposes, output status
                                           ; on an extra port.

           RETI

;*****
;                               RS-232 Transmit Routine
;*****

RS232TX:   JNB   TxCnt.4,TxData ; Go if data bit.
           JNB   TxCnt.0,TxStop ; Go if stop bit.

; Send start bit and do buffer housekeeping.

TxStart:   JB    CTS,TxEx1     ; Is CTS asserted (low) so can we send?
                                           ; If not, try again after 1 bit time.
           CLR   TxPin        ; Set start bit.
           MOV   TxShift,TxDat ; Get byte to transmit from buffer.
           CLR   TxFull       ;
           MOV   TxCnt,#08h    ; Init bit count for 8 bits of data.
                                           ; (note: counts UP).

TxEx1:     MOV   TxTime,#TxBitLen ; Reset time slice count.
           SJMP  CheckRx      ; Restore state and exit.

; Send Next Data Bit.

TxData:    MOV   A,TxShift     ; Get un-transmitted bits.
           RRC   A            ; Shift next TX bit to carry.
           MOV   TxPin,C      ; Move carry out to the TXD pin.
           MOV   TxShift,A    ; Save bits still to be TX'd.
           INC   TxCnt        ; Increment TX bit counter
           MOV   TxTime,#TxBitLen ; Reset time slice count.
           SJMP  CheckRx      ; Restore state and exit.

; Send Stop Bit and Check for More to Send.

TxStop:    SETB  TxPin        ; Send stop bit.
           JB    TxFull,TxEx2 ; More data to transmit?
           CLR   TxOn        ; If not, turn off TX active flag, and
           CLR   RTS         ; make sure that whoever is on the
                                           ; other end knows it's OK to send.

           JB    RxOn,TxEx2   ; If receive active, timer stays on,
           CLR   TR          ; otherwise turn off timer.

TxEx2:     MOV   TxCnt,#11h    ; Set TX bit counter for a start.
           MOV   TxTime,#TxBitLen-1 ; Reset time slice count, stop bit
                                           ; > 1 bit time for synch.
           SJMP  CheckRx      ; Restore state and exit.

```

A software duplex UART for the 751/752

AN446

```

;*****
;
;                               RS-232 Receive Routine
;*****

RS232RX:  MOV    C,RxPin           ; Get current serial bit value.
          JNB   RxCnt.4,RxData    ; Go if data bit.
          JNB   RxCnt.0,RxStop    ; Go if stop bit.

;Verify start bit.

RxStart:  JC    RxErr            ; If bit=1, then not a valid start.
          MOV   RxCnt,#08h        ; Init counter to expect data.
          MOV   RxTime,#RxBitLen  ; Reset time slice count.
          SJMP TOEx              ; Restore state and exit.

; Get Next Data Bit.

RxData:   MOV   A,RxShift        ; Get partial received byte.
          RRC   A                 ; Shift in new received bit.
          MOV   RxShift,A         ; Store partial result in buffer.
          INC   RxCnt             ; Increment received bit count.
          MOV   RxTime,#RxBitLen  ; Reset time slice count.
          SJMP TOEx              ; Restore state and exit.

; Store Data Byte, "push"ing it into the FIFO buffer.

RxStop:   CLR   EA                ; Don't interrupt the following.
          MOV   A,RxBuf           ; "PUSH" the receive buffer.
          XCH  A,RxBuf+1
          XCH  A,RxBuf+2
          MOV   RxBuf,RxShift     ; Add just completed data to buffer.
          INC   RxDatCnt          ; Increment the received byte count.
          SETB EA                ; Re-enable interrupts.

          SETB RxAvail           ; There is data in the RX buffer.
          PUSH PSW                ; Save Carry (received bit)for later.
          MOV   A,RxDatCnt        ; Check receiver buffer status.
          CJNE A,#4,RxChk1       ; Is RX buffer overrun?
          SETB OverrunErr        ; Set status reg overrun error flag.
          MOV   RxDatCnt,#3       ; Re-set buffer counter to "full".

RxChk1:   CJNE A,#3,RxChk2       ; Is RX buffer full?
          SETB RxFull            ; Set buffer full status.

RxChk2:   POP   PSW               ; Retrieve last received bit in Carry.
          JC    RxEx              ; If bit=0, then not a valid stop.
RxErr:    SETB FramingErr        ; Remember had start or stop status.

RxEx:     JB    TxOn,RxTimerOn   ; If transmit active, timer stays on,
          CLR   TR                ; otherwise turn timer off.
RxTimerOn: CLR RxOn              ; Turn off receive active.
          SETB RxTime.7           ; Set bit for no service to
          ; RX Time Slice Branches.
          SETB IE.0              ; Re-enable RS-232 receive interrupts.
          AJMP TOEx              ; Restore state and exit.

;*****
;
;                               Subroutines
;*****

; BaudRate - Determine and set the baud rate from switches.
; Note: if the baud rate is altered, the actual change will only occur when
; a transmit or receive is begun while the timer was not already running
; (i.e.: not already busy transmitting or receiving).

```

A software duplex UART for the 751/752

AN446

```

BaudRate:  MOV  DPTR,#BaudTable      ; Set pointer to baud rate table.
           ANL  A,#03h              ; Limit displacement for lookup.
           RL   A                   ; Double the table index since these
                                   ;   are 2 byte entries.
           PUSH ACC                 ; Save the table index for second byte.
           MOVC A,@A+DPTR          ; Get first byte, and save as the high
           MOV  BaudHigh,A         ;   byte of the baud rate timer value.
           POP  ACC                 ; Get back the table index.
           INC  A                   ; Advance to next table entry.
           MOVC A,@A+DPTR          ; Get second byte, and save as the low
           MOV  BaudLow,A          ;   byte of the baud rate timer value.
           RET

; Entries in BaudTable are for a timer setting of 1/4 of a bit time at the given
; baud rate. The two values per entry are the high and low bytes of the value
; respectively.

; Values are calculated as follows:
;
;                               Osc Frequency
; 1/4 Bit cell time (in machine cycles) = -----
;                                       Baud Rate * 48

; Example for 9600 baud with a 16MHz crystal:
; 16,000,000 / 9600 * 48 = 34.7222... machine cycles per quarter bit time.
; Rounded, this is 35. The hexadecimal value for 35 is 23.
; 10000 hex - 23 hex (truncated to 16 bits) = FFDD. Thus, the BaudTable entry
; for 9600 baud is FF, DD hex.

BaudTable: DB  0FEh,0EAh          ; 1200 baud.
           DB  0FFh,75h          ; 2400 baud.
           DB  0FFh,0BBh        ; 4800 baud.
           DB  0FFh,0DDh        ; 9600 baud.

; TxSend - Initiate RS-232 Transmit.

TxSend:    JB   TxFull,$          ; Make sure TX buffer is free.
           SETB TxFull           ; Reserve the buffer for our use.
           MOV  TxDat,A          ; Put character in buffer.
           JB   TxOn,TSTimerOn   ; Exit if transmitter already running.

           SETB TxOn            ; Transmit active flag set.
           MOV  TxCnt,#11h       ; Init bit counter to expect a start.
           MOV  TxTime,#TxBitLen ; Reset time slice count.
           JB   RxOn,TSTimerOn   ; Exit if receiver already active.

           MOV  RTH,BaudHigh     ; Set up timer for selected baud rate.
           MOV  RTL,BaudLow
           MOV  TH,BaudHigh
           MOV  TL,BaudLow
           SETB TR                ; Start up the bit timer.

TSTimerOn: RET

; PrByte - Output a byte as ASCII hexadecimal format.

PrByte:    PUSH ACC              ; Print ACC contents as ASCII hex.
           SWAP A
           ACALL HexAsc         ; Print upper nibble.
           ACALL TxSend
           POP  ACC
           ACALL HexAsc         ; Print lower nibble.
           ACALL TxSend
           RET

; HexAsc - Convert a hexadecimal nibble to its ASCII character equivalent.

HexAsc:    ANL  A,#0Fh          ; Make sure we're working with only
                                   ;   one nibble.
           CJNE A,#0Ah,HA1      ; Test value range.
HA1:       JC   HAVa109         ; Value is 0 to 9.

```

A software duplex UART for the 751/752

AN446

```

      ADD  A,#7          ; Value is A to F, needs pre-adjustment.
HAVal09: ADD  A,#'0'     ; Adjust value to ASCII hex.
      RET

; GetRx - Retrieve a byte from the receive buffer, and return it in A.

GetRx: CLR  EA          ; Make sure this isn't interrupted.
      DEC  RxDatCnt     ; Decrement the buffer count.
      MOV  A,RxDatCnt   ; Get buffer count.
      JNZ  GRX1         ; Test for empty receive buffer.
      CLR  RxAvail      ; If empty, clear data available status.
GRX1:  ADD  A,#RxBuf     ; Create a pointer to end of buffer.
      MOV  Temp,R0      ; Save R0.
      MOV  R0,A         ; Put pointer where we can indirect.
      MOV  A,@R0       ; Get last buffer data.
      MOV  R0,Temp      ; Restore R0.
      CLR  RxFull       ; Buffer can't be full anymore.
      SETB EA          ; Re-enable interrupts.
      RET

;*****
;
;                               Reset
;*****
Reset: MOV  SP,#2Fh     ; Initialize stack start.
      MOV  TCON,#0     ; Set timer off, INT0 to level trigger.
      MOV  P3,#0       ; Turn off all status outputs.

; For this demo, we only set up the baud rate once at reset:

      MOV  A, P1       ; Read baudrate bits from P1.
      RR  A            ; The switches are on bits 2 and 1.
      ACALL BaudRate   ; Set up the selected baud rate.

      MOV  FLAGS,#0    ; Init all status flags.
      MOV  RxDatCnt,#0 ; Clear buffer count.
      MOV  IE,#93h     ; Turn on timer 0 interrupt and
                        ; external interrupt 0.
      CLR  RTS         ; Assert RTS so we can receive.

; The main program loop processes received data and sends it back to the
; transmitter in hexadecimal format. This insures that we can always fill
; up the receiver buffer (since the returned data is longer than the
; received data) for testing purposes. Example: if the letter "A" is
; received, we will echo "A41 ".

MainLoop: JNB  RxAvail,$ ; Make sure an input byte is available.
          ACALL GetRx    ; Get data from the receiver buffer.
          ACALL TxSend   ; Echo original character.
          ACALL PrByte   ; Output the char in hexadecimal format,
          MOV  A,#20h    ; followed by a space.
          ACALL TxSend   ;
          SJMP  MainLoop ; Repeat.

      END

```


Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

Author: David Chen, Shanghai Philips Technology Applications Lab

SUMMARY

This Application Note describes how to use the 87C751 microcontroller to implement a gang programmer for the Philips PCF8582 and PCF8581 I²C EEPROMs. Included are software, schematic, and PCB layout to allow copying from one 16-pin source EEPROM into ten slave EEPROMs. When the process is completed, a green LED turns on and a small buzzer sounds. A red LED by each slave socket indicates a failed IC.

The PCF8582 is a 256x8 CMOS EEPROM with I²C interface. It comes in a variety of 8- and 16-pin plastic packages. This particular layout uses the 16-pin Dual In-Line package. The PCF8581 is a 128x8 CMOS EEPROM.

This Gang Programmer was designed to program default set-up parameters into Philips Television sets which use I²C EEPROMs. The design is being presented here, not just to show how to make an I²C interfaced programmer, but to also serve as an example of using the 87C751 I²C interface.

The programmer reads data from a source EEPROM and copies the data into ten slave EEPROMs. It can also compare the data between the master and the target. There are just two push buttons on the unit, Copy and Reset. Reset goes directly to the Reset pin on the '751, and shorts out the power-on reset capacitor. Copy is connected to INT0, and gets the micro's attention so that the copy process can start.

The programmer is powered from a wall mount, 9V @ 300mA supply. A 5V regulator is on the board. A yellow LED is used to indicate that power has been applied to the board. A green and a pink LED indicate the status of the programming operation, and there is a red LED at each socket to indicate a failure in the programming process to that IC. If everything goes well, the green LED will signal the successful completion of the programming process and a buzzer will also sound. The pink LED will come on if there are any errors in any EEPROM, and the appropriate red LED will indicate the faulty IC. If the pink LED is on and none of the red LEDs are on, there has been an error in the I²C communication link.

OPERATION

Since the PCF8582 only has three Chip Select lines, and there are eleven parts in the system, a different addressing scheme had to be devised. In this system, A0 and A1 have four possible combinations of addresses. A2 is then used as a bank select which is driven by P1.2, P1.3 and P1.4 on the '751. The 10 red LEDs and the pink and green LEDs are multiplexed, since there are not enough port pins left to directly drive them. Port 3 drives a 74LS244 which in turn drives the anodes of the LEDs through 100Ω resistors. The two banks for the multiplexing are driven to ground by discrete transistors which receive their base drive from P1.0 and P1.1.

Jumpers on P1.6 and P1.7 are used to select different options. Jumping P1.6 to ground selects the PCF8581 and leaving that jumper off selects the PCF8582. If P1.7 is jumpered to ground, the programmer will only do a compare between the source and the slaves. This compare operation is done in 16-byte segments. If no jumper, then the normal programming sequence will be done.

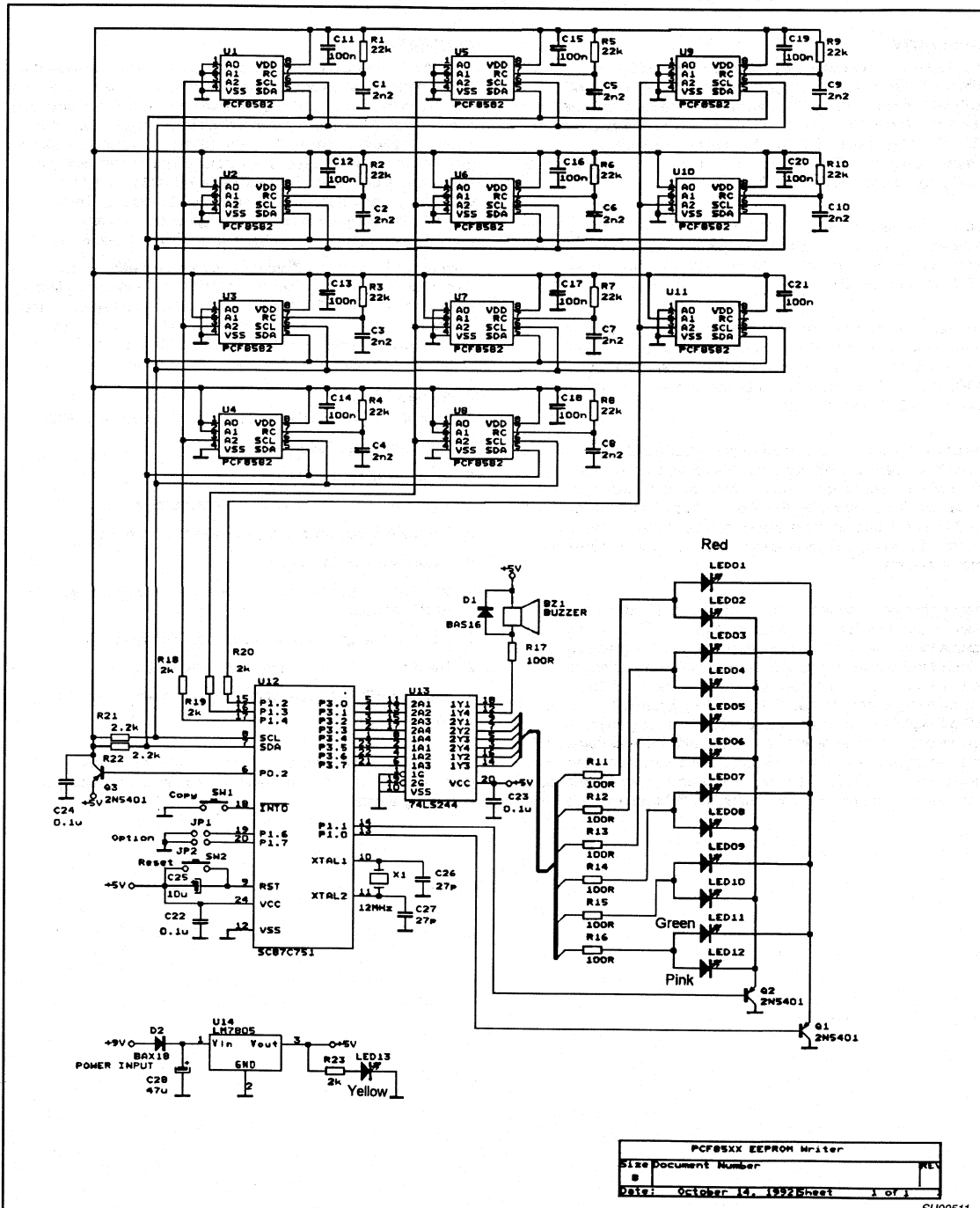
Program Flow

1. Turn on source EEPROM and initialize '751 pointers in RAM
2. Read two bytes of source data
3. Enable target EEPROM array
4. Write the 2-byte data to each of the ten targets
5. Increment subaddress and get next data from source
6. Repeat write to other targets
7. Wait 70ms to allow the information to be written internally to the EEPROM cells
8. Read each target to verify that correct write was done
9. Repeat the 2-byte write and read cycles until done
10. Disable EEPROM power and turn on the green LED and beep
11. Wait for next input

The '751 keeps track of any problems with the programming process, and it skips the verify and subsequent programming of bad or missing targets.

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

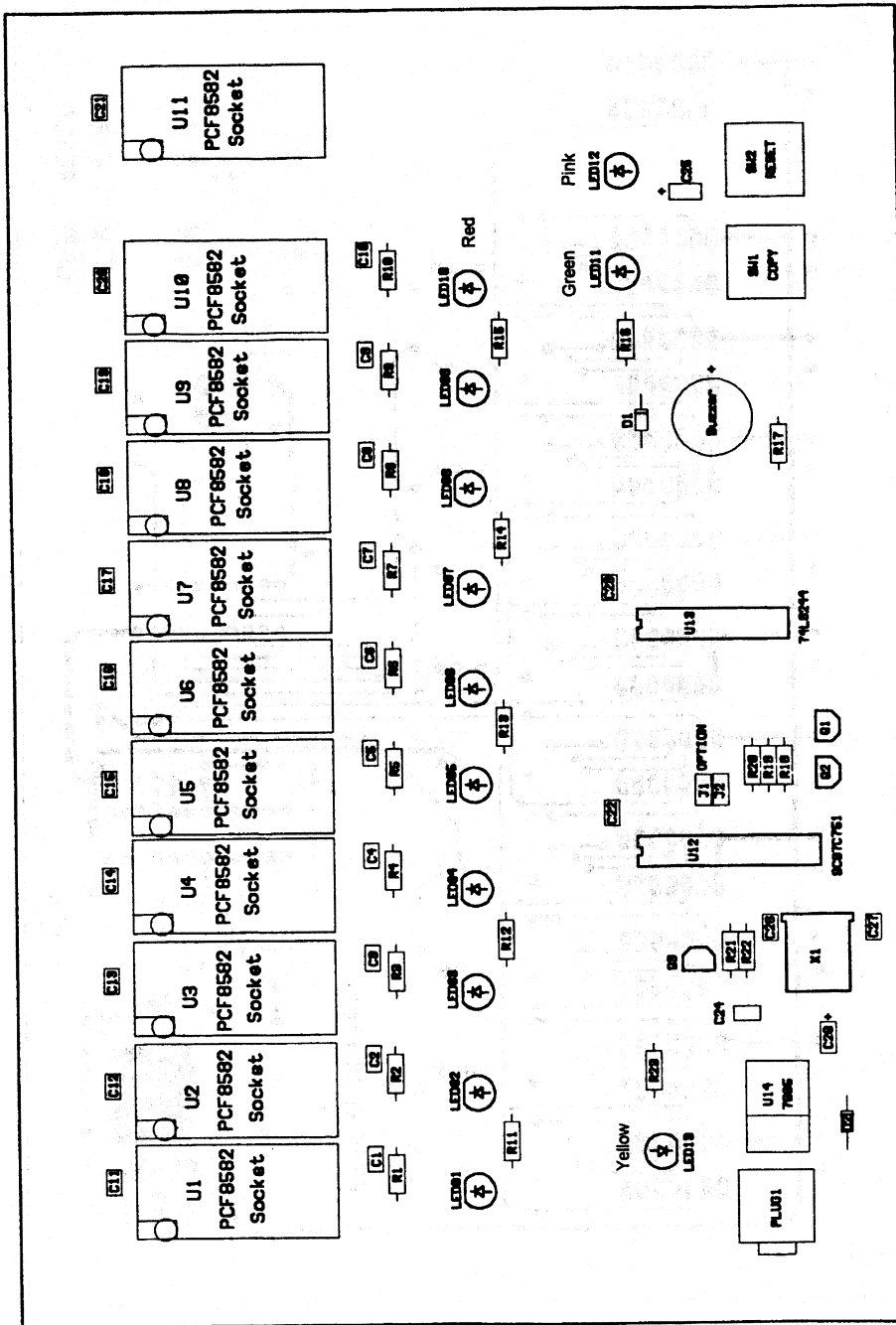


PCF85XX EEPROM Writer		
Size	Document Number	REV
8		
Date:	October 14, 1992	Sheet 1 of 1

SU00511

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

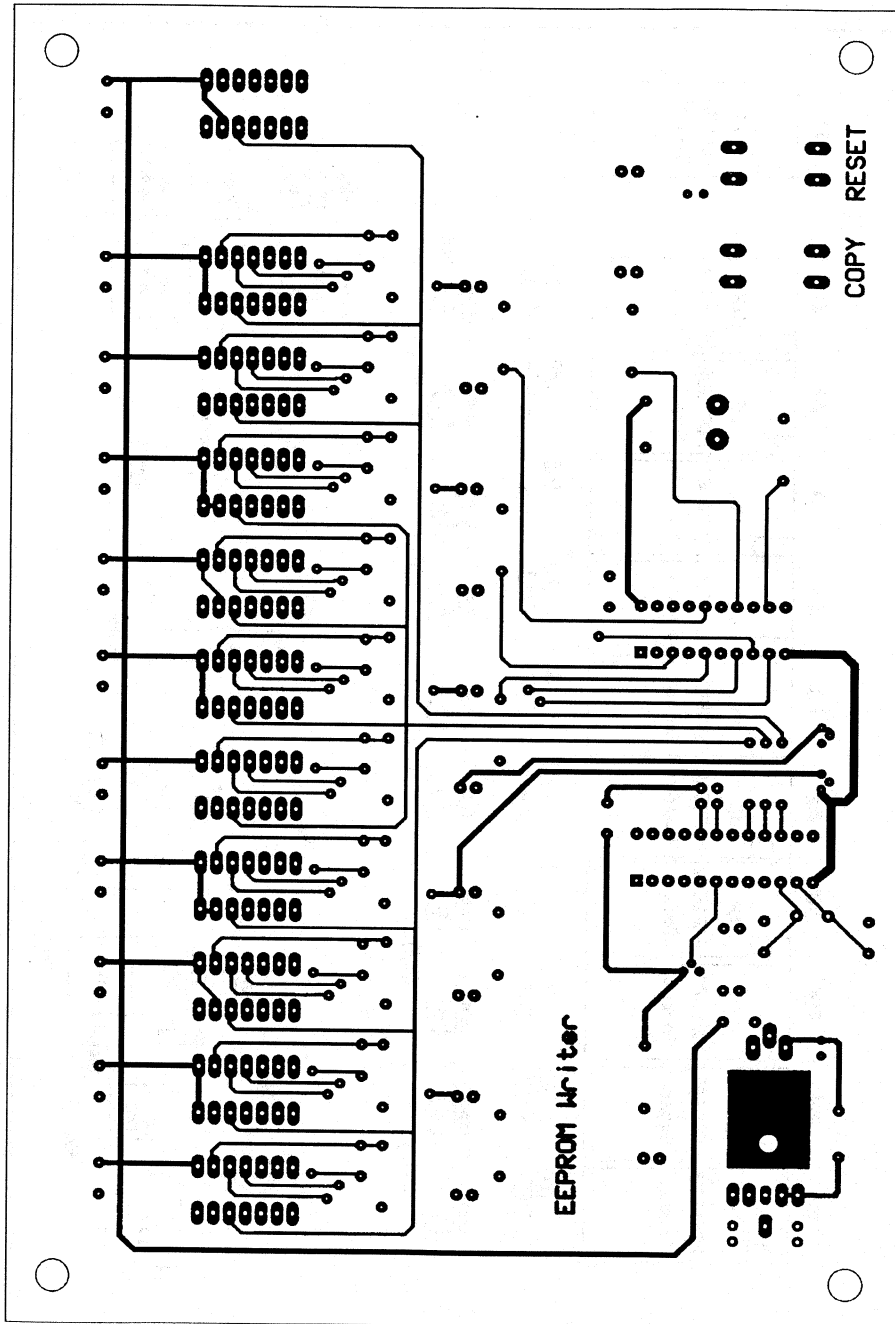
AN453



SU00512

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

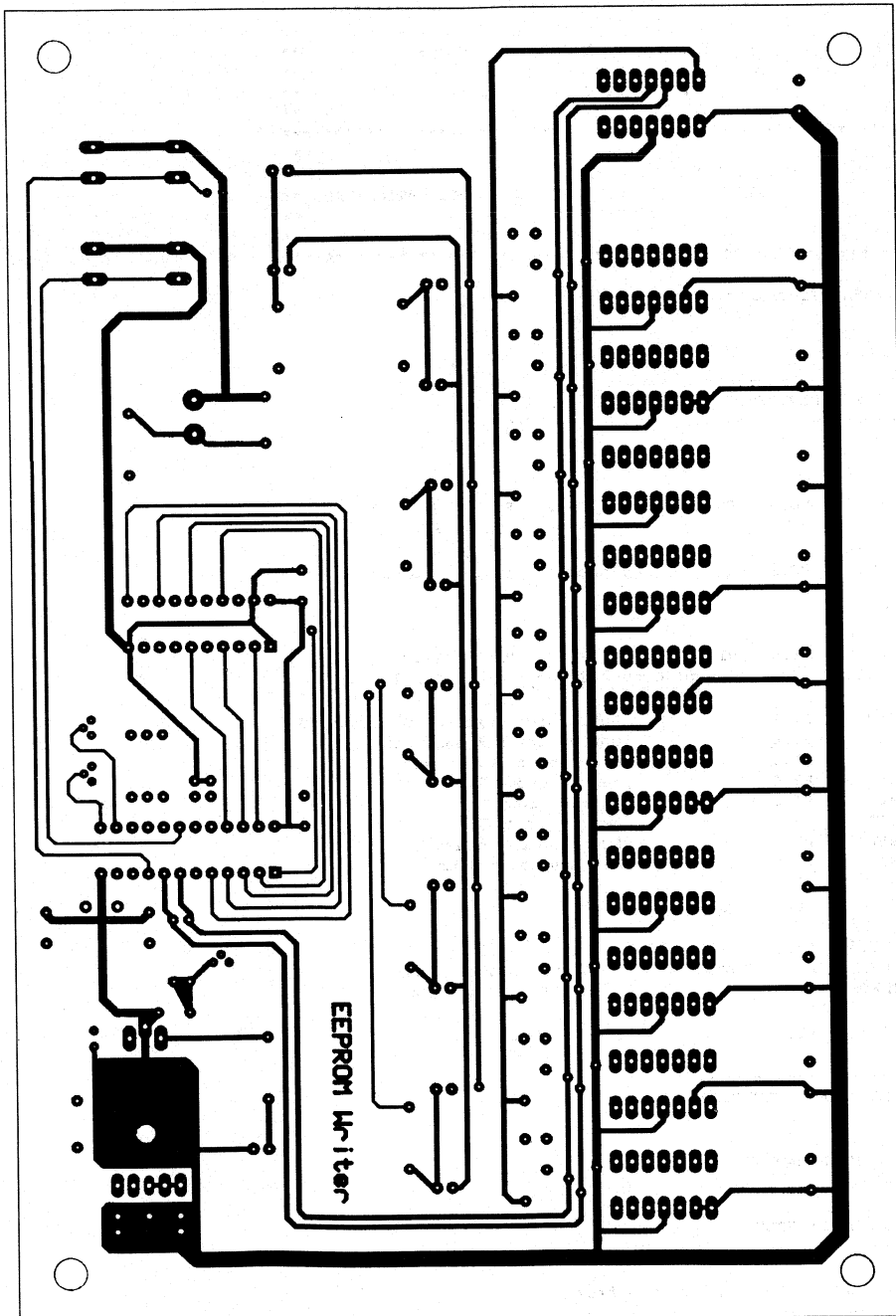
AN453



SU00513

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453



SU00514

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

;
; *****
; **                               **
; **           P C F 8 5 X X   P R O G R O M E R           **
; **                               **
; **           B Y   S C 8 7 C 7 5 1                       **
; **                               **
; *****
; **                               **
; **           EDITOR: DAVID Y. CHEN                         **
; **           SHANGHAI PHILIPS TECHNOLOGY APPLICATION LAB. **
; **           TEL: 86-21-3777760                           **
; **                               **
; *****
;
$Title(PCF8582/8581 Programmer Source Code)
$Mod751
$Debug

;
; -----
; PORT DEFINITIONS
; -----
;
; PORT 0
; -----
; P0.7  P0.6  P0.5  P0.4  P0.3  P0.2  P0.1  P0.0
;                               Power  SDA   SCL
;
Power      BIT      P0.2          ;Power supply for eeprom

;
; PORT 1
; -----
; P1.7  P1.6  P1.5  P1.4  P1.3  P1.2  P1.1  P1.0
; BComp B8582  GO   Group1 Group2 Group3 Col02 Col01
;
BComp      BIT      P1.7          ;Option for compare only when "0"
B8582      BIT      P1.6          ;Option for PCF8581 when "0"
GO         BIT      P1.5          ;Copy start key input
Group1     BIT      P1.4          ;eeprom group 1
Group2     BIT      P1.3          ;eeprom group 2
Group3     BIT      P1.2          ;eeprom group 3
Col02      BIT      P1.1          ;LED display column 2
Col01      BIT      P1.0          ;LED display column 1

;
; PORT 3      (LED display output)
; -----
; P3.7  P3.6  P3.5  P3.4  P3.3  P3.2  P3.1  P3.0
; LED11 LED09 n.c. BUZZER LED07 LED05 LED03 LED01
; /LED12 /LED10 /LED08 /LED06 /LED04 /LED02
;
BZ         BIT      P3.4          ;Beep output

;
; -----
; REGISTER DEFINITION
; -----
;
; REGISTER BANK 0
; -----
;
; R0,R1,R2: FREE
; -----
;
; R3: EEPROM Slave Address Buffer
; -----
;
; R4: EEPROM Subaddress Buffer
; -----

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

; R5: EEPROM Number Counter
; -----

; R6: EEPROM Group Counter
; -----

; R7: RAM Buffer Pointer
; -----

; -----
; EQUIVALENT DEFINITION
; -----

BIT0      EQU      1
BIT1      EQU      2
BIT2      EQU      4
BIT3      EQU      8
BIT4      EQU     010H
BIT5      EQU     020H
BIT6      EQU     040H
BIT7      EQU     080H

CBIT0     EQU     0FEH
CBIT1     EQU     0FDH
CBIT2     EQU     0FBH
CBIT3     EQU     0F7H
CBIT4     EQU     0EFH
CBIT5     EQU     0DFH
CBIT6     EQU     0BFH
CBIT7     EQU     07FH

SourceAdr EQU     0ACH      ;Source eeprom slave address
TargetAdr EQU     0A8H      ;Target eeprom slave address

RTNL      EQU     -200      ;Timer reload for 5 mSec
RTNH      EQU     -20       ;Timer reload

; Masks for I2CFG bits.
BCLRTI    EQU     BIT5
CTVAL     EQU     BIT1      ;CT1, CT0 bit values for I2C.
BTIR      EQU     BIT4      ;Mask for TIRUN bit.
BMRQ      EQU     BIT6      ;Mask for MASTRQ bit.

; Masks for I2CON bits.
BCXA      EQU     BIT7      ;Mask for CXA bit.
BIDLE     EQU     BIT6      ;Mask for IDLE bit.
BCDR      EQU     BIT5      ;Mask for CDR bit.
BCARL     EQU     BIT4      ;Mask for CARL bit.
BCSTR     EQU     BIT3      ;Mask for CSTR bit.
BCSTP     EQU     BIT2      ;Mask for CSTP bit.
BXSTR     EQU     BIT1      ;Mask for XSTR bit.
BXSTP     EQU     BIT0      ;Mask for XSTP bit.

; Register redefinition
SlvAdr    EQU     R3        ;Slave address buffer
SubAdr    EQU     R4        ;Slave subadress buffer

; -----
; RAM and BIT Defination
; -----

LEDRAM1   DATA    28H      ;EEPROM status register1

; BIT7 BIT6 BIT5 BIT4 BIT3 BIT2 BIT1 BIT0
; LED11 LED09 BEP1 LED07 LED05 LED03 LED01

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

BWAIT      BIT      LEDRAM1.7      ;Wait_LED bit (LED11)
BLED09     BIT      LEDRAM1.6      ;EEPROM_LED09 bit
BEP1       BIT      LEDRAM1.4      ;BUZZER
BLED07     BIT      LEDRAM1.3      ;EEPROM_LED07 bit
BLED05     BIT      LEDRAM1.2      ;EEPROM_LED05 bit
BLED03     BIT      LEDRAM1.1      ;EEPROM_LED03 bit
BLED01     BIT      LEDRAM1.0      ;EEPROM_LED01 bit

LEDRAM2    DATA    029H          ;EEPROM status register2

;          BIT7  BIT6  BIT5  BIT4  BIT3  BIT2  BIT1  BIT0
;          LED12 LED10          BEP2  LED08 LED06 LED04 LED02

BERROR     BIT      LEDRAM2.7      ;Error_LED bit (LED12)
BLED10     BIT      LEDRAM2.6      ;EEPROM_LED10 bit
BEP2       BIT      LEDRAM2.4      ;BUZZER
BLED08     BIT      LEDRAM2.3      ;EEPROM_LED08 bit
BLED06     BIT      LEDRAM2.2      ;EEPROM_LED06 bit
BLED04     BIT      LEDRAM2.1      ;EEPROM_LED04 bit
BLED02     BIT      LEDRAM2.0      ;EEPROM_LED02 bit

Flags      DATA    02AH          ;Software status flags.
NoAck      BIT      Flags.0        ;Indicates missing acknowledge in I2C
Fault      BIT      Flags.1        ;Indicates a bus fault of some kind.
Retry      BIT      Flags.2        ;Indicates that last I2C transmission
; failed and should be repeated.
BGo        BIT      Flags.3        ;Flag of Copy_key input
BCol       BIT      Flags.4        ;Flag of LED display column
BBZ        BIT      Flags.5        ;Flag of Buzzer output

; RAM locations used by I2C routines.
BitCnt     DATA    2BH           ;I2C bit counter.
ByteCnt    DATA    2CH           ;Byte counter
RcvDat     DATA    2DH           ;start address of transmit buffer
XmtDat     DATA    2Eh          ;start address of transmit buffer
StackSave  DATA    2Fh          ;Saves stack address for bus recovery.

Stack      DATA    30H          ;Stack address

;
; -----
; GENERAL INTERRUPT ROUTINE
; -----
;
ORG        00H                   ;Reset
AJMP      POR                    ;Power on reset proc

;
ORG        03H                   ;Int0
AJMP      GOIN                   ;Copy_key input

;
ORG        0BH                   ;Counter/Timer0 interrupt
AJMP      Timer

;
ORG        13H
RETI                               ;No external interrupt1 used

;
ORG        1BH                   ;Timer I (I2C timeout) interrupt
AJMP      TimerI

;
ORG        23H                   ;No I2C interrupt used
RETI
;

```


Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

;-----
; I2C bus time our proc
;-----
;
TimerI:   SETB   CLRTI           ;Clear timer I interrupt.
          CLR    TIRUN
          ACALL  ClrInt         ;Clear interrupt pending.
          MOV    SP,StackSave   ;Restore stack for return to main.
          AJMP  Recover        ;Attempt bus recovery.
ClrInt:   RETI
;-----
;
;-----
; Timer 0 interrupt proc
;-----
;Timer0 interrupt is used for LED scan (5mSec period)

Timer:    JB     BCol,Timer1    ;If LED_column 2 was in display
          MOV    P3,LEDRAM1     ;Drive LED_column 1
          SETB   COL02          ;Turn off LED_column 2
          CLR    COL01          ;Light LED_column 1
          AJMP  Timer2
;
Timer1:   MOV    P3,LEDRAM2     ;Drive LED column 2
          SETB   COL01          ;Turn off LED_column 1
          CLR    COL02          ;Light LED_column 2
Timer2:   CPL    BCol          ;Compl. flag of LED column
          RETI
;-----
;
;-----
; External Interrupt 0 proc
;-----
;External interrupt 0 is used to accept Copy_key input

GOIN:    SETB   BGo            ;Set flag for Copy_key input
          RETI
;-----
;
;-----
; Send data byte(s) to EEPROM
;-----
; Entry: SlvAdr = slave address
;        SubAdr = subaddress of eeprom
;        ByteCnt = # of bytes to be sent
;        XmtDat = start address of transmit buffer
; Return: NoAck, Retry flags

SendData: CLR    NoAck          ;Clear error flags.
          CLR    Fault
          CLR    Retry
          MOV    StackSave,SP   ;Save stack address for bus fault.
          MOV    A,SlvAdr       ;Get slave address.
          ACALL  SendAddr       ;Get bus and send slave address.
          JB     NoAck,SDEX      ;Check for missing acknowledge.
          JB     Fault,SDataErr ;Check for bus fault.

          MOV    A,SubAdr       ;Get slave subaddress.
          ACALL  XmitByte       ;Send subaddress.
          JB     NoAck,SDEX      ;Check for missing acknowledge.
          JB     Fault,SDataErr ;Check for bus fault.
          MOV    R0,XmtDat      ;Set start of transmit buffer.

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

SDLoop:    MOV     A,@R0           ;Get data for slave.
           INC     R0
           ACALL  XmitByte        ;Send data to slave.
           JB     NoAck,SDEX      ;Check for missing acknowledge.
           JB     Fault,SDatErr   ;Check for bus fault.
           DJNZ   ByteCnt,SDLoop ;If ByteCnt not zero.
           ; Then continue send data.
SDEX:      ACALL  SendStop        ;Else send an I2C stop.
           RET
;
; Handle a transmit bus fault.

SDatErr:   AJMP   Recover         ;Attempt bus recovery.
;         -----
;
;         -----
;         Receive data bytes from EEPROM
;         -----
; Entry:   SlvAdr = slave address
;         SubAdr = subaddress of eeprom
;         ByteCnt = # of data bytes to be read
;         RcvDat = start address of receive buffer
; Return:  NoAck, Retry, Data in RcvDat buffer

RcvData:   CLR     NoAck          ;Clear error flags.
           CLR     Fault
           CLR     Retry
           MOV     StackSave,SP   ;Save stack address for bus fault.

           MOV     A,SlvAdr       ;Get slave address.
           ACALL  SendAddr        ;Send slave address.
           JB     NoAck,RDEX      ;Check for missing acknowledge.
           JB     Fault,RDatErr   ;Check for bus fault.

           MOV     A,SubAdr       ;Get slave subaddress.
           ACALL  XmitByte        ;Send subaddress.
           JB     NoAck,RDEX      ;Check for missing acknowledge.
           JB     Fault,RDatErr   ;Check for bus fault.

           ACALL  RepStart        ;Send repeated start.
           JB     Fault,RDatErr   ;Check for bus fault.

           MOV     A,SlvAdr       ;Get slave address.
           SETB   ACC.0           ;Set bus read bit.
           ACALL  SendAd2        ;Send slave address.
           JB     NoAck,RDEX      ;Check for missing acknowledge.
           JB     Fault,RDatErr   ;Check for bus fault.

           MOV     R0,RcvDat      ;Set pointer of receive buffer.
           DJNZ   ByteCnt,RDLoop  ;If ByteCnt = 1
           SJMP   RDLast         ; Then goto receive last byte

RDLoop:    ACALL  RDack           ;Get data and send an acknowledge.
           JB     Fault,RDatErr   ;Check for bus fault.
           MOV     @R0,A          ;Save data.
           INC     R0             ;Increase receive buffer pointer.
           DJNZ   ByteCnt,RDLoop  ;Repeat until last byte.

RDLast:    ACALL  RcvByte        ;Get last data byte from slave.
           JB     Fault,RDatErr   ;Check for bus fault.
           MOV     @R0,A          ;Save data.

           MOV     I2DAT,#80h     ;Send negative acknowledge.
           JNB    ATN,$           ;Wait for NAK sent.
           JNB    DRDY,RDatErr    ;Check for bus fault.

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

RDEX:      ACALL  SendStop      ;Send an I2C bus stop.
           RET
;
; Handle a receive bus fault.

RDataErr:  AJMP   Recover       ;Attempt bus recovery.
;
;
; -----
; I2C Bus proc subroutine
; -----
; Send address byte.
; Enter with address in ACC.

SendAddr:  MOV    I2CFG,#BMRQ+BTIR+CTVAL ;Request I2C bus.
           JNB   ATN,$          ;Wait for bus granted.
           JNB   Master,SAErr    ;Should have become the bus master.

SendAd2:   MOV    I2DAT,A        ;Send first bit, clears DRDY.
           MOV    I2CON,#BCARL+BCSTR+BCSTP ;Clear start, releases SCL.
           ACALL XmitAddr       ;Finish sending address.
           RET

SAErr:     SETB   Fault         ;Return bus fault status.
           RET
;
; -----
; Byte transmit routine.
; Enter with data in ACC.
; XmitByte : transmits 8 bits.
; XmitAddr : transmits 7 bits (for address only).

XmitAddr:  MOV    BitCnt,#8      ;Set 7 bits of address count.
           SJMP  XmitBit2

XmitByte:  MOV    BitCnt,#8      ;Set 8 bits of data count.
XmitBit:   MOV    I2DAT,A        ;Send this bit.
XmitBit2:  RL     A              ;Get next bit.
           JNB   ATN,$          ;Wait for bit sent.
           JNB   DRDY,XMErr     ;Should be data ready.
           DJNZ  BitCnt,XmitBit ;Repeat until all bits sent.
           MOV   I2CON,#BCDR+BCXA ;Switch to receive mode.
           JNB   ATN,$          ;Wait for acknowledge bit.
           JNB   RDAT,XMBX     ;Was there an ack?
           SETB  NoAck         ;Return no acknowledge status.

XMBX:     RET

XMErr:    SETB   Fault         ;Return bus fault status.
           RET
;
; -----
; Byte receive routines.
; RDAck : receives a byte of data, then sends an acknowledge.
; RcvByte : receives a byte of data.
; Data returned in ACC.

RDAck:    ACALL  RcvByte       ;Receive a data byte.
           MOV   I2DAT,#0      ;Send receive acknowledge.
           JNB   ATN,$          ;Wait for acknowledge sent.
           JNB   DRDY,RdErr    ;Check for bus fault.
           RET
;
RcvByte:   MOV    BitCnt,#8      ;Set bit count.
           CLR   A              ;Init received byte to 0.

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

RBit:      ORL      A,I2DAT      ;Get bit, clear ATN.
           RL       A           ;Shift data.
           JNB     ATN,$        ;Wait for next bit.
           JNB     DRDY,RdErr   ;Should be data ready.
           DJNZ   BitCnt,RBit   ;Repeat until 7 bits are in.
           MOV     C,RDAT      ;Get last bit, don't clear ATN.
           RLC     A           ;Form full data byte.
           RET

;
RdErr:     SETB    Fault        ;Return bus fault status.
           RET

;
; I2C stop routine.

SendStop:  CLR     MASTRQ       ;Release bus mastership.
           MOV     I2CON,#BCDR+BXSTP;Generate a bus stop.
           JNB     ATN,$        ;Wait for atn.
           MOV     I2CON,#BCDR   ;Clear data ready.
           JNB     ATN,$        ;Wait for stop sent.
           MOV     I2CON,#BCARL+BCSTP+BCXA ;Clear I2C bus.
           CLR     TIRUN        ;Stop timer I.
           RET

;
; I2C repeated start routine.
; Enter with address in ACC.

RepStart:  MOV     I2CON,#BCDR+BXSTR;Send repeated start.
           JNB     ATN,$        ;Wait for data ready.
           JNB     STR,$        ;Wait for repeated start sent.
           MOV     I2CON,#BCARL+BCSTR ;Clear start.
           RET

;
; Bus fault recovery routine.

Recover:   MOV     I2CFG,#BCLRTI+CTVAL ;Request I2C bus.
           ACALL  FixBus        ;See if bus is dead or can be 'fixed'.
           JC     BusErr        ;If not 'fixed'
           SETB   Retry         ;If bus OK, return to main routine.
           CLR    Fault        ;
           CLR    NoAck        ;
           SETB   TIRUN        ;Enable timer I.
           RET

; This routine tries a more extreme method of bus recovery.
; This is used if SCL or SDA are stuck and cannot otherwise be freed.
; (will return to the Recover routine when Timer I times out)

BusErr:    CLR     MASTRQ       ;Release bus.
           MOV     I2CON,#0BCh   ;Clear all I2C flags.
           MOV     LEDRAM1,#0FFH ;Turn on all of LEDs
           MOV     LEDRAM2,#0FFH
           ACALL  Beep          ;Beep alarm for bus error
           SJMP   $-4

; This routine attempts to regain control of the I2C bus after a bus fault.
; Returns carry clear if successful, carry set if failed.

FixBus:    CLR     MastRQ       ;Turn off I2C functions.
           SETB   C            ;
           SETB   SCL          ;Insure I/O port is not locking I2C.
           SETB   SDA          ;
           JNB   SCL,FixBusEx   ;If SCL is low, bus cannot be 'fixed'.
           JB    SDA,RStop      ;If SCL & SDA are high, force a stop.

```

Using the 87C751 microcontroller to gann program PCF8582/PCF8581 EEPROMs

AN453

```

FixBus1:    MOV     BitCnt,#9           ;Set max # of tries to clear bus.

ChekLoop:  CLR     SCL                   ;Force an I2C clock.
           ACALL  SDelay
           JB     SDA,RStop             ;Did it work?
           SETB  SCL
           ACALL  SDelay
           DJNZ  BitCnt,ChekLoop       ;Repeat clocks until either SDA clears
           ; or we run out of tries.
           SJMP  FixBusEx               ;Failed to fix bus by this method.

RStop:     CLR     SDA                   ;Try forcing a stop since SCL & SDA
           ACALL  SDelay                 ; are both high.
           SETB  SCL
           ACALL  SDelay
           SETB  SDA
           ACALL  SDelay
           JNB   SCL,FixBusEx           ;Are SCL & SDA still high? If so,
           JNB   SDA,FixBusEx           ; assume bus is now OK, and return
           CLR   C                       ; with carry cleared.

FixBusEx:  RET
;
;
;-----
; Power on reset for I/O and RAM initialization
;-----
;

POR:       MOV     P0,#03H               ;Switch power supply on for eeprom
           MOV     P1,#0E3H              ;Init. port 1 and disable eeproms
           MOV     P3,#30H               ;Turn off all of LEDs

ResRAM:    MOV     R0,#63                 ;Load RAM counter for clear loop
           MOV     @R0,#0
           DJNZ   R0,ResRAM

           JNB    SCL,BusErr             ;If SCL low then bus error & can't be fixed
           JB     SDA,InitDsp            ;Else If SDA high then bus ok
           SETB  C                       ; Else go to fix bus
           ACALL  FixBus1
           JC     BusErr                 ;If bus not fixed then goto error alarm

InitDsp:   SETB  POWER                   ;Switch power supply off for eeprom
           MOV     LEDRAM1,#0FFH         ;Turn on all of LEDs
           MOV     LEDRAM2,#0FFH

           MOV     SP,#Stack              ;Load stack
           MOV     RTL,#RTNL              ;Load timer for 5mSec.
           MOV     RTH,#RTNH
           MOV     TL,#RTNL               ;Setup timer
           MOV     TH,#RTNH
           MOV     TCON,#BIT4+BIT2        ;Start timer
           MOV     IE,#BIT7+BIT3+BIT1+BIT0 ;Enable interrupt for
           ; Copy_key inputTimer0 & Timer1

           ACALL  Beep                    ;Init. beep
           MOV     LEDRAM1,#0B0H         ;Turn off all of LEDs except wait_led
           MOV     LEDRAM2,#030H

;
Wait:      JNB    GO,$                    ;Wait for Copy_key released
           CLR    BGo                     ;Clear flag of Copy_key
           JNB    BGo,$                   ;Wait for Copy_key input

           MOV     LEDRAM1,#30H           ;Clear all of LEDs
           MOV     LEDRAM2,#30H

           CLR    POWER                   ;switch power supply for eeprom & bus
           ACALL  Delay30                 ;Delay 30 mSEC.

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

MOV     RcvDat,#08H      ;Load pointer for receive buffer address
MOV     XmtDat,#08H     ;Load pointer for tranx buffer address
MOV     R7,#128         ;Load ram location counter for PCF8582
JB      B8582,$+5       ;If option for PCF8582 then skip
MOV     R7,#64          ;Load ram location counter for PCF8581
MOV     SubAdr,#0       ;Load eeprom subaddress
JNB     BComp,PROG03    ;If option for eeprom compare only
;
;
;-----
;
;
;-----
EEPROM Programming Main Loop
;-----
;
;
;-----
PROG01: ACALL  ENSRROM      ;Enable source eeprom
MOV     SlvAdr,#SourceAdr;Load source eeprom slave address
MOV     R6,#3           ;Load eeprom group counter

PROG01A: MOV     ByteCnt,#2   ;Load byte_counter to read source data
ACALL  RcvData         ;Read data from source eeprom
JB      Retry,PROG01A  ;If need retry then try again
JNB     NoAck,PROG02   ;If have ack. then continu
AJMP   Error          ;Else Source eeprom no ack. & goto error
;
PROG02: MOV     R5,#4       ;Load eeprom number counter within group
MOV     A,R6
CJNE   A,#1,$+5       ;If in last eeprom group
MOV     R5,#2          ; then load eeprom number counter with 2
ACALL  ENTGROM        ;Enable target eeprom group
MOV     SlvAdr,#TargetAdr;Load target eeprom slave address

PROG02A: ACALL  RdFlag      ;Read target eeprom status
JC      PROG02C       ;If the target eeprom was error
; then skip it

PROG02B: MOV     ByteCnt,#2   ;Else load byte_counter for data write
ACALL  SendData       ; write data to target eeprom
JB      Retry,PROG02B ;If need retry then try again
JNB     NoAck,PROG02C ;If have ack. then continu
ACALL  SetFlag        ;Else set error flag for the target eeprom

PROG02C: INC     SlvAdr      ;Inc. slave address for next target eeprom
INC     SlvAdr
DJNZ   R5,PROG02A    ;If one group eeprom copy not finished
; then go back to copy next target eeprom
DJNZ   R6,PROG02     ;If three group eeprom copy not finished
; then go back to copy next group eeprom
ACALL  Delay30        ;Delay 70 mSEC
INC     SubAdr        ;Inc. eeprom subaddress
INC     SubAdr        ; for next 2 byte copy loop
DJNZ   R7,PROG01     ;If eeprom location copy not finished
; then go back for 2 byte copy loop

MOV     SubAdr,#0     ;Load eeprom subaddress for verification
MOV     R7,#16        ;Load ram location counter for PCF8582
JB      B8582,PROG03 ;If option for PCF8582 then skip
MOV     R7,#8         ;Else load ram location counter with 8

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

;
; -----
; EEPROM Programming Main Loop
; -----
;
PROG03:  ACALL  ENSRROM      ;Enable source eeprom
        MOV   SlvAdr,#SourceAdr;Load source eeprom slave address
        MOV   R6,#3       ;Load eeprom group counter
        MOV   RcvDat,#08H  ;Load pointer for receive buffer address

PROG03A: MOV   ByteCnt,#16   ;Load byte_counter to read source data
        ACALL RcvData      ;Read data from source eeprom
        JB   Retry,PROG03A ;If need retry then try again
        JB   NoAck,Error   ;If no ack. then go to error

        MOV   RcvDat,#18H  ;Load pointer for tranx buffer address
;
PROG04:  MOV   R5,#4       ;Load eeprom number counter within group
        MOV   A,R6
        CJNE A,#1,$+5     ;If in last eeprom group
        MOV   R5,#2       ; then load eeprom number counter with 2
        ACALL ENTGROM     ;Enable target eeprom group
        MOV   SlvAdr,#TargetAdr;Load target eeprom slave address

PROG04A: ACALL  RdFlag     ;Read target eeprom status
        JC   PROG04C     ;If the target eeprom was error
                    ; then skip it
PROG04B: MOV   ByteCnt,#16   ;Else load byte_counter to read data from
        ACALL RcvData      ; target eeprom for verification
        JB   Retry,PROG04B ;If need retry then try again
        JB   NoAck,PROG04C ;If no ack. then go to set error flag
        ACALL Compare     ;Compare the data read from source and
        JZ   PROG04D     ; target eeprom
                    ;If the data is ok then continu
PROG04C: ACALL  SetFlag    ;Else set error flag

PROG04D: INC   SlvAdr      ;Inc. slave address for next target eeprom
        INC   SlvAdr
        DJNZ R5,PROG04A   ;If one group eeprom verify not finished
                    ; then go back to verify next eeprom
        DJNZ R6,PROG04   ;If three group eeprom verify not finished
                    ; then go back to verify next group eeprom

        MOV   A,SubAdr
        ADD   A,#16       ;incr. eeprom subaddress for next 16 byte
        MOV   SubAdr,A    ; verification loop
        DJNZ R7,PROG03   ;If eeprom location verify not finished
                    ; then go back for 16 byte verify loop
        MOV   A,LEDAM1
        ANL  A,#BIT6+BIT3+BIT2+BIT1+BIT0
        JNZ  Error1      ;go to turn on Error_LED
        MOV   A,LEDAM2
        ANL  A,#BIT6+BIT3+BIT2+BIT1+BIT0
        JNZ  Error1      ;go to turn Error_LED

PROG05:  CLR   GROUP3     ;Disable all of eeprom group
        SETB  POWER      ;Switch power supply off for eeprom & bus
        SETB  BWAIT      ;Turn on Wait_LED
        ACALL Beep       ;Beep for copy completement
        AJMP  Wait       ;Go to wait for next Copy input
;
; -----
Error:   MOV   LEDRAM1,#30H ;Turn off all of LED
        MOV   LEDRAM2,#30H
Error1:  SETB  BERROR     ;Turn on Error_LED
        AJMP  PROG05
;
; -----

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

;-----
; Compare Code Between Source and Target EEPROM
;-----
;Compare 16 bytes between 08h-17h and 18h-27h
;Output: A = 0 SAME
;        A <> 0 NOT SAME

Compare:    MOV     R0,#07H           ;Load pointer for RAM block compare
            MOV     R1,#17H
            MOV     R2,#16           ;Load byte counter for compare

Compare1:   INC     R0
            INC     R1
            MOV     A,@R0
            XRL    A,@R1
            JNZ    CompExit         ;If two byte not same then exit with A<>0
            DJNZ   R2,Compare1     ;If compare loop finished then exit with
;                                       ;(A)=0

CompExit:   RET
;-----

;-----
; Buzzer driver
;-----

Beep:       SETB    BBZ             ;Set buzzer output flag
            MOV     R0,#6           ;Load counter

Beep0:      MOV     R1,#0FFH

Beep1:      MOV     R2,#14
            CPL     BBZ             ;Compl. buzzer output flag
            MOV     C,BBZ
            MOV     BZ,C           ;output to drive buzzer
            MOV     BEP1,C
            MOV     BEP2,C

Beep2:      ACALL   SDelay
            DJNZ   R2,Beep2
            DJNZ   R1,Beep1
            DJNZ   R0,Beep0
            SETB   BEP1
            SETB   BEP2
            SETB   BZ
            RET
;-----

;-----
; Enable the Source EEPROM
;-----

ENSRROM:   CLR     GROUP1         ;Disable eeprom group1 and group2
            CLR     GROUP2
            SETB   GROUP3         ;Enable eeprom group3 for source eeprom
            RET
;-----

```


Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

;
; -----
; Enable the Target EEPROM
; -----

ENTGROM:  CLR     GROUP1           ;Disable all of eeprom group first
          CLR     GROUP2
          CLR     GROUP3
          MOV     A,R6
          CJNE   A,#01,$+7       ;If target eeprom not in group3 then skip
          SETB   GROUP3         ;Else enable eeprom group3
          AJMP   ENTGROM1
          CJNE   A,#02,$+7       ;If target eeprom not in group2 then skip
          SETB   GROUP2         ;Else enable eeprom group2
          AJMP   ENTGROM1
          SETB   GROUP1         ;Enable eeprom group1

ENTGROM1:  RET
;
; -----
; Time Delay 30mSEC
; -----

Delay30:  MOV     R0,#60

Dly30A:   MOV     R1,#0FFH
          DJNZ   R1,$
          DJNZ   R0,DLY30A
          RET
;
; -----
; Read EEPROM Status
; -----

RdFlag:   MOV     A,#3           ;Eeprom number =
          CLR     C              ; ((eeprom group number)-1) * 4
          SUBB   A,R6           ; + (eeprom number in group)
          MOV     B,A           ;((eeprom group number)-1) =
          MOV     A,#4         ; (3-(eeprom group counter))
          MUL    AB            ;((eeprom group number)-1) * 4
          MOV     R2,A         ;Backup
          MOV     A,R6
          CJNE   A,#01,$+7     ;If it is not eeprom group 3 then skip
          MOV     A,#03       ;Else (eeprom number in group) =
          AJMP   $+4          ; (3 - (eeprom number counter in group))
          MOV     A,#05       ;It is not eeprom group 3, then
          CLR     C           ; (eeprom number in group) =
          SUBB   A,R5         ; (5 - (eeprom number counter in group))
          ADD    A,R2         ;Get eeprom number from 1 to 10
          DEC    A           ;Get address offset of eeprom status table
          RL     A
          RL     A
          MOV    DPTR,#ReadTable ;Load status table address
          JMP    @A+DPTR      ;Jump to read eeprom status flag
;
ReadTable: MOV    C,BLED01     ;Read eeprom_1 status in carry
          AJMP  TableExit
          MOV    C,BLED02     ;Read eeprom_2 status in carry
          AJMP  TableExit
          MOV    C,BLED03     ;Read eeprom_3 status in carry
          AJMP  TableExit
          MOV    C,BLED04     ;Read eeprom_4 status in carry
          AJMP  TableExit
          MOV    C,BLED05     ;Read eeprom_5 status in carry
          AJMP  TableExit

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```

MOV     C,BLED06      ;Read eeprom_6 status in carry
AJMP   TableExit
MOV     C,BLED07      ;Read eeprom_7 status in carry
AJMP   TableExit
MOV     C,BLED08      ;Read eeprom_8 status in carry
AJMP   TableExit
MOV     C,BLED09      ;Read eeprom_9 status in carry
AJMP   TableExit
MOV     C,BLED10     ;Read eeprom_10 status in carry

TableExit:  RET
;
;
; -----
; Set EEPROM Status
; -----
;

SetFlag:   MOV     A,#3      ;Eeprom number =
           CLR     C        ; ((eeprom group number)-1) * 4
           SUBB   A,R6      ; + (eeprom number in group)
           MOV   B,A        ; ((eeprom group number)-1) =
           MOV   A,#4      ; (3-(eeprom group counter))
           MUL   AB        ; ((eeprom group number)-1) * 4
           MOV   R2,A      ;Backup
           MOV   A,R6
           CJNE  A,#01,$+7  ;If it is not eeprom group 3 then skip
           MOV   A,#03      ;Else (eeprom number in group) =
           AJMP  $+4        ; (3 - (eeprom number counter in group))
           MOV   A,#05      ;It is not eeprom group 3, then
           CLR   C        ; (eeprom number in group) =
           SUBB  A,R5      ; (5 - (eeprom number counter in group))
           ADD   A,R2      ;Get eeprom number from 1 to 10
           DEC   A        ;Get address offset of eeprom status table
           RL    A
           RL    A
           MOV   DPTR,#SetTable ;Load status table address
           JMP   @A+DPTR    ;Jump to set eeprom status flag
;

SetTable:  SETB   BLED01    ;Set LED_1 error flag
           AJMP  TableExit
           SETB  BLED02    ;Set LED_2 error flag
           AJMP  TableExit
           SETB  BLED03    ;Set LED_3 error flag
           AJMP  TableExit
           SETB  BLED04    ;Set LED_4 error flag
           AJMP  TableExit
           SETB  BLED05    ;Set LED_5 error flag
           AJMP  TableExit
           SETB  BLED06    ;Set LED_6 error flag
           AJMP  TableExit
           SETB  BLED07    ;Set LED_7 error flag
           AJMP  TableExit
           SETB  BLED08    ;Set LED_8 error flag
           AJMP  TableExit
           SETB  BLED09    ;Set LED_9 error flag
           AJMP  TableExit
           SETB  BLED10    ;Set LED_10 error flag
           AJMP  TableExit
;
; -----

```

Using the 87C751 microcontroller to gang program PCF8582/PCF8581 EEPROMs

AN453

```
; -----  
; Short delay routine  
; -----  
; (10 machine cycles).  
  
SDelay:  NOP  
        NOP  
        NOP  
        NOP  
        NOP  
        NOP  
        NOP  
        NOP  
        RET  
;  
-----  
END
```

Interfacing the 83C576/87C576 to the ISA bus

AN454

INTRODUCTION

The most interesting feature of the Philips 83C576, and the principal subject of this application note is its Universal Peripheral Interface (UPI). The UPI is a microprocessor slave interface which allows the '576 to communicate with a microprocessor or microcontroller host with minimal support logic. The UPI acts as a "bus gasket" between the 8051 core inside the '576 and the host. Commands and Data can easily be exchanged over this interface. Along with the hardware interface, a simple, effective bidirectional software protocol can be implemented for reliable data transfer. Each device can pace the exchange of data using a double bi-directional data register implemented as part of the UPI inside the '576. As part of the UPI definition, hardware flow control is supported so that both the host and 8051 core can each establish whether the other has written or read any data to the UPI. This flow control scheme is the heart of a synchronous interface that is independent of the performance of the host or 8051 core.

The UPI is ideal for the PC environment and provides an almost seamless interface to the PC host via the ISA bus. This application note demonstrates in both hardware and software terms how to interface the '576 to a PC host. The UPI interface occupies two locations in the memory or IO space of each device. The first location is the Data register and the second is the Status register. The Data register is simply a pair of registers one directed to the host and the other directed to the 8051 core. This allows both the host and 8051 core to write to the data register simultaneously without effecting each-others data. The write cycle events are recorded as 'buffer flags' (IBF and OBE) in the Status register. One further unique feature of this interface is the concept of 'Commands and Data' in that any data written to the Status register address is directed to the Data register and a flag is set in the Status register recording this cycle as a Command. This flag is known as the 'AF' flag and indicates that a write cycle has been performed on the Status register.

From a software perspective the host device is the 'master' of the UPI. Even though from a hardware sense, each device can pace the cycles, the provision of a single AF flag precludes the implementation of a multi-master protocol. This limitation, however, is not severe, in that this provides a simple scheme for exchanging data between the devices. This interface provides the mechanisms

for many protocol schemes, the most effective one is used in PCs over the keyboard interface and operates as follows:

HOST (Master)	SLAVE ('576 core)
Write a command to the Status register.	AF flag set, the byte in the data register is a Command
Poll the IBF flag in the Status register; wait for it to be 0.	Read the command.
Write the Information byte to the Data register.	AF=0, read the information byte from the Data register; Execute the Command.
Poll OBE, wait for it to be cleared in the Status register; this means that a response is available in the data register.	Write a Response byte to the data register.
Read the Response byte.	

DESCRIPTION

The purpose of this Application Note is to demonstrate the use of the '576 in a PC environment. This example shows the use of a '576 as a peripheral to the PC/AT. The example shows the implementation of a data acquisition card with digital IO and analog IO. Some software is described to show how to exchange data and commands between the PC and the '576. In this example, the PC is the host of the transfers and the '576 is the slave. An example circuit is also described which details the electrical interface to the PC/AT ISA bus.

Figure 1 shows the basic configuration of the acquisition system.

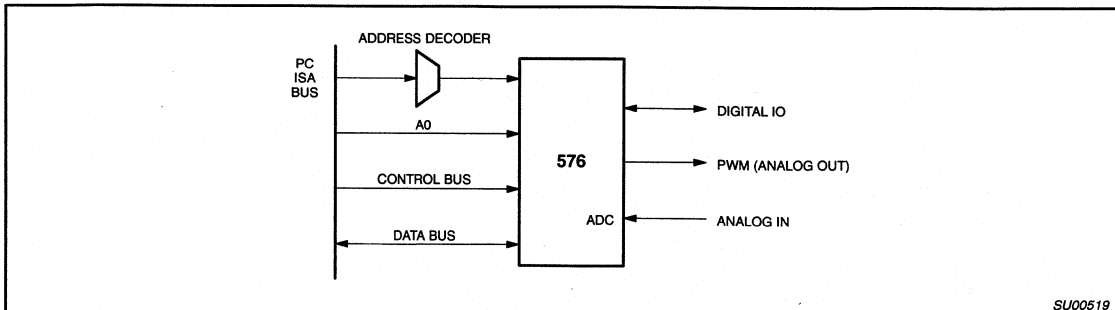


Figure 1. Block Diagram

SU00519

Interfacing the 83C576/87C576 to the ISA bus

AN454

UPI DEFINITION

The Universal Peripheral Interface (UPI) functions as a microprocessor slave interface. This allows the '576 to interface directly to a microprocessor bus as a slave or peripheral device.

The UPI is an 8-bit bidirectional data register, P0, with an associated status register, UCS. The data buffer is comprised of two registers; the input register and, the output register. The input register can be written by the host and read by the '576. The output register can be written by the '576 and read by the host. The status register may be read or written by the '576 core but may only be read by the host. The Status register bits can also be affected by hardware events, for example, a host write cycle to the data register will set the IBF flag.

The host control interface for these registers is comprised of four signals, -RD, -WR, an address line A0, and a chip select -CS. Data transfer is directed to the UPI as shown below:

-CS	A0	-RD	-WR	CONDITION
0	0	0	1	Read Output Data Register
0	1	0	1	Read Status Register
0	0	1	0	Write Input Data Register, AF=0
0	1	1	0	Write Input Data Register, AF=1
1	X	X	X	Disable IO

Write cycles to the data register with A0 = 1 cause the AF flag to be set in the status register. These cycles are generally interpreted as commands. Write cycles to the data register with A0 = 0 cause the AF flag to be cleared in the status register. These cycles are usually interpreted as data.

The status register has 4 control bits and 4 user defined status bits. They are defined as follows:

UCS

7	6	5	4	3	2	1	0
ST7	ST6	ST5	ST4	UE	AF	IBF	OBE

- UCS.7 ST7 User defined status bit
- UCS.6 ST6 User defined status bit
- UCS.5 ST5 User defined status bit
- UCS.4 ST4 User defined status bit
- UCS.3 UE UPI enable bit. 0 = Disabled, 1 = Enabled.
- UCS.2 AF Address Flag – contains the state of the A0 (Address) pin on the last write cycle.
0 = write cycle with A0 cleared
1 = write cycle with A0 set
- UCS.1 IBF Input Buffer Full Flag – set by hardware on the rising edge of a write command to the Input Data Register. Cleared by hardware on the completion of a read cycle of the Input Data Register by the '576.
- UCS.0 OBE* Output Buffer Empty Flag – Cleared by hardware on the completion of a write cycle to the Output Data Register by the '576. Set by hardware on the rising edge of the read command of the Output Data Register by the host.

* NOTE: This flag is OBE when read by the MCU, but is inverted or OBF (Output Buffer Full) when read by an external host.

General Handshaking Protocol

Host write cycles with A0 set are directed to the Input Data Register. The '576 can distinguish the cycles by monitoring the AF flag (Address Flag) in the status register.

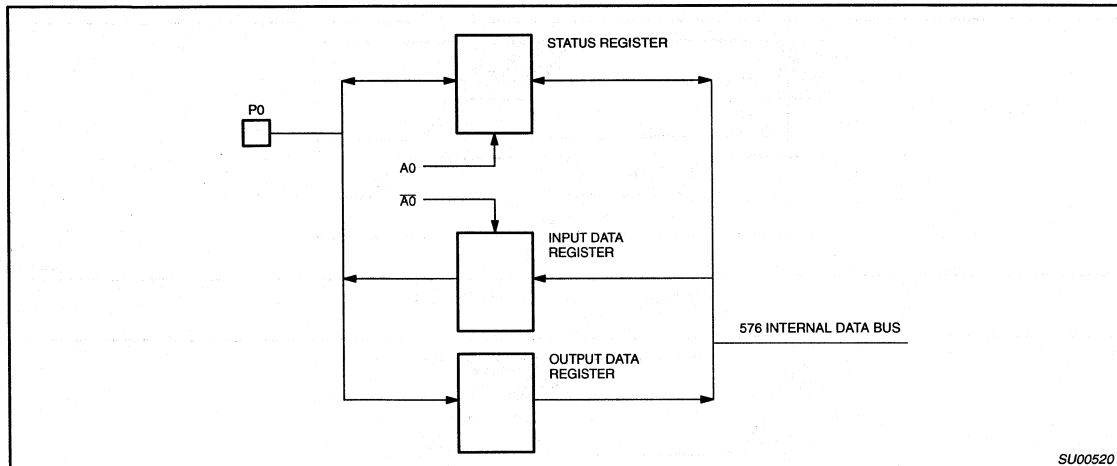


Figure 2. Internal Structure of the UPI Registers

SU00520

Interfacing the 83C576/87C576 to the ISA bus

AN454

ISA BUS INTERFACE

The data acquisition system is comprised of the '576 microcontroller and an address decoder. The '576 provides an almost seamless interface to the PC/AT IO channel. The interface to the PC is known as the ISA (Industry Standard Architecture) bus. This bus is an asynchronous, command driven bus. Most notebook and PDA chipsets provide programmable or fixed address decoders assigned to an output pin. This pin is used to control the $\overline{\text{CS}}$ input of the '576 thus providing a completely seamless interface to the ISA bus.

Address Path

In this application the '576 is interfaced as an IO device. The IO space on a PC is 0 to 03FFH (1Kbytes), therefore SA0 to SA9 must be decoded to locate the '576 in a single region. SA0, however, is used to select between the Data and Status registers, thus the '576 occupies two locations in IO space. For the acquisition card we selected 0300H for the Data register and 0301H for the Status register. 300H and 301H have been chosen because they are assigned to a "prototyping area" in the PC ISA IO space map. The address decoder must reject the address during DMA cycles. For this purpose, AEN is used in the decoder and qualified for its low, inactive state (AEN, when active indicates that the current cycle is a DMA cycle). Output Y0 of U2 is asserted (low) when SA5 to SA7 are low, SA8 is high and AEN is low. U3 further decodes the address and output Y0 is asserted (low) when Y0 of U2 is low together with SA1 to SA4 low and SA9 high. Output Y0 of U3 is fed to the $\overline{\text{CS}}$ input of the '576, it is asserted (low) when the address is 300H with AEN low or 301H with AEN low.

Data Path

The '576 interfaces to the least significant 8 bits of the data bus, SD0 to SD7. When the '576 is configured in UPI mode, PORT0 is configured as push-pull outputs for host read cycles. ISA requires 24mA I_{OL} for this interface. The '576 can only handle 15mA. The 24mA requirement goes back to the XT-TTL days and with current CMOS motherboards 12mA is probably sufficient with a fully loaded system.

Interrupt

An interrupt is employed to completely demonstrate the integration of the '576 to the PC. The first free interrupt is IRQ10. For this reason alone, we need to connect to the extended AT ISA slot where this interrupt line is available. Interrupts on PCs are edge sensitive, sometimes shared and are usually pulled-up on the motherboard just to make life more complicated. If we want to generate an interrupt, port pin P2.3 must be driven low then high. The 8259 interrupt controller in the PC will see the interrupt on the rising edge. Once the host acknowledges the interrupt by say, reading the Data register (detected by the assertion of OBE) the '576 must drive P2.3 high, returning IRQ10 to a high impedance state.

Timing

Another aspect of the design is to consider the timing implications of the UPI. The diagrams below show the relationships of the control, address and data signals.

SYMBOL	PARAMETER	LIMITS
t_{AS}	Address Setup time	21ns MIN
t_{PW}	Command pulse width	600ns MIN (assumes 8.33MHz bus)
t_{DS}	Data setup time	15ns MIN
t_{DH}	Data hold time	15ns MIN
t_{CR}	Cycle recovery time	55ns MIN

The address setup time is lengthened by the two 3 to 8 line decoders, U2 and U3. The ISA worst case is 9ns, we are adding 2 further delays of 6ns giving 21ns of address setup. If the ISA bus is clocked at a higher rate as in some PDAs and some notebooks, the Command pulse width will be shortened. The ISA bus can be clocked as high as 11.1MHz, yielding a minimum command pulse width of 450ns.

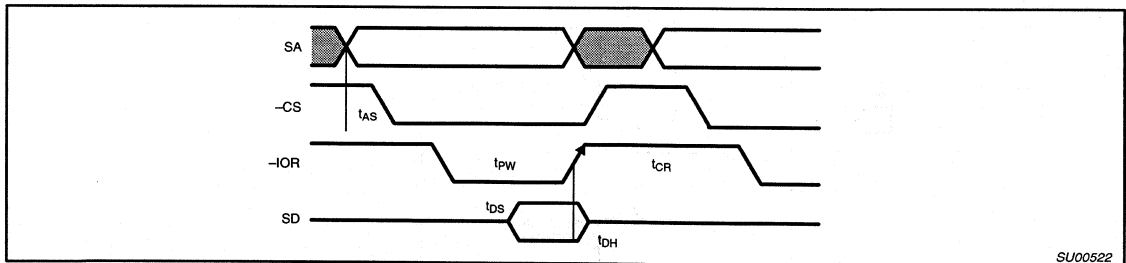


Figure 3. ISA Cycle IO Read Timing

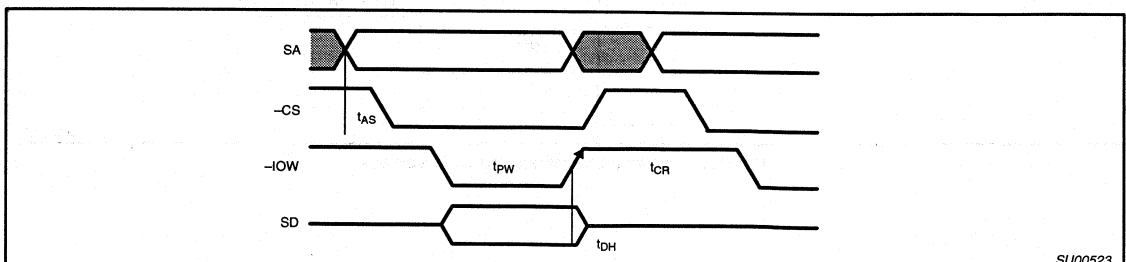


Figure 4. ISA Cycle IO Write Timing

Interfacing the 83C576/87C576 to the ISA bus

AN454

SOFTWARE

The example below shows a typical protocol employed here to exchange data between the PC and the '576.

The protocol that we are employing is only a subset of what can be achieved with this Command / Data, two byte protocol. To complete a transaction, the PC sends two bytes, a Command and Information byte, then the '576 returns two bytes; a response and information byte. Sometimes the information bytes will be meaningless; that's okay, as long as we stick to the protocol.

COMMANDS

	Command	Information	
01H	Get Analog Channel	Channel	[0, 1, 2, 3]
02H	Get Digital Byte	NULL	
03H	Put Digital Byte	Data Byte	
04H	Increase PWM	Channel	[0, 1]

RESPONSES

	Response	Information
01H	Get Analog Channel Complete	M.S. 8 bits of ADC conversion
02H	Get Digital Byte Complete	Data Byte
03H	Put Digital Byte Complete	NULL
04H	Increase PWM Complete	NULL

DRIVERS

PC Driver

The PC must write a command. This is done by making an IO cycle to port 0301H. This sets the AF flag in the status register and causes an IBF interrupt in the '576. The PC can poll the status register to see when the IBF flag has cleared, this means that the '576 has read the Command in the Input Data Register. The PC then sends the Information byte; this is done in the same way except that the cycle is made to 300H.

'576 Driver

Communication from the '576 to the PC is established by driving the interrupt request line, IRQ10, high. The PC first reads the status register to check if the interrupt was from the '576 instead of some other external device. If the OBF flag is set, the PC knows that the data in the Output Data Register is valid. The PC then reads to Output Data Register which causes the OBF flag to become cleared. Once the '576 detects this read cycle, the interrupt line, P2.3, can be returned to its low state. The AF bit in the Status register indicates whether the data is a Response byte to a previous command (AF set) or an associated Information byte (AF cleared).

Interfacing the 83C576/87C576 to the ISA bus

AN454

```

/*
Code to demonstrate the use of the UPI to the '576
in an interrupt driven mode.

Written in Microsoft C7.0 for PC AT

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <conio.h>

#define UC      unsigned char
#define UL      unsigned long
#define UI      unsigned int

#define P8259A_IMR    0x21    /* ;interrupt mask register i/o address */
#define P8259B_IMR    0xa1    /* ;interrupt mask register i/o address */
#define EOI          0x20
#define P8259A       0x20
#define P8259B       0xa0

#define ENABLE_IRQ10    ~0x04
#define ENABLE_IRQ2     ~0x04

#define UPI_DATA_REG    0x300
#define UPI_STATUS_REG  0x301
#define IBF              0x02
#define OBF              0x01

unsigned char interrupt_received = 0;

/* declare a variable to store the pointer of the current interrupt service routine */
void (_interrupt_far *old_int)();

/*Module put_byte().

put_byte() has two parameters; the address of the IO port (port) and the value to be written to the IO
port (data). A dummy variable is declared for lint purposes, since the C function outp() returns an
unsigned short. First put_byte() uses the C outp() function to write the byte to the IO port then,
put_byte() monitors the IBF flag in the status register to see if the '576 has read the byte. When the PC
writes the byte to the input register, the '576 automatically sets the IBF flag. When the '576 core reads
the input register, the IBF flag is automatically cleared. The while() loop breaks when the IBF is 0. The
C function inp() is used to read the status register, the returned value is logically ANDed with the
constant IBF (0x02) and the result is complemented and evaluated by the while() statement. */

void
put_byte (
    unsigned short port,
    unsigned char data) {

    unsigned short dummy;

    dummy = outp(port, UPI_DATA_REG);    // send the byte
    while (~(inp(UPI_STATUS_REG) && IBF)); // check to see if '576 has read it
}

```


Interfacing the 83C576/87C576 to the ISA bus AN454

```
/*Module get_byte()
```

get_byte() has 1 parameter and 1 return value. The parameter is used to pass the address of the IO port to read. The return value is used to pass back the contents of the read port as a byte. get_byte first waits for the byte to be written to the UPI port by the '576. This is achieved by monitoring the OBE flag in the status register. The C function inp() is used to read the contents of the status register. The returned word is masked (ANDed) with the constant OBF (0x01) and the negated result is evaluated by the while() statement. If the IBF flag is 0, the while loop continues to read and evaluate the status register, if the IBF flag is set the while loop breaks. When the while loop breaks, there must be something to read in the output data register, the C function inp() is used to read the input data register whose address is past in the variable 'port'. The result is cast as a byte and returned to the calling routine. */

```
unsigned char
get_byte (
    unsigned short port) {
    while (!(inp(UPI_STATUS_REG) && OBF));           // wait for the byte
    return ((byte)inp(port));
}
```

```
/*Module send_command()
```

send_command assembles a two byte message comprising of a command byte followed by an information byte. If we direct the first byte to the address of the status register, it will appear in the output data register and the AF flag will automatically be set. The '576 will read the status register before the data register, the set AF flag will tag the data as a command. The information byte is directly written to the output data register, thus the AF flag will be cleared and the '576 will interpret the byte as information. */

```
void
send_command (
    unsigned char command,
    unsigned char information ) {
    put_byte(UPI_STATUS_REG,command);
    put_byte(UPI_DATA_REG,information);
}
```

```
/* Module get_response()
```

This module has no parameters and 1 return value. get_response strips out the information byte from a two-byte message from the '576. The first byte is always the echoed command and is discarded by assigning it to a dummy variable. The send byte is always the information and is read using the get_byte() function and returned to the calling function. */

```
unsigned char
get_response (
    void ) {
    unsigned char dummy;
    dummy = get_byte(UPI_DATA_REG);           // get the response
    return(get_byte(UPI_DATA_REG));          // return the info byte
}
```

Interfacing the 83C576/87C576 to the ISA bus

AN454

```
/* Module EnableIRQ10()
```

This module enables the hardware interrupt, IRQ10 via the 8259 programmable interrupt controller (PIC) inside the PC. IRQ 10 is a cascaded interrupt driven by a second PIC attached to IRQ2 of the first. Therefore both PICs need to be unmasked. The C function `disable()` issues a CLI instruction to the processor disabling interrupts. The current value of the interrupt mask for PICA is read and ANDed with the mask cleared for IRQ2. The current value of the interrupt mask for PICB is read and ANDed with the mask cleared for IRQ10. The new masks are written to the PICs consecutively. Interrupts are re-enabled using the C function `enable()` which issues an STI instruction to the processor. */

```
void
EnableIRQ10 (void) {

    unsigned short mask;

    _disable();
    mask = _inp(P8259A_IMR);
    mask &= ENABLE_IRQ2
    _outp (P8259A_IMR,mask);

    mask = _inp(P8259B_IMR);
    mask &= ENABLE_IRQ10;
    _outp (P8259B_IMR,mask);
    _enable();
}
```

```
/* Module Irq10_isr()
```

This module is the service routine for an IRQ10 interrupt. It simply sets a global flag which is monitored by the main loop and then issues a "non specific end-of-interrupt (EOI)" to each PIC. The EOI flags are written to reset the interrupt signal that the PICs assert to each other and the processor."

```
void
__cdecl __interrupt __far Irq10_isr(void) {

    interrupt_recieved = 1;

    _outp(P8259A,EOI);
    _outp(P8259B,EOI);
}
```

Interfacing the 83C576/87C576 to the ISA bus AN454

```
void
main(void) {

    printf("\n576 Interrupt Interface Utility\n");

    /* save the pointer of the current interrupt service routine using the C function _dos_getvect() */
    old_int = _dos_getvect(0x12);

    /* store the pointer of our interrupt service routine (Irq10_isr) using the C function _dos_setvect() */
    _dos_setvect(0x12, Irq10_isr);

    /* clear the global flag */
    interrupt_received = 0;

    /* unmask and enable interrupts */
    EnableIRQ10();

    printf("\nGet Analog Channel      01 00 returned ");
    /* send the command */
    send_command(1,0);
    /* wait for an interrupt */
    while(~interrupt_received);
    /* print the result on sdtio (the screen) */
    printf("%02x",get_response());
    /* clear the global interrupt flag */
    interrupt_received = 0;

    printf("\nGet Digital Byte      02 00 returned ");
    send_command(2,0);
    while(~interrupt_received);
    printf("%02x",get_response());
    interrupt_received = 0;

    printf("\nPut Digital Byte      03 55 returned ");
    send_command(3,0x55);
    while(~interrupt_received);
    printf("%02x",get_response());
    interrupt_received = 0;

    printf("\nSet PWM      04 01 returned ");
    send_command(4,1);
    while(~interrupt_received);
    printf("%02x",get_response());
    interrupt_received = 0;

    /* restore the old interrupt vector */
    _dos_setvect(0x12, old_int);

}
```

Interfacing the 83C576/87C576 to the ISA bus

AN454

```
/*
    Code to demonstrate the use of the UPI
    on the '576.

    Written in Franklin C for the '576.
*/

#include <reg51.h>

// Constants

#define UC    unsigned char
#define UI    unsigned int

// Special Function Registers

sbit    IRQ            = P2^3;

sfr     UCS            = 0x86;

sbit    UE            = UCS^3;
sbit    AF            = UCS^2;
sbit    IBF           = UCS^1;
sbit    OBE           = UCS^0;

sfr     ADC            = 0xB1;
sbit    ADF           = ADC^7;
sbit    ADC6          = ADC^6;
sbit    AD8M          = ADC^5;
sbit    AMOD1         = ADC^4;
sbit    AMOD0         = ADC^3;
sbit    ASCA2         = ADC^2;
sbit    ASCA1         = ADC^1;
sbit    ASCA0         = ADC^0;
sfr     ADC0H         = 0xAA;
sfr     ADC1H         = 0xAB;
sfr     ADC2H         = 0xAC;
sfr     ADC3H         = 0xAD;
sfr     ADC4H         = 0xAE;
sfr     ADC5H         = 0xAF;
sfr     PWM0          = 0xBE;
sfr     PWM1          = 0xBF;
sfr     PWMP          = 0xBD;
sfr     PWCON         = 0xBC;
```

Interfacing the 83C576/87C576 to the ISA bus

AN454

```
/* Module get_analog_channel
```

This module initializes the ADC for mode 0 and 8 bit conversions. This module has 1 parameter and 1 return value. The channel to be converted is passed as an unsigned char parameter and converted to 3 single bit values. The converted values are set in the ADC control register. The conversion is started by setting the enable bit (ADCE). The conversion is monitored for completion by polling the ADF bit. The result from the selected channel is returned to the calling function using the switch() statement */

```
UC
```

```
get_analog_channel(  
    UC channel ) {
```

```
    AMOD0 = 0;        // Mode 0  
    AMOD1 = 0;  
    AD8M = 1;        // 8 bit mode  
    ASCA2 = channel^2;  
    ASCA1 = channel^1;  
    ASCA0 = channel^0;  
    ADCE = 1;        // start conversion
```

```
/* poll the ADF flag. This loop breaks when ADF = 1 */  
    while(~ADF);    // wait for conversion to complete
```

```
/* return the ADC value based on the selected channel */
```

```
    switch (channel) {  
        case 0:  
            return(ADC0H);  
            break;  
        case 1:  
            return(ADC1H);  
            break;  
        case 2:  
            return(ADC2H);  
            break;  
        case 3:  
            return(ADC3H);  
            break;  
        case 4:  
            return(ADC4H);  
            break;  
        case 5:  
            return(ADC5H);  
            break;  
        default:  
            return(0xFF);  
            break;  
    }
```

```
}
```

Interfacing the 83C576/87C576 to the ISA bus

AN454

```
/* Module get_digital_byte()
```

This module has 1 parameter and 1 return value. The parameter is included for consistency with the other functions and has no meaning in this case. The return value is the combined result of reading 6 bits from P3 and 2 bits from P2. */

```
UC
get_digital_byte (UC dummy) {

// declare a temporary variable
    UC byte;

// read P3 and strip the 2 M.S. bits
    byte = P3 && 0x3f;

// read P2, strip 6 M.S. bits, shift result 6 places left and OR with the temporary variable
    byte |= ((P2 && 0x03) << 6);

// return the result
    return (byte);
}
```

```
/* Module put_digital_byte()
```

This module take the byte, passed as a parameter, masks the unused bits of the ports and ORs the masked result with the current port values. */

```
UC
put_digital_byte (
    UC byte ) {

// clear the current port value
    P3 &= 0xc0;
// AND the parameter with a mask and OR the result with the cleared port value
    P3 |= (byte && 0x3f);
    P2 &= 0x3f;
    P2 |= ((byte && 0xc0) >> 6);
    return(byte);
}
```

```
/* Module increase_pwm()
```

This module increments the current PWM value of the selected channel. The channel is passed to this function as a parameter */

```
UC
increase_pwm (
    UC channel ) {

// test the channel parameter
    if (channel == 0)
// increase PWM0
        PWM0++;
    else
// increase PWM1
        PWM1++;

    return(channel);
}
```

Interfacing the 83C576/87C576 to the ISA bus

AN454

```
void
main () {
// declare temporary variables for the command and information bytes
    UC command;
    UC information;

// initialize the PWM controller
    PWMP = 0x80;
    PWCON = 0xFF;

// do forever
    while(1) {

// wait for a command from the PC ie, when the IBF flag is set
        while (~IBF);           // wait for the command

// read the command byte. This read clears the IBF automatically
        command = P0;

// wait for the next byte from the PC
        while (~IBF);           // wait for the information
        information = P0;

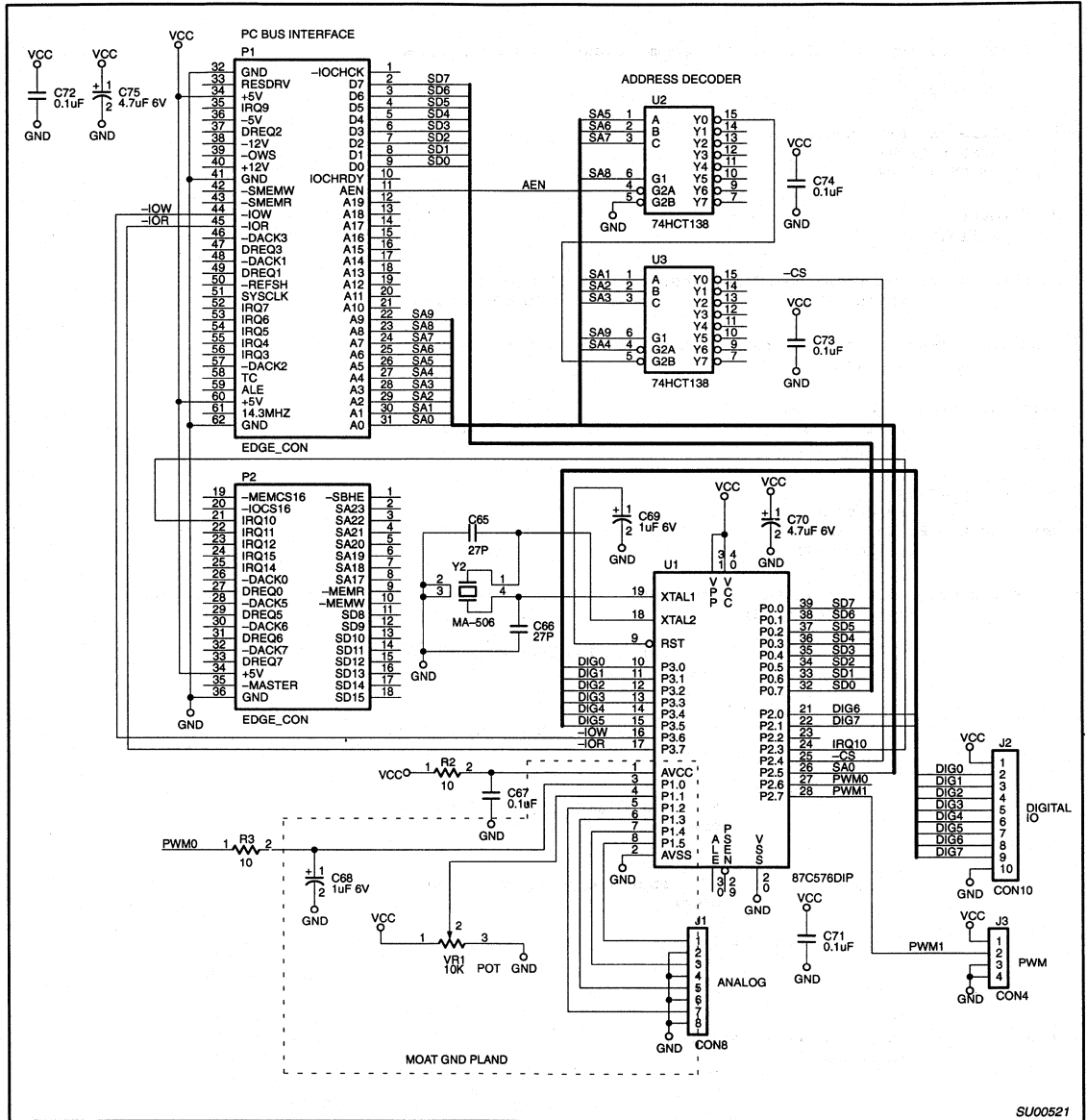
// echo the command
        P0 = command;
// wait for the PC to read it
        while (OBE);

// parse the command and call the appropriate function, passing the information as a parameter
        switch (command) {
            case 1 : {
                P0 = get_analog_channel(information);
                break;
            }
            case 2: {
                P0 = get_digital_byte(information);
                break;
            }
            case 3: {
                P0 = put_digital_byte(information);
                break;
            }
            case 4: {
                P0 = increase_pwm(information);
                break;
            }
            default: {
                P0 = 0xFF;
                break;
            }
        }

// wait for the PC to read the information.
        while (OBE);
    }
}
```

Interfacing the 83C576/87C576 to the ISA bus

AN454



SU00521

Section 8

Other 80C51 Application Notes & Articles

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

AN408	80C451 operation of port 6	577
AN417	256k Centronics printer buffer using the 87C451 microcontroller	588
AN418	Counter/timer 2 of the 83C552 microcontroller	601
AN420	Using up to 5 external interrupts on 80C51 family microcontrollers	608
AN424	8051 family warm boot determinations	610
AN440	RAM loader program for 80C51 family applications	612
AN443	IEEE Micro Mouse using the 87C751 microcontroller	621
AN447	Automatic baud rate detection for the 80C51	642
AN448	Determining baud rates for 8051 UARTs and other UART issues	645
ESG89001	Electro magnetic compatibility and printed circuit board (PCB) constraints ..	648
EIE/AN91001	Workbench EMC evaluation method	667
EIE/AN91006	A/D conversion with P83CL410 PCF1252-x	684
EIE/AN91009	Driver for 8xC851 E2PROM	700
EIE/AN92001	Low RF-emission applications with a P83CE654 microcontroller	715
EIE/AN93017	Using the analog-to-digital converter of the 8XC552 microcontroller	727
	Chips push CAN bus into embedded world	745
	Add Text Overlay to Any Video Display	747

80C451 operation of port 6

AN408

INTRODUCTION

The features of the 80C451 are shared with the 80C51 or are conventional except for the operation of port 6. The flexibility of this port facilitates high-speed parallel data communications. This application note discusses the use of port 6 and is divided into the following sections:

1. Port 6 as a processor bus interface.
2. Using port 6 as a standard pseudo bidirectional I/O port.
3. Implementation of parallel printer ports.

This information applies to all versions of the part: 80C451, 83C451, and the 87C451.

PORT 6 AS A PROCESSOR BUS INTERFACE

Port 6 allows use of the 80C451 as an element on a microprocessor type bus. The host processor could be a general purpose MPU or the data bus of a microcontroller like the 80C451 itself. This feature allows single or multiple 80C451 controllers to be used on a bus as flexible peripheral processing elements. Applications could include keyboard scanners, serial I/O controllers, servo controllers, etc.

OPERATION

On reset, port 6 is programmed correctly for use as a bus interface (see 2). This prevents the interface from disrupting data on the bus of the host processor during power-up. Software initialization of the CSR (Control Status Register) is not required. A dummy read of port 6 may be required to clear the IBF (Input Buffer Full) flag since it could be set by turn on transients on the bus of the host processor. On reset, the CSR of the 83C451 is programmed to allow the following:

1. AFLAG is an input controlling the port select function. If AFLAG is high, the contents of the CSR is output on port 6 when the port is read by the host. If AFLAG is low, then the contents of the output latch is output when port 6 is read by the host.
2. BFLAG is an input controlling the port enable function. In this mode when BFLAG is high, the input latch and the output drivers are disabled and the flags are not affected by the IDS (Input Data Strobe) or ODS (Output Data Strobe) signals. When BFLAG is low, the port is enabled for reading and writing under the control of IDS and ODS pins.

Figure 1 shows one possible example of an 80C451 on a memory bus. This arrangement allows the main processor to query port 6 for flag status without interrupting the 80C451. If the address decoder, shown in Figure 1, enables port 6 on the 80C451 when the address is 8000H or 8001H, and the address line A0 controls the port select feature, then the host processor can read and write to port 6 using address 8000H. Since the port select function is being controlled by the address line A0, the CSR contents can be read by the host processor at address 8001H.

By testing the CSR contents in this way, the host processor can tell if new data has been written to the port 6 output latch since it last read the port or if the 80C451 has read the last byte that the host wrote to the port. Conversely, the 80C451 can poll the flags in its CSR to see if the host processor has written to or read from port 6 since the last time it serviced the port.

If desired, an interrupt source for the 80C451 can be derived easily from the port enable source as shown by the dashed line in Figure 1.

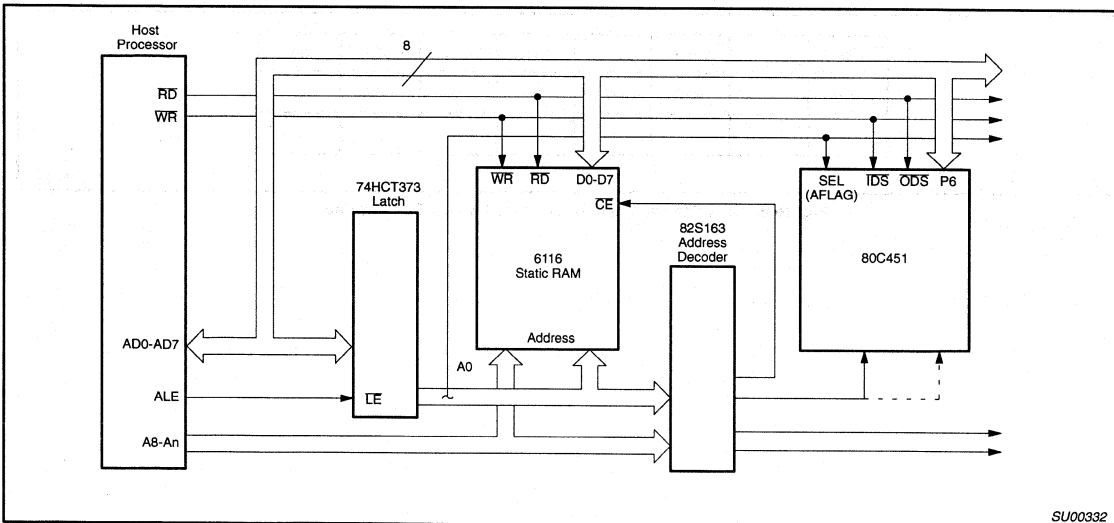


Figure 1. An 83C451 on a Microprocessor Memory Bus

80C451 operation of port 6

AN408

SOFTWARE EXAMPLES

To write to port 6 on the bus shown in Figure 1, the host processor first reads the CSR contents at address 8001H, and tests

the input buffer full flag (CSR bit 0). If the flag is clear, the host writes a byte to address 8000H. This loads the input buffer latch of port 6 and sets the input buffer full flag.

Conversely, the 80C451 polls the IBF flag and reads a byte from port 6 when it finds the flag set. The flag is automatically reset when this internal read occurs.

80C451 ROUTINE TO READ ONE BYTE FROM HOST VIA PORT 6

```

RCVR:      JNB CSR.0,RCVR      ;TEST IBF FLAG
           MOV A,P6           ;WHEN FLAG IS SET READ BYTE
           RET
    
```

80C451 ROUTINE TO WRITE ONE BYTE TO THE 83C451 PORT 6

If the host processor is an 80C51, the following routine will write a byte of data to the 80C451. The data involved is passed to the routine through register 1.

```

XMIT:      MOV DPTR,8001H
TEST:      MOVX A,@DPTR       ;READ THE CSR
           JB ACC.0,TEST      ;TEST IBF FLAG
           MOV DPTR,8000H
           MOV A,R1
           MOVX @DPTR,A      ;WRITE DATA TO THE 451
           RET
    
```

80C451 ROUTINE TO WRITE ONE BYTE TO HOST VIA PORT 6

Routines for data transfer in the opposite direction are similar to the above two. The 80C451 version is given below.

```

XMIT:      JB CSR.1,XMIT      ;TEST OBF FLAG
           MOV P6,A          ;WRITE DATA
           RET
    
```

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBF	IDSM	OBF	IBF
1	1	1	1	1	1		

SU00333

Figure 2. CSR Programmed to Allow Port 6 as a Bus Interface

80C451 operation of port 6

AN408

USING PORT 6 AS A STANDARD QUASI-BIDIRECTIONAL I/O PORT

To use port 6 as a common I/O port, all of the control pins are tied to ground (see Figure 3). On hardware reset, bits 2 - 7 in the CSR are set to one. Port operation and electrical characteristics become identical to port 1 on the 80C51 and the 80C451 ports 1, 4, and 5. No software initialization is required.

If desired, AFLAG and BFLAG can be used as outputs while port 6 is operating as a standard quasi-bidirectional I/O port (see Figure 4). In this case, only IDS and ODS are tied to ground and the CSR is initialized to allow operation of AFLAG and BFLAG as simple outputs (see Figure 5).

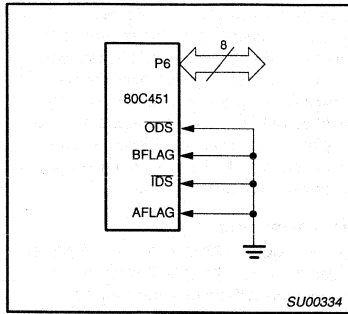


Figure 3. Standard I/O Port on Reset

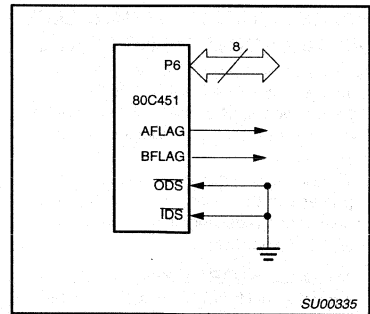


Figure 4. Standard I/O Port on Reset with AFLAG and BFLAG as Outputs

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
1	X	0	X	X	1		

SU00336

Figure 5. CSR Programmed to Allow AFLAG and BFLAG to Operate as Outputs and Port 6 as a Standard I/O Port

DATA TRANSFER SIGNAL PINS		
Pin No.	Ground Return Pin No.	Signal
1	19	STROBE
2	20	DATA 1
3	21	DATA 2
4	22	DATA 3
5	23	DATA 4
6	24	DATA 5
7	25	DATA 6
8	26	DATA 7
9	27	DATA 8
10	28	ACKNLG
11	29	BUSY

TYPICAL AUXILIARY PIN FUNCTIONS	
Pin No.	Signal
12	PAPER OUT
14	AUTO LINE FEED
16	LOGIC GROUND
17	CHASSIS GND
30	GROUND RETURN
31	RESET PRINTER
32	ERROR
33	GROUND RETURN
36	SLCT IN

SU00337

Figure 6. Parallel Printer Interface Pin Functions

80C451 operation of port 6

AN408

IMPLEMENTATION OF PARALLEL PRINTER PORTS USING PORT 6

The 80C451 is an excellent choice for a printer controller. The 80C451 has the facilities to permit all of the intelligent features of a common printer to be handled by a single chip:

1. The features of port 6 allow a parallel printer port to be designed with only line driving and receiving chips required as additional hardware.
2. The onboard UART allows RS232 interfacing with only level shifting chips added.
3. The 8-bit parallel ports 0 to 6 are ample to drive onboard control functions, even when ports are used for external memory access, interrupts, and other functions.
4. The RAM addressing ability of ports 0 and 2 can be used to address up to 64k bytes of a hardware buffer/spooler. AFLAG and BFLAG as simple outputs (see Figure 5).
5. The 64k byte ROM addressing capability allows space for the most sophisticated software.

In addition, either end of a parallel interface can be implemented using port 6, and the interfaces can be interrupt driven or polled in either case.

THE INTERFACE

Data transfer on a parallel printer interface occurs across eleven signal lines. The other conductors on the standard plug are used as ground returns or for auxiliary functions (see Figure 6). Only the data transfer signals will be considered.

The Data Transfer Format

The parallel printer interfaces are far more standardized in features than their serial

counterpart. However, at least three significant variations exist in handshake style in printers using generic parallel interfaces. This fact influences the design of both port hardware and software. A good transmitter should be able to drive devices with all three styles of handshakes, and a good receiver should generate the handshake most likely compatible with any transmitter.

The Variations

Type 1—Figure 7 shows a common style of handshake and is the style that will be implemented in the receiver examples. A busy signal and an acknowledge strobe pulse are generated for every byte received.

Type 2—Another style of handshake generates a busy signal only when the printer will not be able to accept more data for a relatively long time. Acknowledge pulses are created after every byte received. When the busy signal is generated after a byte is received, the associated acknowledge pulse does not occur until *after* the busy signal returns to logic zero (see Figure 7).

Type 3—A third handshake style does not generate acknowledge pulses, but a busy signal is produced after every byte is received.

PARALLEL PRINTER INTERFACES USING POLLING

Transmitter Operation

This application illustrates the flexibility of the port 6 logic in solving an applications problem. We need to be able to handle all types of acknowledge signals that might be received by the transmitter. We will use the \overline{ODS} pin and output buffer full flag logic to record the receipt of the acknowledge pulse (see Figure 8), but not all parallel receivers generate acknowledge pulses. We could poll

the busy signal line, but not all receivers generate busy signals for each byte received; so lack of a busy signal does not imply that we can send another byte. We can, however, expect an acknowledge pulse very shortly after the end of a busy signal if one is going to arrive at all. So we can send a new data byte after having received either a positive transition on the acknowledge line, or shortly after receiving a negative edge on the busy line.

The CSR is programmed to the output only mode. In this mode, the \overline{ODS} pin does not control the output drivers but only the output buffer full flag. The flag serves to record the positive transition of the acknowledge signal. The input latch is not used, but the \overline{IDS} pin is used to set the input buffer full flag. This is used to record the negative transition at the end of the busy signal. Dummy reads by the 80C451 of port 6 will be used to clear the flag. In this example, the AFLAG mode is set only to place the port in the output only mode. The AFLAG pin is not actually used (see Figure 10).

The transmitter's CSR (control status register) is programmed to the following mode (see Figure 9):

1. CSR bit 6 controls the BFLAG output and therefore the strobe line.
2. The OBF (output buffer full) flag controls the AFLAG output.
3. The OBF is cleared on the positive edge of the \overline{ODS} input.
4. The IBF flag is cleared on the negative edge of the \overline{IDS} strobe.

NOTE:

With this combination of modes set, port 6 is in the output only mode.

80C451 operation of port 6

AN408

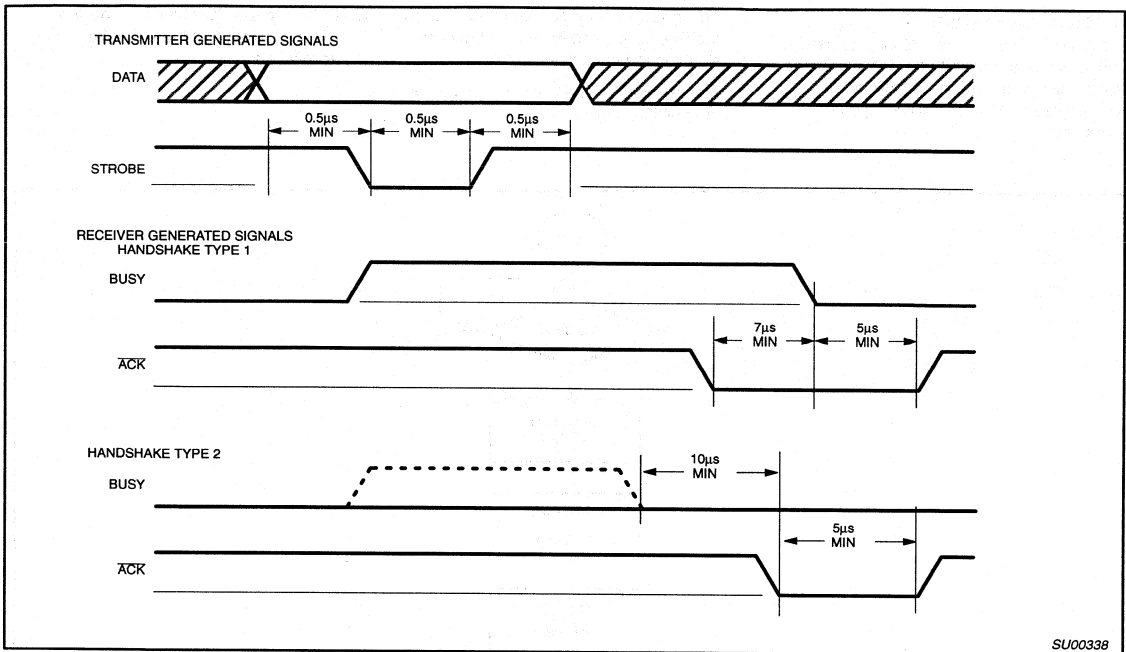


Figure 7. Parallel Printer Interface Signals

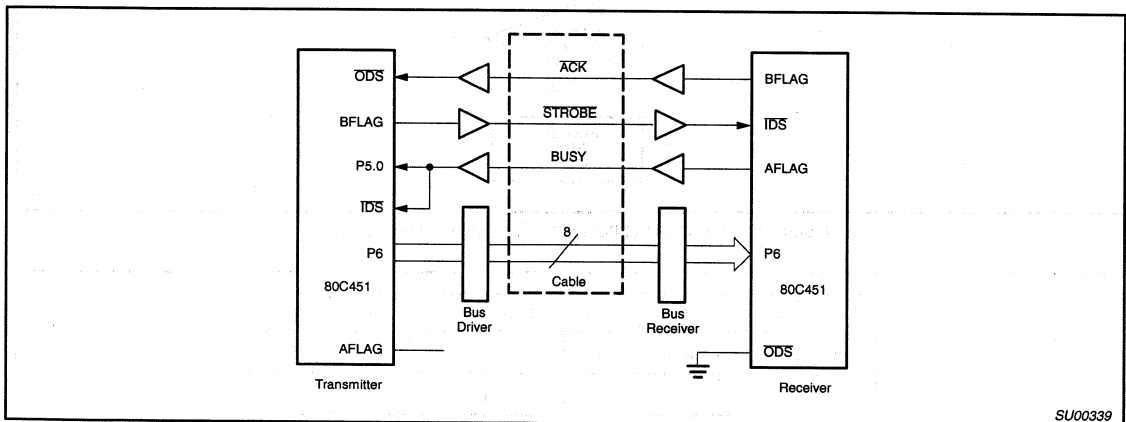


Figure 8. Interconnection for a Parallel Interface Using Polling

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
0	1	1	0	0	1		

SU00340

Figure 9. CSR Programmed for Polled Transmitter Operation

80C451 operation of port 6

AN408

Receiver Operation

In receiver operation, the IDS input is used to latch in the data transmitted on receipt of the strobe pulse. The receiver's CSR is programmed to allow the following (see Figure 11):

1. The input buffer full flag is output through the BFLAG pin and is used as the busy signal to the transmitter.
2. The IBF flag is set and data is latched on the positive edge of IDS.
3. Writing to the CSR bit 4 controls the AFLAG output and therefore the acknowledge line.

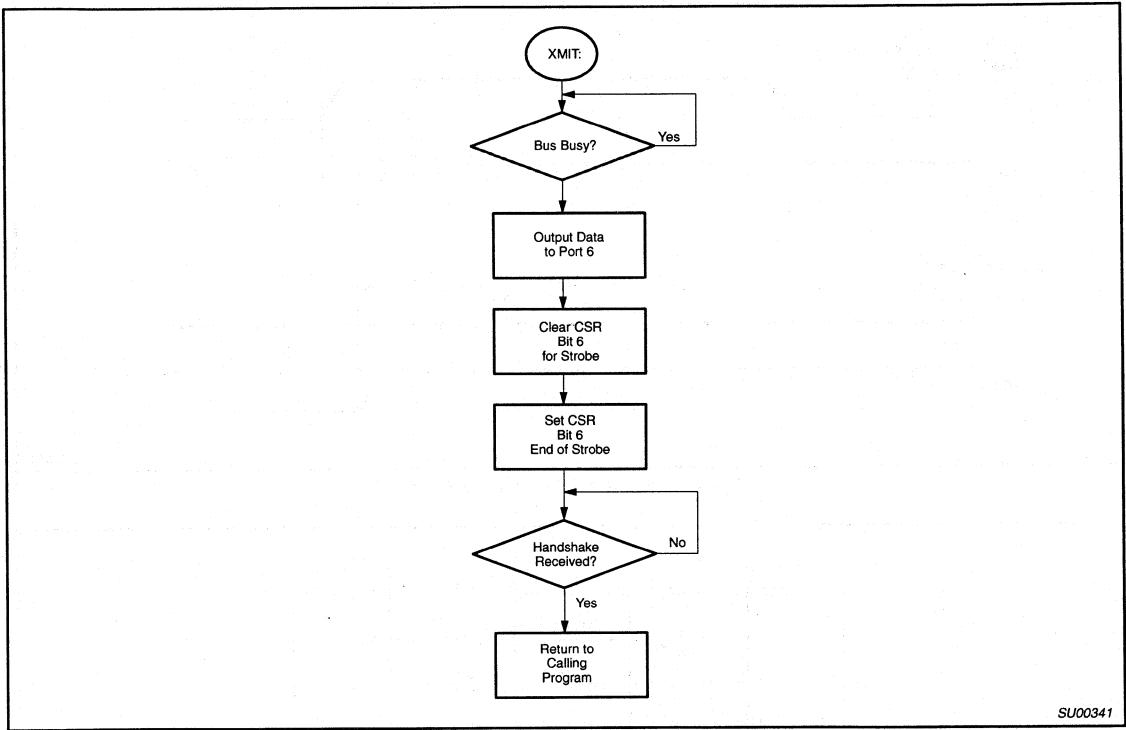


Figure 10. Flow Chart of Polled Parallel Transmitter Operation

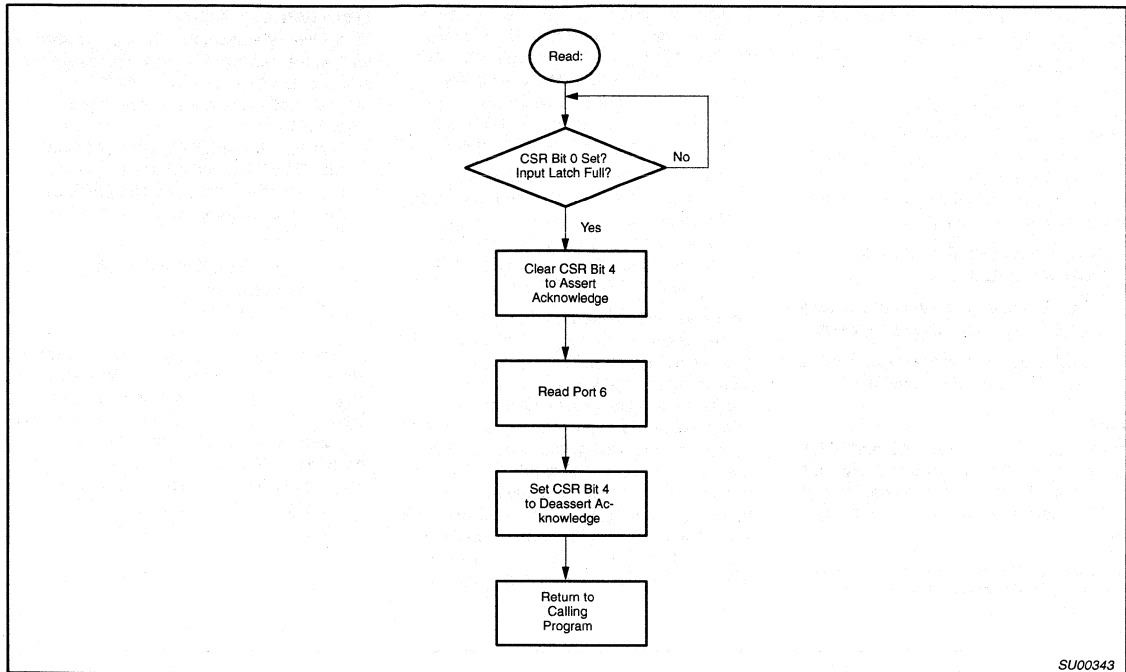
CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
1	0	0	1	1	0		

SU00342

Figure 11. CSR Programmed for Polled Parallel Receiver Operation

80C451 operation of port 6

AN408



SU00343

Figure 12. Flow Chart of Polled Parallel Receiver Operation

SOFTWARE EXAMPLES

This polled parallel transmit routine outputs one byte passed to it in the accumulator.

```

P_INIT:    MOV CSR,#064H    ;INITIALIZE PORT 6 OPERATING MODE
P_OUT:    JB P5.0          ;WAIT IF BUSY SIGNAL IS HIGH
           MOV P6,ACC      ;OUTPUT DATA
           MOV R1,P6       ;DUMMY READ TO CLEAR IBF FLAG
           MOV R1,#02H     ;INITIALIZE DELAY COUNTER
           CLR CSR.6       ;START STROBE PULSE
           DJNZ R1,$       ;TIME 6 MICROSECOND STROBE PULSE
           SETB CSR.6      ;END STROBE PULSE
WAIT:     JNB CSR.1,OUT    ;EXIT IF ACKNOWLEDGE RCV'D
           JNB CSR.0,WAIT  ;EXIT IF NEGATIVE BUSY EDGE RCV'D
  
```

This polled parallel receive routine places one byte in the accumulator each time it is called.

```

P_INIT:    MOV CSR,#09CH    ;INITIALIZE PORT 6 OPERATING MODE
           MOV R7,P6       ;DUMMY READ TO CLEAR IBF FLAG
P_IN:     JNB CSR.0        ;INPUT BUFFER LATCH FULL?
           CLR CSR.4       ;BEGIN ACKNOWLEDGE PULSE
           MOV R7,#02H     ;INITIALIZE DELAY COUNTER
           DJNZ R7,$       ;TIME ACKNOWLEDGE PULSE
           MOV A,P6        ;READ BYTE - CLEAR BUSY SIGNAL
           MOV R7,#02H     ;INITIALIZE DELAY COUNTER
           DJNZ R7,$       ;TIME ACKNOWLEDGE PULSE
           SETB CSR.4      ;END ACKNOWLEDGE PULSE
           RET
  
```

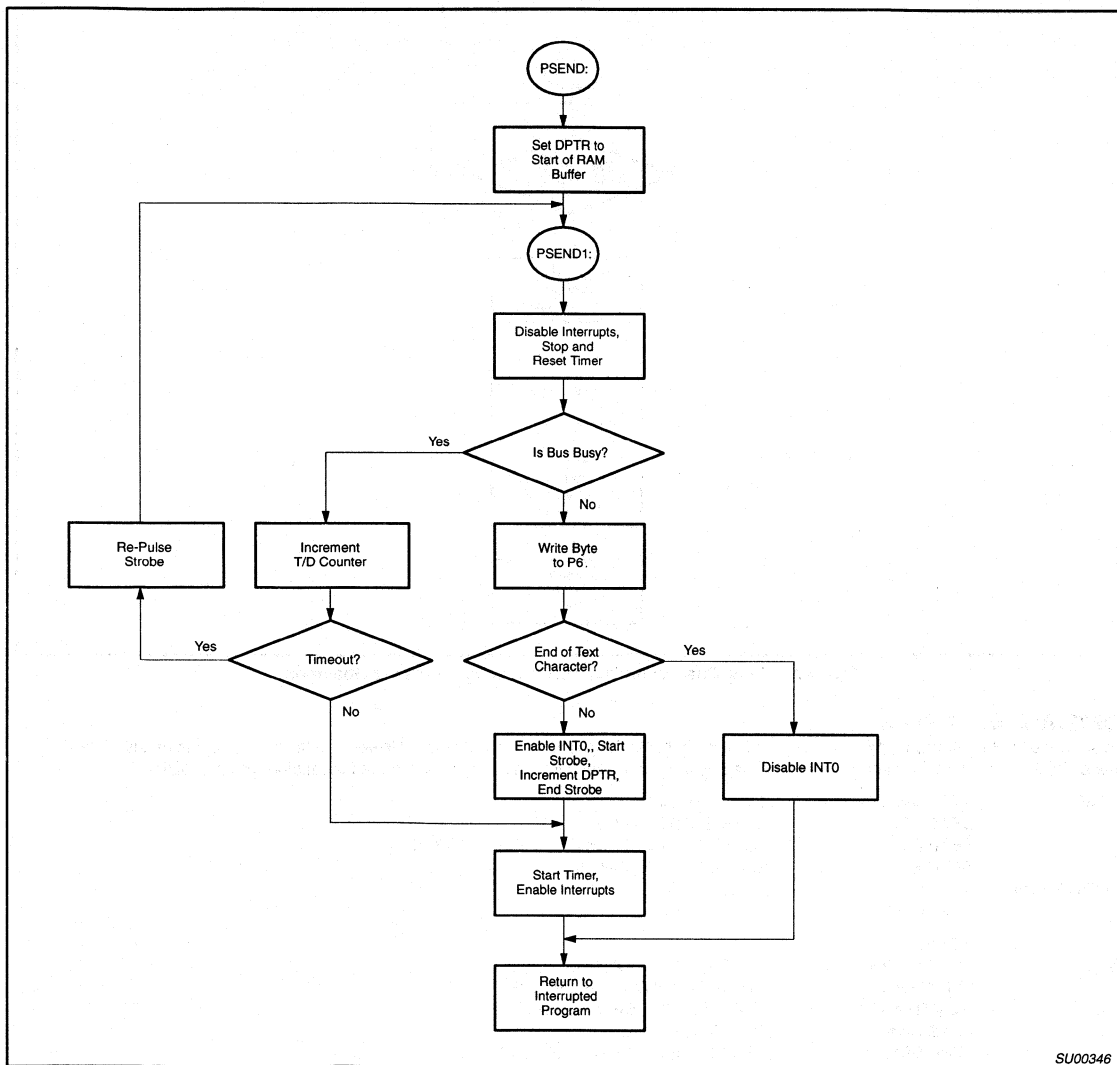

80C451 operation of port 6

AN408

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
0	1	1	0	1	1		

SU00345

Figure 14: CSR Programmed for Use as an Interrupt Driven Parallel Transmitter



SU00346

Figure 15. Flow Chart for an Interrupt Driven Parallel Transmitter

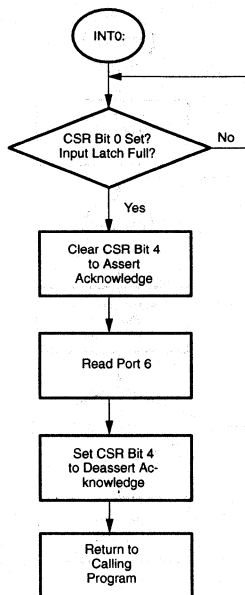
80C451 operation of port 6

AN408

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
1	0	0	1	1	0		

SU00347

Figure 16. CSR Programmed for Use as an Interrupt Driven Parallel Receiver



SU00348

Figure 17. Flow Chart of Interrupt Driven Parallel Receiver Operation

SOFTWARE EXAMPLES

The software for the interrupt driven parallel receiver is similar to the polled receiver example. However, after an interrupt is received, this routine checks to confirm that data has been latched by the positive edge of the strobe pulse before proceeding with the routine.

```

INIT:      MOV CSR,#090H      ;INITIALIZE CSR
           SETB EX0          ;ENABLE INTERRUPT 0
           SETB IT0          ;SET NEG EDGE TRIGGERED INTERRUPTS
           SETB EA           ;ENABLE ALL INTERRUPTS
           ORG EXTIO         ;INTERRUPT 0 VECTOR
           JMP RCVR

RCVR:      JNB CSR.0,#       ;CONFIRM DATA LATCHED
           CLR CSR.4         ;START ACKNOWLEDGE PULSE
           MOV R7,#02H       ;INITIALIZE THE DELAY COUNTER
           DJNZ R7,#         ;TIME ACK PULSE
           MOV A,P6          ;READ BYTE - RESET BUSY LINE
           MOV R7,#02H       ;INITIALIZE THE DELAY COUNTER
           DJNZ R7,$         ;TIME ACK PULSE
           SETB CSR.4        ;END ACK PULSE
           RET1
  
```

80C451 operation of port 6

AN408

This is the software for the interrupt driven parallel transmitter example.

; XMIT ROUTINE DRIVEN BY ACK PULSE GENERATED INTERRUPTS, OR TIME GENERATED INTERRUPTS
 ; FOR NON ACKNOWLEDGING PRINTERS. READS DATA BUFFER IN EXTERNAL RAM STARTING AT 100H
 ; AND READING UNTIL 04H IS FOUND.

```

ORG RESET
JMP 26H
ORG TIMER0
JMP PSEND1
ORG EXTIO
JMP PSEND1
ORG 26H
    MOV CSR,#064H      ;PORT 6 MODE
    MOV TMOD,#002H    ;CONFIGURE TIMER 0 TO 16 BITS
    SETB T00          ;INT0 IS EDGE TRIGGERED
    SETB EA           ;ENABLE INTERRUPTS
PSEND:  MOV DPTR,#0100H ;SET DPTR TO START OF TEXT
        ;BUFFER
PSEND1: CLR EA        ;DISABLE INTERRUPTS AND STOP
        ;TIMER
        CLR TR0       ;IF ENABLED
        CLR ET0
        MOV R7,00H    ;CLEAR TIMEOUT COUNTER
        MOV R6,00H
        MOV TH0,#-4   ;SET TIMER INTERRUPT PERIOD
        MOV TL0,#00H
        JB 0C8H,BB    ;BUS BUSY
        MOV ACC,#00H  ;CLEAR ACCUMULATOR
        MOVX 1,@DPTR  ;RETRIEVE FIRST BYTE
        MOV 06,ACC    ;OUTPUT FIRST BYTE
        CJNE A,#004H,CONT1 ;LOOK FOR END OF TEXT
        JMP EOTB
CONT1:  SETB ERX0      ;ENABLE INTO
        CKR 0EEH      ;START STROBE PULSE
        INC DPTR
        MOV ACC,DPH   ;LOOK FOR PHYSICAL END OF
        JB ACC.2,EOTB ;TEXT BUFFER
        SETB 0EEH
        JMP CONT
EOTB:   CLR EX0       ;END OF TEXT FOUND, DISABLE
        ;INT0
        SETB 0EEH
        SETB EA
        RETI
BB:     INC R7         ;COUNT TIMER TIMEOUTS ON
        ;BUS BUSY
        CJNE R7,#00H,CONT ;LOOK FOR OVERFLOW
        INC R6        ;COUNT OVERFLOWS
        CJNE R6,#10H,CONT ;TIMEOUT APPROX 5 SEC
        JMP TO
CONT:   SETB TR0      ;ENABLE TIMER INTERRUPT
        SETB ET0      ;START TIMER
        SETB EA
        RETI
TO:     CLR 0C9H      ;SEND NEW STROBE PULSE IN
        ;RESPONSE TO TIMEOUT
        NOP
        NOP
        MOV R6,#00H   ;RESET TO COUNTER
        MOV R7,#00H
        SETB 0C9H     ;END OF STROBE PULSE
        JMP PSEND1

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

DESCRIPTION

This application note describes a stand alone Centronics type parallel printer buffer using the 87C451 expanded I/O microcontroller. This type of unit would typically be placed between a personal computer and its printer. It captures the data to be printed at high speed, freeing the personal computer to go to other tasks, and sends data to the printer as required. As described here, 256k dynamic RAMs are used, providing over one quarter million characters of storage. If desired the design is easily modified to work with 1 megabit DRAMs. Although written with the 87C451 in mind, this design is applicable to the 80C451 and 83C451.

Design Objectives

The objectives kept in mind during the design of this device were: provide a substantial size of buffer, keep the parts count and the power consumption to a minimum, and use readily available components.

A buffer size of 256k bytes was chosen because, although a 64k byte buffer is very easily implemented using the 8051 family's 64k external data storage capabilities, it is a little too small for today's printing applications that print a page of text in graphics mode, using up twenty times as many bytes as standard printing mode. Presenting a method for controlling 256k DRAMs shows off the I/O capabilities of the 87C451, and it is very easy to add the extra address line for one megabit devices if a larger buffer is needed.

The 8XC451 Microcontroller

The 8XC451 is an 8-bit microcontroller based on the familiar 8051 family of devices. In fact, it is an 80C51 with three added ports: P4, P5, and P6. Ports 4 and 5 give 12 (16 in PLCC) additional quasi-bidirectional I/O lines. Port 6 provides another 8 bits of I/O, plus 4 handshake lines that can be programmed to operate in several useful modes for interfacing. The 8XC451 comes in three versions: ROMless 80C451, 83C451 with 4k x 8 ROM, and 87C451 with 4k x 8 EPROM.

In this note, port 6 is used in the I/O mode as a Centronics compatible printer output port. Additionally, the /IDS and BFLAG pins normally associated with port 6 are used as part of the input port logic. For a complete discussion of port 6 operating modes and programming, see the application note AN408 titled "83C451 Microcontroller Operation of Port 6."

Circuit Description

Figure 1 is a schematic diagram of the printer buffer circuit. Other than the 87C451 (U1), and the eight 256k DRAMs (U5-U12), only

two 74LS244 buffers (U2, U3) and a 76HCT374 (U4) octal flip-flop are needed. The U2 and U3 buffers are included to provide full drive capability for the output port and some of the handshake signals on the input port, as the output buffers on the 87C451 can only drive 3 LSTTL loads. U4 has 8-bit data strobed into it by the /STB pulse of the input port.

As the code size for this application is quite small (less than 1k bytes), the on-chip instruction memory is quite sufficient for program storage. For a production version, the 87C451 could be replaced with the 83C451 with a 4k x 8 masked ROM on chip. Note that port 0 and port 1 are not used in the present design; thus the 80C451 may be used in this application with the addition of an external address latch and EPROM.

The /RAS, /CAS, and /WR signals for the DRAM array are provided by port 3 bits /WR, /RD, and T1. Note that as in the 80C51, all port 3 signals are multifunctional. That is, each can be treated as a regular quasi-bidirectional port bit, or as having the special function indicated by its name. This feature is an advantage when using /WR and /RD as /RAS and /CAS control signals for a DRAM array. Treated as a normal port bit, the /WR pin is cleared and set by individual CLR and SETB instructions for a normal length RAM read or write cycle. However, when performing a refresh cycle, /RAS (port 3/WR) can be pulsed low using a dummy MOVX @R0,A (move to external data memory) instruction. This allows DRAM refresh to be done much more quickly than would otherwise be possible.

Port 1 and one bit from port 4 form the 9-bit address required when addressing the DRAM array. The data inputs to the array come from the parallel input data lines which are latched by U4. The RAM data outputs are fed to port 5. By making the data outputs available to the processor, it is possible to add some additional features to the firmware, such as control codes for printing multiple copies of a document, data compression, data conversion, etc. which are not implemented in this design.

Port 6 Operation

The /IDS (input data strobe) and BFLAG pins are normally used in conjunction with the port 6 bidirectional mode. In this mode, the /IDS pin is used to strobe data into the port 6 input latches, and BFLAG is used as flag output. In this application, however, these two bits are used to good effect as part of the (separate) input port logic. When a byte of data is strobed into U4 by the printer port of the host computer, the /STB signal connected to /IDS

sets the input buffer full flag (IBF). BFLAG is programmed to mirror the contents of IBF, and therefore becomes asserted. This makes it ideal to be used as the BUSY output for the input port. After the input port data has been read and stored in the RAM buffer, BFLAG is de-asserted by performing a dummy read of port 6, which clears IBF. To complete the input port logic, one of the port 3 pins, P3.4, is used as the acknowledge signal, and is asserted/de-asserted by software. The /ODS pin is tied to ground to permanently enable the port 6 output drivers. This does not cause difficulty as no data is being input into the port.

Note that programming port 6 to operate in the bidirectional mode as described above means the loss of /ODS as an acknowledge input. The acknowledge input is normally used to clear the OBF (output buffer full) flag, indicating that the printer is ready for another character. On the other hand, operating port 6 in the "output only" mode causes the loss of BFLAG as BUSY output. Because the input port requires an instant BUSY indication while the output port only needs to remember the occurrence of an acknowledge pulse, it makes sense to program port 6 to operate in the bidirectional mode, with /ODS grounded to enable the output drivers. The /INT1 pin can be used instead of /ODS to record the occurrence of an acknowledge pulse with the interrupt system.

Priority and Execution of Tasks

There are three tasks that must be performed in this system: Receive—servicing the input port and storing the input character; Transmit—sending stored characters to the output port as required; and Refresh—performing DRAM refresh. The timers and interrupt system are used to manage the execution and priority of these tasks. Figure 2 and Figure 3 illustrate the flow charts of these tasks. Firmware, broken into sections, performing these three functions as well as an initialization routine is provided.

The 51C256 DRAMs require a 256 row refresh every 4 milliseconds. Rather than do an entire refresh cycle every 4 milliseconds, it is done as 64 rows every millisecond. This leaves time for other tasks to get service "slices" more frequently. As DRAM refresh is obviously the highest priority, timer 0 is used as the refresh interval timer, and is programmed to the 16-bit mode, and set to the higher priority level in the interrupt priority (IP) register. The refresh code is written in-line rather than in a loop to maximize speed.

An interesting point to note is that when there are no characters stored, the DRAM does not need to be refreshed. If power consumption

256k Centronics printer buffer using the 87C451 microcontroller

AN417

is of concern, the 87C451 could be programmed to go into idle mode whenever the buffer were empty. A character strobed into the input port would cause an interrupt, restarting the 87C451; DRAM refresh would be maintained until the buffer was once again empty.

The next highest priority should be input port service, as the reason for having a printer buffer is to get the data out of the computer as quickly as possible. Therefore, the input port /STB signal is connected to the /INT0 pin (as well as U4's clock pin and /IDS). Interrupt 0 is programmed in the interrupt priority register to be at the lower interrupt level so it cannot prevent refresh service. The interrupt 0 service routine stores the input character at the next location in the DRAM array, using the technique of a circular FIFO buffer. The routine also sends back an acknowledge pulse by clearing and setting the P3.4 pin, and then clears the BUSY (BFLAG) pin by performing a dummy read of port 6 (unless this character caused the buffer to be completely full).

During periods of access to the DRAM array by the input and output routines, the global interrupt enable bit (EA) is cleared so that the refresh interrupt does not disturb the contents of ports 1 and 4, or the /RAS, /CAS, and /WR signals.

The printer (output port) service routine runs all the time, except when the CPU is called to service the other conditions, therefore having the lowest priority. If there are characters in the buffer, polling is used to check for output port BUSY status. If the printer is not busy, then the character is sent, and the output port /STB pin (P4.3) is cleared and set. The output port /ACK line is connected to the /INT1 pin, so that the negative going edge of the /ACK signal is recorded as an interrupt pending. A very short INT1 service routine sets a software flag to indicate that the printer acknowledge the last character.

Possible Enhancements

There are a number of features that could be added to this design. As mentioned previously, the microcontroller could be put into the idle mode when the buffer is empty, conserving power.

The software could be enhanced to provide features such as multiple copies of a document, data compression, data conversion, automatic printer setup, etc. The PC operating system could be suitably modified to send a header for each file to be printed, containing these parameters. There is plenty of room for operating firmware expansion, and plenty of horsepower left in the 87C451 to handle these features.

The two serial port pins RxD and TxD were deliberately left unused so that input and/or

output ports are easily implemented for serial interfaces or printers using the built-in UART. The pins used for parallel port handshaking could then be used as serial handshaking lines, providing the standard "modem" signals.

Combining the above two features, this circuit could act as a "splitter." By connecting a daisy-wheel printer to the serial port, a dot-matrix printer to the parallel port, and sending an "address" flag in the file header, simultaneous letter-quality and draft printing could be done.

The size of the DRAM array is easily expanded to one megabyte or large devices by connecting the additional address pins to port 4 bits 1 and 2. Only slight modifications to the operating firmware would be required.

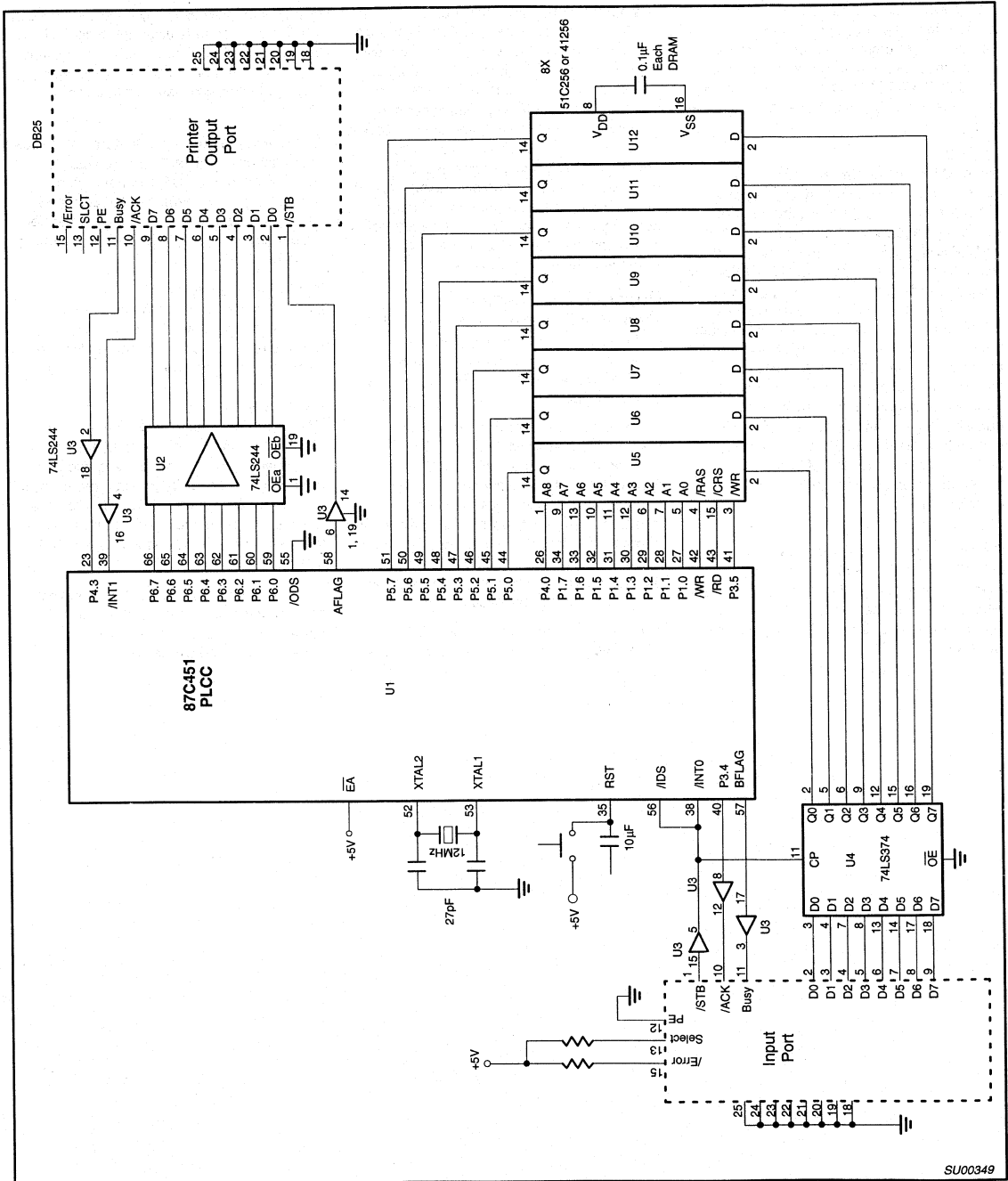
Conclusion

The SC8XC451 microcontrollers provide plenty of I/O pins that previously had to be implemented by clumsy I/O expansion methods. The flexibility of port 6 means that this device can be used in a wide variety of applications requiring special port functions, while still using the industry standard 8051 instruction set.

The Application Note, describing a typical parallel printer buffer, makes full use of the 8XC451 features, yet allows room for enhancement and expansion.

256k Centronics printer buffer using the 87C451 microcontroller

AN417



SU00349

Figure 1. Schematic Diagram

256k Centronics printer buffer using the 87C451 microcontroller

AN417

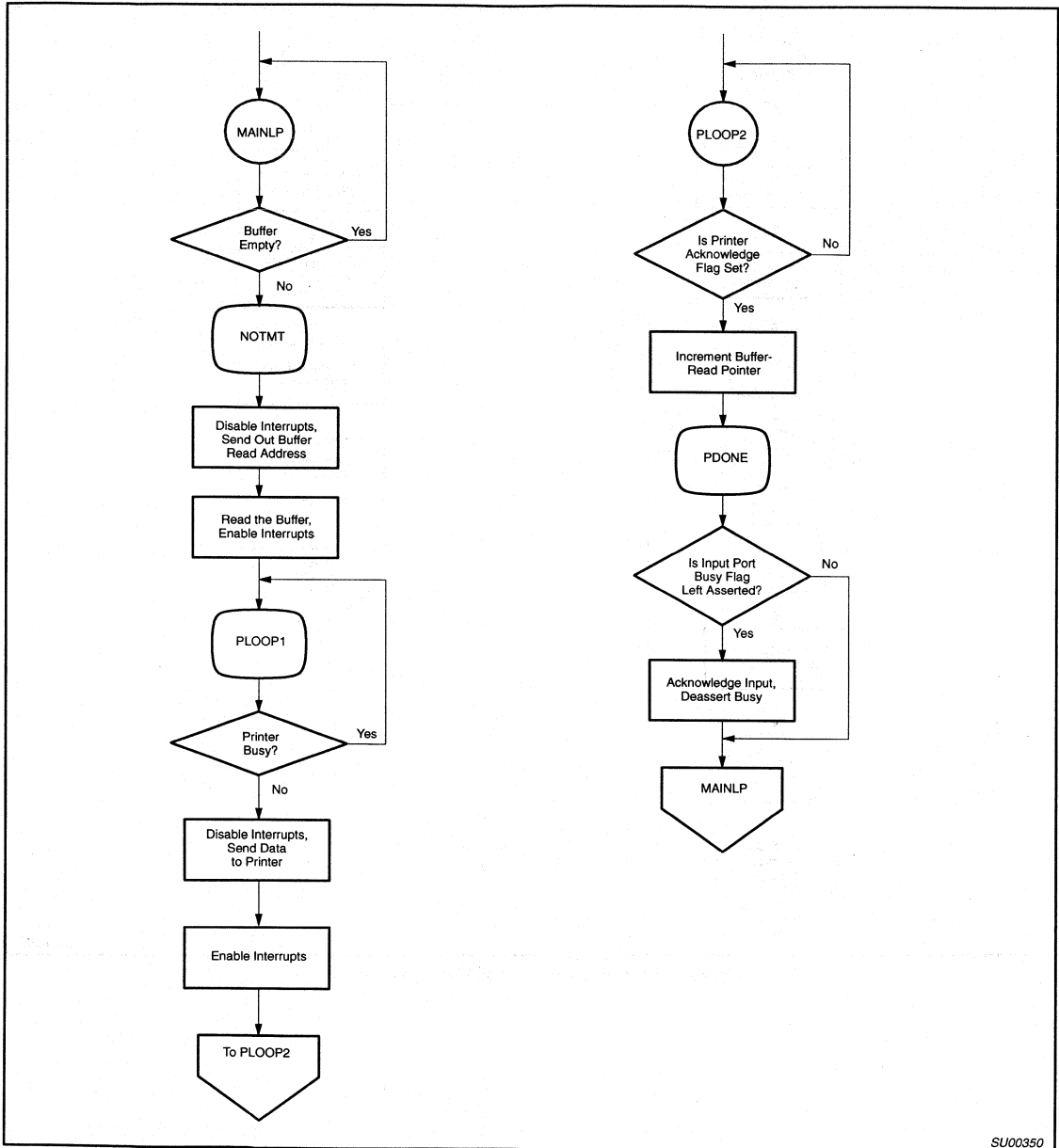


Figure 2. Flowchart of Transmit Operation

SU00350

256k Centronics printer buffer using the 87C451 microcontroller

AN417

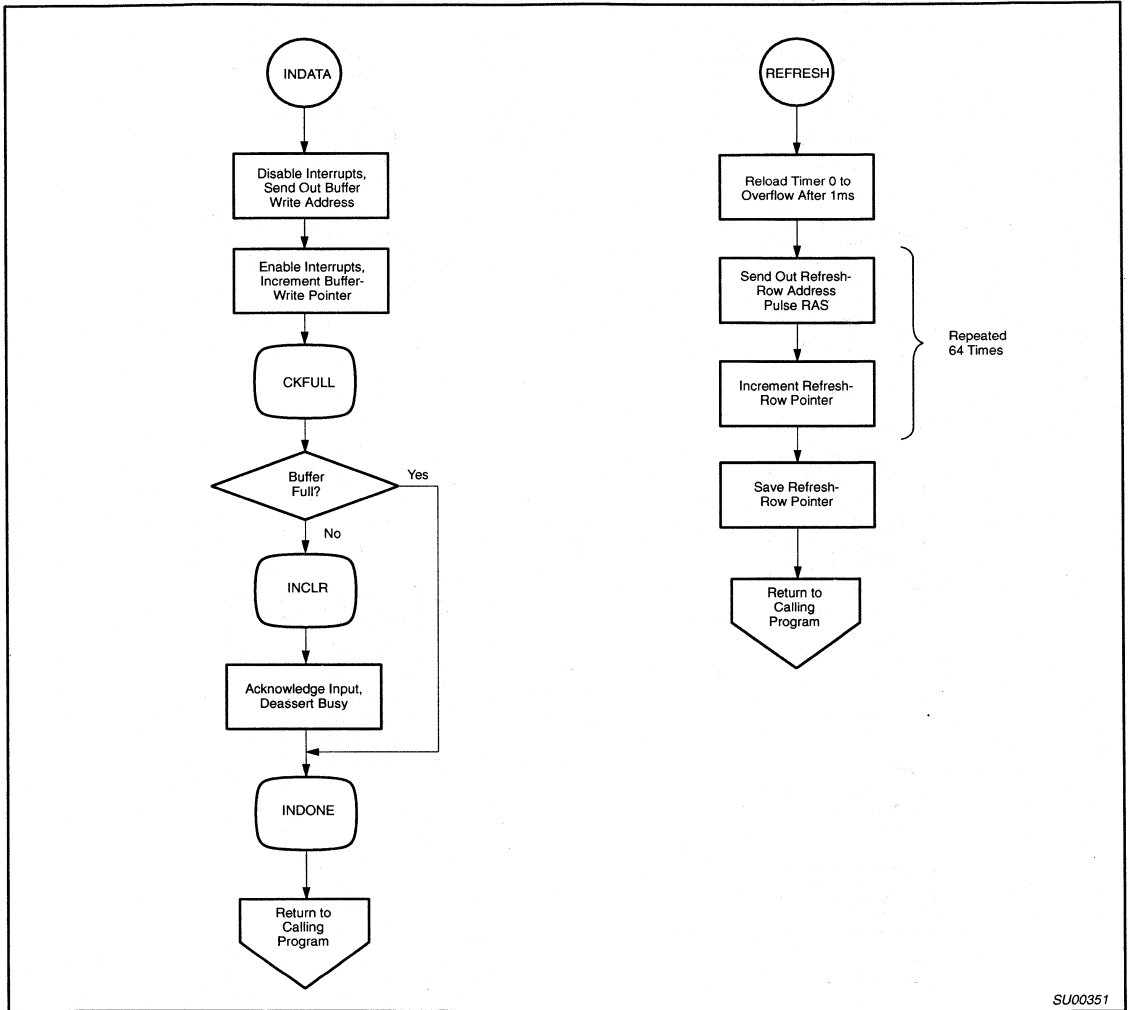


Figure 3. Flowchart of Receive and Refresh Operation

SU00351

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```
XMIT:      .MOV DPTR,8001H
TEST:      MOVX A,@DPTR      ;READ THE CSR
           JB ACC.0,TEST    ;TEST IBF FLAG
```

```
*****
256K PRINTER BUFFER PROGRAM USING THE 8xC451
FOR CENTRONICS PARALLEL PRINTER PORTS

PHILIPS SEMICONDUCTORS
OCTOBER, 1988
*****
```

```
$Mod451
$title(*XC451 Printer Buffer)
$date(10/28/88)
```

PORT USAGE:

- P0 Not used (reserved for data/address bus when external program memory is used).
- P1 Lower 8 bits of DRAM address (A0 – A7).
- P2 Not used (reserved for high-order address bus when external program memory is used).
- P3.0 (Reserved for serial port.)
- P3.1 (Reserved for serial port.)
- P3.2 (/INT0) Input port strobe input (interrupt).
- P3.3 (/INT1) Output port acknowledge input (interrupt).
- P3.4 Input port acknowledge output.
- P3.5 DRAM write enable output.
- P3.6 (/WR) DRAM row address select output.
- P3.7 (/RD) DRAM column address select output.
- P4.0 Upper bit of DRAM address (A8).
- P4.1 Reserved as an extra address line for 1 megabit DRAMS.
- P4.2 Not used.
- P4.3 Output port busy input (OBUSY).
- P4.4–P4.7 Unused (not available on 64-pin DIP package).
- P5 DRAM output data.
- P6 Parallel output port.
- /IDS Input port strobe input (ISTB).
- BFLAG Input port busy output (IBUSY).
- AFLAG Output port strobe output (OSTB).
- /ODS Port 6 output enable, tied low.

Internal Register/RAM Usage:

```
REFCNT EQU 020h ; Low order refresh byte.;
```

The following refer to the circular FIFO buffer implemented in the DRAM array.

```
INLOW EQU 22h ; Incoming address low byte.
INMID EQU 23h ; Incoming address mid byte.
INHI EQU 24h ; Incoming address high byte.
OUTLOW EQU 25h ; Outgoing address low byte.
OUTMID EQU 26h ; Outgoing address mid byte.
OUTHIGH EQU 27h ; Outgoing address high byte.
OACK EQU 28h ; Holds flag for output port acknowledge.
FOACK BIT OACK.0 ; Bit-address of output port acknowledge flag.
```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

; Miscellaneous Equates:
;
TIME      EQU      -1000          ; Value for 1000 timer clocks = 1 millisecond.
TIMEHI    EQU      HIGH TIME     ; High byte of timer value.
TIMELO    EQU      LOW TIME      ; Low byte of timer value.
RAS       BIT      P3.6          ; DRAM column address select.
CAS       BIT      P3.7          ; DRAM row address select.
DRAMWR    BIT      P3.5          ; DRAM write control line.
IACK      BIT      P3.4          ; Input port ACK output.
ISTB      BIT      P3.2          ; Input port strobe line (INT0).
OBUSY     BIT      P4.3          ; Output port BUSY input.
OSTB      BIT      MA0           ; Output port strobe (MA0 bit in port 6 CSR).
;

```

```

; *****
;
; Reset and Interrupt Jump Table
;

```

```

      ORG      00h          ; Power-on reset.
      AJMP     START

      ORG      03h          ; INT 0.
      AJMP     INDATA      ; Data at input port.

      ORG      0Bh          ; Timer 0.
      AJMP     REFRESH     ; Refresh DRAM array.

      ORG      13h          ; INT 1.
      AJMP     OPACK       ; Output port acknowledge.
; *****

```

```

;
; Power up reset routine:
; Set up refresh timer, enable timer interrupt and
; external interrupt, initialize circular buffer pointers.
;

```

```

START:  ORG      18h
        MOV     SP,#40h      ; Initialize stack pointer.
        MOV     A,#00
        MOV     REFCNT,A    ; Initialize refresh counter.
        MOV     INLOW,A     ; Initialize FIFO pointers.
        MOV     INMID,A
        MOV     INHI,A
        MOV     OUTLOW,A
        MOV     OUTMID,A
        MOV     OUTHI,A

; Initialize interrupt priority register so that DRAM refresh
; (TF0) gets high priority, input port service (IE0) and output
; port acknowledge service get lower priority. All other
; interrupts set to lower priority level.
;
        MOV     IE,#00000111b ; Timer0, INT0, and INT1 enabled.
        MOV     IP,#00000010b ; Timer0 high priority.
        MOV     TLO,#TIMELO
        MOV     TH0,#TIMEHI
        MOV     TMOD,#00000001b ; Operate Timer0 in mode 1.
        MOV     TCON,#00010101b ; Timer0 run, I0 and I1 = edge.
;
;

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

; Initialize Port 6 Control and Status Register.
; - 'BFLAG' mode set to output value of IBF
;   (input port BUSY signal : IBUSY)
; - 'AFLAG' set as logic 1 output
;   (output port strobe signal : OSTB)
; - 'IDS' active on negative level
;   (input port strobe signal : ISTB)
MOV   CSR,#10011100b
MOV   A,P6           ; Dummy read of P6 to clear IBF (IBUSY).
SETB  EA             ; Enable interrupts.

```

```

; Main Routine:
; Executes while not performing DRAM refresh or servicing
; input port interrupt.

```

```

; Check if buffer is not empty by comparing input and output
; pointers. If not empty, go to NOTMT to output a byte.

```

```

MAINLP:  MOV   A,INLOW           ; Compare pointers.
         CJNE  A,OUTLOW,NOTMT
         MOV   A,INMID
         CJNE  A,OUTMID,NOTMT
         MOV   A,INHI
         CJNE  A,OUTH,NOTMT
         SJMP  MAINLP

```

```

; Buffer is not empty: compute row & column addresses for
; a read cycle from DRAM.

```

```

NOTMT:  MOV   R4,OUTLOW         ; Save low byte of row.
         MOV   R5,OUTMID       ; Save upper bit of row.
         MOV   A,OUTH          ; Shift to align correctly.
         RRC   A
         MOV   R7,A            ; Save upper column bit.
         MOV   A,OUTMID       ; Get low byte of column.
         RRC   A
         MOV   R6,A           ; Shift in bit from OUTH.
         ; Save.

```

```

; Now do actual DRAM access to get the data byte at computed
; address. Disable interrupts so we don't lose what we put
; out on the ports.

```

```

CLR     EA                   ; Disable interrupts.
MOV     P1,R4                ; Low byte row address.
MOV     A,R5                 ; Get high byte row address.
ORL     A,#0FEh              ; Make sure OBUSY stays high.
MOV     P4,A
CLR     RAS                  ; /RAS low.
MOV     P1,R6                ; Low byte column address.
MOV     A,R7                 ; High byte column address.
ORL     A,#0FEh              ; Make sure OBUSY stays high.
MOV     P4,A
CLR     CAS                   ; /CAS low.
MOV     R4,P5                ; Get the data byte

SETB    CAS                  ; /CAS high.
SETB    RAS                  ; /RAS high.
CLR     FOACK                 ; Clear acknowledge flag.
SETB    EA                   ; Re-enable interrupts.

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

PLOOP1:  JB      OBUSY,PLOOP1  ; Loop if printer busy.
;
;
;      CLR      EA              ; Disable interrupts.
;      MOV      P6,R4          ; Move byte to output port.
;      CLR      MA0           ; Assert output port strobe.
;      NOP                     ; Kill some time.
;      NOP
;      NOP
;      NOP
;      SETB     MA0           ; De-assert output port strobe.
;      SETB     EA              ; Re-enable interrupts.
;
;      Following waits for /ACK to occur on output port. Loops on
;      acknowledge flag which is set by INT1 service routine when
;      /ACK occurs.
;
PLOOP2:  JNB      FOACK,PLOOP2  ; Wait till /ACK occurs.
;
;      INC      OUTLOW         ; Increment output buffer pointer.
;      MOV      A,OUTLOW
;      CJNE     A,#00,PDONE
;      INC      OUTMID
;      MOV      A,OUTMID
;      CJNE     A,#00,PDONE
;      MOV      A,OUTH1
;      INC      A
;      ANL      A,#03h        ; Eliminate unused address bits
;      MOV      OUTH1,A       ; and save.
;
;      Check if input port busy flag was left asserted, indicating that
;      the buffer was full after last input. If so, acknowledge input
;      port and de-assert input busy signal.
;
PDONE:   JNB      IBF,MAINLP    ; Not busy, return to main loop.
;      CLR      EA              ; Disable interrupts.
;      CLR      IACK           ; Assert /IACK.
;      NOP                     ; Wait 7 microseconds.
;      NOP
;      NOP
;      NOP
;      NOP
;      NOP
;      MOV      A,P6          ; Dummy read of P6 clears IBF (IBUSY).
;      NOP                     ; Wait 5 microseconds.
;      NOP
;      NOP
;      SETB     IACK          ; De-assert /IACK.
;      SETB     EA              ; Re-enable interrupts.
;      AJMP     MAINLP        ; Return to main loop.
;

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

.....
:
: Interrupt 1 Service Routine:
:
: - Called when output port asserts /ACK.
: - Sets FOACK flag and returns.
:
OPACK:          SETB          FOACK
              RETI
:
:
:

```

```

.....
:
: DRAM Refresh (Timer0) Interrupt Service:
:
: - Called once every millisecond by timer interrupt.
: - Refreshes 64 rows and then returns.
: - Therefore refreshes all rows every 4 milliseconds.
: (Note that 41256/51C256 DRAM only requires a 256 row refresh.)
:
REFRESH:

```

```

REFRESH:  PUSH    PSW
          MOV     TH0,#TIMEHI    ; Reload timer registers.
          MOV     TL0,#TIMELO
:
          MOV     P1,REFCNT      ; Get next row to refresh.
          MOVX   @R0,A          ; Pulse /RAS (WR).
          INC     P1
          MOVX   @R0,A          ; 1
          INC     P1
          MOVX   @R0,A          ; 2
          INC     P1
          MOVX   @R0,A          ; 3
          INC     P1
          MOVX   @R0,A          ; 4
          INC     P1
          MOVX   @R0,A          ; 5
          INC     P1
          MOVX   @R0,A          ; 6
          INC     P1
          MOVX   @R0,A          ; 7
          INC     P1
          MOVX   @R0,A          ; 8
          INC     P1
          MOVX   @R0,A          ; 9
          INC     P1
          MOVX   @R0,A          ; 10
          INC     P1
          MOVX   @R0,A          ; 11
          INC     P1
          MOVX   @R0,A          ; 12
          INC     P1
          MOVX   @R0,A          ; 13
          INC     P1
          MOVX   @R0,A          ; 14
          INC     P1
          MOVX   @R0,A          ; 15
          INC     P1
          MOVX   @R0,A          ; 16
          INC     P1
          MOVX   @R0,A          ; 17
          INC     P1
          MOVX   @R0,A          ; 18
          INC     P1
          MOVX   @R0,A          ; 19
          INC     P1
          MOVX   @R0,A          ; 20
          INC     P1

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

MOVX  @R0,A      ;21
INC   P1
MOVX  @R0,A      ;22
INC   P1
MOVX  @R0,A      ;23
INC   P1
MOVX  @R0,A      ;24
INC   P1
MOVX  @R0,A      ;25
INC   P1
MOVX  @R0,A      ;26
INC   P1
MOVX  @R0,A      ;27
INC   P1
MOVX  @R0,A      ;28
INC   P1
MOVX  @R0,A      ;29
INC   P1
MOVX  @R0,A      ;30
INC   P1
MOVX  @R0,A      ;31
INC   P1
MOVX  @R0,A      ;32
INC   P1
MOVX  @R0,A      ;33
INC   P1
MOVX  @R0,A      ;34
INC   P1
MOVX  @R0,A      ;35
INC   P1
MOVX  @R0,A      ;36
INC   P1
MOVX  @R0,A      ;37
INC   P1
MOVX  @R0,A      ;38
INC   P1
MOVX  @R0,A      ;39
INC   P1
MOVX  @R0,A      ;40
INC   P1
MOVX  @R0,A      ;41
INC   P1
MOVX  @R0,A      ;42
INC   P1
MOVX  @R0,A      ;43
INC   P1
MOVX  @R0,A      ;44
INC   P1
MOVX  @R0,A      ;45
INC   P1
MOVX  @R0,A      ;46
INC   P1
MOVX  @R0,A      ;47
INC   P1
MOVX  @R0,A      ;48
INC   P1
MOVX  @R0,A      ;49
INC   P1
MOVX  @R0,A      ;50
INC   P1
MOVX  @R0,A      ;51
INC   P1
MOVX  @R0,A      ;52
INC   P1
MOVX  @R0,A      ;53
INC   P1
MOVX  @R0,A      ;54
INC   P1

```


256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

MOVX  @R0,A      ;55
INC   P1
MOVX  @R0,A      ;56
INC   P1
MOVX  @R0,A      ;57
INC   P1
MOVX  @R0,A      ;58
INC   P1
MOVX  @R0,A      ;59
INC   P1
MOVX  @R0,A      ;60
INC   P1
MOVX  @R0,A      ;61
INC   P1
MOVX  @R0,A      ;62
INC   P1
MOVX  @R0,A      ;63
INC   P1

INC   P1          ; Adjust for next time
MOV   REFCNT,P1  ; and save.
POP   PSW
RETI

```

.....

Data at Input Port:

```

; This routine is called via interrupt INTO whenever data
; is strobed into the input port. It saves the data into the
; DRAM array and increments the input pointer. If the output
; pointer is now equal to the input pointer, then the buffer
; is full, and we leave the busy flag set so that no more
; data can be input until some is output and the buffer is
; no longer full.

```

```

INDATA:  PUSH   PSW
         PUSH   ACC
         MOV    R1,INLOW      ; Lower 8 bits of row to R1.
         MOV    R2,INMID     ; Upper bit of row to R2.
         MOV    A,INHI       ; Get upper 2 bits.
         RRC    A            ; LSB to carry.
         MOV    R0,A
         MOV    A,INMID
         RRC    A            ; Shift bit into MSB.
         MOV    R3,A         ; Save.

         CLR    EA           ; Disable interrupts.
         MOV    P1,R1        ; LSB Row address.
         MOV    A,R2         ; MSB row address.
         ORL   A,#0FEh      ; Make sure OBUSY stays high.
         MOV    P4,A         ; MSB row address.
STBLP:   JNB   ISTB,STBLP    ; Check for end of strobe before DRAM write.
         CLR    RAS          ; /RAS low.
         CLR    DRAMWR       ; /WR low.
         MOV    P1,R3        ; LSB column address.
         MOV    1,R0         ; MSB column address.
         ORL   A,#0FEh      ; Make sure OBUSY stays high.
         MOV    P4,A         ; MSB column address.
         MOVX  A,@R0         ; Pulse /CAS low.
         SETB  RAS          ; /RAS high.
         SETB  DRAMWR       ; /WR high.
         SETB  EA           ; Re-enable interrupts.

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

INC     INLOW           ; Increment input buffer pointer.
MOV     A,INLOW
CJNE   A,#00,CKFULL
INC     INMID
MOV     A,INMID
CJNE   A,#00,CKFULL
MOV     A,INH1
INC     A
ANL    A,#03h          ; Eliminate unused address bits.
MOV     INH1,A
;
; Compare input pointer to output pointer to see if the buffer is full.
CKFULL: MOV     A,INLOW
        CJNE   A,OUTLOW,INCLR
        MOV     A,INMID
        CJNE   A,OUTMID,INCLR
        MOV     A,INH1
        CJNE   A,OUTH1,INCLR
;
; If we get here, the buffer is full, so skip the acknowledge pulse.
        SJMP   INDONE
;
; Send acknowledge pulse on /IACK line for 7 microseconds,
; de-assert input BUSY signal halfway through.
INCLR:  CLR     EA           ; Disable interrupts.
        CLR     IACK        ; Assert /IACK.
        NOP                    ; Wait 7 microseconds.
        NOP
        NOP
        NOP
        NOP
        NOP
        MOV     A,P6         ; Dummy read of P6 clears IBF (IBUSY).
        NOP                    ; Wait 5 microseconds before clearing /IACK.
INDONE: POP     ACC
        POP     PSW
        SETB   IACK        ; De-assert /IACK.
        SETB   EA           ; Re-enable interrupts.
        RETI
;
        END

```

Counter/timer 2 of the 83C552 microcontroller

AN418

INTRODUCTION TO THE 83C552

The 83C552 is an 80C51 derivative with several extended features: 8k ROM, 256 bytes RAM, 10-bit A/D converter, two PWM channels, two serial I/O channels, six 8-bit I/O ports, and four counter/timers. The architecture of the 83C552 is identical to that of the 80C51, making the two devices fully code compatible. The additional peripheral functions are added to the 80C51 Special Function Register space, and the interrupt structure is modified accordingly. This information is detailed in other references on the 83C552. The focus of this application note is on one of the timers of the 83C552, Counter/Timer 2.

This counter/timer includes capture, compare, and high-speed output capabilities which facilitate many control oriented tasks. The objective of this note is to make users of the 83C552 aware of this counter/timer subsystem and assist the use of this subsystem by a detailed explanation of its operation supported by actual application examples.

TIMER 2 OF THE 83C552

Timer 2 of the 83C552 is in fact a timing controller and has an associated programmable array. The Timer 2 subsystem consists of three parts:

1. The time base consists of a 16-bit timer with a 3-bit prescaler. The master clock for the subsystem can be derived from the on-chip oscillator (fosc) or an external input, T2. It has an external reset, RT2, by which a signal applied to this input can reset the timer if the external reset is enabled.
2. A capture system consisting of four capture registers and four capture inputs which can be used for a wide variety of time measurements on external signals.
3. A compare system consisting of three compare registers and eight associated high-speed outputs which can be activated upon a match between the 16-bit timer and one of the compare registers.

For reference a complete block diagram of the 83C552 Counter/Timer 2 subsystem is shown in Figure 1.

16-BIT COUNTER/TIMER

The description of Counter/Timer 2 in the following paragraphs is intended to be a general overview. Details on architecture, address locations, interrupt structure, and timer operation are given in the 83C552 Users Manual. This users manual may be useful to complement the material presented in this application note. References to registers, bits, I/O ports, and on-chip hardware will relate directly to 83C552 Users Manual nomenclature. This application note will focus on the use of Counter/Timer 2 as a powerful input capture and high-speed output facilitator through some specific examples and not on the detailed coding.

The counter/timer consists of a 16-bit counter which is readable by software through special function registers TM2L and TM2H. The timer itself has two overflow flags, one after the entire 16-bit counter and one attached to the eighth stage. This latter flag reflects an overflow from the first byte of the counter. These two flags are present in register TM2IR and are labeled T2BO for the overflow from the first byte and T2OV for the overflow from the entire 16 bits. These flags may be used to generate an interrupt.

The counter timer is controlled directly through the special function register TM2CON, the timer 2 control register. This register also contains certain status flags.

The prescaler divides the input clock by a programmable ratio. The prescaler divide value is programmable to divide by 1, 2, 4, or 8 as controlled by T2PO and T2P1 in TM2CON.

The input clock to the prescaler is either fosc/12 or the external input, T2. The clock input to the prescaler may also be shut off. This clock input selection is controlled by bits T2MS0 and T2MS1 in TM2CON.

If T2 is used as the input clock to the timer 2 subsystem, the hardware logic samples this input and looks for a low-to-high transition. If the logic detects a logic 0 at the T2 input in state S2P1 of the microcontroller and a logic 1 in state S5P1, then this is recognized as a low-to-high transition, and the prescaler is incremented. The prescaler is incremented in the second cycle after the cycle in which the transition was detected. If the transition is detected before S2P1 is finished, the prescaler is incremented in the next cycle. This timing is shown in Figure 2. Note that this sampling rate is twice that of the normal 80C51 timers, T0 and T1; therefore T2 has

twice the maximum external counting rate as compared to the standard timers.

Any programming of the clock source or the prescaler divide ratio results in a reset of the prescaler. This allows the state of the timer subsystem to be in a known state upon programming. The main 16-bit timer cannot be reset by software but it is reset by activating the reset pin or using the external reset, RT2. The external reset, RT2, can be enabled or disabled by bit T2ER in TM2CON. These resets reset the prescaler as well as the 16-bit counter.

Only one interrupt is available from the 16-bit counter timer. Two bits in TM2CON control whether TM2L, TM2H, or both flags will be used to generate the interrupt. A selection for no interrupt is also possible.

Capture System

The capture system is a powerful tool to measure the width of pulses or repetition rates. There are four independent inputs for the signals to be analyzed, CTI0 through CTI3. These inputs are alternate functions to port 1. Each input is connected to a dedicated capture register. A transition at any of these inputs will cause the content of the 16-bit counter/timer to be loaded into the respective capture register. The capture can occur upon various conditions of the input signal as specified by certain bits in the capture control register, CTCON. Each input can be set to cause a capture on a low-to-high transition, a high-to-low transition, or on both transitions. Upon a capture taking place, each input causes an interrupt flag to be set in the Timer 2 Interrupt Flag Register, TM2IR. If enabled, an interrupt will be generated.

One of the capture inputs is shown in more detail in Figure 3. All of the other capture inputs are similar to this one. The capture input is gated with the capture enable bits CTN0 and CTP0, which are located in CTCON. According to the status of these bits, the desired edges are selected to generate the capture enable pulse. The input pulse transient detection is at the input of the enable pulse generator. The input signal is sampled at S1P1 of the machine cycle. If a logic 1 is detected when a logic 0 was detected at the same time in the previous cycle, then the event is taken as a transition. An enable pulse is sent to the capture register, and the contents of timer 2 is copied into the capture register at the end of this machine cycle. The interrupt flag CTI0 is also set.

Counter/timer 2 of the 83C552 microcontroller

AN418

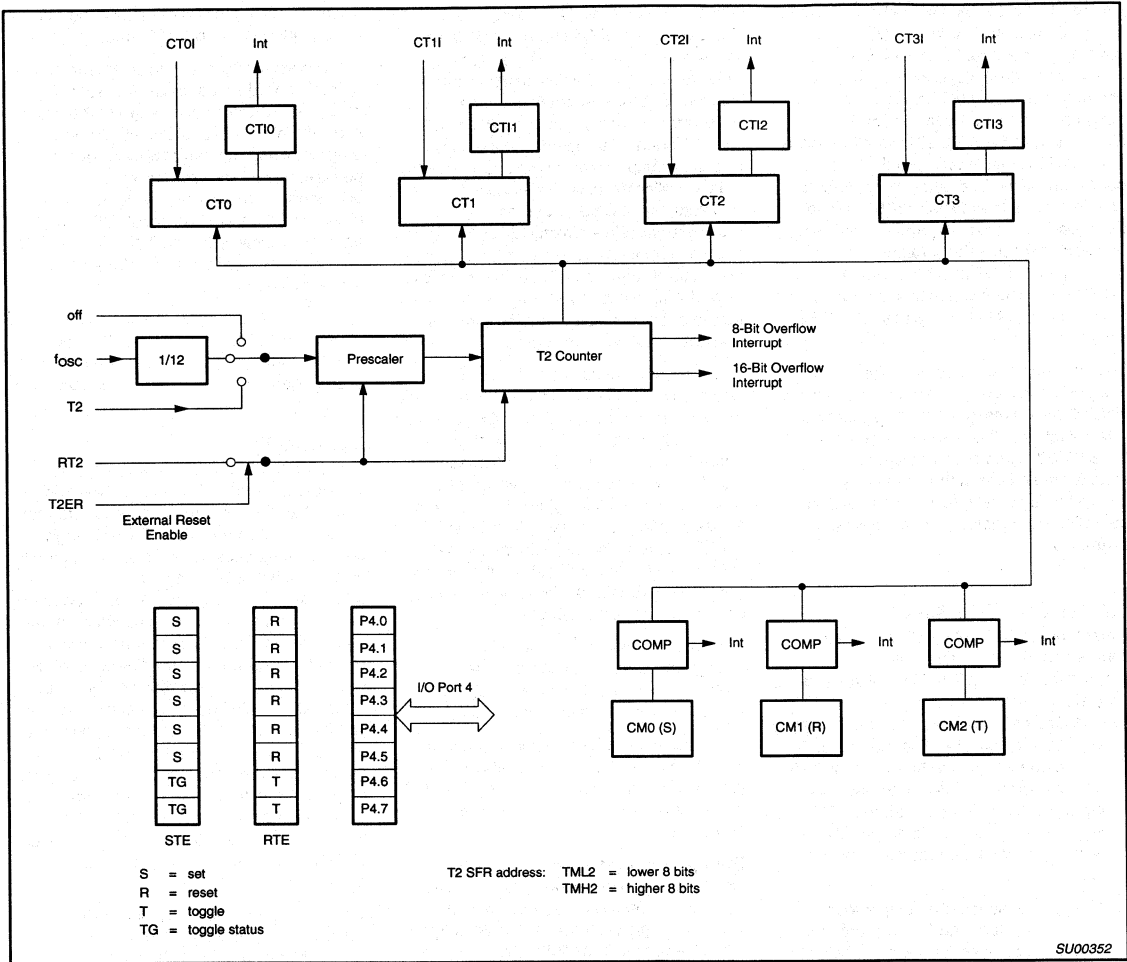


Figure 1. 83C552 Counter/Timer 2 Block Diagram

Counter/timer 2 of the 83C552 microcontroller

AN418

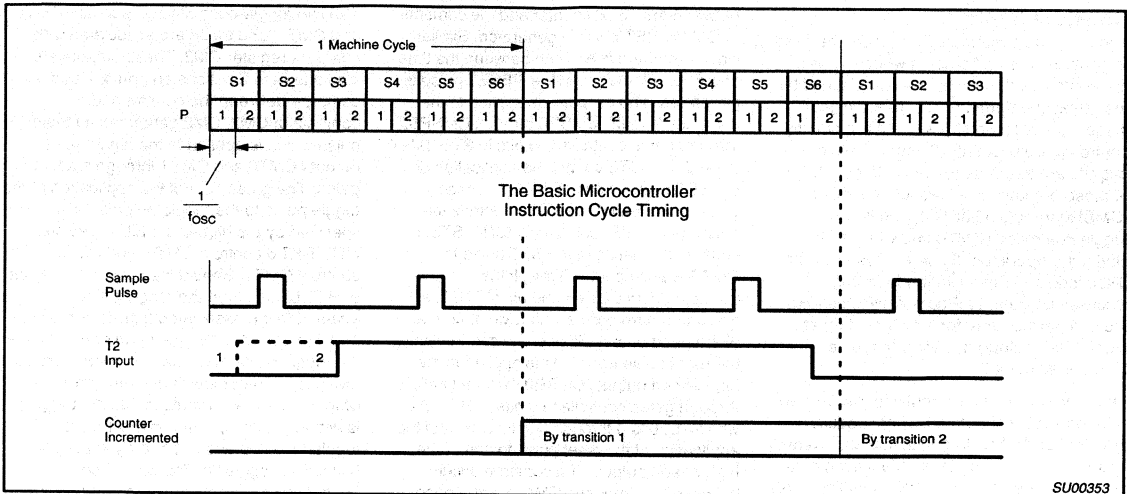


Figure 2. The Sampling of the External Clock Signal

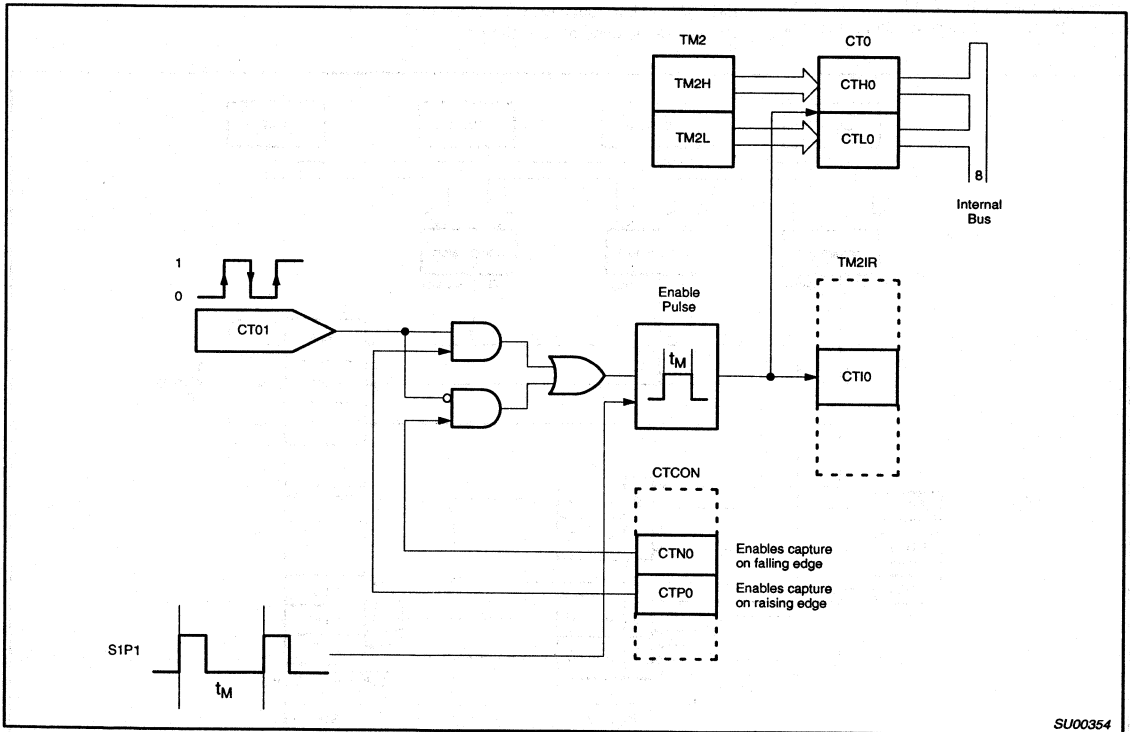


Figure 3. Capture Subsystem for CTO1

Counter/timer 2 of the 83C552 microcontroller

AN418

Compare System

The compare system of Timer 2 can be used to generate a set of outputs whose transitions are controlled directly by the time defined by the 16-bit counter/timer. There are eight of these high-speed outputs which are directly controlled from Counter/Timer 2. These outputs are alternate functions to port 4. Six of these outputs are set-reset controlled (CMSR0 through CMSR5), and two are toggle controlled (CMT0 and CMT1). To clarify the operation, these two types will be discussed separately. In the following discussions, refer to Figure 4, which shows the compare system for P4.5 (a set-reset high-speed output) and P4.6 (a toggle high-speed output).

There are two compare registers associated with the set-reset outputs. These registers are CM0 and CM1. In addition, there are two enable registers: one to enable setting of an output and the other to enable resetting of an output. These registers are STE and RTE, respectively. The contents of CM0 and CM1 are continuously compared to the contents of the 16-bit counter. Whenever there is a match

between the 16-bit counter and the contents of CM0, a SET pulse is generated. Similarly, whenever there is a match between the timer and the contents of CM1, a RESET pulse is generated. The set pulse is applied to the set-reset outputs, CMSR0 through CMSR5, through gates controlled by bits in STE. Bits 0 through 5 in STE control the application of the SET pulse to one of the high-speed outputs. For example, STE.0 controls the gating of the SET pulse to CMSR0, STE.1 controls the gating of the SET pulse to CMSR1, and so forth. Thus, if the corresponding SET enable bit in STE is a 1 and a compare occurs with CM0, then that high-speed output will become set. Similarly, the reset pulse from CM1 is applied to the high-speed outputs CMSR0 through CMSR5 through gates controlled by bits in RTE. As with STE, bits 0 through 5 in RTE control the application of the reset pulse to one of the high-speed outputs. If a compare occurs between the timer and CM1, a high-speed output will be reset if its corresponding enable bit in RTE is a 1. Compares with CM0 and CM1 set interrupt flags which, if enabled, can be used to generate an interrupt.

The two toggle-controlled outputs are CMT0 and CMT1, and these are associated with compare register CM2. These outputs are also alternate functions on port 4. Upon a compare between the counter and the contents of CM2, CM2 generates a toggle pulse which is applied to the high-speed outputs CMT0 and CMT1 through a set of gates. The gates control the application of the toggle pulse to the toggle outputs as specified by the high-order bits of register RTE. RET.6 controls CMT0, and RET.7 controls CMT1. Should the corresponding bit of RTE be set, then the toggle pulse is enabled to the associated high-speed output and that output will toggle upon generation of the toggle pulse from CM2. The structure of these toggled outputs is different from the other high-speed outputs in that the toggling is actually accomplished in a separate toggled flip-flop and not directly in the port latch. This toggle flip-flop cannot be controlled directly by software and powers up in an indeterminate state. The state of the toggle flip-flops is readable in STE bits 6 and 7.

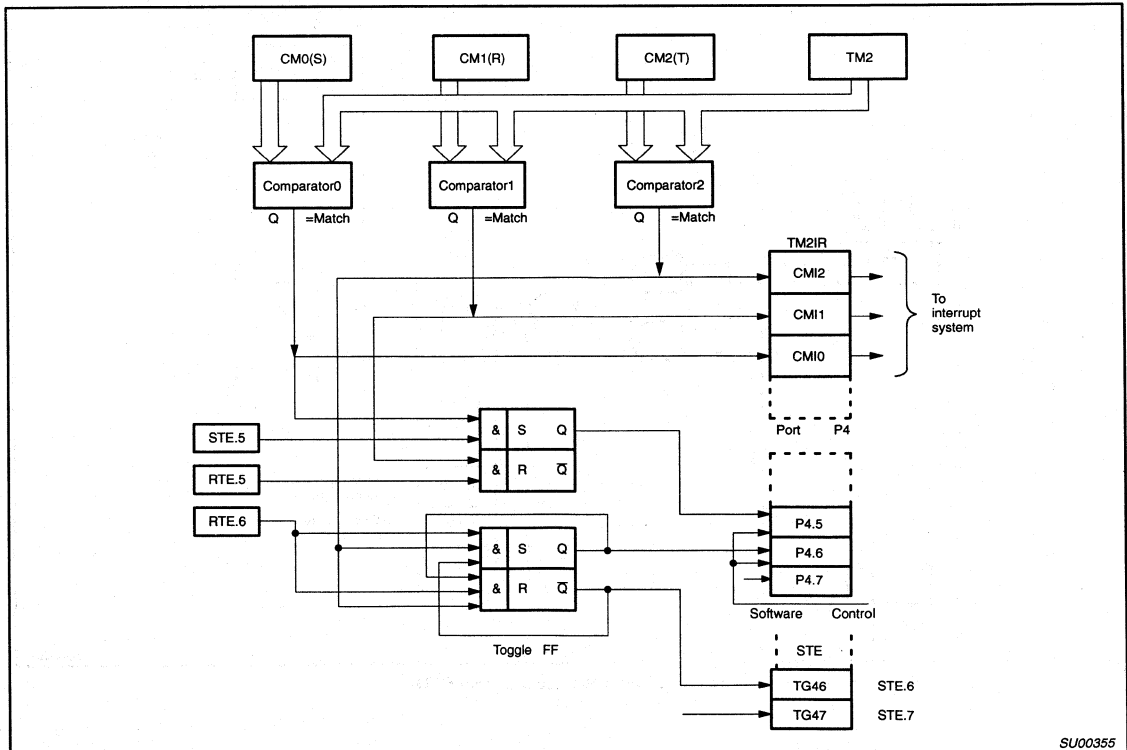


Figure 4. Compare System for P4.5 and P4.6

SU00355

Counter/timer 2 of the 83C552 microcontroller

AN418

APPLICATION EXAMPLE— TIMED FUEL INJECTION

In modern automobiles, optimal combustion is necessary to meet emission standards and improve fuel consumption. Optimal combustion depends on several factors and is enhanced by proper fuel injection based upon these factors which vary according to engine speed and other factors. Thus the task is to control the opening and closing of the engine fuel injectors of each cylinder relative to the crankshaft reference point.

For the application example here, we will not consider the factors which determine the timing relationships. These are assumed to be given quantities. The example here will focus upon the implementation of the injector timing control signals and how they are generated using the Counter/Timer 2 system. The illustration considers a four cylinder engine. While this is an automotive application which serves to clearly illustrate Counter/Timer 2 Subsystem operation, it is clear that many systems share similar timing requirements, and the techniques employed here are applicable to a wide class of timing tasks. The 83C552 will also support six cylinder engine control.

Figure 5 shows the injection timing required for two consecutive revolutions of the engine crankshaft. Start and stop of the injection are given relative to a reference point on the crankshaft. The cylinders are numbered in the order of the injection sequence (not with reference to their physical location). Start of the injection is usually given in angular measure with respect to top dead center, and

the injection duration is assumed to be a time value calculated from engine environmental factors and operating parameters. The angle for the start of the injection must be converted into time with respect to the reference point.

The injector drivers are assumed to be connected to the port 4 high-speed outputs CMSR0 through CMSR3. To obtain the top dead center reference point, the signal from the appropriate sensor is connected to the capture input CTOI. The interrupt for this capture input is enabled so that software can synchronize its operation to this time reference and make use of the top dead center time in the injector timing calculations. The software synchronization takes two forms. First, the captured time is an absolute reference for all real-time output operations. This time is available in capture register CTO. Note that at 12 MHz operation, Timer 2 can have a resolution as fine as 1 microsecond with a total time before overflow of over 65 milliseconds, and these times are adjustable by increasing the prescaler divide ratio. A proper selection can make the timing calculations relatively simple. Second, at the time the input is captured, flags which keep track of the phases of the crankshaft cycle are reset when cylinder 1 is at top dead center. These flags are used in the interrupt service routines to tell which action is required for that phase of the crankshaft.

Consider now the sequence of events in one rotation of the engine crankshaft and refer to Figure 5 during the discussion. Assume that

the engine is running, that all relevant parameters are available, and that it has been determined that the processor is responding to the interrupt associated with the top dead center capture, CTOI. Interrupts for CTOI, CM0 (compare register 0), and CM1 (compare register 1) are enabled. Upon entering the interrupt service routine for CTOI, the previous value of the captured top dead center time is subtracted from the present value, and the crankshaft rotation time is determined. This is used to compute the time to open the first injector from the required angle at which the injector is to open. This time is made available for the interrupt service routine which responds to a compare from CM0. The interrupt service routine is exited.

The next interrupt to occur per the figure for this example is a result of a compare with CM1 which will be a result of the injector stop time for cylinder 4 having been reached. The flags in an internal status register are employed to keep track of the cylinder number that is presently active for both injection stop and injection start times. After identifying this interrupt from the flags, the processor uses the injector start time for cylinder 1 (previously loaded into CM0) and the predetermined duration to calculate the injector stop time for cylinder 1. This value is loaded into compare register CM1 and the reset enable bit for high-speed output CMSR0 is programmed to a 1. This is bit RTE.0 The reset enable bit for the cylinder 4 injector is set to 0 (bit RET.3). The interrupt routine is now exited.

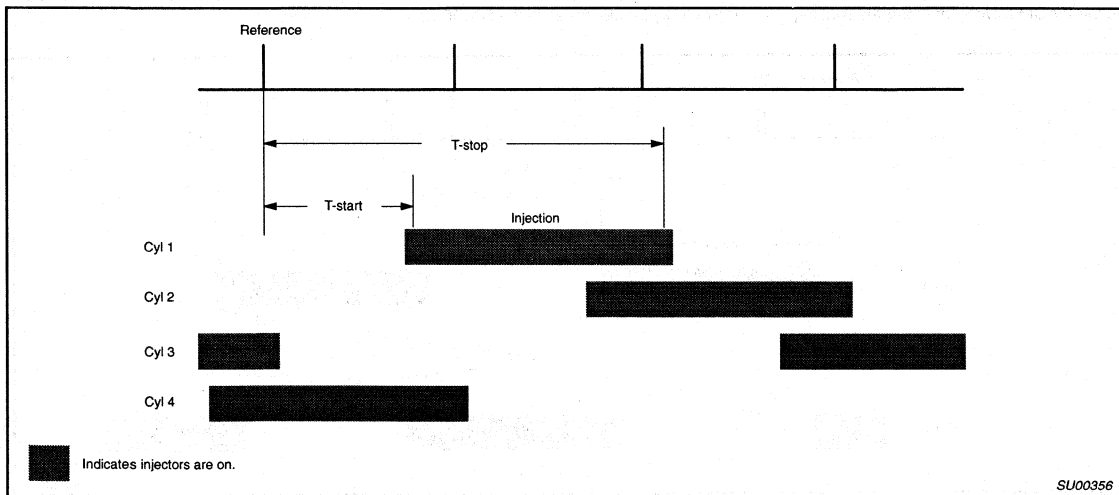


Figure 5. Four Cylinder Injection Timing

Counter/timer 2 of the 83C552 microcontroller

AN418

The next interrupt to occur will be for the start time for the injector for cylinder 2. This and all subsequent cases follow the same sequence of events as for the cylinder 1 CM0 interrupt described above. In this case, calculations are made for cylinder 2 and loaded into CM0, STE.1 is programmed to a 1, and STE.0 is programmed to a 0. Similarly, the next interrupt for CM1 is treated in the same way, and the sequence of events rotates around through all cylinders in turn. The flag bits associated with this operation keep track of the injector sequencing.

While this example shows the injection stop time of one cylinder overlapping into the injector on time of the subsequent cylinder, close examination of the operations described above reveal that the start and stop events are independent and can overlap or not as required. In this way all injectors may be driven independently and have overlapping on times.

Given that this is an example applicable to general usage, it is possible that interrupt service routine could be relatively long as it would be in an actual injector application. Since the service routine has other interrupts disabled, the length may cause real-time conflicts. To eliminate this potential problem, the interrupt service routines are divided into two parts. In the first part, all other interrupts are disabled, and the essential register loading is done to prepare for the next interrupt. After this is completed, all interrupts are enabled and the ancillary service routine functions are performed prior to a return to the main routine.

As an example, consider the interrupt service routine for CM0. Upon entering the routine,

all interrupts are disabled. Then the following actions are performed:

- Set bit in STE to start next injector
- Clear bit in STE for injector just started
- Load CM0 with start time for next injector
- Clear CMIO interrupt flag in TM2IR

Now that the essential set-up is made for the next interrupt, all interrupts are now enabled. However, the return to the main program is not invoked until the following ancillary processing is completed:

- Calculate the next absolute start time for the next injector (the next load value for CM0)
- Increment the flag so that the next entry to this interrupt service routine will be able to identify the next injector to start.

The process performing these calculations can be interrupted to service real-time functions.

APPLICATION EXAMPLE—TIMED IGNITION

In electronic ignition systems, multiple ignition coils may be used and each coil is fired by electronic means rather than with the old style mechanical breakers. In a four cylinder engine, there may be two ignition coils, one coil providing spark for a pair of cylinders. Both plugs fire at the same time. For one cylinder, the spark occurs at the appropriate time while for the other cylinder, the spark occurs at the end of the exhaust stroke and has no effect. With timing references to crankshaft top dead center provided by an external sensor, the ignition timing for the engine may be generated in the 83C552 and

applied to the electronic drivers for the ignition coils.

To illustrate the toggle high-speed outputs of the 83C552 Counter/Timer 2 subsystem, the following example will discuss the ignition timing in a four cylinder engine employing the two coil approach with one coil for a pair of cylinders. The coil timing is illustrated in Figure 6. A reference time is used which is a given interval prior to top dead center so that the times used in the illustration can be always after the reference. There are two times of interest for each coil: the load time and the ignition point.

Ignition advance is usually given in degrees crankshaft angle prior to top dead center. As with injection, this angle is assumed to be derived from other calculations and is a given value for this illustration. This angle must be converted in to a time with respect to the reference point. The load time (the time at which the coil has to be switched on to reach the current that will give sufficient energy for an adequate spark) must be subtracted from the desired ignition point. At the ignition time, the coil will be switched off and the spark will be generated.

The coil driver electronics are connected to port bits P4.6 and P4.7. Ignition coil 1 is connected to P4.6, and ignition coil 2 is connected to P4.7. These outputs are the toggle high-speed outputs controlled by the 16-bit compare register, CM2. The program simply needs to set up the compare and control registers to turn the coils on and off at the appropriate times. It is assumed in this example that the ignition and load times are given quantities and have been determined previously.

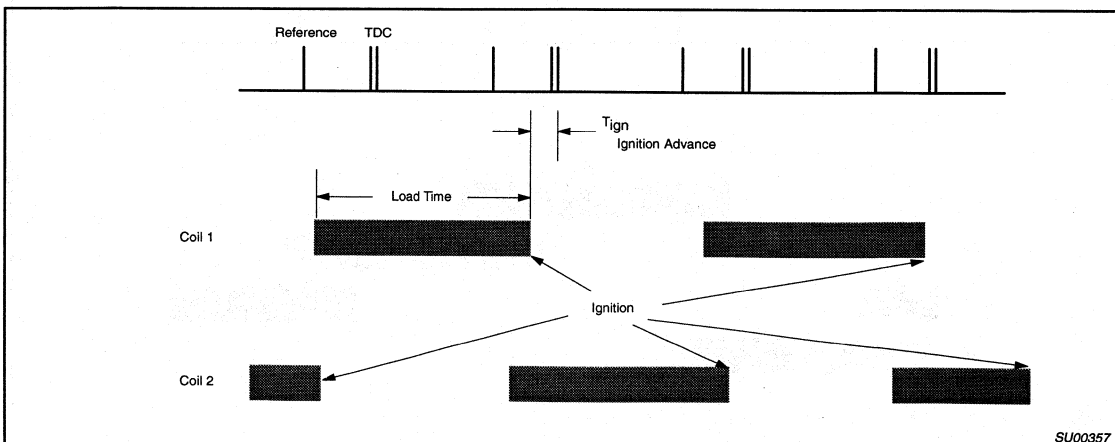


Figure 6. Four Cylinder, Two Coil Ignition Timing

SU00357

Counter/timer 2 of the 83C552 microcontroller

AN418

Consider now the sequence of events in two rotations of the engine crankshaft and refer to Figure 6. Assume that the engine is running, that all relevant parameters are available, and that it has been determined that the processor is responding to the interrupt associated with a compare to CM2. The top dead center time and crankshaft rotation speed have been already determined through the top dead center capture, CTOI. This is the same as in the injector example. The interrupt for CTOI is enabled. From the top dead center time, the times to turn on and turn off the coil drivers are computed and made available in data storage locations in the microcontroller. It is also convenient to have flags to identify the step in the complete ignition cycle. The flags are cleared in the interrupt service routine for top dead center of cylinder 1.

Upon entering the interrupt service routine, other interrupts are disabled. Examination of the flags reveals that the state of the ignition sequence is that coil 1 has been turned on to begin the current build up (load time). The next event will therefore be turning off coil 2 to cause ignition. The interrupt service routine then performs the following actions: The time to turn off coil 2 is moved into compare register 2, CM2. Bit 6 of RTE is cleared; this disconnects the output of CM2 from the toggle flip-flop of P4.6 (coil 1). Bit 7 of RTE is set; this connects the output of CM2 to the toggle flip-flop of P4.7 (coil 2). The flags are

incremented to indicate that the next interrupt will be a result of coil 2 turning off and causing ignition. The other interrupts can be enabled and a return to the main program can be executed. After the other interrupts are enabled and before a return is made to the main program, it may be convenient to do any necessary calculations to determine the time value to be loaded into CM2 in the next CM2 interrupt.

Since the flip-flops are toggled, it is likely that upon power up of the microcontroller, the toggle flip-flops will not be in the desired state. To get the toggle flip-flops in the correct state in the ignition cycle, the flip-flops must be toggled if they are in the wrong state. To determine if this is necessary, the state of the toggle flip-flops can be read from the STE register. The state of the P4.6 flip-flop is present in STE bit 6 and the state of the P4.7 flip-flop is present in STE bit 7. Comparing the actual state to the required state determines which if any or both of the flip-flops must be toggled. If a toggle is necessary to put one or both of the flip-flops in the correct state, the corresponding bits in RTE would be set for those flip-flops requiring the toggle, and CM2 would be loaded with a value that is slightly larger than the present contents of timer 2. If desired for reliability purposes, the state of the flip-flops could be checked periodically against the ignition cycle flags to determine if a correction is necessary.

CONCLUSION

This application note has examined one aspect of the 83C552 CMOS 80C51 derivative microcontroller. The Counter/Timer 2 Subsystem has been applied to a complex timing task of gasoline engine injector valve and ignition coil timing control. While this is a specific application to the automotive interests, the results are applicable to a wide variety of time measurement and control applications. The 83C552 would be ideal for many electromechanical systems such as copy machines, fax machines, industrial process control equipment, automatic transmission control, and anti-skid and anti-lock braking control.

These application areas are those which can successfully employ the 83C552 Counter/Timer 2; however, the other features should not be overlooked. When combined with the 10-bit A to D Converter, the Pulse Width Modulator, the I²C serial bus, and peripheral device family, the 83C552 provides minimum component count solutions for cellular radio systems, professional audio systems, and medical instrumentation products such as bedside patient monitors and analyzers for home care and sports use.

Using up to 5 external interrupts on 80C51 family microcontrollers

AN420

80C51 family microcontrollers are equipped with up to two inputs which may be used as general-purpose interrupts. A typical device provides a total of 5 interrupt sources. Timer 0 and Timer 1 generate vectored interrupts, as does the Serial Port. Applications that require more than two externally signaled vectored interrupts, and do not use one or more of the counters or the serial port, can be configured to use these facilities for additional external interrupt inputs.

This note describes a method to configure the timer/counters and the serial port for use as interrupt inputs (see Figure 1). Minimum response time is a goal for this configuration.

Another popular method to implement extra interrupt inputs is to poll under software control a port pin configured as an input. This method is necessary when the on-chip peripherals are in use. Applications where this approach is recommended are ones in which the processor spends more than half of the time executing a "wait loop," or a short code sequence which jumps or branches back on itself without performing any functions. In this case, the instructions that will check the state of input used as an interrupt source are inserted into this sequence. Consequently, this input is ignored when other routines are being executed. This input may have to be latched externally, or the processor may miss the signal while executing other routines.

Dedicated interrupt inputs that vector the processor to individual service routines (as the two general-purpose interrupt inputs work) do not have the drawbacks of the method described above.

COUNTER/TIMER CONFIGURATION

Timers 0 and 1 are placed in mode 2, which configures the timer/register as an 8-bit counter with automatic reload. The counter and reload register are loaded with FF hexadecimal which is stored in TH1 and TL1 or TH0 and TL0.

To prepare one of the timers for this kind of operation, a number of control bits have to be set up. The following is a list of these bits and their values:

In TMOD:	In TCON:	In IE:
GATE = 0	TRI = 1	ETI = 1
C/T = 1		EA = 1
M1 = 1		
M0 = 0		

Where "i" is the timer number being used as the external interrupt. The TMOD value would be 66 hexadecimal if both timers are being used as external interrupt sources, x6 hex for timer 0, and 6x hex for timer 1. The interrupt priority may also be set in the IP register.

A falling edge on the corresponding Timer 0 or Timer 1 input (T0 or T1) will cause the

counter to overflow and generate a timer interrupt. The counter will be automatically loaded with another FF from the reload register, so the interrupt can occur again as soon as the interrupt service routine completes. Counter/Timer operation is described in detail elsewhere in this manual.

SERIAL PORT CONFIGURATION

The serial port can be placed in mode 2, which is a 9-bit UART with the baud rate derived from the oscillator. The external interrupt is signaled through this port on the RxD receive data pin. Reception is initiated by a detected 1-to-0 transition at RxD. The signal must stay at 0 for at least five-eighths of a bit period for this level to be recognized. Refer to the description of baud rates to determine the length of a bit period at the oscillator frequency selected for the application. The input signal should remain low for at least one bit period and for not more than 9 bit periods.

To prepare the serial port for use as an external interrupt, the following bits must be set up:

In SCON:
SM0 = 1
SM1 = 0
SM2 = 0
REN = 1

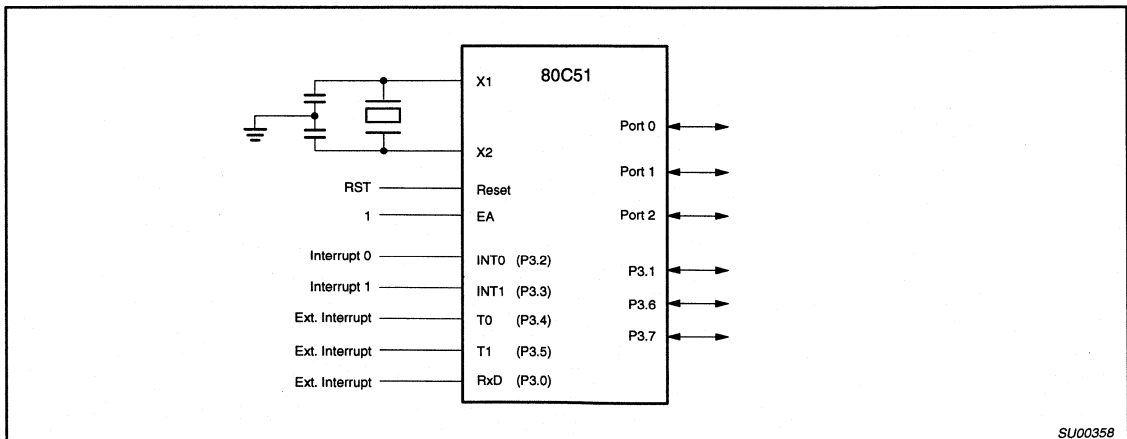


Figure 1. 80C51 Five Interrupt Configuration

SU00358

Using up to 5 external interrupts on 80C51 family microcontrollers

AN420

The Serial Port Interrupt is then used as a general-purpose interrupt. The contents of receive buffer should be ignored, and will subsequently be overwritten during the next interrupt.

Note that the response time for this input will be slower than for the Counter/Timer inputs. This is due to the fact that the RI is generated after the eighth serial data bit time after the falling edge on RxD.

; Demonstration program for five external interrupts.

\$MOD51

\$TITLE (Five Vectored External Interrupts)

; Interrupt Jump Table

```
ORG 0H           ;Reset
AJMP Setup
```

```
ORG 3H           ;External interrupt 0.
RETI             ;(not implemented in this demo)
```

```
ORG 0BH          ;Timer 0 interrupt.
AJMP Tim0
```

```
ORG 13H          ;External interrupt 1.
RETI             ;(not implemented in this demo)
```

```
ORG 1BH          ;Timer 1 interrupt.
AJMP Tim1
```

```
ORG 23H          ;Serial port interrupt.
AJMP Serial
```

; Begin setup code

```
Setup MOV SP,#7FH ;Initialize the stack pointer.
```

; Configure both timers

```
MOV TMOD,#66H   ;Put both counters into mode 2.
```

```
MOV A,#0FFH
```

```
MOV TL0,A       ;Load FF hex into both counters
```

```
MOV TH0,A
```

```
MOV TL1,A
```

```
MOV TH1,A
```

```
SETB ET0        ;Enable Timer 0 interrupt.
```

```
SETB ET1        ;Enable Timer 1 interrupt.
```

```
SETB TR0        ;Enable Timer 0 to run.
```

```
SETB TR1        ;Enable Timer 1 to run.
```

; Configure the serial port

```
SETB ES         ;Enable serial port interrupt.
```

```
MOV SCON,#90H  ;Put the serial port in mode 2.
```

```
SETB EA        ;Enable interrupt system.
```

```
Wait: NOP      ;Wait for an interrupt.
```

```
JMP Wait
```

```
Serial: NOP    ;Serial interrupt service routine.
```

```
CLR RI        ;Clear receiver interrupt flag.
```

```
RETI
```

```
Tim0: NOP     ;Timer 0 interrupt service routine.
```

```
RETI
```

```
Tim1: NOP     ;Timer 0 interrupt service routine.
```

```
RETI
```

```
END
```

8051 family warm boot determinations

AN424

DESCRIPTION

For some classes of applications, it may be desirable to know if the application of the reset signal to a microcontroller is due to an initial power-on sequence, or is the result of an external signal such as an operator pressing a reset pushbutton, or the result of a watchdog timer or similar event.

While there are perhaps numerous hardware solutions that can be employed, a simple software solution can offer a high degree of confidence in making this determination. The task is to determine the differences in state of resources internal to the microcontroller that would occur as a result of these two types of reset conditions. With respect to the 80C51 family of microcontrollers, on-chip resources consist of the special function registers (SFRs) and the internal data memory (RAM). Most of the SFR locations are initialized as a result of a reset condition and thus cannot be used for this determination. The data memory contents are unaffected by reset. Thus, valid data loaded into the RAM of the 80C51 while executing a program would not be affected by the application of an external reset signal provided the power source for the microcontroller has not been removed (as is the case for a "warm boot").

The contents of data memory as a result of an initial application of power, however, is indeterminate. While this effect has not been extensively characterized, empirical observation suggests that it is highly random in nature. If it is assumed, for the moment, that the behavior of a given byte of data memory is such that it will power-up with a

value that is totally random, then there is a one in eight chance that it will power-up with a predetermined value. If the assumption is extended to two bytes, a 16-bit number, then there is one in 2^{16} chance that both bytes will power-up with predetermined values. Extending this to four bytes results in a one in 2^{32} chance; a very small probability. This is the basis for the software determination of a warm or cold boot condition.

The technique consists of evaluating the contents of four consecutive bytes of data memory following a reset condition to determine whether these bytes had been previously loaded with known data values. If the contents of all four bytes match predetermined values, this is interpreted to be a warm boot condition. If there is no match, it is then interpreted to be a cold boot condition. At this point, it is necessary to load these four bytes with predetermined data to prepare for the possibility of a subsequent warm boot condition.

The software example included in this application brief can be used to perform this warm or cold boot determination.

The symbols WARM1 through WARM4 represent the predetermined values. The symbol WARM is the address of the first of the four consecutive bytes in data memory. It is set to 30H to avoid conflict with the four register banks, the stack, and the bit-addressable locations in data memory. The symbol WARMBT is a bit-addressable location used as a status bit. It is set as the

result of a warm boot and cleared as a result of a cold boot.

The label START is the location of the first instruction to be executed following a reset (address = 0000H). An instruction is located here to jump into the main body of the program to bypass the interrupt vector locations.

The main program body begins by loading register R0 with the address of the first byte in data memory to be evaluated. The contents of this first byte is compared with the first predetermined value. If there is no match, the conclusion is that it is a cold boot. However, if a match is found, this does not imply that it is a warm boot since all four bytes must match, and therefore the remaining three bytes must also be evaluated. Register R0 is incremented to point to the second byte and then compared to the second predetermined value. Comparison of the bytes proceeds until either a no match condition is found or until all four bytes have been evaluated successfully. If all four bytes compared favorable, then a status bit (WARMBT) is set to indicate a warm boot and the remainder of the application program is completed.

An unsuccessful comparison results in branching to the label COLD. This section of code clears the status bit (WARMBT) to indicate a cold boot, and loads the four bytes of data memory with the predetermined values preparing the system for a subsequent possible warm boot. Program flow then continues with the remainder of the application program.

8051 family warm boot determinations

AN424

```

1
2      ;warm boot application example
3
0030      4      WARM   EQU    30H           ;first location of the four bytes in RAM
0055      5      WARM1  EQU    55H           ;first predetermined value
00AA      6      WARM2  EQU    0AAH          ;second predetermined value
0033      7      WARM3  EQU    33H           ;third predetermined value
00CC      8      WARM4  EQU    0CCH          ;fourth predetermined value
0000      9      WARMBT  EQU    0           ;warm boot status bit
10
0000      11     ORG     0
12
0000 020026 13     START: JMP     MAIN           ;bypass interrupt vectors
14
0026      15     ORG     26H
16
0026 7830   17     MAIN:  MOV     R0,#WARM         ;pointer for first byte
0028 B65511 18     CJNE   @R0,#WARM1,COLD ;test first byte
002B 08     19     INC     R0                 ;pointer for second byte
002C B6AA0D 20     CJNE   @R0,#WARM2,COLD ;test second byte
002F 08     21     INC     R0                 ;pointer for third byte
0030 B63309 22     CJNE   @R0,#WARM3,COLD ;test third byte
0033 08     23     INC     R0                 ;pointer for fourth byte
0034 B6CC05 24     CJNE   @R0,#WARM4,COLD ;test fourth byte
0037 D200   25     SETB   WARMBT          ;this is a warm start
0039 02004B 26     JMP     INIT              ;continue with rest of application
003C C200   27     COLD:  CLR     WARMBT          ;this is a cold boot
003E 7830   28     MOV     R0,#WARM         ;pointer for first byte
0040 7655   29     MOV     @R0,#WARM1       ;load the four bytes for future test
0042 08     30     INC     R0                 ;
0043 76AA   31     MOV     @R0,#WARM2       ;
0045 08     32     INC     R0                 ;
0046 7633   33     MOV     @R0,#WARM3       ;
0048 08     34     INC     R0                 ;
0049 76CC   35     MOV     @R0,#WARM4       ;
004B      36     INIT:  ;continue with the application
      37
      38     END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

```

COLD . . . . . C ADDR 003CH
INIT . . . . . C ADDR 004BH
MAIN . . . . . C ADDR 0026H
START . . . . . C ADDR 0000H NOT USED
WARM . . . . . NUMB 0030H
WARM1 . . . . . NUMB 0055H
WARM2 . . . . . NUMB 00AAH
WARM3 . . . . . NUMB 0033H
WARM4 . . . . . NUMB 00CCH
WARMBT . . . . . NUMB 0000H

```

RAM loader program for 80C51 family applications

AN440

Author: Greg Goodhue

The following program allows an 80C51 family microcontroller to load most of its code into a RAM over a serial link after power up and execute out of the RAM for normal operation. This can allow a final product to have firmware updates done by a simple diskette mailing. Such a program is often called a "bootstrap loader".

For this example, it is assumed that the code download is done via a serial communication

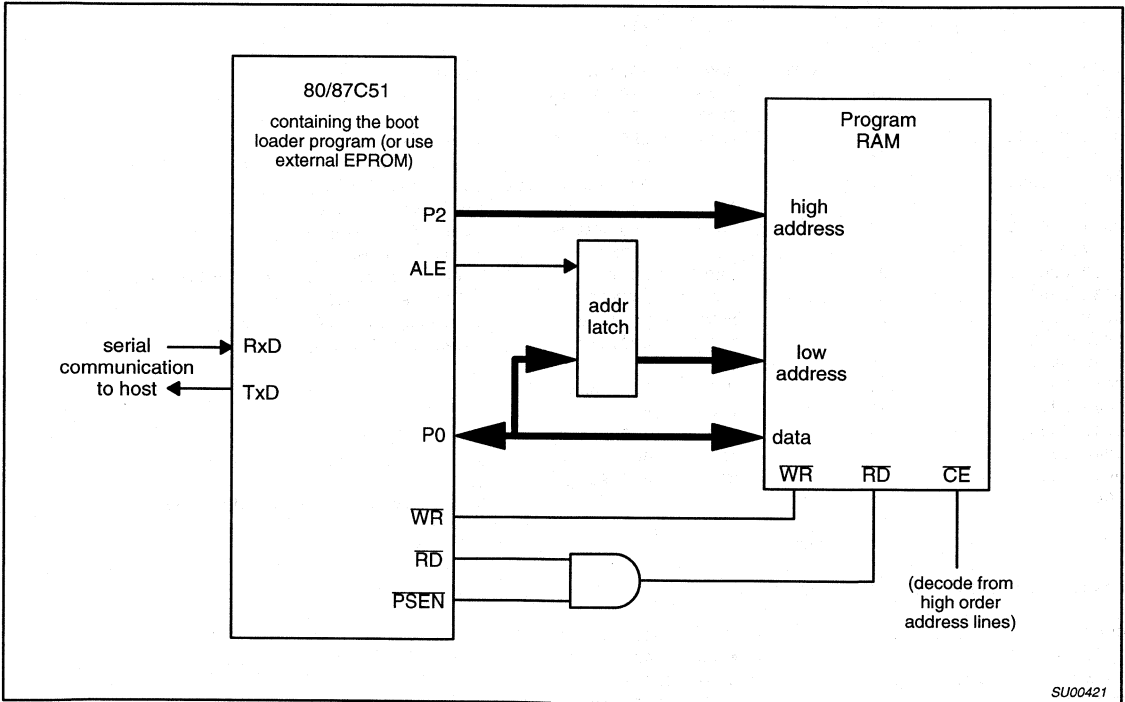
link, although the program could be adapted to other forms of download. The comments at the beginning of the listing are intended to document the program and its use completely.

An additional comment would be that any static routines (low level routines that are unlikely to change over time) can probably be put into the permanent program memory (on-chip or off-chip ROM or EPROM) along

with the bootstrap loader to save program RAM space for other things.

The source code file for this program is available for downloading from the Philips computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.

BASIC DIAGRAM OF RAM CONNECTIONS FOR THE BOOT LOADER



SU00421

RAM loader program for 80C51 family applications

AN440

```

;=====
;
;           Bootstrap Loader for Hexadecimal Files
;           written by G. Goodhue, Philips Electronics
;
; This program allows downloading a hexadecimal program file over an
; asynchronous serial link to a code RAM in an 80C51 system. The downloaded
; code may then be executed as the main program for the system. This technique
; may be used in a system that normally connects to a host PC so that the code
; may come from a disk and thus be easily updated. The system RAM must be
; wired to the 80C51 system so that it appears as both data and program memory
; (wire the RAM normally, but use the logical AND of RD and PSEN for the
; output enable.)
;
; To use the bootstrap program, an Intel Hex file is sent through the serial
; port in 8-N-1 format at 9600 baud. The baud rate and format may be altered
; by making small changes in the serial port setup routine (SerStart).
;
; Note that there is no hardware handshaking (e.g. RTS/CTS or XON/XOFF)
; implemented between the host and the bootstrap system. This was done to keep
; the protocol between the two systems as simple as possible.
;
; Since the bootstrap program does not echo the data file, there is no chance
; of an overrun unless the 80C51 is running very slowly and/or the
; communication is very fast. An 80C51 running at 11.0592 MHz (the most
; commonly used frequency in systems with serial communication) will be able
; to easily keep up with 38.4K baud communication without handshaking.
;=====
; The download protocol for this program is as follows:
;
; - When the bootstrap program starts up, it sends a prompt character ("=")
;   up the serial link to the host.
;
; - The host may then send the hexadecimal program file down the serial link.
;   At any time, the host may send an escape character (1B hex) to abort and
;   restart the download process from scratch, beginning from the "=" prompt.
;   This procedure may be used to restart if a download error occurs.
;
; - At the end of a hex file download, a colon (":") prompt is returned. If
;   an error or other suspicious circumstance occurred, a flag value will
;   also be returned as shown below. The flag is a bit map of possible
;   conditions and so may represent more than one problem. If an error
;   occurs, the bootstrap program will refuse to execute the downloaded
;   program.
;
; Exception codes:
;   01 - non-hexadecimal characters found embedded in a data line.
;   02 - bad record type found.
;   04 - incorrect line checksum found.
;   08 - no data found.
;   10 - incremented address overflowed back to zero.
;   20 - RAM data write did not verify correctly.
;
; - If a download error occurs, the download may be retried by first sending
;   an escape character. Until the escape is received, the bootstrap program
;   will refuse to accept any data and will echo a question mark ("??") for
;   any character sent.
;
; - After a valid file download, the bootstrap program will send a message
;   containing the file checksum. This is the arithmetic sum of all of the
;   DATA bytes (not addresses, record types, etc.) in the file, truncated to
;   16 bits. This checksum appears in parentheses: "(abcd)". Program
;   execution may then be started by telling the bootstrap program the
;   correct starting address. The format for this is to send a slash ("/")
;   followed by the address in ASCII hexadecimal, followed by a carriage
;   return. Example: "/8A31<CR>"

```

RAM loader program for 80C51 family applications

AN440

```
; - If the address is accepted, an at sign ("@" ) is returned before executing
; the jump to the downloaded file.
```

```
; The bootstrap loader can be configured to re-map interrupt vectors to the
; downloaded program if jumps to the correct addresses are set up. For
; instance, if the program RAM in the system where this program is to be used
; starts at 8000 hexadecimal, the re-mapped interrupts may begin at 8003 for
; external interrupt 0, etc.
```

```
=====
$Title(Bootstrap Loader for Hexadecimal Files)
$Date(04-13-92)
$MOD51
```

```
=====
;                               Definitions
=====
```

```
LF           EQU      0Ah           ; Line Feed character.
CR           EQU      0Dh           ; Carriage Return character.
ESC          EQU      1Bh           ; Escape character.
StartChar    EQU      ':'           ; Line start character for hex file.
Slash        EQU      '/'          ; Go command character.
Skip         EQU      13           ; Value for "Skip" state.

Ch           DATA    0Fh           ; Last character received.
State        DATA    10h           ; Identifies the state in process.
DataByte     DATA    11h           ; Last data byte received.
ByteCount    DATA    12h           ; Data byte count from current line.
HighAddr     DATA    13h           ; High and low address bytes from the
LowAddr      DATA    14h           ; current data line.
RecType      DATA    15h           ; Line record type for this line.
ChkSum       DATA    16h           ; Calculated checksum received.
HASave      DATA    17h           ; Saves the high and low address bytes
LASave      DATA    18h           ; from the last data line.
FilChkHi    DATA    19h           ; File checksum high byte.
FilChkLo    DATA    1Ah           ; File checksum low byte.

Flags        DATA    20h           ; State condition flags.
HexFlag      BIT      Flags.0       ; Hex character found.
EndFlag      BIT      Flags.1       ; End record found.
DoneFlag     BIT      Flags.2       ; Processing done (end record or some
; kind of error.

EFlags       DATA    21h           ; Exception flags.
ErrFlag1    BIT      EFlags.0       ; Non-hex character embedded in data.
ErrFlag2    BIT      EFlags.1       ; Bad record type.
ErrFlag3    BIT      EFlags.2       ; Bad line checksum.
ErrFlag4    BIT      EFlags.3       ; No data found.
ErrFlag5    BIT      EFlags.4       ; Incremented address overflow.
ErrFlag6    BIT      EFlags.5       ; Data storage verify error.
DatSkipFlag BIT      Flags.3        ; Any data found should be ignored.
```


RAM loader program for 80C51 family applications

AN440

```

;=====
;                               Reset and Interrupt Vectors
;=====
; The following are dummy labels for re-mapped interrupt vectors. The
; addresses should be changed to match the memory map of the target system.
ExInt0    EQU    8003h           ; Remap address for ext interrupt 0.
T0Int     EQU    800Bh           ; Timer 0 interrupt.
ExInt1    EQU    8013h           ; External interrupt 1.
T1Int     EQU    801Bh           ; Timer 1 interrupt.
SerInt    EQU    8023h           ; Serial port interrupt.

        ORG     0000h
        LJMP    Start           ; Go to the downloader program.

; The following are intended to allow re-mapping the interrupt vectors to the
; users downloaded program. The jump addresses should be adjusted to reflect
; the memory mapping used in the actual application.

; Other (or different) interrupt vectors may need to be added if the target
; processor is not an 80C51.

        ORG     0003h
;       LJMP    ExInt0           ; External interrupt 0.
        RETI

        ORG     000Bh
;       LJMP    T0Int           ; Timer 0 interrupt.
        RETI

        ORG     0013h
;       LJMP    ExInt1          ; External interrupt 1.
        RETI

        ORG     001Bh
;       LJMP    T1Int           ; Timer 1 interrupt.
        RETI

        ORG     0023h
;       LJMP    SerInt          ; Serial port interrupt.
        RETI

;=====
;                               Reset and Interrupt Vectors
;=====
Start:    MOV     IE,#0           ; Turn off all interrupts.
          MOV     SP,#5Fh         ; Start stack near top of '51 RAM.
          ACALL  SerStart        ; Setup and start serial port.
          ACALL  CRLF            ; Send a prompt that we are here.
          MOV     A,#'='         ; "<CRLF> ="
          ACALL  PutChar         ;
          ACALL  HexIn           ; Try to read hex file from serial port.
          ACALL  ErrPrt          ; Send a message for any errors or
          ; warnings that were noted.
          MOV     A,EFlags        ; We want to get stuck if a fatal
          JZ      HexOK          ; error occurred.

ErrLoop:  MOV     A,#'? '        ; Send a prompt to confirm that we
          ACALL  PutChar         ; are 'stuck'. " ? "
          ACALL  GetChar         ; Wait for escape char to flag reload.
          SJMP   ErrLoop

HexOK:    MOV     EFlags,#0       ; Clear errors flag in case we re-try.
          ACALL  GetChar         ; Look for GO command.
          CJNE   A,#Slash,HexOK  ; Ignore other characters received.
          ACALL  GetByte         ; Get the GO high address byte.
          JB     ErrFlag1,HexOK   ; If non-hex char found, try again.
          MOV     HighAddr,DataByte ; Save upper GO address byte.
          ACALL  GetByte         ; Get the GO low address byte.
          JB     ErrFlag1,HexOK   ; If non-hex char found, try again.
          MOV     LowAddr,DataByte ; Save the lower GO address byte.
          ACALL  GetChar         ; Look for CR.
          CJNE   A,#CR,HexOK     ; Re-try if CR not there.

```

RAM loader program for 80C51 family applications

AN440

```

; All conditions are met, so hope the data file and the GO address are all
; correct, because now we're committed.

        MOV     A,#'@" ; Send confirmation to GO. " @"
        ACALL  PutChar
        JNB    TI,$    ; Wait for completion before GOing.
        PUSH  LowAddr  ; Put the GO address on the stack,
        PUSH  HighAddr ; so we can Return to it.
        RET     ; Finally, go execute the user program!

;=====
;                               Hexadecimal File Input Routine
;=====

HexIn:   CLR     A      ; Clear out some variables.
        MOV     State,A
        MOV     Flags,A
        MOV     HighAddr,A
        MOV     LowAddr,A
        MOV     HASave,A
        MOV     LASave,A
        MOV     ChkSum,A
        MOV     FilChkHi,A
        MOV     FilChkLo,A
        MOV     EFlags,A
        SETB    ErrFlag4 ; Start with a 'no data' condition.

StateLoop: ACALL  GetChar ; Get a character for processing.
          ACALL  AscHex  ; Convert ASCII-hex character to hex.
          MOV    Ch,A    ; Save result for later.
          ACALL  GoState ; Go find the next state based on
          ; this char.
          JNB   DoneFlag,StateLoop ; Repeat until done or terminated.

          ACALL  PutChar ; Send the file checksum back as
          MOV    A,#'(' ; confirmation. " (abcd) "
          ACALL  PutChar
          MOV    A,FilChkHi
          ACALL  PrByte
          MOV    A,FilChkLo
          ACALL  PrByte
          MOV    A,#')'
          ACALL  PutChar
          ACALL  CRLF
          RET     ; Exit to main program.

; Find and execute the state routine pointed to by "State".

GoState: MOV     A,State ; Get current state.
        ANL     A,#0Fh  ; Insure branch is within table range.
        RL     A      ; Adjust offset for 2 byte insts.
        MOV    DPTR,#StateTable
        JMP    @A+DPTR ; Go to appropriate state.

StateTable: AJMP   StWait ; 0 - Wait for start.
          AJMP   StLeft ; 1 - First nibble of count.
          AJMP   StGetCnt ; 2 - Get count.
          AJMP   StLeft ; 3 - First nibble of address byte 1.
          AJMP   StGetAd1 ; 4 - Get address byte 1.
          AJMP   StLeft ; 5 - First nibble of address byte 2.
          AJMP   StGetAd2 ; 6 - Get address byte 2.
          AJMP   StLeft ; 7 - First nibble of record type.
          AJMP   StGetRec ; 8 - Get record type.
          AJMP   StLeft ; 9 - First nibble of data byte.
          AJMP   StGetDat ; 10 - Get data byte.
          AJMP   StLeft ; 11 - First nibble of checksum.
          AJMP   StGetChk ; 12 - Get checksum.
          AJMP   StSkip ; 13 - Skip data after error condition.
          AJMP   BadState ; 14 - Should never get here.
          AJMP   BadState ; 15 - " " " "

```

RAM loader program for 80C51 family applications

AN440

; This state is used to wait for a line start character. Any other characters
; received prior to the line start are simply ignored.

```
StWait:  MOV     A,Ch           ; Retrieve input character.
         CJNE   A,#StartChar,SWEX ; Check for line start.
         INC   State         ; Received line start.
SWEX:    RET
```

; Process the first nibble of any hex byte.

```
StLeft:  MOV     A,Ch           ; Retrieve input character.
         JNB   HexFlag,SLERR   ; Check for hex character.
         ANL   A,#0Fh         ; Isolate one nibble.
         SWAP  A               ; Move nibble to upper location.
         MOV   DataByte,A     ; Save left/upper nibble.
         INC   State         ; Go to next state.
         RET   State         ; Return to state loop.
SLERR:   SETB   ErrFlag1     ; Error - non-hex character found.
         SETB   DoneFlag     ; File considered corrupt. Tell main.
         RET
```

; Process the second nibble of any hex byte.

```
StRight: MOV     A,Ch           ; Retrieve input character.
         JNB   HexFlag,SRERR   ; Check for hex character.
         ANL   A,#0Fh         ; Isolate one nibble.
         ORL   A,DataByte     ; Complete one byte.
         MOV   DataByte,A     ; Save data byte.
         ADD   A,ChkSum       ; Update line checksum,
         MOV   ChkSum,A      ; and save.
         RET   State         ; Return to state loop.
SRERR:   SETB   ErrFlag1     ; Error - non-hex character found.
         SETB   DoneFlag     ; File considered corrupt. Tell main.
         RET
```

; Get data byte count for line.

```
StGetCnt: ACALL  StRight      ; Complete the data count byte.
         MOV   A,DataByte
         MOV   ByteCount,A
         INC   State         ; Go to next state.
         RET   State         ; Return to state loop.
```

; Get upper address byte for line.

```
StGetAd1: ACALL  StRight      ; Complete the upper address byte.
         MOV   A,DataByte
         MOV   HighAddr,A
         INC   State         ; Go to next state.
         RET   State         ; Return to state loop.
```

; Get lower address byte for line.

```
StGetAd2: ACALL  StRight      ; Complete the lower address byte.
         MOV   A,DataByte
         MOV   LowAddr,A
         INC   State         ; Go to next state.
         RET   State         ; Return to state loop.
```

; Get record type for line.

```
StGetRec: ACALL  StRight      ; Complete the record type byte.
         MOV   A,DataByte
         MOV   RecType,A
         JZ    SGRDat        ; This is a data record.
         CJNE  A,#1,SGRErr   ; Check for end record.
         SETB  EndFlag       ; This is an end record.
         SETB  DatSkipFlag   ; Ignore data embedded in end record.
         MOV   State,#11     ; Go to checksum for end record.
         SJMP SGREX
```

RAM loader program for 80C51 family applications

AN440

```

SGRDat:   INC      State           ; Go to next state.
SGREX:    RET                               ; Return to state loop.

SGRErr:   SETB    ErrFlag2         ; Error, bad record type.
          SETB    DoneFlag         ; File considered corrupt. Tell main.
          RET

; Get a data byte.

StGetDat: ACALL    StRight          ; Complete the data byte.
          JB      DatSkipFlag,SGD1 ; Don't process the data if the skip
                                     ; flag is on.
          ACALL    Store            ; Store data byte in memory.
          MOV     A,DataByte        ; Update the file checksum,
          ADD     A,FilChkLo        ; which is a two-byte summation of
          MOV     FilChkLo,A        ; all data bytes.
          CLR     A
          ADDC   A,FilChkHi
          MOV     FilChkHi,A
          MOV     A,DataByte

SGD1:     DJNZ    ByteCount,SGDEX   ; Last data byte?
          INC     State             ; Done with data, go to next state.
          SJMP   SGDEX2

SGDEX:    DEC     State             ; Set up state for next data byte.
SGDEX2:   RET                               ; Return to state loop.

; Get checksum.

StGetChk: ACALL    StRight          ; Complete the checksum byte.
          JNB    EndFlag,SGC1       ; Check for an end record.
          SETB   DoneFlag          ; If this was an end record,
          SJMP   SGCEX             ; we are done.

SGC1:     MOV     A,ChkSum          ; Get calculated checksum.
          JNZ    SGCErr            ; Result should be zero.
          MOV     ChkSum,#0         ; Preset checksum for next line.
          MOV     State,#0         ; Line done, go back to wait state.
          MOV     LASave,LowAddr    ; Save address byte from this line for
          MOV     HASave,HighAddr   ; later check.
SGCEX:    RET                               ; Return to state loop.

SGCErr:   SETB    ErrFlag3         ; Line checksum error.
          SETB    DoneFlag         ; File considered corrupt. Tell main.
          RET

; This state used to skip through any additional data sent, ignoring it.

StSkip:   RET                               ; Return to state loop.

; A place to go if an illegal state comes up somehow.

BadState: MOV     State,#Skip       ; If we get here, something very bad
          RET                               ; happened, so return to state loop.

; Store - Save data byte in external RAM at specified address.

Store:    MOV     DPH,HighAddr      ; Set up external RAM address in DPTR.
          MOV     DPL,LowAddr
          MOV     A,DataByte
          MOVX   @DPTR,A           ; Store the data.
          MOVX   A,@DPTR           ; Read back data for integrity check.
          CJNE   A,DataByte,StoreErr ; Is read back OK?

          CLR     ErrFlag4         ; Show that we've found some data.
          INC     DPTR             ; Advance to the next addr in sequence.
          MOV     HighAddr,DPH     ; Save the new address
          MOV     LowAddr,DPL
          CLR     A
          CJNE   A,HighAddr,StoreEx ; Check for address overflow
          CJNE   A,LowAddr,StoreEx ; (both bytes are 0).
          SETB   ErrFlag5         ; Set warning for address overflow.

```

RAM loader program for 80C51 family applications

AN440

```

StoreEx:    RET

StoreErr:   SETB    ErrFlag6      ; Data storage verify error.
            SETB    DoneFlag      ; File considered corrupt. Tell main.
            RET

;=====
;                               Subroutines
;=====

; Subroutine summary:

; SerStart - Serial port setup and start.
; GetChar  - Get a character from the serial port for processing.
; GetByte  - Get a hex byte from the serial port for processing.
; PutChar  - Output a character to the serial port.
; AscHex   - See if char in ACC is ASCII-hex and if so convert to hex nibble.
; HexAsc   - Convert a hexadecimal nibble to its ASCII character equivalent.
; ErrPrt   - Return any error codes to our host.
; CRLF     - output a carriage return / line feed pair to the serial port.
; PrByte   - Send a byte out the serial port in ASCII hexadecimal format.

; SerStart - Serial port setup and start.

SerStart:   MOV     A,PCON          ; Make sure SMOD is off.
            CLR     ACC.7
            MOV     PCON,A
            MOV     TH1,#0FDh      ; Set up timer 1.
            MOV     TL0,#0FDh
            MOV     TMOD,#20h
            MOV     TCON,#40h
            MOV     SCON,#52h     ; Set up serial port.
            RET

; GetByte - Get a hex byte from the serial port for processing.

GetByte:    ACALL   GetChar        ; Get first character of byte.
            ACALL   AscHex        ; Convert to hex.
            MOV     Ch,A          ; Save result for later.
            ACALL   StLeft       ; Process as top nibble of a hex byte.
            ACALL   GetChar      ; Get second character of byte.
            ACALL   AscHex        ; Convert to hex.
            MOV     Ch,A          ; Save result for later.
            ACALL   StRight      ; Process as bottom nibble of hex byte.
            RET

; GetChar - Get a character from the serial port for processing.

GetChar:    JNB     RI,$           ; Wait for receiver flag.
            CLR     RI            ; Clear receiver flag.
            MOV     A,SBUF        ; Read character.
            CJNE    A,#ESC,GCEX   ; Re-start immediately if Escape char.
            LJMP   Start
GCEX:      RET

; PutChar - Output a character to the serial port.

PutChar:    JNB     TI,$           ; Wait for transmitter flag.
            CLR     TI            ; Clear transmitter flag.
            MOV     SBUF,A        ; Send character.
            RET

; AscHex - See if char in ACC is ASCII-hex and if so convert to a hex nibble.
; Returns nibble in A, HexFlag tells if char was really hex. The ACC is not
; altered if the character is not ASCII hex. Upper and lower case letters
; are recognized.

```

RAM loader program for 80C51 family applications

AN440

```

AscHex:    CJNE    A,#'0',AH1      ; Test for ASCII numbers.
AH1:       JC      AHBAd           ; Is character is less than a '0'?
           CJNE    A,#'9'+1,AH2    ; Test value range.
AH2:       JC      AHVal09        ; Is character is between '0' and '9'?

           CJNE    A,#'A',AH3      ; Test for upper case hex letters.
AH3:       JC      AHBAd           ; Is character is less than an 'A'?
           CJNE    A,#'F'+1,AH4    ; Test value range.
AH4:       JC      AHValAF        ; Is character is between 'A' and 'F'?

           CJNE    A,#'a',AH5      ; Test for lower case hex letters.
AH5:       JC      AHBAd           ; Is character is less than an 'a'?
           CJNE    A,#'f'+1,AH6    ; Test value range.
AH6:       JNC    AHBAd           ; Is character is between 'a' and 'f'?
           CLR     C
           SUBB   A,#27h           ; Pre-adjust character to get a value.
           SJMP  AHVal09          ; Now treat as a number.

AHBad:     CLR     HexFlag         ; Flag char as non-hex, don't alter.
           SJMP  AHEx             ; Exit
AHValAF:   CLR     C
           SUBB   A,#7             ; Pre-adjust character to get a value.
AHVal09:   CLR     C
           SUBB   A,#'0'          ; Adjust character to get a value.
           SETB  HexFlag          ; Flag character as 'good' hex.
AHEx:      RET

```

; HexAsc - Convert a hexadecimal nibble to its ASCII character equivalent.

```

HexAsc:    ANL     A,#0Fh          ; Make sure we're working with only
           CJNE    A,#0Ah,HA1      ; one nibble.
           JC      HAVal09         ; Test value range.
           ADD    A,#7             ; Value is 0 to 9.
           ADD    A,#'0'          ; Value is A to F, extra adjustment.
HAVal09:   ADD    A,#'0'          ; Adjust value to ASCII hex.
           RET

```

; ErrPrt - Return an error code to our host.

```

ErrPrt:    MOV     A,#':'          ; First, send a prompt that we are
           CALL   PutChar         ; still here.
           MOV    A,EFlags        ; Next, print the error flag value if
           JZ     ErrPrtEx        ; it is not 0.
           CALL   PrByte
ErrPrtEx:  RET

```

; CRLF - output a carriage return / line feed pair to the serial port.

```

CRLF:     MOV     A,#CR
           CALL   PutChar
           MOV    A,#LF
           CALL   PutChar
           RET

```

; PrByte - Send a byte out the serial port in ASCII hexadecimal format.

```

PrByte:   PUSH    ACC              ; Print ACC contents as ASCII hex.
           SWAP   A
           CALL   HexAsc           ; Print upper nibble.
           CALL   PutChar
           POP    ACC
           CALL   HexAsc           ; Print lower nibble.
           CALL   PutChar
           RET

```

=====

END

IEEE Micro Mouse using the 87C751 microcontroller

AN443

Author: Tracy Ching

DESCRIPTION

Micro Mouse is an IEEE contest first proposed by the author of IEEE Spectrum in 1977. It consists of an autonomous robot known as a "mouse" which navigates through a maze of 256 two-inch-high, seven-inch squares in a 16 × 16 arrangement. The robot is self powered and has no knowledge of the maze configuration prior to releasing it in the maze. The first time it is released into the maze, its prime objective is to find a path from the starting square which is located in a corner of the maze to the destination square which can be located in the center or a different corner depending on competition level. The destination square for the advanced level can be found only by using a smart algorithm which will make the mouse gravitate towards the center without becoming lost. The destination square for a novice contest can be found by using a wall hugging algorithm. The analogy for this algorithm is to imagine a blind person holding their right hand out against a wall and following the walls until they reach the destination. A maximum of ten runs or 15 minutes is given to each mouse. Leaving the starting square constitutes one run. The mouse with the fastest run time from the starting square to the destination square wins.

The contest is held with different competition levels for novices and advanced. The novice level is typically held for college students trying to exercise newly acquired hardware and software skills, whereas the advanced level encompasses international talent and may require the implementation of a proportional integral derivative (PID) controller in software to utilize commutated or brushless DC motors and complex navigation algorithms.

DESIGN OBJECTIVES

Several design objectives were as follows: minimize weight, minimize part size and count, minimize power consumption which allows the use of smaller and lighter batteries, minimize cost and maximize speed.

The main objective was to keep the total weight at a minimum to obtain the fastest acceleration from the stepper motors and to reduce wheel slippage during deceleration and turning. The objectives are inter-related such that changing one will affect the other. Hence, having a low part count means lighter weight, faster acceleration and lower cost.

A typical mouse may consist of a microcontroller with supporting memory and GLU logic to interface the motor controllers

and sensors. The 87C751 is suitable for this application with its small size. The need for external RAM is eliminated by using the internal RAM to store minimum maze information suitable for the novice level. Nickel cadmium batteries are used because of the cost, size and power density obtainable versus other battery types. Nickel metal hydride was not available during development but would be a good choice over nickel cadmium batteries.

The 87C751 Microcontroller

The 87C751 is an 8-bit microcontroller based on the 8051 microcontroller family. It is code compatible with the exception of the MOVX, LJMP, and LCALL instructions. The MOVX instruction and external memory accesses are not supported. LJMP and LCALL instructions are not needed since AJMP and ACALL can reach the entire program memory range (2k bytes) of the 751. The 87C751 contains a 2k × 8 EPROM, a 64 × 8 RAM, 19 I/O lines, a 16 bit auto-reload counter/timer, a fixed rate timer, a five source fixed priority interrupt structure, a bidirectional Inter-Integrated circuit (I²C) bus interface, and an internal oscillator. The 87C751 comes in an erasable quartz package (87C751), one time programmable (87C751), and mask ROM (83C751).

HARDWARE DESCRIPTION

Figure 1 is a schematic diagram of the mouse. The stepper motors are 4 volts 0.95 amperes per coil giving about 14 oz-inches of torque. Each motor is driven by an Allegro UCN-5804B unipolar stepper motor translator/driver which contains the sequencing logic and high current darlington outputs. The sequencing logic only requires clock, direction of rotation, and output enable signals from the 87C751 which relieves the chore of having to cycle through a sequencing table for both motors therefore reducing code size. Fast recovery diodes are used to protect the darlington outputs from negative voltage due to motor winding flyback.

The infrared (IR) emitters are Optek OP-240A. The sensors are Optek OPL-560-OC which have a built-in light amplifier and TTL open collector output. Each sensor bank is enabled via a high side P-channel MOSFET. A 74LS04 (U4) is used to drive the P-channel MOSFET. Data from each bank of eight IR sensors is fed into port 3. By using open collector output sensors, the need for latching the data using a 3-State buffer is eliminated. Port pins P0.0 and P0.1 are used for enabling either the left or right

sensor bank via the 74LS04 (U4) and the MOSFETs. Each sensor bank hangs over the two inch high walls whereby the light emitted from the OP-240 is reflected into the OPL-560-OC if a wall is present. Two sensor pairs in the middle of the array are typically sensing the presence of a wall and the remaining sensors are used for guidance to keep the mouse running parallel to the walls.

A 74LS573 connected to port 3 which latched data into eight LEDs was used in the debugging process. P1.7 was used on the latch signal for the 74LS573. Since the eight LEDs and latch were not needed for the final product, they were removed thus reducing weight and battery drain. LEDs D1 and D2 were also used in the debugging process and are currently used for visual feedback when selecting options during operation.

Power for the digital circuitry is fed by an 8.4 volt NiCAD battery pack, packaged in a 9 volt battery case, via a 7805 regulator. The circuit can run constantly with the sensors taking "snapshots" of the walls intermittently for at least 30 minutes satisfying the 15 minute maximum requirement. Power for the motors is fed by four "A" size NiCAD cells which have enough energy to run the motors for 15 minutes before becoming useless at about 1 volt per cell.

TASK PRIORITIES

Several tasks take place during program execution which are as follows in order of importance: pulsing the stepper motors according to a velocity profile table, gathering sensor data, deciding whether to accelerate, decelerate, turn left or right.

In-line coding is used to avoid calling subroutines that cause the program counter to be pushed onto the stack and use valuable RAM for the turn decisions. Interrupt subroutines are an exception.

Interrupt timer 0 (T0) vectors to the routine which supplies a pulse to the stepper motor drivers. This is the most important task because the stepper motors require a smooth train of pulses in order to prevent jerky or sporadic motions. Thus, T0 must have the highest priority and must not be interrupted. Although timer 0 is a 16 bit timer, only eight bits are used with the higher byte set to FFH. T0 takes care of pulsing the left and right motor drivers at different times by using two external registers which are used as prescalers. Each motor can be assigned a different prescale value via the assigned register in order to step each motor at a different step rate.

IEEE Micro Mouse using the 87C751 microcontroller

AN443

The velocity profile table for T0 was designed using a spreadsheet program with visual graphing. This aided the ability to derive an exponential table for the fastest stepper motor acceleration using qualitative observations.

The first part of the in-line code contains the routine to accelerate the mouse from a stopped position. A pointer for each motor is incremented until the end of the velocity profile table is reached. During acceleration, the left and right sensors are strobed and stored after each step caused by T0.

The routine that strobes the sensors for wall data returns several results through the use of flags. The routine first stores the sensor data in registers. This information is used to determine if the sensor array or mouse is too far right, left, or aligned in a square and to set the corresponding flags. It also returns the presence of a front wall using the innermost sensor pairs. The deceleration routine is similar to the acceleration routine except the pointers decrement through the velocity profile table skipping a few values each time. Deceleration uses less steps than acceleration.

The routine which decides whether to continue acceleration, deceleration, or turn left or right keeps track of step count or position of the mouse within a square in order to store wall information at the proper time. After the previous routines have stored the data for the front, left, and right walls, a decision is obtained. If the mouse is not in a back-up mode, the wall information, status of the back-up flag and left/right algorithm flag forms an offset byte. If the mouse is in a back-up mode, the decision from above plus the previous decision on the stack is used to form the offset byte. This offset is stored in the accumulator. The decision which contains the direction to turn is obtained using the MOVC instruction which points to the table using the data pointer. The logic table is shown in Tables 1 and 2 below.

External interrupt 0 (INT0) vectors to the routine which brings the mouse to a halt. INT0 subroutine disables T0, sets output enable high on the stepper motor drivers, and sets the return address for the program counter to 0000H. This causes the mouse to restart program execution without losing the internal RAM.

**Table 1. Logic Table:
Non-Back-Up Mode**

(1)	(2)	(3)	(4)
R	NONE	R	push R onto stack
R	F	R	push R onto stack
R	R	S	push S onto stack
R	R, F	L	push L onto stack
R	L	R	push R onto stack
R	L, F	R	push R onto stack
R	L, R	S	ignore
R	L, R, F	180	turn on B-U flag
L	NONE	L	push L onto stack
L	F	L	push L onto stack
L	R	L	push L onto stack
L	R, F	L	push L onto stack
L	L	S	push S onto stack
L	L, F	R	push R onto stack
L	L, R	S	ignore
L	L, R, F	180	turn on B-U flag

NOTES:

Abbreviations used:

R = right

L = left

F = front

S = straight

B-U = back-up flag

stack = RAM used for storing decisions (not for the program counter)

Non back-up mode:

(1) wall hugging algorithm (user selected)

(2) walls surrounding present square

(3) direction to turn

(4) operations to perform

**Table 2. Logic Table:
Back-Up Mode**

(1)	(2)	(3)	(4)
R	R	R	push S onto stack, clear B-U
R	L	L	pop stack only
R	S	S	push L onto stack, clear B-U
L	R	R	pop stack only
L	L	L	push S onto stack, clear B-U
L	S	S	push R onto stack, clear B-U
S	R	R	push L onto stack, clear B-U
S	L	L	push R onto stack, clear B-U
S	S	S	pop stack only

NOTES:

Abbreviations used:

R = right

L = left

F = front

S = straight

B-U = back-up flag

stack = RAM used for storing decisions (not for the program counter)

Back-up mode:

(1) previous decision from top of stack

(2) decision from the above table for this current square

(3) direction to turn (should be equal to (2))

(4) operations to perform (all operations require lowering the stack by one before pushing data)

IEEE Micro Mouse using the 87C751 microcontroller

AN443

OPERATION

Reset

True reset is accomplished only during power on. After completing the maze, the operator brings the mouse to a stop using the start/stop switch which effectively disables the motors and resets the program counter to 0000H, restarting the mouse's program with the exception that the RAM contains vital maze information.

Starting/Stopping

After power-up, the mouse remains idle with the motors and sensors disabled, awaiting an interrupt from switch SW1. The first time the switch is pressed, the mouse will start. The second time the switch is pressed, the mouse will stop and the cycle repeats.

Selecting right or left wall hugging

After power on reset, the program defaults to right wall hugging. Pressing switch SW2 will cause the mouse to follow the left walls. If it is pressed again, it will follow the right walls— toggling between the left and right wall hugging algorithm each time it is pressed.

Increasing motor speed

After completing the first run, pressing switch SW2 causes the timer value to increase by an index of one, increasing the speed of the motors. This allows the mouse to be run at increasing speeds each run in order to attain the fastest possible time before crashing into

a wall or incurring a condition in the stepper motors called "pull-out". Pull-out is a condition wherein the fields are changing in the coils faster than the rotor can maintain synchronism with the coils, causing the rotor to stall.

SOFTWARE LIMITATIONS

The 87C751 has 64 bytes of RAM. The program requires 31 bytes of RAM leaving 33 bytes for storing decisions. One hundred thirty two decisions can be stored in 33 bytes of RAM since four decisions are stored per byte. Although there are 16 times 16 squares with possibly 256 decisions to be made, this condition is not very probable since every square typically is not made into a turning point. Long straight ways are used as well as turning points. The probability that the RAM will fill to the maximum capacity is very slim in the novice level where there are typically 50 decisions to be made. Hence, RAM which can hold 132 decisions is adequate but not infallible.

FUTURE ENHANCEMENTS

Although the 87C751 has minimal RAM, I²C RAM could be easily added by switching the MOSFET drivers using the LED port pins and placing the I²C RAM on port pins P0.0 and P0.1. This would allow the implementation of a full mapping algorithm thus allowing entry to the advanced class. The code size for the I²C routines and recursive algorithm to find

the center of the maze would add about 800 bytes more of code. Since some of the code for the novice class would be eliminated, it would bring the total code size to less than 2k bytes. A few maze solving algorithms have been implemented successfully on other mice. These are the flooding or Bellman's algorithm, backtracking algorithm and others. The first two algorithms are recursive, thus the code sizes are quite small.

Another added feature would be the ability to execute a rounded turn rather than pivot. This requires a more complex navigation scheme and velocity profiler to make the motors turn at differing speeds in order to make the rounded turn.

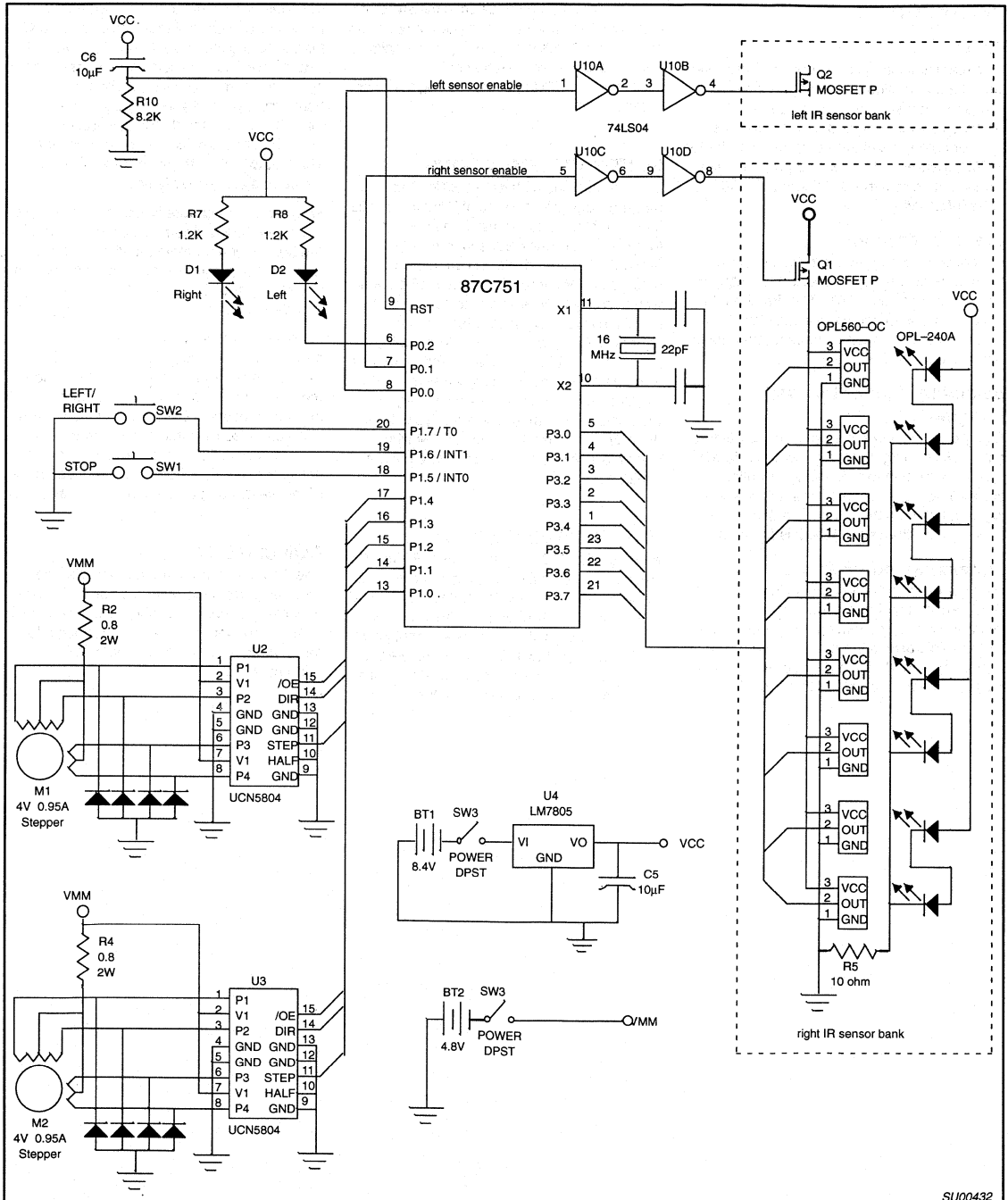
In addition to the enhancements aforementioned, methods to track a wall through the use of an A/D converter which measures the reflectivity strength from infrared sensors pointing at the sides of the walls can also be implemented on the I²C bus. This would eliminate the array of sensors hanging in front of the mouse on top of the walls thus decreasing the weight.

CONCLUSION

The 87C751 microcontroller provides the required computing resources and I/O ports necessary to control a robotics device known as a "Micro Mouse". The I²C interface allows for an abundance of variations on this robotics device.

IEEE Micro Mouse using the 87C751 microcontroller

AN443



SU00432

Figure 1. Schematic Diagram of the Maze Mouse

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 1

```

LOC  OBJ          LINE      SOURCE
1          ;*****
2          ; 87C751 Micro Mouse version 4.0 started 9-27-91      *
3          ;                               version 4.3 finished 4-16-92      *
4          ; Algorithms and programming by Tracy Ching          *
5          ; Some symbols commented out for use as in-line coding *
6          ; rather than being used as subroutines (S.R.)       *
7          ;*****
8
9 +1 $include      (751.equ)
=1 10          ;These are all RAM addresses
=1 11          ;equate table of constants
=1 12          ;ind_reg      equ    00          ;used for ind addr of regs
=1 13          ;map_org      equ    01          ;pointer for storing map of maze
=1 14          ;              equ    02          ;decide storage-local,
=1 15          ;              equ    03          ;do180, turn90 use it
=1 16          ;              equ    04          ;snapshot R wall storage
=1 17          ;              equ    05          ;snapshot L wall storage
=1 18          ;              equ    06          ;store_val, decel, int 1, gen purp
=1 19          ;              equ    07          ;pause setting, gen purp
=1 20
0008      =1 21          step_count    equ    08h          ;step count used by decide
0009      =1 22          temp_reg1     equ    09h          ;
000A      =1 23          ls373        equ    0ah          ;storage for 74LS373 debug data leds
000B      =1 24          count_fw     equ    0bh          ;count for wall, step count
000C      =1 25          l_timr       equ    0ch          ;value to be used in the isr
000D      =1 26          r_timr       equ    0dh          ;
000E      =1 27          l_ptr        equ    0eh          ;points at accel table
000F      =1 28          r_ptr        equ    0fh          ;
0010      =1 29          map_offset    equ    10h          ;points to the specific two bits in map ptr
=1 30          ;          equ    11h
=1 31          ;          equ    12h
=1 32          ;          equ    13h
=1 33          ;          equ    14h
=1 34          ;          equ    15h
=1 35          ;          equ    16h
=1 36          ;          equ    17h
=1 37
=1 38          ;PCON      EQU    87H
=1 39          ;TA        EQU    0C7H
008B      =1 40          rtl          equ    8bh
008D      =1 41          rth          equ    8dh
=1 42
=1 43
=1 44          ;flag declarations, they are erased after every restart
0020      =1 45          ss_bits      equ    20h          ;sens flag reg
0000      =1 46          too_r        bit    20h.0        ;too close to right wall
0001      =1 47          too_l        bit    20h.1        ;              left wall
0002      =1 48          r_wall      bit    20h.2        ;right wall present, snapshot storage
0003      =1 49          l_wall      bit    20h.3        ;left wall present, snapshot storage
0004      =1 50          f_wall      bit    20h.4        ;front wall present, snapshot storage

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

```

MCS-51 MACRO ASSEMBLER      751MAIN                                04/16/92  PAGE    2

LOC OBJ          LINE      SOURCE
0005          =1    51    aligned      bit    20h.5    ;sensors detect OK straightness
0006          =1    52    far_r        bit    20h.6    ;used to sync step count
0007          =1    53    far_l        bit    20h.7    ;ditto - left
              =1    54
0008          =1    55    r_turn      bit    21h.0    ;decide sets this for turn S.R.
0009          =1    56    l_turn      bit    21h.1    ;ditto
000A          =1    57    s_s_int     bit    21h.2    ;start stop bit
000B          =1    58    time_int    bit    21h.3    ;timer 0 int has occurred
000C          =1    59    r_int       bit    21h.4    ;r motor was stepped
000D          =1    60    l_int       bit    21h.5    ;l mot
000E          =1    61    slow_r      bit    21h.6    ;make int_0 incr timer flag
000F          =1    62    slow_l      bit    21h.7    ;ditto left
              =1    63
0010          =1    64    get_out     bit    22h.0    ;used to flag accel to end so decel can do
0011          =1    65    r_decide    bit    22h.1    ;r wall present, decide storage
0012          =1    66    l_decide    bit    22h.2    ;l wall present, decide storage
0013          =1    67    f_decide    bit    22h.3    ;f wall present, decide storage
0014          =1    68    make_180    bit    22h.4    ;decide says do 180
0015          =1    69    prev_r      bit    22h.5    ;decide uses for detect wall transition
0016          =1    70    prev_l      bit    22h.6    ;ditto left
0017          =1    71    cw180       bit    22h.7    ;toggle dir of 180
              =1    72
0018          =1    73    det_L2H     bit    23h.0    ;used to detect the low to high wall
0019          =1    74    look4f      bit    23h.1    ;skips to look for front wall
001A          =1    75    back_up     bit    23h.2    ;tells decide that it is backing up
001B          =1    76    look        bit    23h.3    ;start looking at wall
001C          =1    77    count_en    bit    23h.4    ;enables step counting
001D          =1    78    skip_decide bit    23h.5    ;TEMPORARY (testing only)
001E          =1    79    genp2       bit    23h.6    ;chk R or L sens flag
001F          =1    80    genp1       bit    23h.7    ;gen purpose, watch for conflicts betwx S.R.s
              =1    81
              =1    82
              =1    83
              =1    84    ;these are the flags that don't get erased after restarting
0020          =1    85    done        bit    24h.0    ;tells the prog that it has gone thru
0021          =1    86    l_r_bit     bit    24h.1    ;hug left or right bit, set=left, clr=right
0022          =1    87    temp_bit1   bit    24h.2    ;local bit var
              =1    88
              =1    89
              =1    90    ;hardware definitions
0082          =1    91    l_led       bit    p0.2
0097          =1    92    r_led       bit    p1.7
0090          =1    93    mot_en      bit    p1.0
0081          =1    94    r_sens      bit    p0.1
0080          =1    95    l_sens      bit    p0.0
0092          =1    96    r_step      bit    p1.2
0091          =1    97    r_dir       bit    p1.1
0094          =1    98    l_step      bit    p1.4
0093          =1    99    l_dir       bit    p1.3
0095          =1   100    s_s_sw      bit    p1.5
0096          =1   101    l_r_sw      bit    p1.6
00B0          =1   102    sensors     equ    p3
              103 +1    $include      (extrn751.tab)
              =1   104    ;These are all of the mouse constants.
              =1   105

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 3

```

LOC  OBJ          LINE      SOURCE
=1  106      extrn number      (timer_reload)
=1  107      extrn number      (pivot_speed)
=1  108      extrn number      (acc_steps)
=1  109      extrn number      (half_way)
=1  110      extrn number      (hold_val)
=1  111      extrn number      (decel_acc)
=1  112      extrn number      (decel_steps)
=1  113      extrn number      (decel_decr)
=1  114      extrn number      (steps90)
=1  115      extrn number      (half_90)
=1  116      extrn number      (look90)
=1  117      extrn number      (steps180)
=1  118      extrn number      (half_180)
=1  119      extrn number      (look180)
=1  120      extrn number      (map_org_addr)
=1  121      extrn number      (sens_pat)
=1  122      extrn number      (pause_val)
=1  123      extrn number      (w2nw_cnt)
=1  124      extrn number      (chk4fr)
125
0000          126          org          00
0000 0115    127          ajmp         main
0003          128          org          03
0003 618D    129          ajmp         int_0
000B          130          org          0bh
000B 6123    131          ajmp         tim_0
0013          132          org          13h
0013 6164    133          ajmp         int_1
134
135
136          ;*****
137          ; This section initializes the registers. Note that the maze info *
138          ; is not cleared. *
139          ;*****
140
0015 C220    141      main:          clr          done          ;start intial run
0017 758B00  F  142          mov         r1l,#timer_reload ;reload value for 751
001A 758DFE  F  143          mov         rth,#0ffh          ;
144
001D D281    145      main_1:       setb        r_sens
001F D280    146          setb        l_sens
0021 43907F  147          orl         pl,#7fh          ;lines high
0024 752000  148          mov         20h,#0          ;clear all flags except for
0027 752100  149          mov         21h,#0          ; location 24h bits
002A 752200  150          mov         22h,#0
002D 752300  151          mov         23h,#0
0030 7810    152          mov         r0,#10h          ;clear out 10h to 0h ram
0032 7600    153      main_0:       mov         @r0,#0          ; clears the regs
0034 DBFC    154          djnz        r0,main_0
155          ;initialize values
0036 758125  156          mov         sp,#25h          ;start of STACK
0039 7900    F  157          mov         r1,#map_org_addr;start of mapping memory
003B 75A887  158          mov         ie,#10000111b ;enable ints
003E 300AFD  159          jnb         s_s_int,$          ;stay here until start pressed
160

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 4

```

LOC  OBJ          LINE      SOURCE
161
162                ;*****
163                ; This section accelerates the mouse thru the velocity table.      *
164                ; The table value is lowered if the mouse is too close to a wall.  *
165                ;*****
166
167                accel:                ;loads acc_table and goes from there
0041  71B7          168                acall  snapshot                ;
0043  A202          169                mov   c,r_wall                ; check the walls
0045  9215          170                mov   prev_r,c
0047  A203          171                mov   c,l_wall                ; for both sides
0049  9216          172                mov   prev_l,c
004B  750841        173                mov   step_count,#65        ; start at 75stps, sens mid way
004E  750E00        174                acc_aa:  mov   l_ptr,#0
0051  750F00        175                mov   r_ptr,#0                ;clear offsets
0054  750DFE        176                mov   r_timr,#0feh        ;even up the timers to step
0057  750CFE        177                mov   l_timr,#0feh        ; evenly
005A  D28C          178                setb  tr0                    ;enable timer int
179
005C  300BFD        180                acc_0:  jnb   time_int,$            ;stay here until int happens
005F  C20B          181                clr   time_int
0061  71B7          182                acall snapshot                ;get status of too_l too_r
183
184                ;right routine
0063  300C21        185                jnb   r_int,acc_1            ;skip if right hasn't been stepped yet
0066  C20C          186                clr   r_int                ;ackn r int
0068  E50F          187                mov   a,r_ptr                ;check to see if at end of acc
006A  B40002        F 188                cjne  a,#acc_steps,acc_1c
006D  0171          189                ajmp  acc_1b
006F  050F          190                acc_1c: inc  r_ptr                ; point to next accel value
0071  300113        191                acc_1b: jnb   too_l,acc_1            ;if not too left then fall thru
0074  C3            192                clr   c                    ;subtr halfway from ptr
0075  E50F          193                mov   a,r_ptr
0077  9400          F 194                subb  a,#half_way            ;C set if ptr < half_way
0079  E50D          195                mov   a,r_timr
007B  4006          196                jc    acc_1a
007D  2400          F 197                add   a,#hold_val            ; slow down even more
007F  F50D          198                mov   r_timr,a                ;load it back into timer decel value
0081  0187          199                ajmp  acc_1                ;get out
0083  2400          F 200                acc_1a: add  a,#dec_acc            ;dec_acc used when going fast
0085  F50D          201                mov   r_timr,a                ;load it back into timer decel value
202
203                ;left routine
0087  300D21        204                acc_1:  jnb   l_int,acc_2            ;if L hasn't been stepped, skip
008A  C20D          205                clr   l_int                ;ackn l int
008C  E50E          206                mov   a,l_ptr                ;check to see if at end of acc
008E  B40002        F 207                cjne  a,#acc_steps,acc_2c
0091  0195          208                ajmp  acc_2b
0093  050E          209                acc_2c: inc  l_ptr                ; point to next accel value
0095  300013        210                acc_2b: jnb   too_r,acc_2            ;if not too right then fall thru
0098  C3            211                clr   c                    ;subtr halfway from ptr
0099  E50E          212                mov   a,l_ptr
009B  9400          F 213                subb  a,#half_way            ;C set if ptr < half_way
009D  E50C          214                mov   a,l_timr
009F  4006          215                jc    acc_2a

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 5

```

LOC OBJ          LINE    SOURCE
00A1 2400      F      216      add    a,#hold_val    ; slow down even more
00A3 F50C          217      mov    l_timr,a      ;load it back into timer decel value
00A5 01AB          218      ajmp   acc_2         ;get out
00A7 2400      F      219      acc_2a: add    a,#dec_acc    ;dec_acc used when going fast
00A9 F50C          220      mov    l_timr,a      ;load it back into timer decel value
                221
                222      acc_2:
                223
                224
                225      ;*****
                226      ; This section is the "decision-maker". It makes the decision to *
                227      ; quit acceleration, turn left or right, make a 180 pivot, store *
                228      ; a decision, or get a decision from the stack. *
                229      ;*****
                230
                231      ;decide:
                232      ;r_turn, l_turn, get_out, make_180 as public bits
                233      ;r_decide, l_decide, f_decide as local bits
                234      ;This S.R. decides when to start the decel using a step count
                235      ;when it sees a front wall or when it is time to turn. It also
                236      ;decides which way to turn on a mapping run or a final run.
                237      ;1) look for a wall to no wall transition
00AB 201853      238      jb    det_L2H,di_3_1 ;skip the stuff below if set
00AE A215      239      mov    c,prev_r
00B0 B002      240      anl   c,/r_wall      ;check for r wall transition
00B2 5003      241      jnc   di_1           ;if no transition goto di 1
00B4 750800     242      mov    step_count,#0 ;start counting
00B7 A216      243      di_1: mov    c,prev_l
00B9 B003      244      anl   c,/l_wall      ;check for l wall transition
00BB 5003      245      jnc   di_2           ;if no transition goto di 2
00BD 750800     246      mov    step_count,#0 ;start counting
                247      di_2:
                248      ;2) now detect step count
00C0 E508      249      mov    a,step_count  ;check to see if
00C2 C3        250      clr    c
00C3 9454      251      subb   a,#84         ; 80 <= steps <= 83
00C5 5013      252      jnc   di_3
00C7 E508      253      mov    a,step_count
00C9 C3        254      clr    c
00CA 9450      255      subb   a,#80
00CC 400C      256      jc    di_3
00CE A202      257      mov    c,r_wall      ;store R wall into decide storage
00D0 9211      258      mov    r_decide,c    ; for use later in CASE
00D2 A203      259      mov    c,l_wall      ;
00D4 9212      260      mov    l_decide,c    ; when CASE statment is built below
00D6 C21D      261      clr    skip_decide
00D8 2197      262      di_2_0: ajmp   di_9      ;get out since did above stuff
                263
                264      di_3:
                265      ;3)detect no wall to wall (lo to hi) and check for front wall
                266      ;count must be greater than 120 to allow for spaces in posts
00DA E508      266      mov    a,step_count
00DC C3        267      clr    c
00DD 9478      268      subb   a,#120
00DF 4020      269      jc    di_3_1
00E1 A215      270      mov    c,prev_r      ;the prev should be low

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 6

```

LOC  OBJ          LINE    SOURCE
00E3  A002         271      orl    c,/r_wall      ;check for r wall transition
00E5  400A         272      jc     di_3_0         ;IF no lo to hi then di_3_0
                                273      ;found lo to hi, set step counter
00E7  750800       F 274      mov    step_count,#w2nw_cnt
00EA  C213         275      clr    f_decide
00EC  750B00       276      mov    count_fw,#0    ;clr count for use in decel S.R.
00EF  D218         277      setb   det_L2H
00F1  A216         278      di_3_0: mov    c,prev_l      ;the prev should be low
00F3  A003         279      orl    c,/l_wall     ;check for l wall transition
00F5  400A         280      jc     di_3_1         ;IF no lo to hi then di_3_1
                                281      ;found lo to hi, set step counter
00F7  750800       F 282      mov    step_count,#w2nw_cnt
00FA  C213         283      clr    f_decide
00FC  750B00       284      mov    count_fw,#0
00FF  D218         285      setb   det_L2H
                                286      di_3_1: ;condition that will get you to di_3_9
                                287      ;det_L2H & count > 162
0101  A204         288      mov    c,f_wall      ; trigger from extrn lite during
0103  9213         289      mov    f_decide,c    ; other points in square
0105  400E         290      jc     di_3_9         ;get out if meet condition ELSE contnu
0107  3018CE       291      jnb   det_L2H,di_2_0 ;this section checks second cond
                                292
010A  E50B         293      mov    a,count_fw
010C  B40002       F 294      cjne  a,#chk4fr,di_3_2
010F  2115         295      ajmp  di_3_9
0111  050B         296      DI_3_2: inc   count_fw      ;
0113  2197         297      ajmp  di_9           ;get out
                                298
0115  201DC0       299      di_3_9: ;at this point we have detected it's time to stop, store in RAM, and
0118  D21D         300      ;execute plan.... Therefore set bit to skip all of this
                                301      jb    skip_decide,di_2_0
                                302      setb  skip_decide
                                303
011A  C218         304      clr    det_L2H
011C  750800       305      mov    step_count,#0
011F  202054       306      jb    done,dif_0     ;IF done THEN dif_0 ELSE fall thru
                                307
0122  7400         308      di_4: ;4) at this point F L R are loaded, set up CASE table
                                309      ; L=01 R=10 S=11 stop-mouse=00 all represented as two bits.
0124  A221         310      mov    a,#0
0126  33          311      mov    c,l_r_bit     ;if no back up then we have this...
0127  A212         312      rlc    a              ; |
0129  33          313      mov    c,l_decide   ; |
012A  A211         314      rlc    a              ; | L/R_bit 1=L
012C  33          315      mov    c,r_decide   ; | 0=R
012D  A213         316      rlc    a              ; |
012F  33          317      mov    c,f_decide   ; |
0130  9001A1       318      rlc    a              ;| 0 | 0 | 0 | 0 |L/R| L | R | F |
0133  93          319      mov    dptr,#d_table ;point to decision table
0134  FA          320      movc  a,@dptr       ;get decision
                                321      mov    r2,a         ;save it for later
                                322
0135  201A14       323      jb    back_up,di_4_0 ;IF backing up GOTO di
                                324      ;5) execute the table from here. | A | part is taken care of here
0138  A2E7         325      mov    c,acc.7      ;setting make_180 flag

```


IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 7

```

LOC OBJ          LINE    SOURCE
013A 9214        326          mov    make_180,c
013C 9210        327          mov    get_out,c
013E 921A        328          mov    back_up,c
0140 4055        329          jc     di_9           ;if make_180 then return
                                330          ;this determines | D | direction to turn
0142 5403        331          anl   a,#00000011b  ;mask out junk to get | D |
0144 F5F0        332          mov    b,a
0146 31E3        333          acall  store_val     ;store the decision and then store
0148 31C1        334          acall  inc_map_ptr   ; a halt afterwards and decr
                                335
014A 802E        336          sjmp  dx_58         ;go execute it below
                                337
014C 31D2        338          di_4_0: acall  dec_map_ptr
014E 5117        339          acall  get_val      ;get top of stack, put in B
0150 EA          340          mov    a,r2
0151 5403        341          anl   a,#00000011b  ;keep | D | (dir to turn)
0153 C5F0        342          xch   a,b           ;stack top in A, | D | in B
0155 23          343          rl    a             ;stack top now looks like this
0156 23          344          rl    a             ; 0000XX00
0157 45F0        345          orl   a,b           ;looks like 0000|prev| D |
0159 D2E4        346          setb  acc.4         ;looks like 000|B/U|prev| D |
                                347
015B 93          348          movc  a,@a+dptr     ;get decision
015C FA          349          mov    r2,a         ;save it just in case
015D C214        350          clr   make_180
015F A2E6        351          mov    c,acc.6      ;setting or clearing back-up
0161 921A        352          mov    back_up,c
0163 201A14      353          jb    back_up,dx_58 ;if still backing, dont store
0166 C4          354          swap  a
0167 5403        355          anl   a,#00000011b  ;| C | now in B
0169 F5F0        356          mov    b,a
016B 31E3        357          acall  store_val     ; and now stored in RAM
016D 31C1        358          acall  inc_map_ptr
016F EA          359          mov    a,r2         ;get decision and mask to get
0170 5403        360          anl   a,#00000011b  ; | D | dir to turn
0172 F5F0        361          mov    b,a
0174 8004        362          sjmp  dx_58         ;go execute it below
                                363
                                364          ;6) skip all of the junk above and do this if running finals
0176 5117        366          dif_0: acall  get_val     ;get the turn decision
0178 31C1        367          acall  inc_map_ptr   ;incr map pointer
                                368
                                369
                                370          ;7) interpret A to set the turn flags accordingly
017A E5F0        372          dx_58: mov    a,b           ;r_turn, l_turn flags
017C B40108      373          cjne  a,#01,dx_52   ;cjne's only work on acc not b
017F D208        374          setb  r_turn        ;look for
0181 C209        375          clr   l_turn
0183 D210        376          setb  get_out
0185 8010        377          sjmp  di_9
0187 B40208      378          dx_52: cjne  a,#02,dx_53
018A C208        379          clr   r_turn
018C D209        380          setb  l_turn

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 8

```

LOC OBJ          LINE    SOURCE
018E D210        381          setb   get_out
0190 8005        382          sjmp   di_9
                   383      dx_53:      ;must be b=11 or b=00, straight or stop
                   384          ;if 00 then stop or HALT
0192 B40002      385          cjne   a,#0,di_9
0195 8153        386          ajmp   halt
                   387
                   388      di_9:      ;end of S.R. here at di-9
0197 A202        389          mov    c,r_wall
0199 9215        390          mov    prev_r,c      ;store the wall condition for next
019B A203        391          mov    c,l_wall      ; toggle look
019D 9216        392          mov    prev_l,c
                   393
019F 412C        394          ajmp   decide_x
                   395
                   396      ;*****
01A1 05          397      d_table:      ;this holds all of the answers for the decision table
01A2 05          398          ;byte form looks like this... | A | B | C | D | (two bits ea)
01A3 0F          399          ; A - XY, X=do 180, Y=B-U flag status
01A4 0A          400          ; B - add to stack if in B-U, L=10 R=01 S=11 stop-mouse=00
01A5 05          401          ; C - what to add to stack not in B-U (B-U means back-up mode)
01A6 05          402          ; D - direction to turn
01A7 0F
01A8 CF
01A9 0A          403          db    00000101b,00000101b,00001111b,00001010b
01AA 0A
01AB 0A
01AC 0A
01AD 0F          404          db    00000101b,00000101b,00001111b,11001111b
01AE 05
01AF 0F
01B0 CF
                   405          db    00001010b,00001010b,00001010b,00001010b
01B1 00          406          db    00001111b,00000101b,00001111b,11001111b
01B2 00
01B3 00
01B4 00
01B5 00          407          ;these lines are for the B-U mode decisions, some non-valid
01B6 31          408          db    00h,00h,00h,00h
01B7 42
01B8 23
01B9 00          409          db    00h,00110001b,01000010b,00100011b
01BA 41
01BB 32
01BC 13
01BD 00          410          db    00h,01000001b,00110010b,00010011b
01BE 21
01BF 12
01C0 43          411          db    00h,00100001b,00010010b,01000011b

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 9

```

LOC  OBJ          LINE   SOURCE
                                412
                                413           ; | 0 | 0 | 0 | 0 | L/R | L | R | F |           for normal address
                                414           ; | 0 | 0 | 0 | 0 | B/U | PREV | WAY_2_GO |           for back-up address
                                415           ;                               | 2 bits | 2 bits |
                                416           ;*****
                                417           ;map representation in RAM will look like this...
                                418           ; |most sig 2 bits| XX | XX |least sig 2 bits|
                                419           ; location = | D | C | B | A | where A is the first decision, B is second
                                420           ; etc. A, B, C, D are all two bits. The next byte of RAM is rep the same.
                                421           ; It requires R1 as pointer and MAP_OFFSET 10h
                                422           ; L=10 R=01 S=11 stop-mouse=00 all represented as two bits.
                                423           ;If pointer = 3Fh and map offset=4 then end of RAM
                                424           ;*****
                                425           inc_map_ptr: ;this S.R. incs the map pointer from 0 to 3 and map_org
                                426           ;MAP uses mem-addr 11h-1fh and 2ch-3fh. R1 holds addr.
                                427
01C1  AF10          428           mov     r7,map_offset ;cjne only works on local regs
01C3  BF0309        429           cjne   r7,#3,imp_0 ;IF 3 THEN do below ELSE imp0
01C6  7510FF        430           mov     map_offset,#0ffh ;start at MSB two bits (offset=0)
01C9  B91F02        431           cjne   r1,#1fh,imp_1
01CC  792B          432           mov     r1,#2bh
01CE  09            433           imp_1:  inc     r1 ; and also move pointer high
01CF  0510          434           imp_0:  inc     map_offset
01D1  22            435           ret
                                436
                                437           ;*****
                                438           dec_map_ptr: ;this S.R. decr the map pointer and returns what's on the
                                439           ; top of the stack in B
01D2  AF10          440           mov     r7,map_offset ;cjne only works on local regs
01D4  BF0009        441           cjne   r7,#0,dmp_0 ;IF 0 THEN do below ELSE dmp_0
01D7  751004        442           mov     map_offset,#4 ;point |XX|XX|XX|00| at 00
01DA  B92C02        443           cjne   r1,#2ch,dmp_1
01DD  A920          444           mov     r1,20h
01DF  19            445           dmp_1:  dec     r1 ;map pointer R1
01E0  1510          446           dmp_0:  dec     map_offset
01E2  22            447           ret
                                448
                                449           ;*****
                                450           store_val: ;requires the decision to be in B as 000000XX and pointed to
01E3  C7            451           xch    a,@R1 ;get byte for below but save a
01E4  AEF0          452           mov     r6,b ;decision is in B
01E6  AF10          453           mov     r7,map_offset ;get it
01E8  BF0002        454           cjne   r7,#0,sv_a ;are we at 0 yet?
01EB  8004          455           sjmp   sv_b ; I guess we are
01ED  03            456           sv_a:  rr     a ;roll bits to shift |00|XX|00|00|
01EE  03            457           rr     a ;to get |XX|00|00|00|
01EF  DFFC          458           djnz   r7,sv_a ;keep shifting til |00|00|00|XX|
01F1  C2E0          459           sv_b:  clr   acc.0 ;clear it out to load it below
01F3  C2E1          460           clr   acc.1
01F5  BE0104        461           cjne   r6,#01,sv_0 ;load R if 01
01F8  D2E0          462           setb  acc.0 ;looks like |XX|XX|XX|01|
01FA  800E          463           sjmp   sv_2
01FC  BE0204        464           sv_0:  cjne   r6,#02,sv_1 ;load L if 02
01FF  D2E1          465           setb  acc.1 ;looks like |XX|XX|XX|10|
0201  8007          466           sjmp   sv_2

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

```

MCS-51 MACRO ASSEMBLER      751MAIN                                04/16/92   PAGE   10

LOC  OBJ          LINE      SOURCE
0203 BE0304      467      sv_1:      cjne    r6,#03,sv_2
0206 D2E0        468              setb    acc.0          ;load Straight AKA S
0208 D2E1        469              setb    acc.1          ;looks like |XX|XX|XX|11|
020A AF10        470      sv_2:      mov     r7,map_offset  ;roll until bits in original position
020C BF0002      471              cjne    r7,#0,sv_c    ;are we at 0 yet?
020F 8004        472              sjmp   sv_d           ; I guess we are
0211 23          473      sv_c:      rl     a              ;roll bits to shift |00|00|00|XX|
0212 23          474              rl     a              ;to get                |XX|00|00|00|
0213 DFFC        475              djnz   r7,sv_c       ;keep shifting til  |00|XX|00|00|
0215 C7          476      sv_d:      xch   a,@R1         ;put it all back
0216 22          477              ret
0217 F509        481      get_val:   ;this S.R. returns turn value in B
0219 E7          482              mov    temp_reg1,a    ;save acc same as push acc
021A AF10        483              mov    a,@r1
021C BF0002      484              mov    r7,map_offset  ;get it
021F 8004        485              cjne   r7,#0,gv_0    ;are we at 0 yet?
0221 03          486      gv_0:      rr     a              ;roll bits to shift |00|00|00|XX|
0222 03          487              rr     a              ;to get                |00|00|XX|00|
0223 DFFC        488              djnz   r7,gv_0       ;keep shifting til  |XX|00|00|00|
0225 5403        489      gv_1:      anl   a,#00000011b   ;now have |00|00|00|XX|
0227 F5F0        490              mov    b,a            ;stuff it back
0229 E509        491              mov    a,temp_reg1
022B 22          492              ret
022C 101002      493
022F 015C        494
022F 015C        495      ;*****
022F 015C        496
022C 101002      497      decide_x:  jbc   get_out,decel  ;get out, time to stop
022F 015C        498              ajmp  acc_0          ;do this again
022F 015C        499
022F 015C        500
022F 015C        501      ;*****
022F 015C        502      ; This section decelerates the motors. The table value is lowered *
022F 015C        503      ; if the mouse is too close to a wall. *
022F 015C        504      ;*****
022F 015C        505
022F 015C        506      decel:
0231 AE0B        507              mov    r6,count_fw    ;get no. steps past post from count-fw
0233 7400        F 508              mov    a,#decel_steps
0235 C3          509              clr   c
0236 9E          510              subb  a,r6
0237 FE          511              mov   r6,a            ;R6 now has decel_steps - count_fw
0237 FE          512
0238 7B00        F 513              mov   r3,#decel_decr  ;decel decr is now in r3
023A E50F        514              mov   a,r_ptr         ;check to see how far into accel tab
023C C3          515              clr   c
023D 9400        F 516              subb  a,#acc_steps    ;C set if r_ptr < top_speed
023F 5002        517              jnc   dec_0a
0241 7B01        518              mov   r3,#1          ;this is the decel_decr value
0241 7B01        519
0243 EE          520      dec_0a:   mov   a,r6
0244 8BF0        521              mov   b,r3

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 11

```

LOC  OBJ          LINE    SOURCE
0246  A4          522      mul      ab
0247  04          523      inc      a
0248  F50F        524      mov      r_ptr,a
024A  F50E        525      mov      l_ptr,a
                    526
024C  300BFD      527      dec_0:   jnb      time_int,$      ;stay here until int happens
024F  C20B        528      clr      time_int
0251  71B7        529      acall   snapshot      ;take a look at the walls
                    530      ;right routine
0253  300C08      531      jnb      r_int,dec_1    ;skip if right hasn't been stepped yet
0256  C20C        532      clr      r_int
0258  E50F        533      mov      a,r_ptr      ;get right pointer
025A  C3          534      clr      c
025B  9B          535      subb    a,r3          ;slow down by getting lesser value
025C  F50F        536      mov      r_ptr,a      ;ptr just got decremented
                    537
                    538      ;left routine
025E  300DEB      539      dec_1:   jnb      l_int,dec_0    ;if need to do left then goto dec-1
0261  C20D        540      clr      l_int
0263  E50E        541      mov      a,l_ptr      ;get left pointer
0265  C3          542      clr      c
0266  9B          543      subb    a,r3          ;slow down by getting lesser value
0267  F50E        544      mov      l_ptr,a      ;ptr just got decremented
0269  DEE1        545      dec_2:   djnz   r6,dec_0      ;decr number of steps to decel
026B  C28C        546      clr      tr0          ;stop the timer int
026D  7F00        547      mov      r7,#pause_val ;pause value
026F  9145        548      acall   pause         ;stay stationary for a while
                    549
                    550
                    551      ;*****
                    552      ; This section makes the mouse turn 90 or 180 degrees using the      *
                    553      ; decision from the "decision-maker".                                *
                    554      ;*****
                    555
                    556      ;pivot:   ;routine to pivot the mouse 90 or 180 degrees
0271  E58B        557      mov      a,rtl
0273  C3          558      clr      c
0274  9400        559      subb    a,#pivot_speed ;slower speed
0276  30140D      560      jnb      make_180,turn90 ;do 180 else turn
0279  101706      561      jbc     cw180,piv_0    ;180 cw
027C  D217        562      setb    cw180         ;next time 180 ccw
027E  C293        563      clr      l_dir        ;make L go backwards
0280  4190        564      ajmp    piv_1
0282  C291        565      piv_0:   clr      r_dir        ;make R go bw
0284  4190        566      ajmp    piv_1
                    567
0286  A208        568      turn90:  mov      c,r_turn     ;this section sets the motor dir
0288  B3          569      cpl      c            ; bits according to which
0289  9291        570      mov      r_dir,c      ; dir it has to turn.
028B  A209        571      mov      c,l_turn
028D  B3          572      cpl      c
028E  9293        573      mov      l_dir,c
                    574
                    575      piv_1:  ;THIS IS NEW. Mod adds or subtracts steps depending on sensor info
0290  201404      576      jb      make_180,piv_2y ;if mouse is skewed, it turns more

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 12

```

LOC  OBJ          LINE      SOURCE
0293  7F00        F         577          mov     r7,#half_90      ; or less degrees
0295  8002                578          sjmp    piv_2x
0297  7F00        F         579          piv_2y:    mov     r7,#half_180     ;
580          piv_2x: ;this section determines add or sub count depending on dir to pivot
581          ;(r_dir & too_l) | (l_dir & too_r) = - steps
582          ;(r_dir & too_r) | (l_dir & too_l) = + steps
0299  A291                583          mov     c,r_dir          ;check for first condition
029B  8201                584          anl    c,too_l
029D  9222                585          mov     temp_bit1,c
029F  A293                586          mov     c,l_dir
02A1  8200                587          anl    c,too_r
02A3  7222                588          orl    c,temp_bit1
02A5  5003                589          jnc    piv_2z
02A7  1F                590          dec     r7                ;decr step count to do pivot
02A8  800F                591          sjmp    piv_2v
02AA  A291                592          piv_2z:    mov     c,r_dir          ;check for second condition
02AC  8200                593          anl    c,too_r
02AE  9222                594          mov     temp_bit1,c
02B0  A293                595          mov     c,l_dir
02B2  8201                596          anl    c,too_l
02B4  7222                597          orl    c,temp_bit1
02B6  5001                598          jnc    piv_2v
02B8  0F                599          inc     r7                ;incr step count to do pivot
600
02B9  750F00            601          piv_2v:    mov     r_ptr,#0         ;point at first accel value
02BC  750E00            602          mov     l_ptr,#0
02BF  750DFE            603          mov     r_timr,#0feh     ;even up the timers to step
02C2  750CFE            604          mov     l_timr,#0feh     ; evenly
02C5  D28C                605          setb   tr0                ;enable timer
606
02C7  300CFD            607          piv_2:    jnb    r_int,$           ;stay here until done
02CA  C20C                608          clr    r_int             ;ack r interrupt
02CC  050F                609          inc    r_ptr             ;incr step count
02CE  050E                610          inc    l_ptr             ;incr step count
02D0  E50F                611          mov     a,r_ptr          ;see if at half way mark
02D2  201405            612          jb     make_180,piv_2a
02D5  B507EF            613          cjne   a,07,piv_2       ;IF half way fall thru to piv_3
02D8  8003                614          sjmp    piv_3
02DA  B507EA            615          piv_2a:    cjne   a,07,piv_2       ;if half way, fall thru
616
02DD  B40002            F         617          piv_3:    cjne   a,#look90,piv_4
02E0  D21B                618          piv_3a:    setb   look              ;if so, set look bit
02E2  300CFD            619          piv_4:    jnb    r_int,$           ;stay here until int done
02E5  C20C                620          clr    r_int             ;ack int
02E7  150F                621          dec    r_ptr             ;make it slow down
02E9  150E                622          dec    l_ptr
02EB  301B11            623          jnb    look,piv_5       ;if look is set, do snapshot
624
02EE  71B7                625          acall  snapshot         ;if aligned & wall exist truth table:
02F0  20051C            626          jb     aligned,piv_end
02F3  A201                627          mov     c,too_l
02F5  B093                628          anl    c,/l_dir
02F7  4016                629          jc     piv_end
02F9  A200                630          mov     c,too_r
02FB  B091                631          anl    c,/r_dir

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 13

```

LOC OBJ          LINE    SOURCE
02FD 4010        632          jc    piv_end
                                633
02FF E50F        634    piv_5:    mov    a,r_ptr    ;
0301 70DA        635          jnz    piv_3      ;IF end of count fall thru
                                636
0303 A202        637          mov    c,r_wall
0305 7203        638          orl   c,l_wall
0307 5006        639          jnc   piv_end    ;IF no walls get out
                                640
0309 050F        641          inc   r_ptr      ;keep steppin while you see
030B 050E        642          inc   l_ptr      ; a wall
030D 41E2        643          ajmp  piv_4
                                644
                                645    piv_end:
030F C28C        646          clr   tr0        ;off timer int
0311 850A8B      647          mov   r7,#00001010b ;full speed again
0314 C21B        648          clr   look
                                649
0316 C20C        649          clr   r_int
0318 C20D        650          clr   l_int
031A 43900A      651          orl   r7,#00001010b ;set r dir l dir
031D 7F00        652          mov   r7,#pause_val ;pause for a bit
031F 9145        653          acall pause
                                654
0321 0141        655          ajmp  accel     ;goto accel
                                656
                                657
                                658 ;*****
                                659 ; This is the Timer 0 interrupt subroutine. *
                                660 ; It will step a motor if the "prescale" for the corresponding *
                                661 ; motor overflows (or decrements to zero). *
                                662 ;*****
                                663
                                664    tim_0:    ;used to step the motors
0323 C28C        665          clr   tr0        ;quit counting
0325 C0D0        666          push  psw
0327 C0E0        667          push  acc
0329 C082        668          push  dpl
032B C083        669          push  dph
032D 900000      670          mov   dptr,#acc_tab
0330 D50D16      671          djnz  r_timer,tim_00
0333 C292        672          clr   r_step    ;step the R motor
0335 E50F        673          mov   a,r_ptr    ;load timer value from pointer
0337 93          674          movc  a,@a+dptr  ;get higher byte accel value
0338 F50D        675          mov   r_timer,a  ;load the timer value from table
033A E508        676          mov   a,step_count
033C B48202      677          cjne a,#130,tim_x
033F 8002        678          sjmp  tim_y
0341 0508        679    tim_x:    inc   step_count  ;decide uses this stuff
0343 D20C        680    tim_y:    setb  r_int      ;flag that the R mot was stepped
0345 D20B        681          setb  time_int  ;int has occurred flag
0347 D292        682          setb  r_step
0349 D50C0D      683    tim_00:  djnz  l_timer,tim_ret ;get out if not zero
034C C294        684          clr   l_step    ;step the L motor
034E E50E        685          mov   a,l_ptr    ;load timer value from pointer
0350 93          686          movc  a,@a+dptr  ;get higher byte accel value

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 14

```

LOC  OBJ          LINE      SOURCE
0351  F50C         687          mov     l_timr,a      ;load the timer value from table
0353  D20D         688          setb    l_int        ;flag that the L mot was stepped
0355  D20B         689          setb    time_int     ;int has occurred flag
0357  D294         690          setb    l_step
                                691          tim_ret:
0359  D083         692          pop     dph
035B  D082         693          pop     dpl
035D  D0E0         694          pop     acc
035F  D0D0         695          pop     psw
0361  D28C         696          setb    tr0          ;start counting
0363  32           697          reti
                                698
                                699
                                700          ;*****
0364  202010       701          ; This is the External interrupt 1 subroutine.          *
                                702          ;*****
                                703
0366  10210B       704          int_1:          ;used for setting left or right hug
                                705          ; L/R=0 right algo, L/R=1 left algo
                                706          ;AND also setting speed of run
0367  10210B       707          jb     done,int_1_2 ;incr speed and get out
036A  D221         708          jbc    l_r_bit, int_1_0
036C  C282         709          setb   l_r_bit      ;hug left
036E  7F0A         710          clr    l_led
0370  9145         711          mov    r7,#10       ;value for 2 seconds
0372  D282         712          acall  pause
0374  32           713          setb   l_led
0375  C221         714          reti
0377  7E05         715          int_1_0:      clr    l_r_bit      ;hug right
0379  302002       716          int_1_2:      mov    r6,#5
037C  058B         717          int_1_1:      jnb   done,int_1_3
037E  C282         718          inc    rtl          ;incr the speed
0380  7F02         719          int_1_3:      clr    l_led
0382  9145         720          mov    r7,#2
0384  D282         721          acall  pause
0386  7F02         722          setb   l_led
0388  9145         723          mov    r7,#2
038A  DEED         724          acall  pause
038C  32           725          djnz  r6,int_1_1
                                726          reti
                                727
                                728
                                729          ;*****
038D  C28C         730          ; This is the External interrupt 0 subroutine.          *
                                731          ;*****
                                732
038F  200A06       733          int_0:          ;used for starting and stopping the thing
0392  D20A         734          clr    tr0
0394  C290         735          jb     s_s_int, int_0_0 ;we are here to start up
0396  61AE         736          setb   s_s_int
0398  D290         737          clr    mot_en
039A  D220         738          ajmp  int_0_ret
                                739          int_0_0:      setb   mot_en      ;we are here cause at finish box
                                740          setb   done      ;tell the prog that we are done
                                741

```


IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER

751MAIN

04/16/92 PAGE 15

```

LOC  OBJ          LINE      SOURCE
039C  C20A         742                clr     s_s_int
039E  90001D       743                mov     dptr,#main_1    ;get address
03A1  A881         744                mov     r0,sp           ;set up to start all over
03A3  A683         745                mov     @r0,dph         ;load reti vector to 0004h
03A5  18            746                dec     r0
03A6  A682         747                mov     @r0,dpl
03A8  75A800        748                mov     ie,#00
03AB  758800        749                mov     tcon,#00        ;clears any ints
                                750
03AE  C297         751      int_0_ret:    clr     r_led
03B0  7F14         752                mov     r7,#20          ;value for 2 seconds
03B2  9145         753                acall  pause
03B4  D297         754                setb   r_led
03B6  32           755                reti
                                756
                                757
                                758      ;*****
03B7  75B0FF       759      ; This section strobes the sensors for data.          *
03BA  752000       760      ;*****
                                761
03BA  752000       762      snapshot:      ;takes a look at walls and sets bits accordingly
03BD  C281         763      ;R4, R5 for sensor info. ACC, C, r l sens, bit addressables
03BF  A4           764                mov     p3,#0ffh
03C0  A4           765                mov     ss_bits,#0      ;clear all flags
03C1  A4           766                clr     r_sens          ;enable right sensor bank
03C2  A4           767                mul     ab               ;causes a 6uS wait state
03C3  E5B0        768                mul     ab
03C4  A4           769                mul     ab
03C5  D281        770                mul     ab
03C6  FC          771                mov     a,p3            ;store right wall
03C7  FC          772                setb   r_sens
03C8  33          773                mov     r4,a            ;wall is now repr as a high
03C9  9204        774                rlc     a               ;store R sens 0 for f_wall
03CA  6002        775                mov     f_wall,c
03CB  D202        776                jz     ss_0             ;if no wall goto ss_0
03CC  D202        777                setb   r_wall          ;right wall present
                                778
03CF  C280        779      ss_0:      clr     l_sens          ;enable left sensor bank
03D0  A4           780                mul     ab               ;causes a 6uS wait state
03D1  A4           781                mul     ab
03D2  A4           782                mul     ab
03D3  A4           783                mul     ab
03D4  A4           784                mul     ab
03D5  E5B0        785                mov     a,p3            ;store left wall
03D6  D280        786                setb   l_sens
03D7  FD          787                mov     r5,a            ;wall is now repr as a high
03D8  33          788                rlc     a               ;grab inner sens and or it
03D9  7204        789                orl    c,f_wall
03DA  9204        790                mov     f_wall,c        ;store front wall
03DB  6002        791                jz     ss_2             ;if no wall goto ss_2
03DC  D203        792                setb   l_wall          ;left wall present
                                793
03DE  20045E       794      ss_2:      jb     f_wall,ss_ret    ;if no front wall then cont
03DF  7400         795                mov     a,#0            ;build case statement
03E0  A221         796                mov     c,l_r_bit
03E1  33           797                rlc     a               ; 00000| L/R | L | R |

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 16

```

LOC  OBJ          LINE      SOURCE
03EB  A203         797          mov    c,l_wall      ;   |algo |wall|wall|
03ED  33           798          rlc    a              ;   |   |bit |bit |
03EE  A202         799          mov    c,r_wall      ;   |   |   |   |
03F0  33           800          rlc    a              ;done shifting in bits for case stmt
03F1  B40102        801          cjne   a,#01,ss_4     ;chk R
03F4  810F          802          ajmp   ss_9
03F6  B40302        803          ss_4:  cjne   a,#03,ss_5     ;chk R
03F9  810F          804          ajmp   ss_9
03FB  B40502        805          ss_5:  cjne   a,#05,ss_6     ;chk R
03FE  810F          806          ajmp   ss_9
0400  B40202        807          ss_6:  cjne   a,#02,ss_7     ;chk L
0403  8115          808          ajmp   ss_10
0405  B40602        809          ss_7:  cjne   a,#06,ss_8     ;chk L
0408  8115          810          ajmp   ss_10
040A  B40737        811          ss_8:  cjne   a,#07,ss_ret  ;if eq then chk L else do nothing
040D  8115          812          ajmp   ss_10
      813          ;*****
      814          ss_9:  ;check the right side offset
040F  8CF0           815          mov    b,r4          ;put wall info into acc
0411  C21E           816          clr    genp2         ;0=chk R
0413  8004           817          sjmp   ss_93
0415  8DF0           818          ss_10: mov    b,r5
0417  D21E           819          setb   genp2         ;1=chk L
      820
0419  E5F0           821          ss_93: mov    a,b
041B  5400           822          F      anl    a,#sens_pat   ;masking for cmp, 3 high 4 low
041D  B40804        823          cjne   a,#08h,ss_90  ;check for aligned condition
0420  D205           824          setb   aligned      ;perfectly on wall
0422  8144           825          ajmp   ss_ret
0424  E5F0           826          ss_90: mov    a,b          ;get wall info again
0426  5403           827          anl    a,#00000011b  ;check for too close center if a > 0
0428  600B           828          jz     ss_91         ;if no wall on ones then not too_
042A  201E04        829          jb     genp2,ss_101
042D  D201           830          setb   too_l
042F  8144           831          ajmp   ss_ret
0431  D200           832          ss_101: setb   too_r
0433  8144           833          ajmp   ss_ret
0435  E5F0           834          ss_91: mov    a,b          ;get wall info again
0437  5460           835          anl    a,#01100000b  ;check for too close wall if a > 0
0439  6009           836          jz     ss_ret
043B  201E04        837          jb     genp2,ss_102
043E  D200           838          setb   too_r
0440  8144           839          ajmp   ss_ret
0442  D201           840          ss_102: setb   too_l
0444  22            841          ss_ret: ret
      842
      843          ;*****
0444          pause: ;pause loop using R7 as the loop counter
0445  C28C           845          clr    tr0
0447  903EFE        846          mov    dptr,#03efeh
044A  D582FD        847          djnz  dpl,$
044D  D583FA        848          djnz  dph,$-3
0450  DFF3           849          djnz  r7,pause
0452  22            850          ret
      851

```

IEEE Micro Mouse using the 87C751 microcontroller

AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 17

```

LOC  OBJ          LINE  SOURCE
                                852  ;*****
                                853  halt:          ;This S.R. brings the thing to a halt - mostly used for debug
0453  C2A9         854          clr          et0
0455  D290         855          setb       mot_en
0457  80FE         856          sjmp      $
                                857  ;*****
                                858  EXTRN CODE      (acc_tab) ;from the DOS file 751acc.asm
                                859
                                860  end

```

Automatic baud rate detection for the 80C51

AN447

Author: Greg Goodhue

This note documents a method to automatically establish the correct baud rate for serial communications in many 80C51 family applications. The first character received after a program is started is used to measure the baud rate empirically.

This can eliminate the need to have setup switches whose settings are difficult to remember and all of the other headaches associated with applications that use multiple baud rates. One might assume that a reliable method of accomplishing this might be impossible without severely limiting the characters that could be recognized. The problem is in finding a timing interval that can be measured in a large number of possible characters under a wide variety of conditions.

Measuring a single bit time would be the obvious way to quickly determine what baud rate is being received. However, many ASCII characters don't have an example of a single bit time in the RS-232 pattern. For most characters, the length of the entire transmission from the start bit to the last "visible" transition will fall within certain ranges as long as some reasonable assumptions can be made about the possible baud rates (i.e. that they are standard baud rates). Moreover, many systems now use 8 data bits and no parity for ASCII transmissions. In this format, normal ASCII characters will never have the MSB set and since UARTs send data LSB first/MSB last, the program would always be able to "see" the beginning of the stop bit.

The following baud rate detection routine waits for a start bit (falling edge) on the serial input pin and then starts timer 0. At every subsequent rising edge of the serial data, the timer value is captured and saved. When the timer overflows, the last captured value will

indicate the duration of the serial character from the start bit to the last 0 to 1 transition (hopefully the stop bit).

The table CmpTable contains the maximum timer measurement that is accepted for each baud rate. These values were picked such that a timed interval of only 4 data bit times (plus the start bit time) will still produce the correct baud rate.

There is an assumption in this method that anyone using it needs to be aware of. That is, that this technique depends on only one character being received during the sampling window, which has to be at least as long as a typical character at the slowest baud rate that can be accepted. Essentially this means that the data must normally come from someone typing at a keyboard.

On our PCs, we were not able to fool the program by typing two characters in quick succession. The PC function keys did present a problem because they send two characters in a tight sequence, and fooled the program into detecting the wrong baud rate. In the example program, which is designed for a 12 MHz clock, the total sample interval is about 65 milliseconds, or about twice the duration of an RS-232 character sent at 300 baud.

If parity is used, a possibility of a baud rate determination error happens when the four MSBs and the parity bit of the character received are all ones. This can happen for the lower case letters "p" through "z", plus curly brackets, vertical bar (|), tilde (~), and "delete", depending on whether the system uses odd or even parity. Note that the usual prompt characters that a user would type to get a system's attention (e.g. space, carriage

return, and escape) are NOT subject to this limitation.

Because of the way this program works, the first input character that is used to detect the baud rate is lost since the UART cannot be set to the correct baud rate until after the first character has been timed. Also, most "real" programs using this technique would want to repeat the baud rate detection process if framing errors are detected at the UART during normal operation.

To calculate CmpTable values for other oscillator frequencies and baud rates, use the following equation:

$$\text{Table entry} = \frac{\text{Osc(MHz)}}{\text{Baud Rate}} \times \frac{5}{12}$$

Remember that the table entry is a two byte value, so the result of the above must be split into upper and lower bytes (easy if you have a hexadecimal calculator). It may also be possible to get the assembler to do all of the calculations for you.

The above equation was derived as follows:

$$\frac{\text{maximum timer value}}{\text{(table entry)}} = \frac{\text{minimum recognition time}}{\text{machine cycle time}}$$

$$\text{Minimum recognition} = \frac{\text{bits-to-recognize}}{\text{\#-of-bits}} \times \text{byte time}$$

Note: '#-of-bits' (the number of "visible" bits) is 9, and bits-to-recognize (the minimum # of bits to recognize) is 5 for 8-N-1 communication.

$$\text{byte time} = \frac{1}{\text{baud rate}} \times \text{\#-of-bits}$$

$$\text{machine cycle time} = \frac{\text{Osc frequency}}{12}$$

Automatic baud rate detection for the 80C51

AN447

```

;*****
;
;           Automatic Baud Rate Detection Test
;*****

$Title(Automatic Baud Rate Detection Test)
$Date(12-16-91)
$MOD552

;*****
;
;           Definitions
;*****

RX          BIT    P3.0           ;Location of serial receive pin.
CharH       DATA 30h           ;Holds high byte of frame timer result.
CharL       DATA 31h           ;Holds low byte of frame timer result.
BaudRate    DATA 32h           ;Holds final baud rate determination.
Display     EQU   P4             ;Port to display result for debug.

;*****
;
;           Reset and Interrupt Vectors
;*****

          ORG    8000h

Start:    ACALL  AutoBaud        ;Go try to get a baud rate value.
          MOV    Display,BaudRate ;Display baud rate value for debug.
          SJMP  Start

;*****
;
;           Subroutines
;*****

; AutoBaud Rate Detect Routine.
; Attempts to detect baud rate from first received character, by measuring
; the length of the character. Some characters may not work properly,
; primarily those that end with more than 3 (4?) ones in a row.
; Returns with ACC = baud rate pointer.

AutoBaud: MOV    TMOD,#01h       ;Initialize timer 0 (UART baud rate timer).
          MOV    TH0,#0          ;Put timer 0 in 16-bit counter mode.
          MOV    TLO,#0
          MOV    TCON,#0

          MOV    CharH,#0        ;Initialize timer result.
          MOV    CharL,#0

AB0:      JB     RX,AB0          ;Wait for serial start bit.
          SETB   TR0            ;Start timer.

AB1:      JB     TF0,AB3         ;Check for timer overflow.
          JNB    RX,AB1         ;Check for a rising edge on serial data.
          MOV    CharH,TH0      ;Capture timer value at this serial edge.
          MOV    CharL,TLO

AB2:      JB     TF0,AB3         ;Check for timer overflow.
          JB     RX,AB2         ;Check for falling edge on serial data.
          SJMP  AB1            ;Go back and repeat sampling.

```

Automatic baud rate detection for the 80C51

AN447

```

AB3:      CLR   TR0           ;Maximum sample time has expired, check result.
          CLR   TF0           ;Begin by stopping timer and clearing flag.

          MOV   BaudRate,#19   ;Set up table pointers.
CmpLoop:  MOV   A,BaudRate
          MOV   DPTR,#CmpTable
          MOVC  A,@A+DPTR      ;Get a table entry for comparison.
          DEC   BaudRate
          CJNE  A,CharH,Cmp1    ;Check result range.
          SJMP  CmpLow         ;High byte table = timed value, check low byte.
Cmp1:     JC    CmpMatch       ;A match if table value is < timed value.
          DJNZ  BaudRate,CmpLoop ;Check for end of comparison table.
          SJMP  CmpMatch

CmpLow:   MOV   A,BaudRate
          MOVC  A,@A+DPTR      ;Get a table entry for comparison.
          CJNE  A,CharL,Cmp2    ;Check result range.
          SETB  C               ;Match if equal.
Cmp2:     JC    CmpMatch       ;C set if A < low byte of result.
          DJNZ  BaudRate,CmpLoop ;Check for end of comparison table.

CmpMatch: MOV   A,BaudRate     ;Comparison complete,
          CLR   C               ; get final baud rate index,
          RRC   A
          MOV   BaudRate,A     ; and save.
          RET

```

; Compare table holds timer values for the transition points of the accepted
; baud rates. Entries are LSB, MSB. These values are for 12 MHz operation.

```

CmpTable: DB   40h,0           ;0 - out of range, value too low.
          DB   80h,0           ;1 - 38400 baud.
          DB   0,01h          ;2 - 19200 baud.
          DB   0,02h          ;3 - 9600 baud.
          DB   0,04h          ;4 - 4800 baud.
          DB   0,08h          ;5 - 2400 baud.
          DB   0,10h          ;6 - 1200 baud.
          DB   0,20h          ;7 - 600 baud.
          DB   0,40h          ;8 - 300 baud.
          DB   0,80h          ;9 - out of range, value too high.

```

END

Determining baud rates for 8051 UARTs and other UART issues

AN448

Author: Greg Goodhue

The purpose of this note is to expand upon and clarify some aspects of determining baud rates and crystal frequencies for using a standard 8051 or 80C51 UART for ordinary RS-232 type serial communication. The standard baud rate equation is simplified here and is restated to allow solving for other variables such as the crystal frequency and timer reload values.

The following discussion assumes that the reader has some knowledge of the 8051/80C51 UART and timers. This should be considered a supplement to the information presented in the Philips 80C51 Family Microcontroller Data Book sections on Timer/Counters and the Standard Serial Interface.

Since this discussion assumes the use of a standard UART for RS-232 serial communications, the UART will be used in modes 1 or 3 (variable baud rates) and timer 1 will be used in mode 2 (8-bit auto-reload mode) as the baud rate generator. All of the equations shown here give an option for two clock divisors depending on whether the SMOD bit is used on a CMOS microcontroller. For an NMOS device, always use the default value (SMOD is not = 1).

The basic equation for a timer reload value can be stated as:

$$TH1 = 256 - \frac{\text{Crystal Frequency}/384 \text{ (or 192 if SMOD = 1)}}{\text{Baud Rate}}$$

Example:

To obtain a timer reload value for a 9600 baud serial data rate with an 11.0592 MHz crystal:

$$256 - \frac{11,059,200/384}{9600} = 256 - 3 = 253, \text{ or FD hexadecimal}$$

The equation can also be solved to derive the baud rate or the crystal frequency from the other information as follows:

$$\text{Baud Rate} = \frac{\text{Crystal Frequency}/384 \text{ (or 192 if SMOD = 1)}}{256 - TH1}$$

$$\text{Minimum crystal frequency for a given baud rate} = \text{Baud Rate} \times 384 \text{ (or 192 if SMOD = 1)}$$

Thus, the minimum crystal frequency that may be used for 19.2k baud communication on a CMOS part with SMOD = 1 would be 19200×192 , which gives 3.6864 MHz. When using this equation, the timer reload value TH1 for the maximum baud rate is always 255 (256 - 1) or FF hexadecimal.

Of course, any even multiple of the frequency obtained in this manner will also support the same baud rate with a different timer reload value. For instance, four times 3.6864 MHz is 14.7456 MHz. At that crystal frequency, 19.2k baud is attained with a timer reload value that gives one fourth of the timer overflow rate: 252 (256 - 4) or FC hexadecimal.

Determining baud rates for 8051 UARTs and other UART issues

AN448

CRYSTAL FREQUENCIES USED FOR STANDARD BAUD RATES

The following chart shows possible crystal frequencies for use with the 80C51 UART at standard baud rates. The chart assumes use of the UART in modes 1 or 3 (variable baud rates) and timer mode 2 (8-bit auto-reload mode). The chart also assumes a minimum requirement of at least 9600 baud (including the use of SMOD for baud rate doubling). More crystal frequencies are available if a lower maximum baud rate is required.

The minimum timer count column indicates how many timer counts are required at the stated crystal frequency in order to obtain the

maximum baud rate shown. The last column shows the timer reload value that is used to obtain the minimum timer count. This is simply 256 minus the minimum timer count.

Timer reload values for other baud rates at the same crystal frequency are determined by multiplying the minimum timer count by two successively and calculating a new reload value as previously mentioned. For instance, for 4800 baud at 1.8432 MHz, the timer count would be 2 (twice what it is for 9600 baud), giving a timer reload value of 254 (256 - 2) or FE hexadecimal.

Maximum Standard Crystal (MHz)	Maximum Baud Rate	Timer Count	Timer Reload Value (in hex)
1.8432	9600	1	FF
3.6864	19200	1	FF
5.5296	9600	3	FD
7.3728	38400	1	FF
9.2160	9600	5	FB
11.0592	19200	3	FD
12.9024	9600	7	F9
14.7456	76800 (2 × 38400)	1	FF
16.5888	9600	9	F7
18.4320	19200	5	FB
20.2752	9600	11	F5
22.1184	38400	3	FD
23.9616	9600	13	F3
25.8048	19200	7	F9
27.6840	9600	15	F1
29.4912	153600 (4 × 38400)	1	FF
31.3344	9600	17	EF
33.1776	19200	9	F7
35.0208	9600	19	ED
36.8640	38400	5	FB

Determining baud rates for 8051 UARTs and other UART issues

AN448

THE EFFECT OF USING OFF-FREQUENCY CRYSTALS

Occasionally, one may wish to use an off-frequency crystal in a design, but still want to make use of the UART for debug purposes.

Since most terminals (or other RS-232 devices) will communicate with another device that has a baud rate that is off by several percent, this can often be done successfully. WARNING: running the UART off-frequency is NOT recommended if part of the application's normal operation involves communication with other RS-232 devices.

There is no exact limit on how much frequency error is tolerable, since this depends on the devices communicating, the baud rates, precise frequencies used by both devices, etc. However, a rule of thumb may be used that the communication is likely to work if the frequency is off by less than 5%. This somewhat arbitrary number was arrived at as follows: for a ten-bit serial code (one start, 8 data bits, one stop), a 10% data rate error will put the receiver off by about plus or minus one bit time at the end of one data frame. A one bit-time error seems rather excessive if one wants fairly reliable communications. So, consider using half of that value (5%) as a rule of thumb.

The consequence of all this is that one may often find a more standard "off-the-shelf" crystal frequency to use in an application, if the UART is being used for debugging, factory testing, etc. As an example, consider the well-known "color burst" crystal. At 3.579545 MHz, this crystal is only about 3% slower than the 3.6864 MHz crystal that may be desired for baud rate generation. As such, this lower cost crystal could be used in place of the less standard one in some cases. Another obvious replacement is to use the standard 14.31818 MHz crystal in place of the not-so-standard 14.7456 MHz crystal that appears in the table. This replacement also yields a less than 3% error and may be handy because it gives a fast instruction execution rate for the 80C51, whereas 3.58 MHz may be too slow for many applications.

It should also be remembered that RS-232 communications are most robust if characters are not transmitted back-to-back. This can become more important when the UART is deliberately used out of spec as described here. When data is sent at full speed, there is no chance for the receiver to re-synchronize to the transmitted frames if it once gets out of synch. However, when there is a short pause between characters (about 2 to 3 bit times or longer), the receiver will generally be able to correctly locate start bits without framing errors. In the worst case, a pause of one byte-time or longer in a transmission should ALWAYS re-synchronize any receiver no matter how out of synch it has become.

A LITTLE KNOWN PHENOMENON

In the UART setup code for most applications, the actual timer count register (TL1) is not initialized. In many applications, this DOES have an affect on the way the UART behaves on the first character sent, although the chances of this being noticed are slight. This can be seen by trying to observe the first character sent from the UART on a logic analyzer that is being triggered by the end of microcontroller reset. The first character will begin so far down the time line that it will not be seen at any resolution on the logic analyzer that would show any of the individual bits.

This effect occurs because TL1 has to time-out once before the first character is transmitted. If TL1 is not initialized in the program, it will have a reset value of 0. This could give the first timeout a duration of up to 255 normal bit times, depending on the reload value for TH1 (which again depends on the baud rate and crystal frequency).

Again, in most applications, this would never be an issue. In fact, it may often be an advantage to have a delay before the first serial character is sent after power-up. But if the first serial character should start sooner, TL1 may be initialized to some value other than zero. For no delay, the same value used in TH1 should be used.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

1. INTRODUCTION

The routing of the traces on a Printed Circuit Board (PCB) largely effect the ElectroMagnetic Compatibility (EMC) performance of the PCB with respect to both ElectroMagnetic (EM) radiation as susceptibility to EM-fields.

The PCB will connect electronic components such as passive components, transistors and ICs. Furthermore, cables to interconnect the PCB with other system parts, e.g., another PCB, signal generator, CATV wall-outlet, DC power source or an AC-mains connection, will largely influence the PCB with respect to EMC [7].

In order to get a PCB on which the circuits function properly, the trace routing, the placement of components/connectors and the decoupling used with certain ICs will have to be optimized according to the constraints given in this report.

To reach an economic and functional PCB design, the following items have to be kept in mind:

4. Correct choice of the PCB format (mono, bi- or multi-layer)
5. Take care that "every" signaltrace has its signalreturn nearby
6. Proper decoupling for each IC or group of ICs
7. Allowed tracelengths and allowed loopareas
8. Placement of the connectors
9. Right cable choice with a proper connector
10. Proper use and placement of filters and filterparts.

These items with the appropriate measures will be further explained.

The main target is to get control over your PCB currents.

2. GENERAL

2.1. Conductors

Single conductors have, as a rule of thumb, an inductance of $1\mu\text{H}/\text{m}$. At low frequencies only, below 1kHz , R_{dc} applies. These impedances, together with the currents that will flow through these impedances, will be responsible for the voltage drop between points as Ohms law applies. The voltage drop can be diminished by either reducing the impedance or lowering the current through that impedance.

In typical digital designs the voltage drop will be frequency independent. A square wave current, resulting from a square wave output voltage to a resistive load, can be described as a series of sinewaves of which the amplitude of the harmonics decrease proportional with the frequency (Fourier expansions), see Figure 1b. The impedance of the inductor increases proportional with frequency (see Figure 1a), therefore the product; voltage drop (Figure 1c) remains constant.

When the current has a triangular waveshape, as function of time, due to capacitive loading, the amplitude of the harmonics decreases with the frequency square and the voltage drop across the inductor reduces proportional with frequency.

2.2. Transmissionlines

By using the inductance of a single wire, L_1 , the mutual coupling, M , and the capacitance between the traces, C_1 , a transmissionline, shown in Figure 2, can be defined of which the characteristic impedance, Z_0 , equals:

$$Z_0 = \sqrt{(L_{eff} / C)}$$

where:

$$L_{eff} = L_1 + L_2 - 2 \cdot M, \quad k = \sqrt{(L_1 + L_2) / M}$$

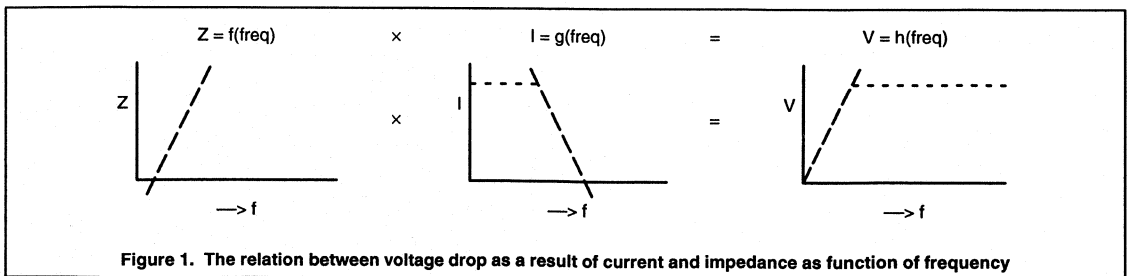
$$\text{and } C = C_1 + C_2.$$

When the coupling, k , between the traces of the transmissionline is high, the effective inductance will decrease rapidly. Some coupling factors are given in Table 1.

An indifferent signal path design (Figure 3a) can be changed into a transmissionline design (Figure 3b). This change will lower the effective inductance, L_{eff} , between the two circuit blocks and will therefore lower the voltage drop between the two references of those circuits.

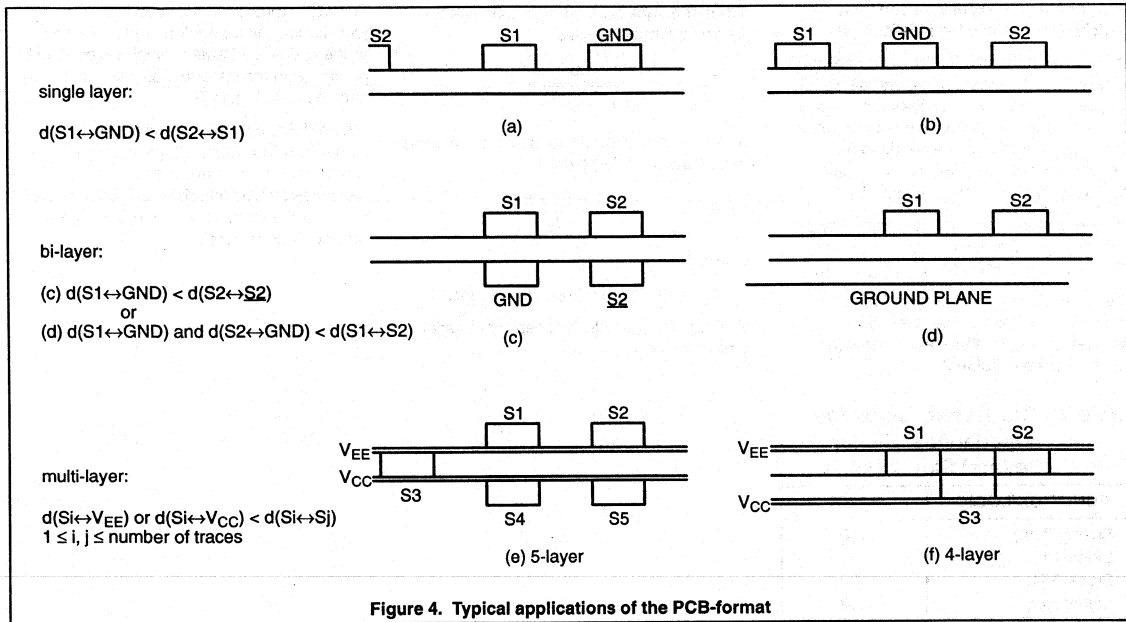
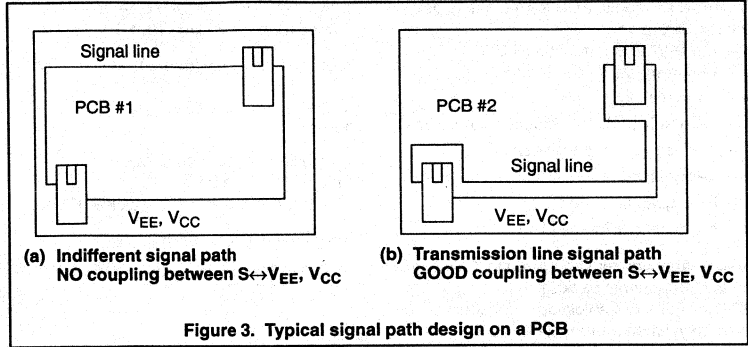
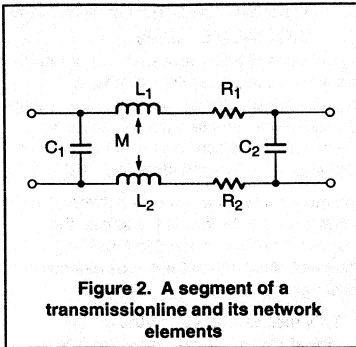
Table 1. Coupling Factors between the Conductors of a Transmissionline

TRANSMISSIONLINE TYPE	COUPLING
Parallel wires	0.5 - 0.7
Bi-layer PCB	0.6 - 0.9
Multi-layer PCB	0.9 - 0.97
Coaxial cable	0.8 - 1.0
RG-58 coax	0.996



Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001



2.3. Capacitive and Inductive Coupling

Separately, the capacitive and inductive values, derived from the definition of the transmissionline, can also be used to calculate the crosstalk between adjacent traces, not being a function signal path. The capacitive coupling, representing and induced current, is given by:

$$I_{Ck} = 1/C_k \cdot dV/dt,$$

where:

C_k = coupling capacitance between adjacent traces; in practice: 100pF/m

(depends upon the vicinity of other traces, see Appendix A),

and the inductive coupling, representing an induced voltage, is given by:

$$V_{Mk} = M_k \cdot di/dt,$$

where:

M_k = mutual coupling between two traces (For further detail see Chapter 4.)

In both coupling modes, the transfer function will typically show a high pass behavior.

3. CHOICE OF THE PCB-MATERIAL

By a proper choice of the PCB-material and the routing of the traces, a good transmissionline with low coupling to other traces can be created. Low coupling, or little crosstalk, can be obtained when the distance, d , between the transmissionline conductors is less than their distance to other adjacent conductors (see Figure 4).

By using these examples of geometry of traces the definition of the transmissionline between S_1, S_2, S_i, j and $(S_2) GND, V_{EE}$ and/or V_{CC} are well defined and the coupling between the traces S_2 and S_1 is low.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The most economic PCB format has to be chosen based on:

- the legal and/or functional EMC requirements for the product,
- trace density,
- assembly and manufacturer capabilities,
- CAD-system capabilities,
- design-costs,
- PCB quantities, and
- the costs of EM-shielding.

Special attention must be given to the integral costs (components packaging/pinning + PCB-format + EM-shielding + construction + assembly) when a product definition is considered by using a NON-shielded cover. In many cases the choice of a proper PCB-format may expel the need for a metallized box within the plastic cover.

To improve immunity and to lower unwanted emission, both in fast analog and all digital applications, transmissionlines are needed. Dependent upon the transition of the output signal, a transmissionline needs to be present between $S \leftrightarrow V_{CC}$, $S \leftrightarrow V_{EE}$, and $V_{EE} \leftrightarrow V_{CC}$, as indicated in Figure 5.

The signal current will be determined by the output-stage symmetry of the circuit. For MOS: $I_{OL} = I_{OH}$, while for TTL: $I_{OL} > I_{OH}$.

The Logic Family and functional reasons determine the typical characteristic impedance, Z_0 , for that transmissionline which is given in Table 2.

For two traces next to each other the following formula applies [10, 11].

$$Z_0 = \frac{120 \ln(\pi \cdot h / (b + c))}{\sqrt{\epsilon} \cdot r}$$

where:

- h = distance between traces
- b = width of the trace
- c = thickness of the trace; typical 17µm,

for two traces on top of each other:

$$Z_0 = \frac{120 \pi (h / (h + b))}{\sqrt{\epsilon} \cdot r}$$

where:

- h = 1.5mm (typical thickness of epoxy).

When the trace is above a groundplane the following formula applies:

$$Z_0 = \frac{87 \ln(6 \cdot h / (.8 \cdot b + c))}{\sqrt{(\epsilon \cdot r + \sqrt{2})}}$$

and in case of a trace between two (ground-) planes the formula yields:

$$Z_0 = \frac{60 \ln(4 \cdot K / (.67 \cdot \pi \cdot b \cdot (.8 + c/b)))}{\sqrt{\epsilon} \cdot r}$$

where:

- K = distance in-between the planes.

Typically the permittivity for epoxy material equals: $\epsilon_r = 4.7$.

4. THE SIGNALTRACE AND ITS SIGNALRETURN

Signaltraces need to have their signal-return-traces as close as possible in order to prevent emission from that looparea enclosed by these traces and to reduce susceptibility due to voltages which can be induced in this loop, e.g., by RF-transmitters and ESD.

Commonly, when the distance between two traces equals the width of the traces, the coupling factor is about 0.5 to 0.6. The effective inductance of the traces has gone down from 1µH/m to 0.4 - 0.5µH/m.

This means that 40 to 50% of the signal-return current may run freely through the other traces of the PCB.

For each signal path between two (sub-)blocks either analog or digital **three** properly defined transmissionlines need to be present with the impedances given in Table 2 and shown in Figure 5.

With TTL logic the sink-current; the high-to-low transition, is higher than the source-current. In this case the transmissionline should be defined between V_{CC} and S instead of V_{EE} and S, which is commonly considered.

Table 2. The Transmissionline Impedances, Z_0 , for Several Signal Paths

FUNCTION/LOGIC	Z_0 (Ω)
Supply (typ.)	<<10
Signal ECL	50
Signal TTL	100
Signal HC(T)	200

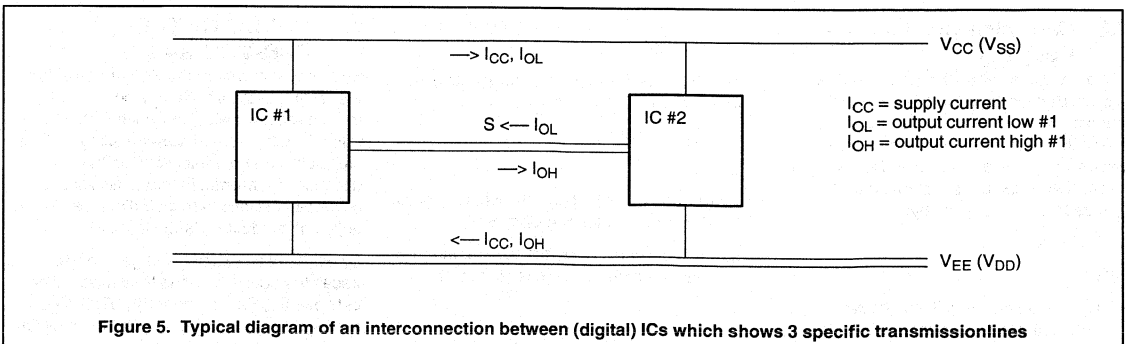


Figure 5. Typical diagram of an interconnection between (digital) ICs which shows 3 specific transmissionlines

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The mutual coupling between two parallel traces can be calculated from the double integral [9]:

$$M_k = \mu / (4 \cdot \pi) \cdot \int_{l_1} \int_{l_2} ds_1 \cdot ds_2 \cdot dr / r$$

where:

- l_1, l_2 = length of traces 1 and 2
- r = relative distance between line segments, ds_1, ds_2 , of each trace.

Substituting the geometry of two parallel lines results in:

$$M_k = 200 [l \cdot \ln \{ (l + \sqrt{l^2 + h^2}) / h \} + \sqrt{l^2 + h^2} + h] [nH]$$

where:

- l = length of the two parallel traces and
- h = distance between the traces (trace thickness and width are neglected).

If the coupling between the two conductors of a transmissionline is too low, a ferrite toroid ($\mu_r > 200$ (-5000)), with some windings, will increase this coupling to ≈ 1 .

By using ferrite toroids one can get full control over the signal- and signal-return currents.

In case of parallel conductors, the characteristic impedance of this transmissionline may be influenced by the ferrite. In case of coaxial cable, the presence of the ferrite will only be noticeable on the outer parameters of the cable.

CONCLUSIONS:

- I. Use traces as thin as possible next to one another instead on top of each other (separation commonly less than 1.5mm + epoxy thickness of a bi-layer).
- II. Create a layout where every signalline has its signal-return at the closest possible interval (applies to both signal- and supply-traces).
- III. If the coupling between the conductors of the transmissionline is insufficient a ferrite toroid may be used.

5. PROPER DECOUPLING WITH EACH IC

ICs will be commonly decoupled by capacitors only. Because capacitors are not ideal, resonances will occur. Above the resonance frequency the capacitor behaves as an inductor, which means that the dI/dt is limited. The value of this capacitor is determined by the voltage-fluctuations which are allowed across the power supply pins of the IC. According to good designers practice, this voltage fluctuation should be less than 25% of the signal-line worst-case noise margin. From the following equation the optimal decoupling capacitor for each logic family output gate can be calculated:

$$I = c \cdot dV/dt$$

The worst-case signal-line noise margins for several logic families are given in Table 3, together with the recommended decoupling

capacitor value, $C_{dec.}$, which need to be added with each output gate.

The values of the decoupling capacitors for fast logic families may no longer be useful if the capacitor incorporates a large series inductance, either caused by the construction of the capacitor, long connecting wires or PCB traces. Additional small ceramic capacitors (100–100pF) need then to be added, *as close as possible to the pins of the IC*, in parallel to these "LF-" decoupling capacitors. The resonance frequency of this ceramic capacitor (including the trace length towards the supply pins of the IC) should be above the bandwidth of the logic [$1 / (\pi \cdot \tau_r)$], where τ_r is the voltage risetime of the logic.

If the decoupling capacitor is placed with every IC the signalreturn current may choose which path is most convenient, V_{EE} or V_{CC} . This choice is determined by the mutual coupling present between the signaltrace and one of the supply traces.

Between two decoupling capacitors, one for each IC, and the inductance, L_{trace} , formed by the supply traces, a series resonant circuit will result. This resonance is only allowed when it occurs at low frequencies (<1MHz) or when the Q of this resonance circuit is low (<2).

This resonance can be kept below 1MHz by using a choke with high RF-losses in series with the V_{CC} network and the decoupled IC. Too less RF-losses can be compensated by either adding a resistor in parallel or in series (Figure 6).

Table 3. Recommended Decoupling Capacitor

FAMILY NOISE-MARGIN	dI / dt			$C_{dec.}$ nF
	volt	mA	ns	
CMOS (5V)	1.75	2	100	0.5
TTL-LS	0.4	50	10	5.0
TTL-F	0.4	50	2–3	22.0
HCT	0.7	50	2–3	12.8
HC (5V)	1.2	50	2–3	7.5
ACT	1.7	175	1–2	35.0

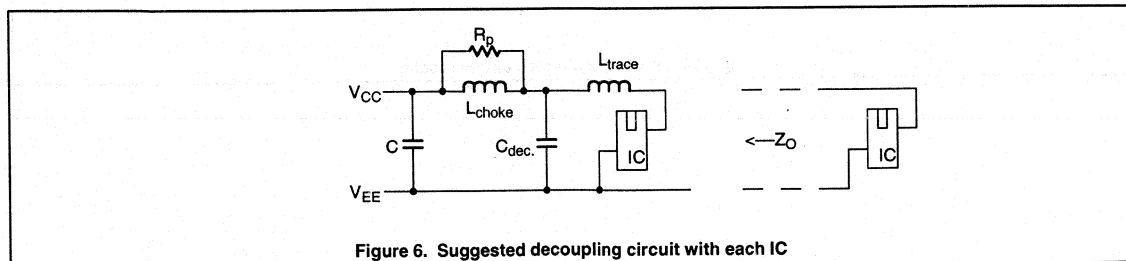


Figure 6. Suggested decoupling circuit with each IC

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The choke may never have an open core, because then it will either act as a RF-transmitter or a ferroceptor for magnetic fields.

Example:

$$1\text{MHz} \times 1\mu\text{H} \rightarrow Z_1 = 6.28\Omega \rightarrow R_s = 3.14\Omega$$

$$Q \leq 2 \quad R_p = 12.56\Omega$$

Above the resonance frequency, the characteristic impedance, Z_0 , of the "transmissionline" (in this case the impedance of the IC sees at its supply terminals) will be equal to:

$$Z_0 = \sqrt{L_{\text{trace}} / C_{\text{decoupling}}}$$

The series inductance of the decoupling capacitor and the inductance of the interconnecting traces have a negligible effect on the RF supply-current distribution,

when a choke of $1\mu\text{H}$, for example, is used. Still it determines the voltage fluctuations between the supply pins of the IC. With a 25% signal-to-noise margin dissipation by the power supply, the recommended maximum inductances, L_{trace} , are given in Table 4.

With the decoupling as suggested in Figure 6, the number of transmissionlines between the two ICs has gone down from 3 to 1 (see Figure 7).

CONCLUSION:

IV. By using proper decoupling with each IC: $L_{\text{choke}} + C_{\text{dec.}}$, only one transmissionline needs to be defined between the circuit blocks.

With high speed logic, $\tau_r < 3\text{ns}$, the total inductance in series with the decoupling capacitor needs to be low (see Table 4). A

trace, in series with the supply pins, of 50mm equals an inductance of 50nH. Together with the load conditions at an output, 50pF typical, this will give a minimum risetime of 3.2ns. If faster risetimes are required, shorter leads from the decoupling capacitor (preferred leadless) and shorter leads within the IC package are necessary. This can be obtained by using, for example, IC-decoupling capacitors, or better, using center (supply) pinned ICs in combination with small leadless ceramic capacitors with a 3E pitch (DIL). A multi-layer board with supply and ground planes can be another option. Further improvements can be reached by applying SO-packages with center pinned supply connections.

CONCLUSION:

V. When using fast logic: multi-layer panels should be used.

Table 4. Allowed (Supply) Series Inductance

FAMILY NOISE-MARGIN	dl / dt			L_{trace} nF
	volt	mA	ns	
CMOS (5V)	1.75	2	100	200.0
TTL-LS	0.4	50	10	20.0
TTL-F	0.4	50	2-3	4.0
HCT	0.7	50	2-3	7.0
HC (5V)	1.2	50	2-3	12.0
ACT	1.7	175	1-2	2.4

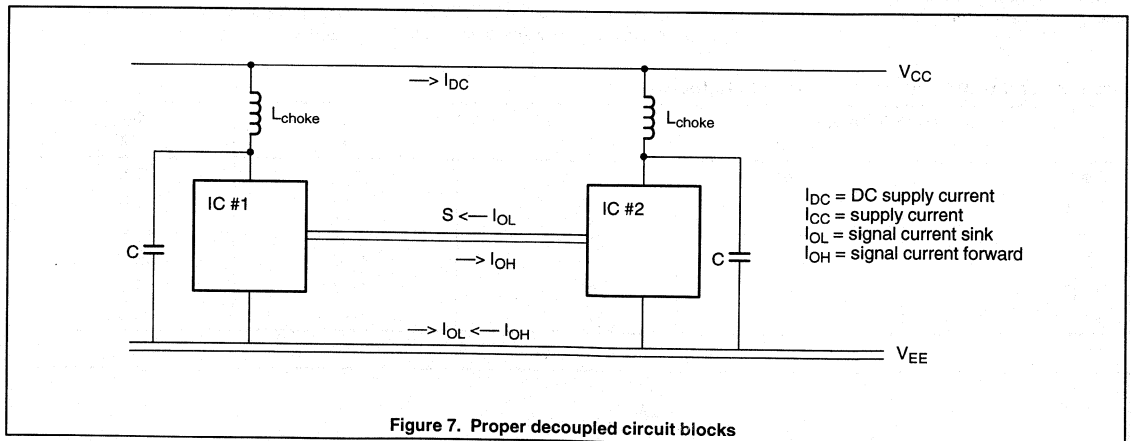


Figure 7. Proper decoupled circuit blocks

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

6. MINIMIZE TRACELLENGTH AND LIMITED LOOPAREAS

The maximum tracelength is determined by reflections which will occur at NON-terminated transmissionlines. The loopareas and tracelengths are limited by the EM-radiation which is allowed by mandatory requirements for the product. The latter requirements will directly apply to the PCB if it is used in an unshielded box/cover.

6.1. Allowed Tracelengths Due to Reflections

The first limitation of the tracelength is determined by functional requirements. A transmissionline can be made reflection free by either adding a load resistor at the end of the line, which without series capacitance will cause DC-dissipation, or by adding a resistor in series with the driver. In this case the output impedance of the circuit plus the series resistor must be equal to the characteristic impedance of the transmissionline.

When the transmissionline is NOT terminated the allowed trace length is determined by the noise-margin of the logic used, its bandwidth and the propagation delay of the line, which is assumed to be 5ns/m. The bandwidth

determines the dynamic noise margin which by approximation is inverse proportional to the disturbance pulse halfwidth time.

Applying the requirement that the noise, in this case the reflected signal, has to be less than 25% of the (dynamic) noise margin the tracelengths in Table 5 result.

CONCLUSION:

VI. A transmissionline should, if necessary, be series-terminated at the drivers side. If the trace lengths are long compared to those given in the table, END-termination is inevitable.

6.2. Allowed Loopareas Due to Radiation

The emission from a PCB (or a complete product) is limited to 100µV/m at 10 meters distance from the object at frequencies above 30MHz [FCC, IEC CISPR publications, class B]. This emission is determined by the product of the looparea, A, the loopcurrent, I, and the permeability of the medium within that loop, µ_r (commonly equal to 1). This product is called the magnetic dipole-moment, M.

In case a number of loops are present, operating at the same frequency or

clock-rate, the limit of the dipole-moment strength should be divided by √(n), in which n = number of loops, hence the signals will add as random noise.

$$M(\text{freq}) = I(\text{freq}) \cdot A \cdot \mu_r$$

The limit value for the magnetic dipole-moment can be calculated from the radiated power [7, 8]:

$$E = (7/r) \cdot \sqrt{P_{\text{rad}}}$$

$$P_{\text{rad}} = 31200 \cdot I^2 \cdot A^2 / \lambda^4 = 31200 \cdot M^2 / \lambda^4$$

where:

I = loopcurrent as function of frequency

A = looparea

λ = wavelength belonging to the frequency component of the loopcurrent

By substitution the following results:

$$E = (7/r) \cdot 176 \cdot I \cdot A / \lambda^2$$

Filling in the requirement, given above, that E ≤ 100µV/m at 10 meters distance from the source the following equation results for the looparea and current as function of frequency:

$$I \cdot A / \lambda^2 \leq 8.1 \cdot 10^{-7} [A], \text{ or}$$

$$M \leq 8.1 \cdot 10^{-7} \cdot \lambda^2 [A \cdot m^2]$$

Table 5. Allowed NON- or Series-Terminated Tracelength

FAMILY NOISE-MARGIN	volt	dt ns	MAXIMUM TRACELLENGTH (m)	
			NON-TERMINATED	SERIES TERMINATED
CMOS	1.75	100	14.3	— ¹
TTL-LS	0.4	10	0.4	0.5
TTL-F	0.4	2-3	0.08	0.15
HCT	0.7	2-3	0.14	∞
HC	1.2	2-3	0.24	— ¹
ACT	1.7	1-2	0.18	— ¹

NOTE:

1. If series termination is used in an asynchronous logic circuit design, attention must be given to the occurrence of metastability; especially symmetrical logic input-circuitry cannot decide whether the input signal is high or low and a non-defined output status may/will result.

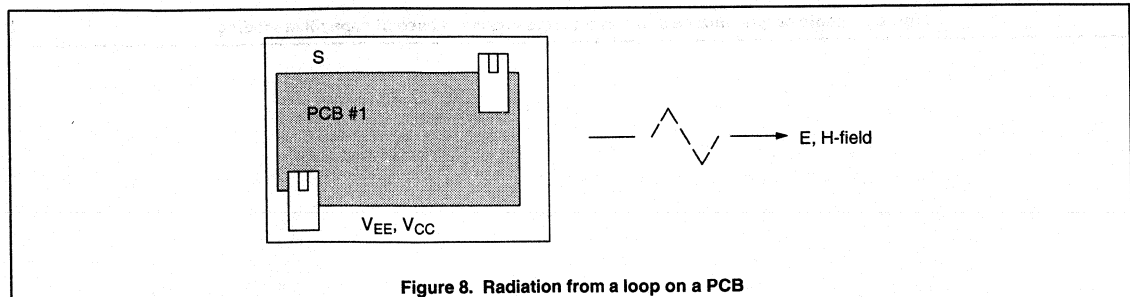


Figure 8. Radiation from a loop on a PCB

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The spectral current amplitude, for logic signals in the frequency domain, decrease above the bandwidth of the logic ($= 1 / \pi \cdot \tau_r$) proportional with frequency square. At this corner frequency, the radiation resistance of the loop still increases proportional with frequency square. Therefore one can calculate the maximum looparea which is determined by the clockrate or repetition rate, the risetime or bandwidth of the logic and the current amplitude in the time-domain. The current waveshape is derived from the voltage waveshape and the current halfwidth time is by approximation equal to the voltage risetime (Figure 9).

The current amplitude at the corner frequency ($= 1 / \pi \cdot \tau_r$) becomes:

$$I(f) = 2 \cdot I \cdot \tau_r / T$$

where:

- I = current amplitude in the timedomain,
- T = 1 / clockrate = period time,
- τ_r = voltage risetime $\equiv \tau_H$ current halfwidth time.

From this equation the maximum looparea at a clockrate for a certain logic family can be calculated. These loopareas are given in Table 6.

CONCLUSION:

VII. The maximum looparea is determined by the clockrate, the logic family (= output current) and the number, n, of simultaneous switching loops on that PCB.

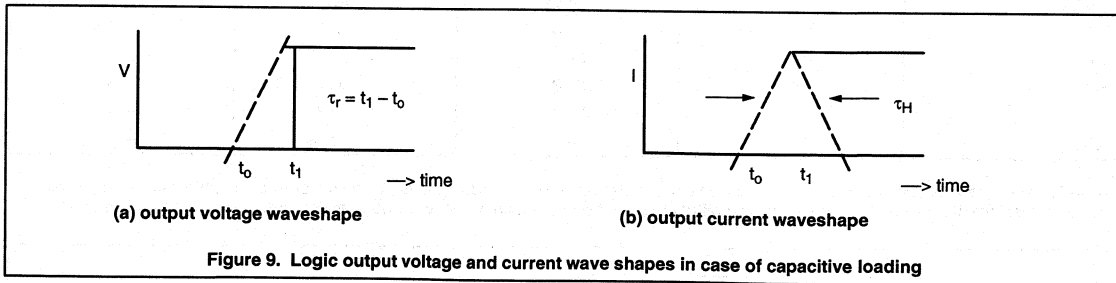
When a bi-layer is used with a thickness of 1.5mm, the maximum allowable tracelength, derived from the looparea, will be much less than the tracelength found from the reflection point of view. If clockrates are used above 30MHz, the use of a multi-layer will be inevitable. In this case the epoxy thickness depends upon the number of layers used and may vary between 60 – 300µm. When only a limited number of high clockrate signals are distributed on the PCB, careful routing, by using side-to-side traces, may lead to acceptable results on a bi-layer.

Table 6. The Allowed Single Looparea for Each Logic Family

FAMILY	dl mA	dt ns	MAXIMUM LOOPAREA IN mm ² AT CLOCKRATE OF:			
			f = 4MHz	f = 10MHz	f = 30MHz	f = 100MHz
CMOS	2	100	4.5 10 ⁶	1.8 10 ⁶	—	—
TTL-LS	50	10	1.8 10 ⁶	7200	2400	—
TTL-F	50	2–3	1.8 10 ⁶	1400	480	144
HCT	50	2–3	1.8 10 ⁶	1400	480	144
HC	50	2–3	1.8 10 ⁶	1400	480	144
ACT	175	1–2	515	206	69	21 (note 1)

NOTE:

1. In this case, when using common DIL packages, the looparea limit will be exceeded and additional shielding measures, together with proper filtering will be inevitable.



Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

6.3. Allowed Tracelength Due to Radiation

The allowed tracelength are even less, when the transmissionline is directly coupled to the system reference and an unshielded outgoing cable leaving the product, then the values found up to now. A simple diagram is given in Figure 10. The voltage drop between the two references with each IC has become the driving source of the antenna formed by reference system and the outgoing cable. The worst case radiation resistance of the antenna is assumed to be 150Ω and frequency independent [7]. The amplitude of the driving source, U, is now limited to:

$$P_{rad} = U^2 / 150\Omega.$$

Applying the radiation requirements as given earlier the voltage drop has to be:

$$U \leq 1.75mV.$$

The voltage drop is determined by the current amplitude, at the logic bandwidth's frequency, and the effective inductance of the transmissionline between these points.

$$U(f) = I(f) \cdot Z(f) = I(f) \cdot j \cdot \omega \cdot (L-M) = I(f) \cdot j \cdot \omega \cdot L \cdot (1-k)$$

Taking Table 1 and the current amplitude in the frequency domain, the tracelengths in Table 7 can be found.

This table shows that many practical applications shall not fulfill the radiation requirements.

In most cases, filtering or shielding of the outgoing cable, which leaves the product will be sufficient. Shielding of the entire product, plus necessary filtering, becomes inevitable

when the magnetic loop constraints are exceeded.

CONCLUSION:

VIII. Circuit designs shall be made in such a way that the voltage drop between references shall not directly excite an antenna being any outgoing cable.

Simple approximations will give the number for the required filtering or shielding performance whenever necessary. These can be found by using the Tables 6 and 7 and counting the number of correlated sources in the product.

In the chapters 7, 8, and 9 some basic information is given about the cable shield performance and filtering techniques.

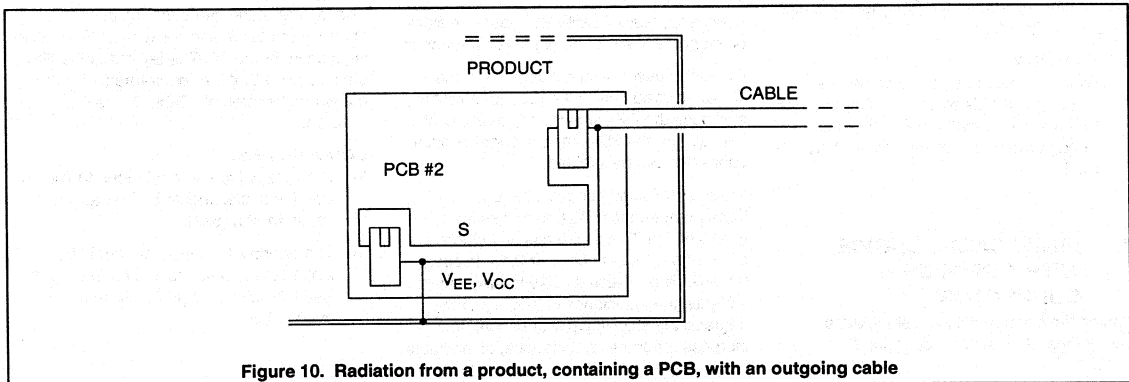


Figure 10. Radiation from a product, containing a PCB, with an outgoing cable

Table 7. Maximum Tracelength in Case of Direct Radiation

FAMILY	di mA	dt ns	ALLOWED TRACELNGTH IN mm BI-LAYER / MULTI-LAYER			
			f = 4MHz	f = 10MHz	f = 30MHz	f = 100MHz
CMOS	2	100	108 / —	44 / —	—	—
TTL-LS	50	10	4.3 / —	1.75 / —	0.6 / —	—
TTL-F	50	2-3	4.3 / 55	1.75 / 40	0.6 / 4.4	— / 2.2
HCT	50	2-3	4.3 / 55	1.75 / 40	0.6 / 4.4	— / 2.2
HC	50	2-3	4.3 / 55	1.75 / 40	0.6 / 4.4	— / 2.2
ACT	175	1-2	— / 15.4	— / 3.2	— / 2.1	— / 0.62

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

7. PLACEMENT OF THE CONNECTORS

All connectors, which provide the interconnections to other panels and/or units, must be placed as close as possible to one another. In this way common-mode currents, which are induced in those cables, will NOT flow through the traces of the circuit on the PCB. In addition, voltage drop between references on the PCB will not excite the (antenna)-cables.

To avoid such common-mode effects, it may be necessary to make a separation between the reference-strip near to the connectors and the groundplane, groundgrid or reference of the circuitry on the PCB. This groundstrip shall, if applicable, be connected to the metal cover of the product. From this separate groundstrip, only high impedances; inductors, resistors, reed relays and opto-couplers are allowed in between these two grounds. This will be explained when the filter networks are described, Chapter 9.

CONCLUSION:

IX. All connectors need to be placed as close as possible to one another in order to prevent external currents running through the traces or reference of the PCB.

8. RIGHT CABLE CHOICE WITH A PROPER CONNECTOR

Cables have, when they are shielded, a transferimpedance, see Appendix B.

Determined by the amplitude and the frequency content of the signals flowing through these cables a choice shall be made. In case cables, leaving the enclosure of the product, contain data above a 10kHz clockrate, shielding will be inevitable (product requirement). This shielding shall be connected to ground (metal cover product) on both ends of the cable, this to assure that the shield acts both as an electric and a magnetic shield.

If separate grounds are used, this shall be done to the "connector-ground" instead of the "circuit-ground".

In case the clockrate is above 10kHz and below 1MHz *and* the risetime of the logic is kept as slow as possible, an optical coverage of 80% or more or a transferimpedance which equals less than 10nH/m will do. Above 1MHz clockrates, better shielded cables are always necessary.

In general, coaxial cable excluded, the shield of the cable shall not be used as signalreturn.

By using passive filters in series with the signal input/outputs to the ground/reference, to reduce the RF-content, the necessity of a high quality shielding and the corresponding connector can be avoided.

A proper shielded cable will have a transferimpedance equal to or less than $l_j \cdot \omega \cdot 10 \text{ nH/m}$. Every wire has an inductance of 1 nH/mm (refer to chapter 2.1.). In case the shielding of such a cable is wrapped into a pigtail, the inductance of that pigtail will degrade the shielding performance, thus increase the transferimpedance, of the cable.

CONCLUSION:

X. A good shielded cable deserves a proper connector.

9. PROPER USE AND PLACEMENT OF FILTERS AND FILTER PARTS

Signal bandwidth reduction shall be achieved by using RC low-pass filters. In case the voltage drop across the series resistor is unacceptable an inductor with high RF-losses shall be used. The LC low-pass filter will always show resonances and therefore its Q must be kept low.

The filter can be used in two directions; namely, to prevent emission from the PCB *and* to improve the immunity of the board to external sources, e.g. RF-transmitters, ESD, etc.

The lay-out of the interconnection of the shield of the cable and a low-pass RCR-filter is given in Figure 12. The lay-out of the filter shall be such that the requirements for the maximum tracelength, Table 7, are not violated.

CONCLUSIONS:

XI. Currents, which do not belong to the circuit signals, should be by-passed using another path.

XII. The bandwidth of signals should be limited to the lease functional bandwidth. Use the slowest logic family suitable for the function.

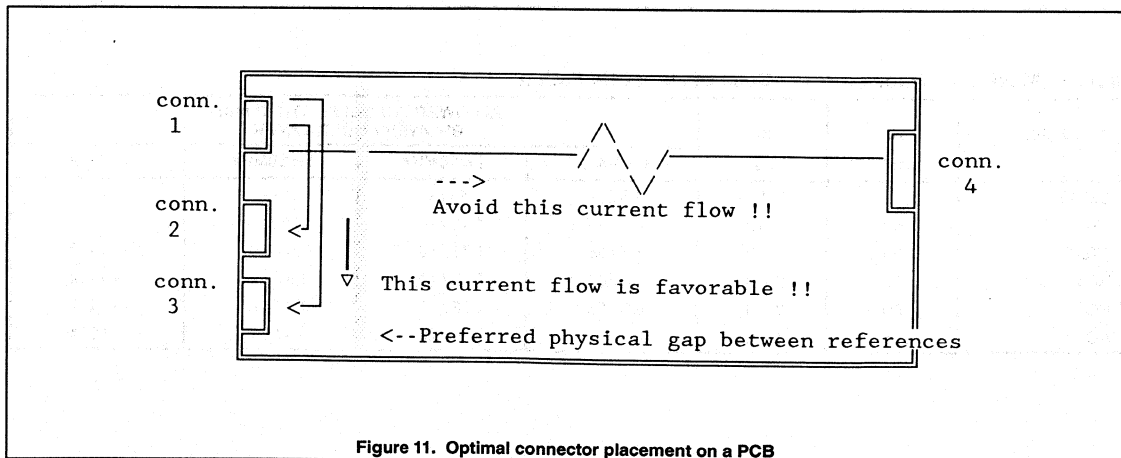
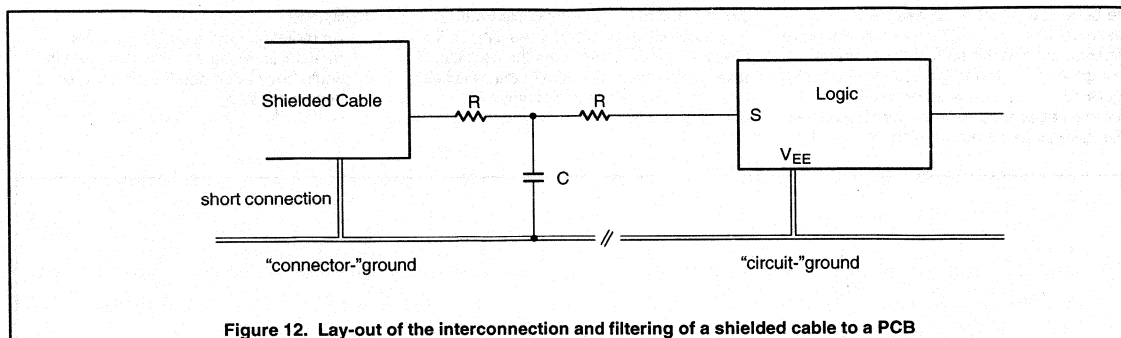


Figure 11. Optimal connector placement on a PCB

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001



10. PCB DEMO-BOARD, ROUTING AND DECOUPLING EFFECTS

An EURO-card PCB (100 × 160 mm²) has been chosen to demonstrate the effects of signal lines and their signal returns with respect to magnetic radiation.

The board contains a relaxation oscillator, created by 3 inverters (NANDs) and an RC-network (1kΩ, 560pF), which will produce a squarewave voltage signal. The frequency will be determined by the used logic and its threshold voltages. This oscillator is placed in one corner of the board together with some switches to change signal-return path and supply decoupling. In the opposite corner of the PCB another quad NAND has been placed as a capacitive load. These NANDs are all cascaded and will change status with some skew. The last NAND is terminated by a resistor. The supply decoupling of this IC can be altered as well. The diagram of the circuit with the switches is given in Figure 13 and the physical layout of the PCB and component placement are given in Figure 14.

The layout has been chosen such that the supply traces are as close as possible to one another, which is commonly arranged by a proper CAD-tool. In parallel to the signal trace a signal return trace has been placed, according to Chapter 4. At the supply pins of the ICs decoupling capacitors are added to each IC. By means of jumpers or switches a

series inductor may be short-circuited or added to the circuit.

In total 4 relevant situations can be evaluated which are given in Table 8.

Situation 1.

Supply decoupling only takes place by the capacitors and the signal return has been established through the supply trace V_{EE} (V_{DD}).

Situation 2.

Supply decoupling only takes place by the capacitors and the signal return has been established through the supply trace V_{EE} (V_{DD}) and a trace in parallel to the signal trace. The coupling between signal and signal return determines that only a small portion of the signal-return current will flow through the supply traces.

Situation 3.

The supply trace, V_{EE} (V_{DD}), has been taken out and the I_{CC} and signal-return current have to flow through the trace next to the signal line. The high-frequency components of the signal-return current shall still flow through the V_{CC} (V_{SS}) trace due to the (de-)coupling capacitors at the supply pins of the ICs.

Situation 4.

By adding the inductors, with sufficient RF-loss, in series with the supply trace, V_{CC} , of both ICs ALL the signal-return

current will have to flow through the trace next to the signal line and radiation from the loop on the PCB has diminished.

The effects with respect to the radiation can be measured both in time as in frequency-domain. The latter has the advantage of showing the differences between situation 3 and 4 which are marginally discernable on an oscilloscope. These RF-effects are of extreme importance with respect to radiation as explained in Chapter 6.

To demonstrate the phenomena on an oscilloscope, a 50 (100) MHz bandwidth version shall be used. A small (electrically shielded) loop shall be used as measuring probe. If not available, a loop made by using a voltage probe of which the ground strap is short-circuited to the measuring tip can be used. This "loop" shall be placed on the PCB as some secondary loop near the supply traces. On the oscilloscope the effects of the positions of the switches can be observed. Measured results in the time domain are given in Appendix C.

In case a spectrum analyzer is used, an electrically shielded measuring loop shall be placed on the PCB as some secondary loop near the supply traces. On the screen the effects of the positions of the switches can be observed. Measured results in the frequency domain are given in Appendix D.

Table 8. A List of the Relevant Configurations of the Switches on the Demo-board with Respect to Emission Measures.

SITUATION	POSITION OF THE SWITCHES			
	SW 1	SW 2	SW 3	SW 5
1	ON	ON	ON	OFF
2	ON	ON	ON	ON
3	OFF	ON	ON	ON
4	OFF	OFF	OFF	ON

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The behaviour of the PCB has been simulated with PHILPAC for a unity sinewave signalsource and the sum of the currents through the V_{EE} and V_{DD} traces are given in Figure 15. In the simulated circuit the parasitic capacitance across the chokes has been taken into account, which leads to the

same result at higher frequencies with respect to situation 3 and 4. As long as the measuring loop is kept from the oscillator area, which itself (also due to the switches) radiates the effects can be shown unambiguously.

REMARK:

As radiation from a certain passive network is reciprocal, the same results could have been obtained in case of an immunity set-up.

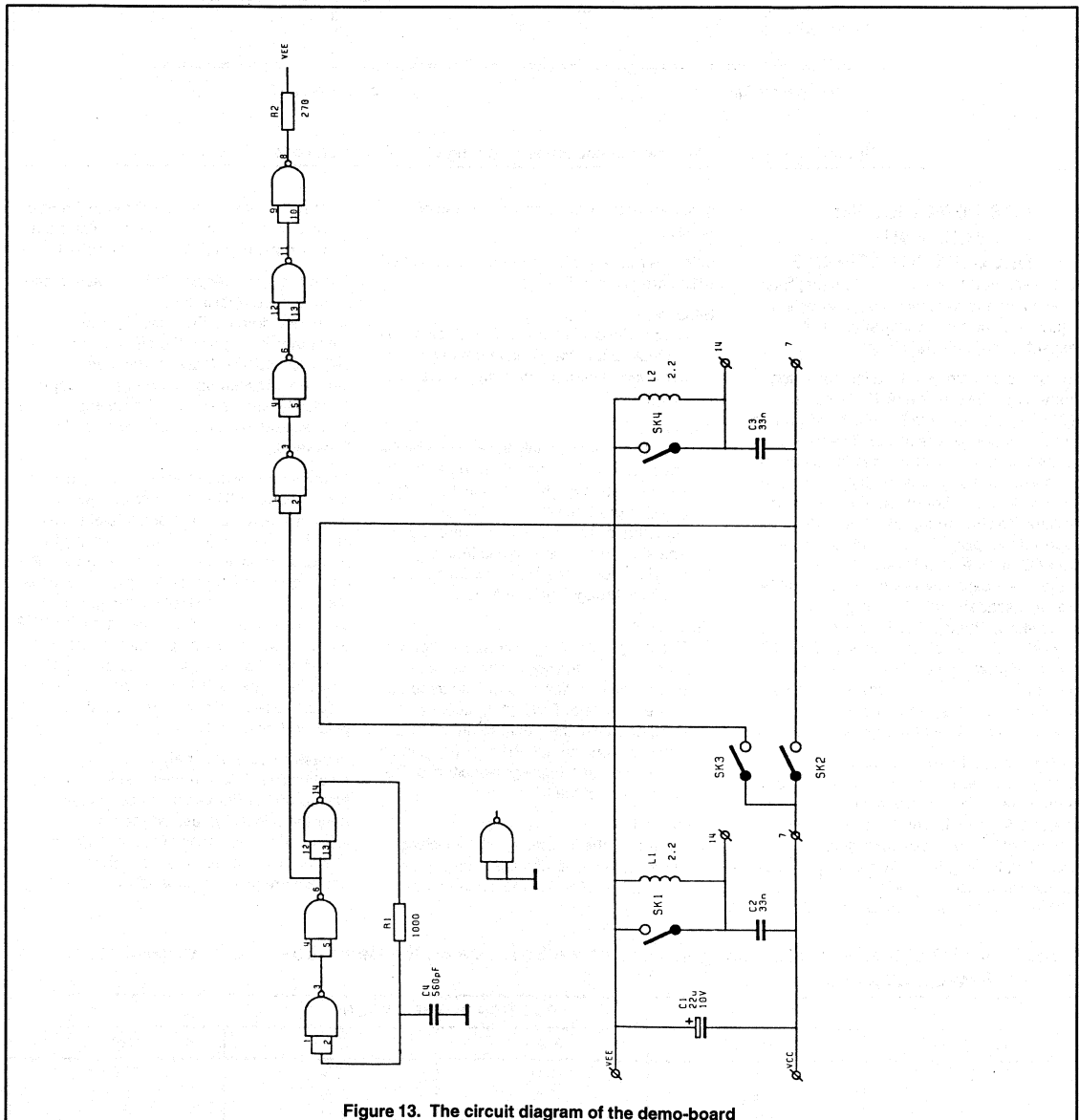


Figure 13. The circuit diagram of the demo-board

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

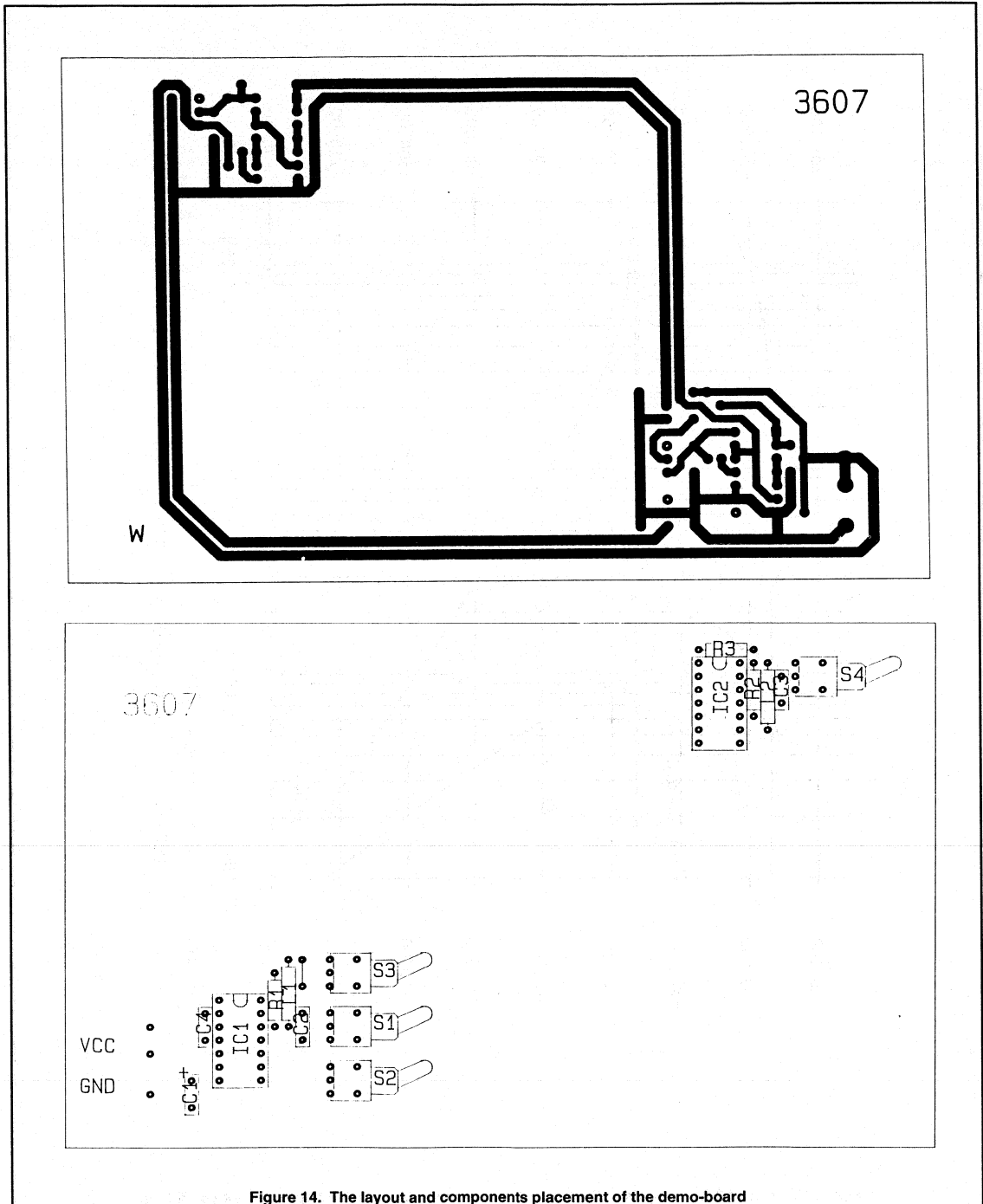


Figure 14. The layout and components placement of the demo-board

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

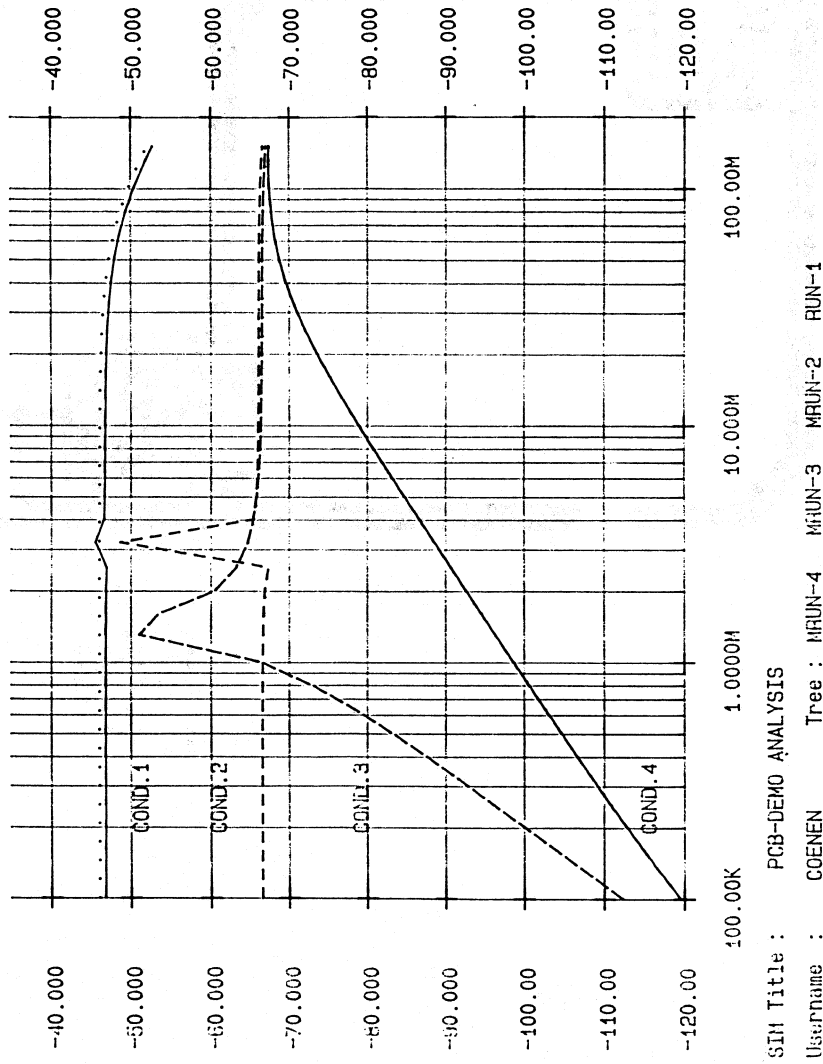


Figure 15. PHILPAC AC analysis of the EM radiation behavior of the demo-board in the 4 conditions

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

11. REFERENCES

- [1] EMC in TV receivers and monitors, D. Teuling, ETV 8702 Philips Components, 1987, Eindhoven.
- [2] Electromagnetic compatibility syllabus, J.J. Goedbloed, November 1987, Philips Central Training Department, The Netherlands.
- [3] Low frequency approach to the electromagnetic radiation of the printed circuit boards, M. Coenen, Philips Research Lab. Note 316/85.
- [4] Radiated emissions from common-mode currents, C.R. Paul, IEEE EMC symposium notes, 1987.
- [5] EMC and loop inductances on printed wiring boards, B. Danker, 7th Symposium on EMC, Zurich, 1987.
- [6] Electromagnetic compatibility design and layout guidelines of printed circuit boards, M.J.C.M. van Doorn, Philips Video Display Products, Pre-development, AR6-60.07, 1987.
- [7] An evaluation method to characterize the EMC performance of PCBs containing ICs, M.J. Coenen, ESG 8801, Philips Components, 1988.
- [8] Antenna theory, analysis and design, C.A. Balanis, Harper and Row Publishers, New York, 1982.
- [9] Electromagnetic theory, J.A. Stratton, McGraw Hill, New York and London, 1941.
- [10] Fast TTL Logic series, Handbook IC15 Philips, 1988.
- [11] Advanced CMOS Logic Data Manual, Signetics/Philips, 1988.
- [12] Taschenbuch der Hochfrequenztechnik, H. Meinke, F.W. Gundlach, Springer Verlag, Berlin, New York, 1968.
- [13] Transmission-line methods aid memory-board design, E.A. Burton, Electronic Design, December 1988.
- [14] EDN's advanced CMOS logic ground-bounce tests, EDN, March 1989.

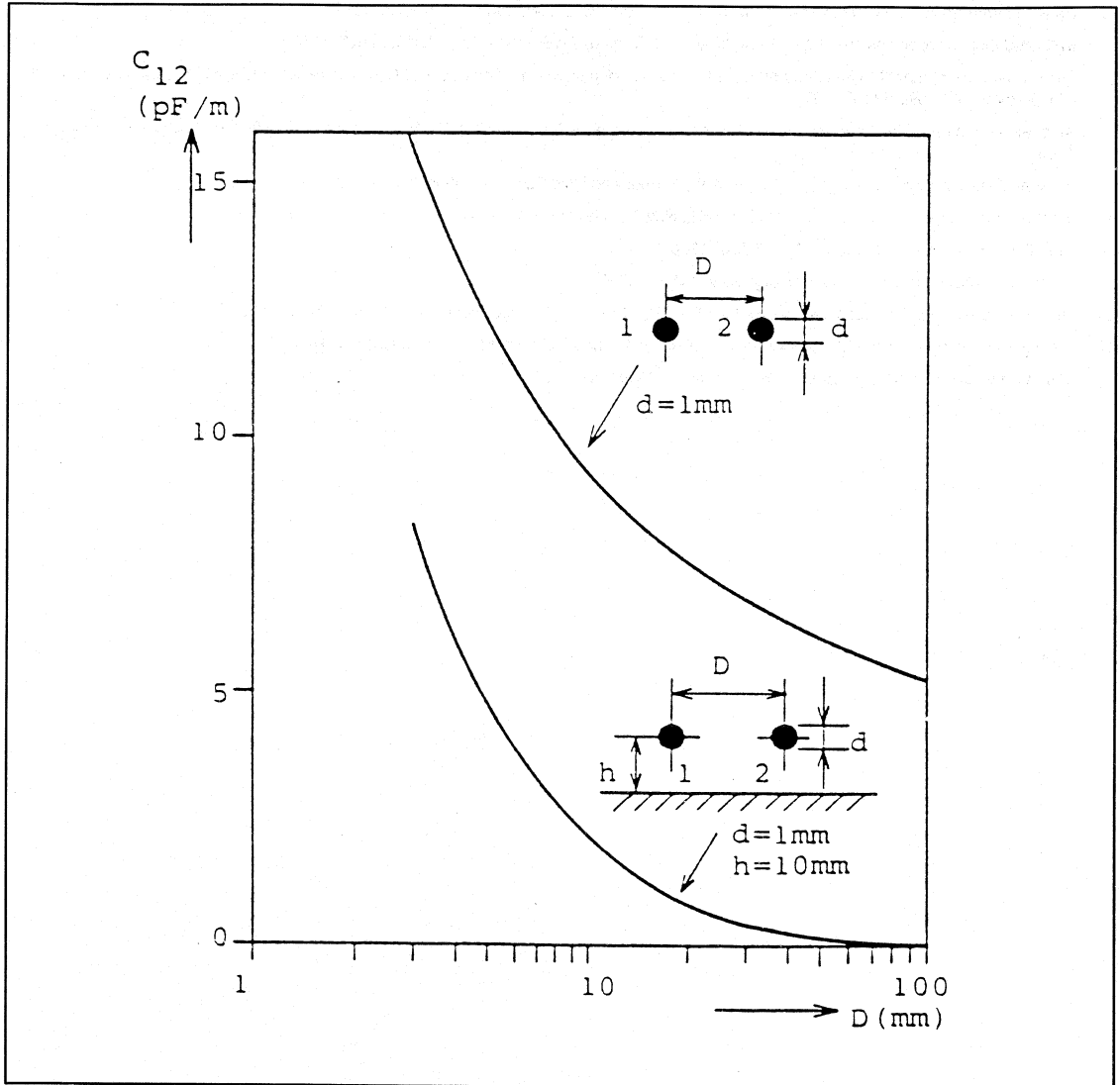
Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

APPENDIX A. CAPACITIVE COUPLING BETWEEN TRACES

In this appendix the graphical presentation is given of the capacitive coupling between two traces in free space and for two traces above a reference plane [12, form. 24.25].

It shows the necessity of a reference plane at a height, h , closer to the traces than the distance, D , to reduce the capacitive coupling between the traces.

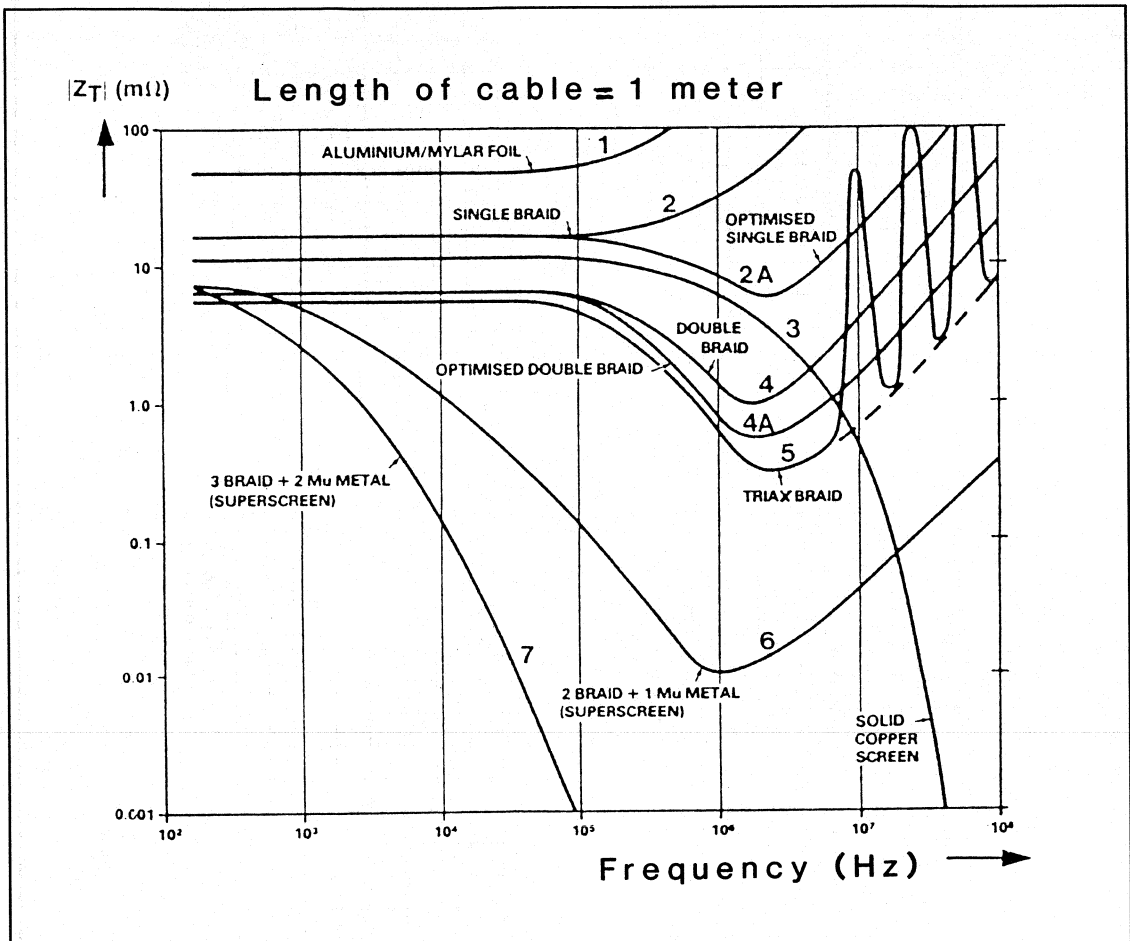


Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

APPENDIX B. THE TRANSFER IMPEDANCE OF VARIOUS CABLE SCREENS

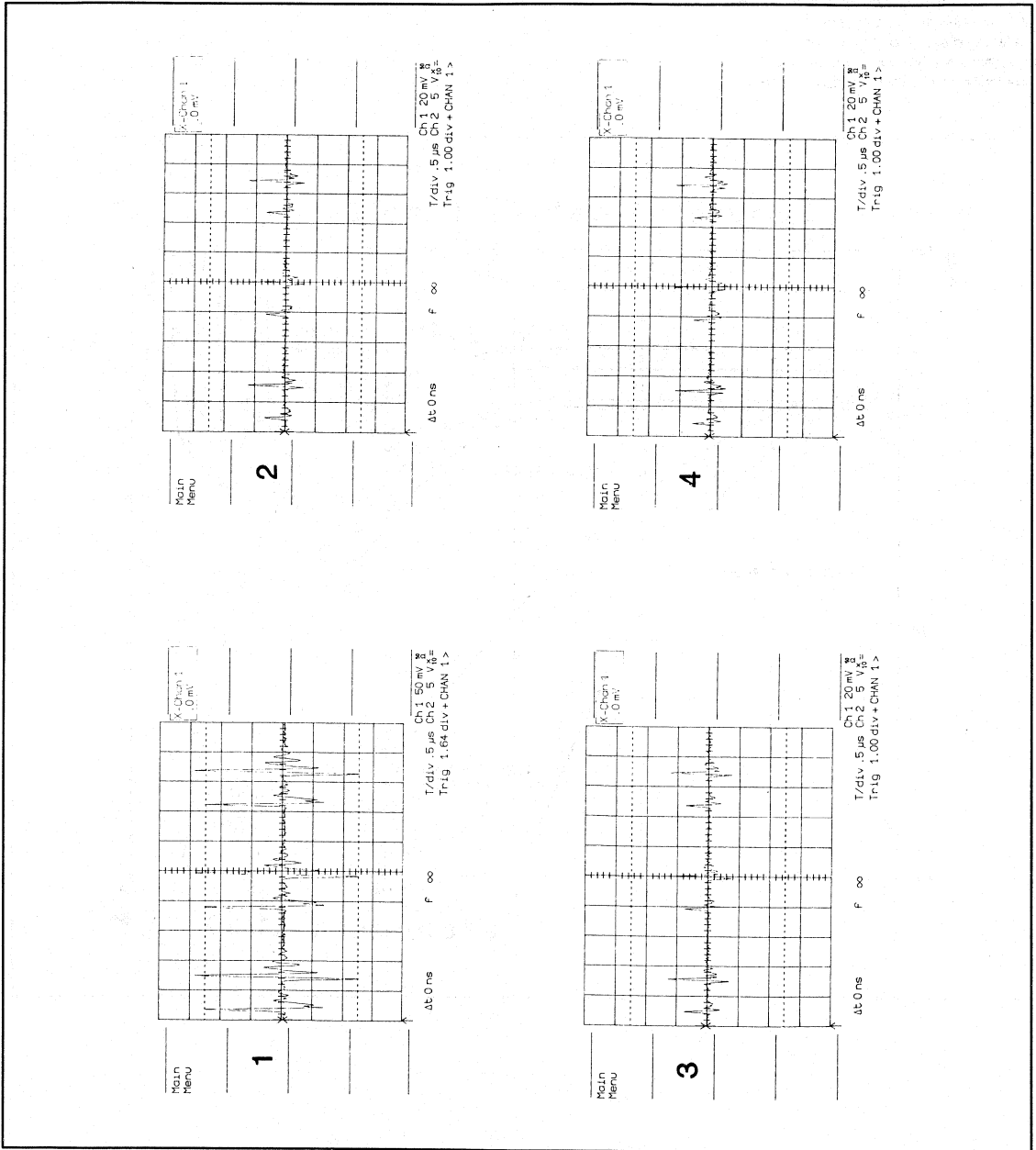
The transfer impedance, Z_T , is the relation between the current through the screen due to an external source and the induced voltage across the nominal load impedances of that cable. Further information about the measuring method to obtain information of the screening efficiency or the transfer impedance can be found in IEC publication 96.



Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

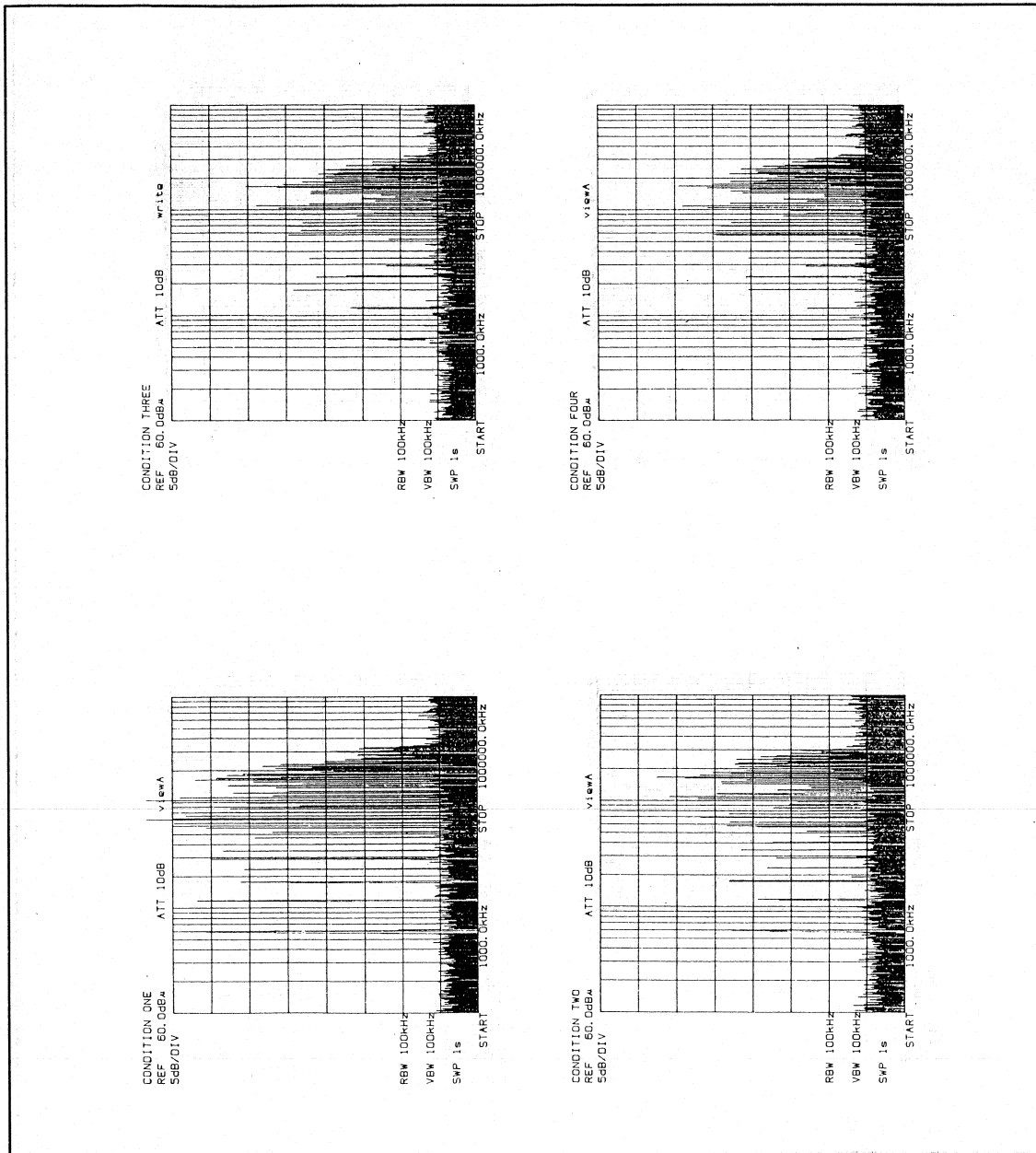
APPENDIX C. MEASURED RESULTS IN THE TIME DOMAIN, 150MHz BANDWIDTH, FROM THE DEMO-BOARD, CONTAINING A 74HCT00, IN THE 4 CONDITIONS DESCRIBED IN CHAPTER 10.



Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

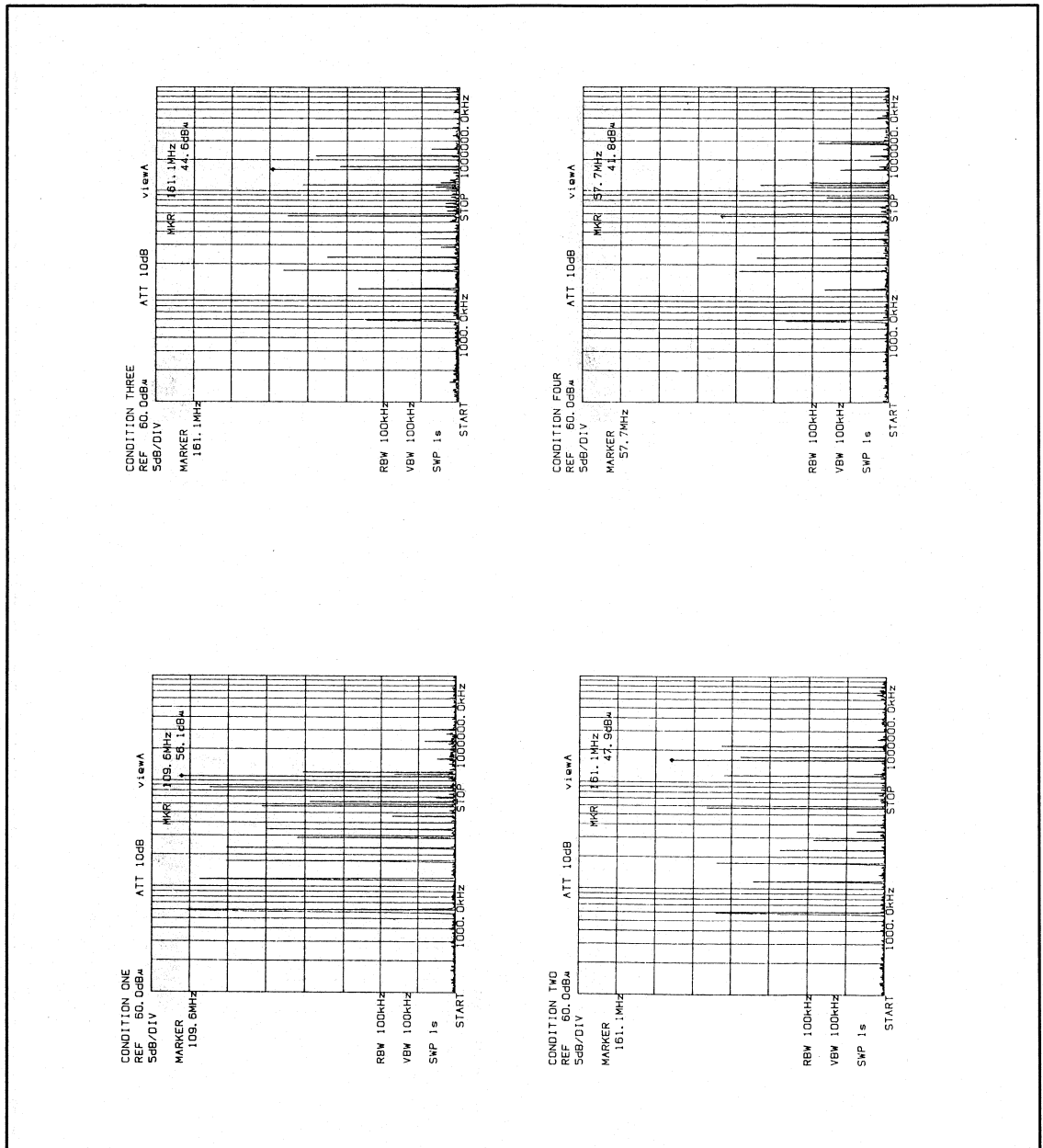
APPENDIX D1. MEASURED RESULTS IN THE FREQUENCY DOMAIN, PEAK DETECTION, FROM THE DEMO-BOARD, CONTAINING A 74HCT00, IN THE 4 CONDITIONS DESCRIBED IN CHAPTER 10.



Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

APPENDIX D2. MEASURED RESULTS IN THE FREQUENCY DOMAIN, PEAK DETECTION, FROM THE DEMO-BOARD, CONTAINING A 74HCT00, IN THE 4 CONDITIONS DESCRIBED IN CHAPTER 10.



Workbench EMC evaluation method

EIE/AN91001

1 INTRODUCTION

In many mass manufactured electrical and electronic products, printed wiring boards (PWBs) are used in a non-shielded application because it would be too expensive to do otherwise, e.g. hand-held cassette players, telephones, televisions, etc. As such, the product related EMC requirements directly apply to the (main) PWB containing semiconductors. To meet economic constraints, EMC solutions need to be taken, as far as possible, within the semiconductors, such that the needs for external measures diminish. In order to select between the various applications suggested by different vendors, an EMC qualification is required too.

Generally, the designer will face problems with CAD tools for PWB and semiconductor lay-out. The main problem is that all routers known are random routers which can do their functional job quite well on a bi- or multi-layer PWB, or with single or double metal processes. It will therefore be such that if the PWB or semiconductor is routed several times, using the same input, the results can be totally different. EMC results are commonly bad, mainly because the router is not told which lines need priority and which need to be close together.

An additional problem to face is that both product and semiconductor developments need to go faster, by using all available tools. By adding as few as possible external (passive) components and considering geometrical constraints given by the automatic assembly tools we hope that it complies with the often stringent **mandatory** EMC requirements.

The above given problems ask for an EMC evaluation method, by which the designer can, at any stage of the design, determine the degree of satisfying the EMC requirements. This EMC evaluation technique must be easy to use, easy to handle and not too expensive to assure usage of this method. The concept is based on the technical information now available in draft IEC 801-6. This concept will be explained in chapter 2.

Workbench EMC evaluation method

EIE/AN91001

2 TEST METHOD

2.1 EMC regulations and standards

With respect to EMC-standards we can concentrate ourselves best to the generic emission and immunity requirements given in European Standards EN 50081-1 and EN 50082-1 which become legally enforceable from 1992 onwards, especially to the European Market. In these standards reference is made to either IEC documents or CENELEC standards in which certain disturbance or emission phenomena are depicted. For most product groups, the IEC CISPR 11/22 documents or CENELEC European Norm 55011/22, class B apply for RF-emission, whereas for immunity phenomena IEC 801-2 through -6 or CENELEC 55024-x apply.

In our particular case, considering IC-development and IC-application, the most important document is draft IEC 801-6 as it can be applied both to immunity and emission. The validity is given by the fact that the EM-radiation properties of cables and wires connected to the Device or Equipment Under Test (DUT or EUT) are substituted by simple passive networks. As passive networks have reciprocity both emission and immunity can be evaluated in the same set-up.

2.2 Basic Concept

With the method, according to draft IEC 801-6, immunity to **conducted** RF-disturbances is tested. The set-up simulates the EM radiation effects by coupling the induced disturbing signals through Coupling/Decoupling Networks (CDNs) via the cables and wires to the PWB under test in a defined way, Fig.1. This method is applicable in the frequency range 150 kHz up to 230 (1000) MHz. The set-up with CDNs simulates "passive" cables which are, from the radiation point of view, at resonant length. The dimension of the PWB(s) between these cables and wires is assumed to be short compared to wavelengths involved.

According to the existing radiation measurement procedures, either the cable contributions are eliminated by adding ferrites on them to serve reproducibility (EN 55020) or the cables shall be adjusted in length and geometry with each frequency to obtain maximum radiation (EN 55011/22). With the conducted "interaction" method, which simulates radiated RF-field phenomena, both problems are solved.

Note 1: This method also reproduces the electric and magnetic near-fields to the PWB, associated with the source of disturbance (E and H in figure 2).

Note 2: The method of publication 801-6 does not provide the injection (or measurement) of the current from an ideal current or voltage source. It rather provides the coupling of the disturbance signal from a real source, having a "radiation" resistance of 150 Ω , i.e. short-circuit current as well as open voltage are limited.

CDNs are mainly defined by the common-mode impedance represented at the EUT-port side, which needs to be about 150 Ω to reference ($\pm 20 \Omega$, freq. ≤ 30 Mhz, $+60/-45 \Omega$, freq. > 30 Mhz). Every CDN fulfilling the impedance requirements given in draft IEC 801-6 can be used. Typical application examples are given in the figures 3a) for shielded and 3b) for unshielded cables. In figure 3c) a simplified drawing for mechanical construction is given. By using new NiZn ferrite ($\mu_r \geq 1000$), the entire frequency range can be covered by using one toroid on which we have 17 windings, $L_{(150 \text{ kHz})} \geq 280 \mu\text{H}$, and one bead-on-cable only.

Workbench EMC evaluation method

EIE/AN91001

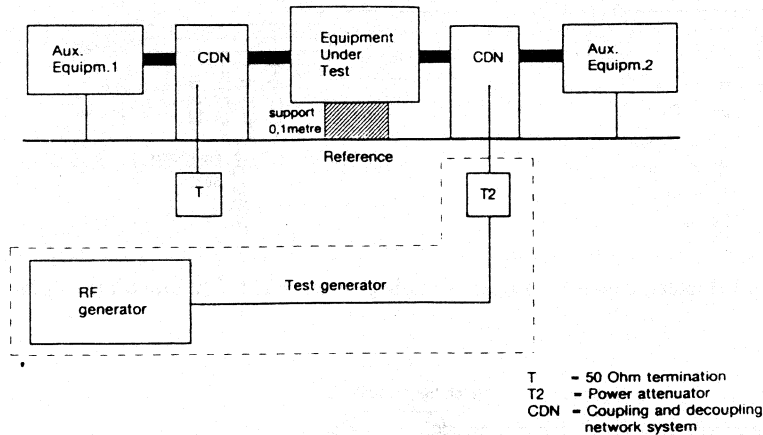
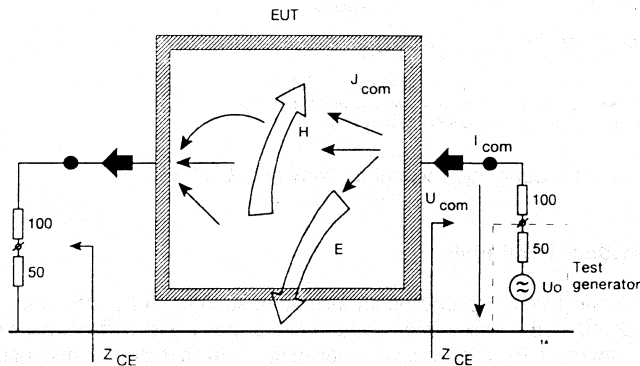


Fig.1. Schematic set-up for immunity test to RF conducted disturbances.



- Z_{CE} : Common-mode EUT-point impedance of the coupling and decoupling network system, $Z_{CE} = 150 \Omega$.
- Note: The 100 Ω resistors are given by the coupling and decoupling networks. The left input is terminated by a (passive) 50 Ω load and the right input is loaded by the test generator.
- U_o : Test generator output voltage
- U_{com} : Common-mode voltage between EUT and reference plane
- I_{com} : Common-mode current through the EUT
- J_{com} : Common-mode current density in the EUT
- E, H: Electric and magnetic fields

Fig.2. Equivalent circuit of Fig.1 to explain the electromagnetic near-fields approximated by common-mode currents and voltages induced by a RF-source according to the immunity method to conducted disturbances.

Workbench EMC evaluation method

EIE/AN91001

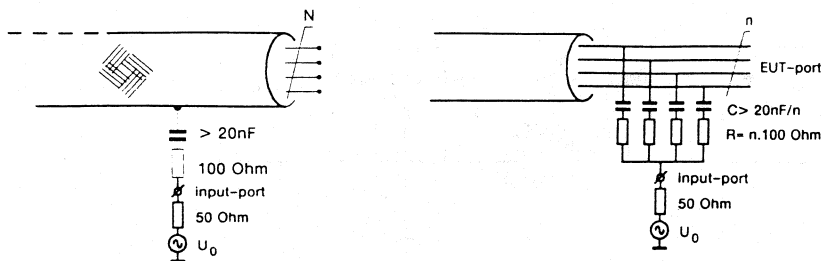


Fig. 3a. Coupling to shielded cables Fig. 3b. Coupling to un-shielded (multi-wire) cables.

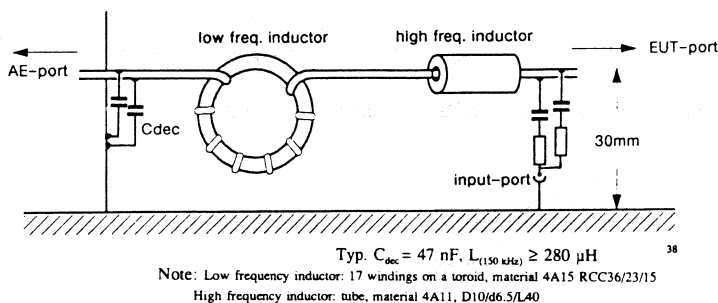


Fig. 3c. Mechanical drawing for the Coupling/Decoupling Network (CDN).

2.3 Test set-up for individual PWB units

According to draft IEC 801-6, the PWB is placed on an insulating support, 0,1 metre above an earth reference plane. If the PWB will be used in a cabinet where a metal part is more near to the PWB than 0,1 metre, e.g. an adjacent PWB or a metal floor plate, then that shorter distance must be used. The earth reference plane shall exceed the projected geometry of the PWB and the used CDNs on all sides by at least 0,2 m.

On all cables measures to obtain high common-mode impedances at the frequencies of interest or CDNs must be inserted. The number of CDNs should be limited (between 3 and 5) by restricting oneself to the representative functions and main (disturbing) current distributions occurring to the application in practice. The CDNs need to be placed directly on the earth reference plane, making proper contact to it, at a maximum distance of 0,3 metre from the PWB. The cables between the CDNs and the PWB shall be as short as possible. Their height over the earth reference plane must be kept between 30 and 50 mm (as long as possible).

All auxiliary equipment, AE, required for the defined operation of the PWB, according to the specifications of the product, must be connected through high common-mode impedances or through CDNs to the PWB, e.g. communication, modem, printer, sensor, etc. This shall also be

done for all auxiliary equipment necessary for ensuring proper data transfer and assessment of the functions.

*Warning 1: As measures to obtain high common-mode impedances **and** CDNs are transparent for functional signals, part of the disturbance signal may enter the auxiliary equipment in this way. As such, filtering measures will be inevitable.*

2.4 Workbench Faraday cage

The present proposal, suitable for small application boards and semiconductor evaluation, is a table-top size Faraday cage where all the connections to DC-supply and other auxiliary equipment are made through filters mounted on the wall of the Faraday cage. The wires and cables from these filters to the PWB need to be wrapped on a ferrite toroid, Philips NiZn material 4A15, to create a high common-mode impedance towards the walls of the Faraday cage (= earth reference plane), similar to the construction given in Fig.3c. Then, by means of resistors defined impedances, 150 Ω (100 Ω in series with an external 50 Ω coaxial load), are applied to simulate the radiation or reception performance of the cables or wires which might be connected to this application board in practical (product) application.

Note 3: When a workbench Faraday cage is used, the earth reference plane is extended all around the PWB under test. As a result of the dimensions of the workbench Faraday cage, distance between PWB and CDN will remain $\leq 0,3$ metre.

To keep this method simple, the maximum number of common-mode measurement points is set to 3. All other wires and cables must be provided with a RF-blocking (high common-mode impedance) device towards the feed-through panel on the wall of the Faraday cage.

The common-mode measurement points on the PWB selected for evaluation are:

- at the DC-power supply connector,
- at the input port connector and
- at the output port connector.

Dependent on the implementation of the application board, containing one or several ICs, the emission or immunity performance can be measured, Fig.6 and 7. In those set-ups the coaxial 50 Ω loads outside the Faraday cage shall be exchanged in turn with either the selective voltmeter (spectrum analyzer) or the disturbance source.

For worst-case testing the three connector positions, which are selected as common-mode points, are distributed all alongside the PWB, Fig.4a. This arrangement is chosen because one can not predict the geometrical lay-out the customer is going to use in his product.

Note 4: Only if an application is required (to fulfil the requirements) such that all connectors are placed on one side only, Fig.4b, it must be tested as such. As a result, this condition then, shall be clearly stated in the application report.

The determination of the common-mode points of the selected ports is based on the cables or wire-geometries likely to occur in product application. Furthermore, dependent on the product application, the appropriate CDN shall be selected.

Workbench EMC evaluation method

EIE/AN91001

- When shielded cables are used, the shielding is referred to as common-mode point of that port (towards the wall of the Faraday cage), no matter how many wires are within that shielding.
- When unshielded cables are used, the common-mode impedance shall be established by placing a number of resistors to each individual wire such that the total common-mode resistance equals $100\ \Omega$ (+ $50\ \Omega$ external) towards the reference, being the wall of the Faraday cage. In series with those resistors capacitors ($C_{\text{total}} \geq 20\ \text{nF}$) must be applied such that the impedance requirements are still met.

In product applications where an IC is fed by a signal source properly defined, i.e. signal and ground return are adjacent tracks, separation $\leq 1\ \text{mm}$, **and** the tracks are short, length $\leq 0,1\ \text{metre}$, a coaxial CDN-type can be applied. The same applies for the output configuration. Commonly, supply traces are not kept adjacent properly from IC to supply and therefore an unbalanced two-wire CDN should be used. If an IC is properly decoupled, supply and ground will be RF-short-circuited and a coaxial CDN can be applied.

Warning 2: With analog circuits, outputs often cannot handle high capacitive loading which is represented by CDNs and its cables. As only the demodulated component at 1 kHz is of interest, an RC-filter need to be used in-between output and CDN, see Fig.5.

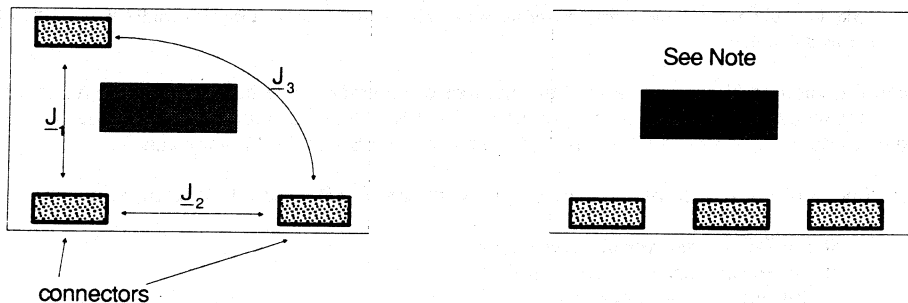


Fig.4. Typical arrangements of inputs, outputs and supply alongside the IC.

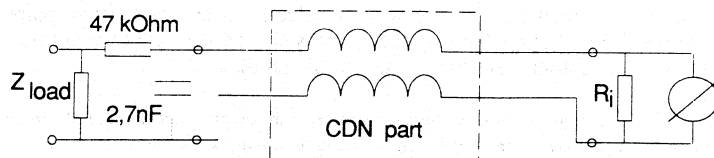


Fig.5. Filter/impedance matching network necessary to lower capacitive loading at IC outputs.

Workbench EMC evaluation method

EIE/AN91001

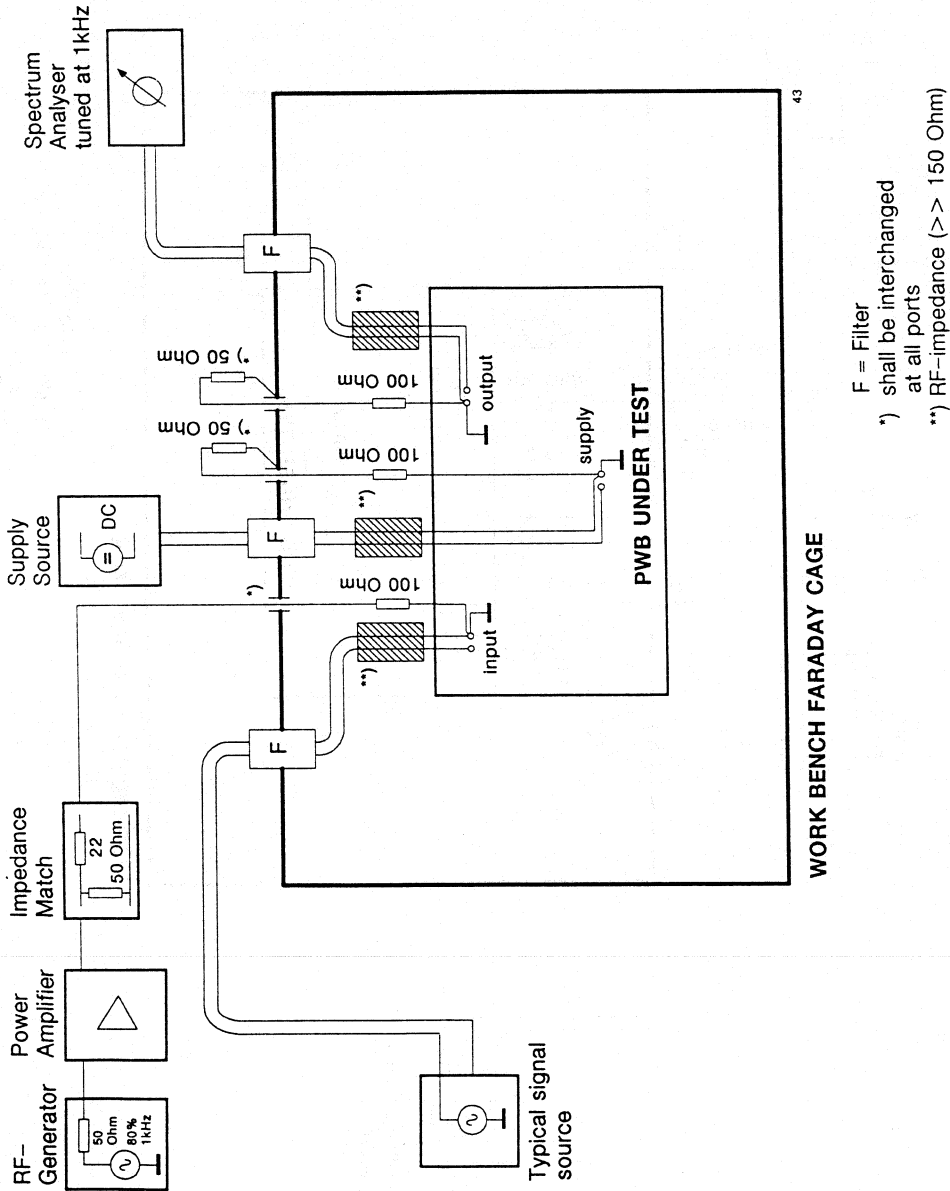


Fig.6. Simplified set-up for immunity testing of the PWB using the workbench Faraday cage.

Workbench EMC evaluation method

EIE/AN91001

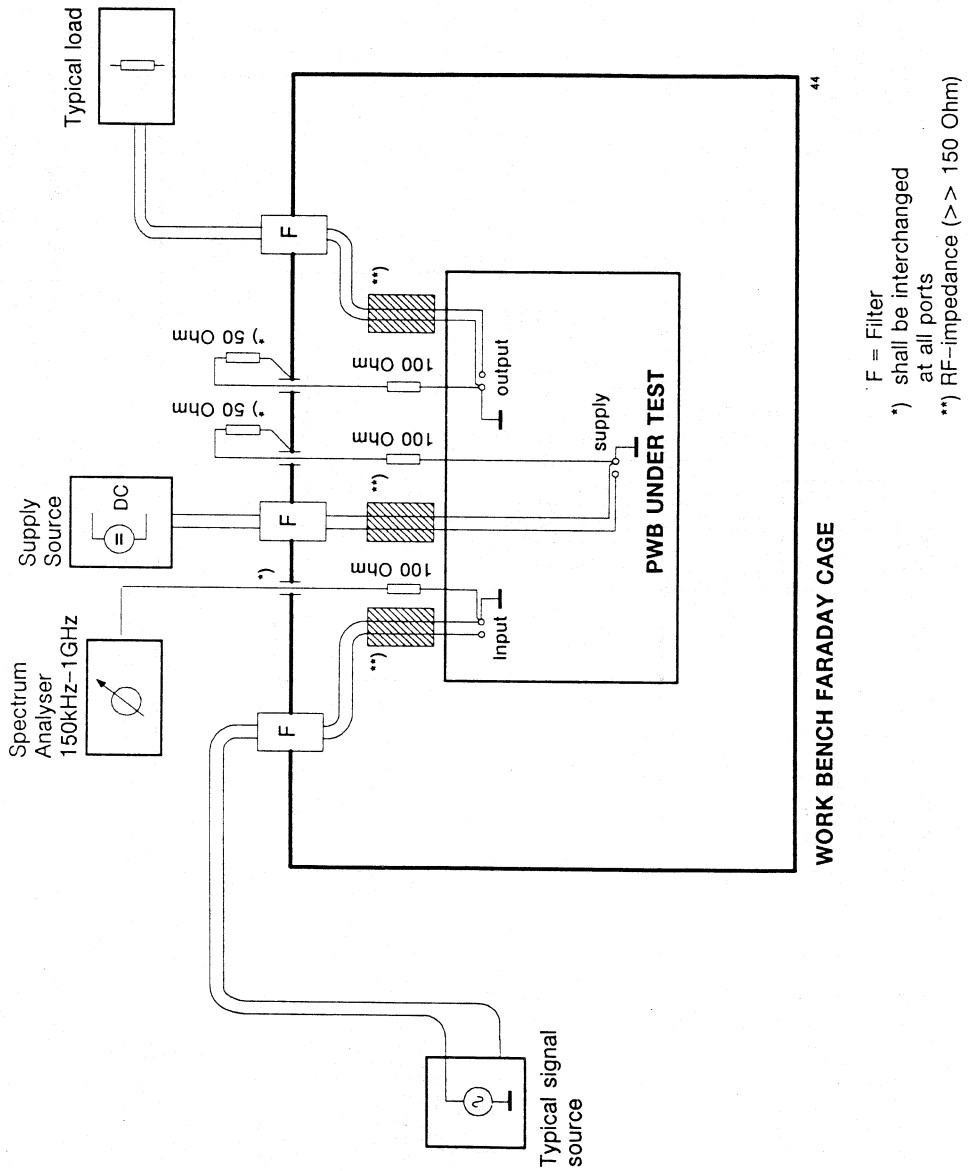


Fig.7. Simplified set-up for emission testing of the PWB using the workbench Faraday cage.

Workbench EMC evaluation method

EIE/AN91001

2.5 Test equipment verification

The verification of the disturbance signal used with immunity testing shall be done as follows:

1. The RF generator is set at 10 MHz **un-modulated**. The output level shall be adjusted such that an rms voltage of e.g. 3 Volts appears at the port which will be connected to the coaxial feed-throughs (with the 100 Ω series resistor behind) of the Faraday cage. This voltage is measured across the output of the 50 Ω impedance matching network using a high input impedance RF-voltmeter.
2. Amplitude modulation must be turned on, using a 1 kHz sinewave (987.5 Hz / 1005 Hz, see chapter 3) and the modulation depth must be set to 80 %. By means of an oscilloscope this AM-signal shall be observed, Neither clipping nor modulation inversion may occur on the signal wave shape.

In this case the peak-to-peak level will be 15,3 Volt.

3 Volt rms = 4,24 Volt peak = 8,48 Volt peak-to-peak un-modulated.

3 Volt 80 % modulated then becomes 15,27 Volt_{p-p}.

3. When this verification is accomplished this signal is provided to the coaxial feed-through connected to the common-mode point under test.

With respect to emission measurements and the verification of the receiver used, further reference is made to IEC CISPR publication 16.

If the emission limit is given in Quasi-Peak or Average limits the emission performance shall be measured in Peak-mode first (because it is fastest to apply).

1. If the PWB satisfies the Average requirements, using the Peak-mode, no further measurements have to be carried out.
2. When the PWB **does not** satisfy the Average requirements but complies with the Quasi-Peak requirements, additional measurements in Average mode need to be carried out.
3. When the PWB **does not** fulfil either of the requirements, both measurements; Quasi-Peak and Average have to be carried out. Normally, there will be insufficient margin and PWB modifications will be necessary.

Note 5: If the RF-signal is a continuous sinusoidal wave, then the indications from either Peak, Quasi-Peak and Average-mode detectors will be equal.

2.6 Requirements

With conducted immunity measurements, the limit applicable to the product, e.g. draft IEC 801-6, can be used directly for verifying the IC application.

When radiated immunity requirements are given, a transformation from electric fieldstrength requirements to induced voltages and currents has to be considered. For indication the following relation can be considered:

Workbench EMC evaluation method

EIE/AN91001

- 1 Volt_{cmf} (150 Ω in series) shall be taken for 1 Volt/metre (travelling wave, far field condition, generated by a tuned dipole antenna, distance ≥ 3 metre) in case the cables and EUT are exposed to the EM-field, i.e. transformation ratio = 0 dB.
- 0,3 Volt_{cmf} (150 Ω in series) shall be taken for 1 Volt/metre (travelling wave, far field condition, generated by parallel plate or strip-line set-ups) in case only the EUT is exposed to the EM-field and cables are excluded, i.e. transformation ratio = -10 dB.

Note 6: The transformation ratio will depend on product size, the way cables are routed and frequency.

When carrying out conducted emission measurements in the frequency range, freq. ≥ 30 Mhz, where the radiation limits are given in μVolt/metre, measured at 10 metre distance, the following approximation can be applied:

$$E_{\text{limit}} = 30 \text{ dB}\mu\text{V/m} (= 30 \mu\text{V/m}) \text{ at } 10 \text{ metre distance from the object.}$$

$$E = (7/d) \cdot \sqrt{P_i}, \quad P_i = U^2/150 \Omega,$$

$$E = (7/10) \cdot \sqrt{(U^2/150)} = U/17,5$$

$$E(\text{dB}\mu\text{V/m}) = U(\text{dB}\mu\text{V}) - 25 \text{ dB.}$$

In some emission standards, e.g. automotive and information technology equipment, conducted emission requirements are given over a large frequency range. Those requirements shall be adhered without any transformation.

In portable applications such as radio or television where an on-top antenna is used, functional requirements will be much more severe than the legal requirements. Those functional limits need to be measured on the product, without any annoying disturbances, and thereafter the relation given in chapter 2.6 can be applied to the found limit.

Workbench EMC evaluation method

EIE/AN91001

3 EXAMPLES.

All applications, e.g. a single op-amp, a transmission/speech circuit for telephone, μ P or video processor are tested on immunity with impedances (symmetrical and a-symmetrical loading) at inputs and outputs applied according to the application instructions (which shall be stated in the application report !!).

When these (input, output) impedances are passive, they can be either outside or inside the workbench Faraday cage. When these are active, they should be outside. When coaxial feed-throughs are used, care must be taken by additional measures (low-pass-filters, see Fig.5) that the RF-energy which will be superimposed on either input or output lines does not adversely affect the performance of the auxiliary equipment **and** that by measures the source or load impedance remains properly defined (even at RF).

The signal wires going from the application board to the feed-through connectors or filters shall be provided with an RF-blocking impedance, represented by 14 - 17 turns on a 4C65 (freq \geq 1 MHz) or better 4A15 ferrite toroid.

By means of three 100 Ω resistors (PR 37 or PR 52, or equivalent power metal film resistors), the common of the input, output and supply is then coupled to the three coaxial feed-through connectors as indicated in paragraph 2.3. Externally, these connections are either coupled to the RF-disturbance generator, the selective voltmeter or terminated by a 50 Ω coaxial resistor.

Note 7: The lay-out of the PWB shall be made such that either the typical performance of the circuit is tested (non-optimized mono- or bi-layer) or on e.g. multi-layer with every precaution taken to have optimal EMC performance of the IC in this application.

Note 8: If the latter application cannot satisfy the EMC requirements, then no designer/customer will be capable of making a proper product with it at reasonable costs.

3.1 Audio applications.

3.1.1 Immunity

The audio circuit shall be set at nominal (gain) conditions. When necessary or obtainable, a 1 kHz generator can be used to make the required setting. The baseband 1 kHz output signal across the normal load shall be measured. This level will be taken as a reference for the immunity performance testing.

The disturbance signal (RF, 80 % modulated by 1 kHz) shall be applied to one of the common-ports of the PWB under test in turn, while the other ports are terminated to reference by 50 Ω . A demodulated signal level (1 kHz only) of 40 dB less than the nominal signal level across that load is acceptable for proper operation. This level can be measured either by a low frequency spectrum analyzer or a sensitive AC-Voltmeter with a 1 kHz band-pass in front (as described in EN 55020). A typical pass-band bandwidth of 500 Hz is sufficient for these kind of measurements.

Workbench EMC evaluation method

EIE/AN91001

Example: The nominal level on the a,b-lines of a telephone system is 100 mV (across 600 Ω). The signal level across the earpiece can be measured and may be 30 mV (example only !!, determined by dynamic sensitivity of the earpiece). The level of the demodulated signal across that same earpiece, with the disturbance signal applied to the PWB shall be less than 0.3 mV.

3.1.2 Emission

In most cases the emission from linear applications is nil unless it contains some oscillator. In the latter case, the disturbance generator shall be replaced by a spectrum analyzer covering the frequency range of interest, commonly 9 kHz to 1 GHz. As the emission will be continuous (at the fundamental and its harmonics), the detector chosen in the spectrum analyzer will not influence the readings. Where possible, the bandwidth requirements as stated in IEC CISPR publication 16 shall be adhered.

3.2 Video applications

3.2.1 Immunity

With video applications it will be such that most applications are based on an interlaced video frame sequence of 25 or 30 Hz. In both situations the spectrum around 1 kHz is fully crowded by harmonics of the frame frequency. It is therefore, that the modulation frequency, nominal 1 kHz should be shifted a little up/down such that it becomes an inter-harmonic of the frame frequency.

Examples: 25 Hz \rightarrow 987,5 Hz
 30 Hz \rightarrow 1005 Hz

To obtain nominal settings of the video application a colour bar signal is applied to the circuit to be fully operational.

The demodulated signal at the outputs, e.g. CVBS, RGB or sync. can be measured by using a spectrum analyzer with a resolution bandwidth of ≤ 6 Hz. This selectivity is required to obtain a signal-to-noise ratio sufficient to discriminate the demodulated signal from the wanted signal. Here the demodulated signal to nominal signal level_(peak-to-peak) ratio has to be about -55 to -60 dB to be just not perceptible on the screen. This limit level can be found by superimposing a 1 kHz signal to the normal signal while observing the picture. The 1 kHz generator level is increased up to a level where it becomes just perceptible on the screen. This level referred to the functional signal is then taken as limit for the signal-to-interference ratio (S/I).

Note 9: Special care shall be taken to assure that the demodulation of the spectrum analyzer is much less than the limit level to be measured. Additional low-pass filters, see Fig.5. may be required !!
 Furthermore, it may be necessary to use a comb-filter to lower the level of the frame harmonics.

Example: The video signal level of the G-signal from the processor board to the video output stage is 3600 mV_{p-p}. Then the demodulated signal shall be less than 3,6 mV_(peak).

Note 10: Disturbing signals may also effect synchronization and as such appear as vertical zig-zag lines on the screen. This effect can only be measured by using a jitter or phase modulation meter between sync-out of the pattern generator and the output signal coming from the PWB.

Workbench EMC evaluation method

EIE/AN91001

3.2.2 Emission

As indicated above emission from the PWB is measured by replacing the disturbance generator by a spectrum analyzer or selective voltmeter. In analog applications, emission will be caused by the video signal itself and some oscillator signals used for demodulation, mixing, etc. According to the emission standard EN 55013, the test page pattern of teletext shall be used as video information. Where possible, the bandwidth and detector requirements as stated in IEC CISPR publication 16 shall be adhered.

3.3 Digital applications

3.3.1 Immunity

With digital applications, the main problem will be the fact that it needs wide address and data buses for operation. As these wide buses are inconvenient for this method, these shall be limited by simplifying the circuit or by introducing parallel-to-series and series-to-parallel decoders. Preferable, these decoders must be used on the PWB under test such that only one serial line will be fed through the wall of the cage. To isolate the decoders from the IC under test, series resistors, e.g. 1 k Ω , shall be used in-between the decoders and the IC. The common-mode points on the PWB needs to be chosen close to the IC to be tested instead of the serial decoder input and output ports indicated in chapter 2.4.

By means of a tap, the signals at the output of the IC shall be monitored by means of an oscilloscope or a logic analyzer with an analog input and adjustable threshold levels. As criteria, the output signal of the IC may not exceed the specified high/low levels other than during functional transitions.

When testing the inputs, the worst case DC-levels shall be superimposed to the input signal (if it does not already contain a DC-component from a previous stage). When the disturbance signal is applied to the common-mode points chosen, the signal levels at the outputs shall be observed.

With more complex circuits it can be such that analog and digital inputs and outputs are available. By closing the loop; data \rightarrow dig.out \rightarrow dig.in \rightarrow analog out \rightarrow analog in \rightarrow compare with initial data and turn on/off a flag, which drives a LED, a powerful test program is carried out.

3.3.2 Emission

For proper operation, the normal program or coding shall be applied to the IC such that the emission measured is representative for a typical application. In case of a μ P, the outputs can be driven such that a digital ramp function (bit 0 = frequency F0, bit 1 = F0/2, etc) is generated over its 4, 8 or 16 bits wide data bus. All buses will have typical length e.g. 0,1 metre. The end of the bus shall be terminated as indicated in the application report e.g. by 50 pF//3.3 k Ω . The common-mode points for the IC under test shall be taken (similar as with immunity) to measure the emission performance of the IC in its application.

Workbench EMC evaluation method

EIE/AN91001

4 THE WORKBENCH CAGE PARAMETERS

The size of the workbench Faraday cage is chosen in such a way that it can contain most typical application and evaluation boards:

Length:	500 mm
Width :	350 mm
Height:	150 mm

The Workbench Faraday cage is made from carbon-free iron 1,5 mm thick. A conductive gasket is used between the box and the cover to make proper contact. The inside of the box is covered with an anti-static insulating material.

The connections through the wall can be made by:

Coax	:	5 x BNC,
Single line	:	4 x π -filter,(2 x 1,35 nF + 8 μ H, 2 Amp., 50 Volt max.),
		6 x Feed-through capacitors, (62 nF, 16 Amp., 500 Volt max.).

The shielding effectiveness in H_{xyz} directions is better than 60 dB in the frequency range 1 to 1000 MHz, measured with two 60 mm electrically shielded loops according to Mil.Std. 220. An example of the shielding effectiveness of the workbench Faraday cage is given in Fig.7.

The characteristics of the π -filters (Low-Pass-Filter at 1 MHz) and the feed-through capacitors are given in Fig.8. The choice for these filters is made such that by additional external measures the filtering performance can be enhanced at low frequencies. For most applications the given performance will be sufficient. These π -filters can also be used at the outputs of digital circuits when using a 100 Ω resistor in series. In this case, the pass-band is limited to about 500 kHz, which is still sufficient to allow e.g. I²C-communication.

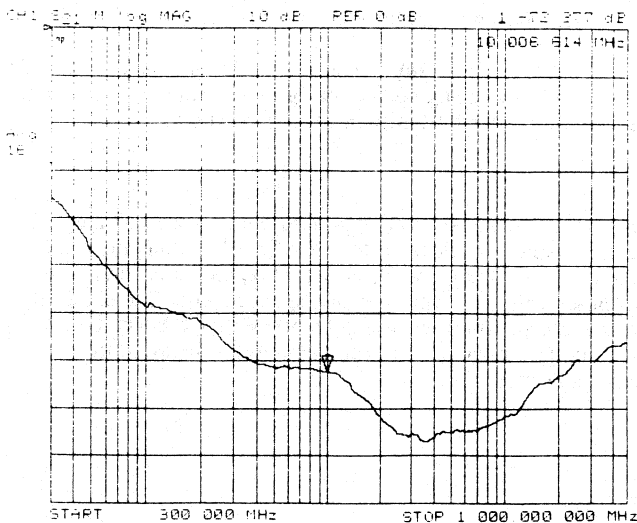


Fig.7. Shielding effectiveness of the workbench Faraday cage.

Workbench EMC evaluation method

EIE/AN91001

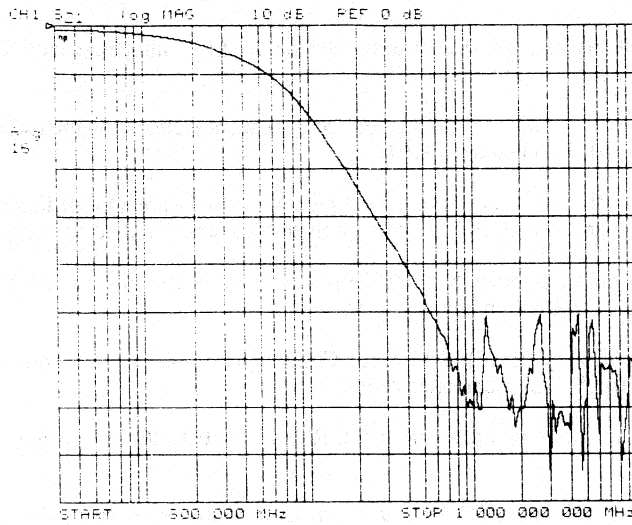


Fig.8a. Performance of the π -feed-through filter used with the workbench Faraday cage.

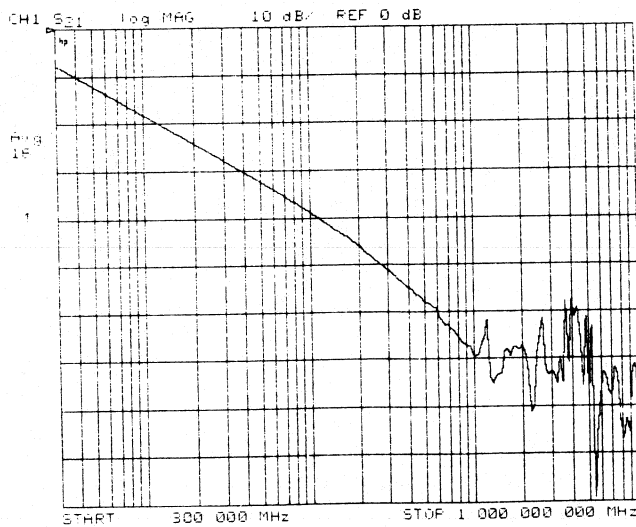


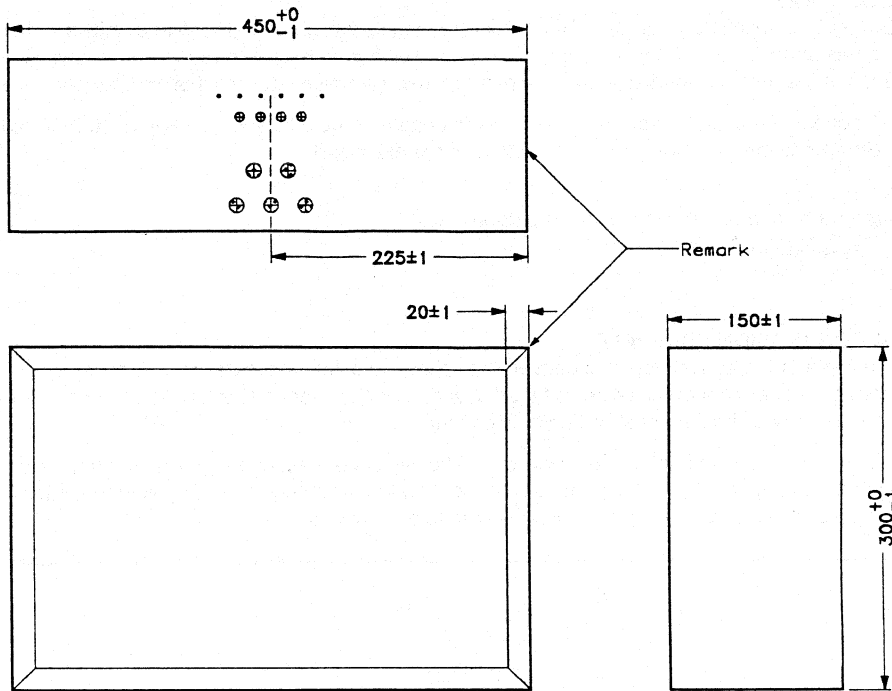
Fig.8b. Filter performance of the feed-through capacitor used with the workbench Faraday cage.

5 REFERENCES

- [1] Draft IEC 801-6, Immunity to conducted disturbances, induced by radio frequency fields above 9 kHz, IEC 65A(secr) , nov. 1991.
- [2] 65A/77B(secretariat)121/88, Immunity to radiated radio-frequency electromagnetic fields, Draft publication IEC 901-3 2nd edition, 1991.
- [3] IEC CISPR publication 20 2nd edition, Limits and methods of measurement of immunity characteristics of sound and television broadcast receivers and associated equipment, 1990.
- [4] An evaluation method to characterize the EMC performance of PCBs containing ICs, M.J.Coenen, ESG 8801, Philips Components, 1988.
- [5] ElectroMagnetic Compatibility (EMC) and Printed Circuit Board (PCB) constraints, M.J.Coenen, ESG 89001, Philips Components, 1989.
- [6] Radiated emissions from common-mode currents, C.R.Paul, IEEE EMC Symposium on EMC, Zürich, 1987.
- [7] Antenna theory, analysis and design, C.A.Balanis, Harper and Row Publishers, New York, 1982.
- [8] Electromagnetic theory, J.A.Stratton, McGraw Hill, New York and London, 1941.
- [9] Taschenbuch der Hochfrequenztechnik, H.Meinke, F.W.Gundlach, Springer Verlag, Berlin und New York, 1968.

Workbench EMC evaluation method

EIE/AN91001



Remark: Welded joints and finished

UN-0 25		TOLERANCES UNLESS OTHERWISE STATED TOLERANTIES TENzijN ANDERS VERMELD		UN-0 603			
P ₀ in mm		DIMENSION MAAT	ANGLE HOEK	LITEN STUK		SAFETY VEILIGHEID	PART DEEL
GENERAL REQUIREMENTS ALGEMENE VEREISTEN	UNIT EENHEID	MATERIAL		PATTERN NO. MODELNR.			
✓	mm	CHROMIUM NICKEL STEEL PLATE 1 mm					
SCALE SCHALE	PROJ. EUROPE.	FINISH AFWERKING		ORDER NO. ORDERNR.		PART DEEL	
1:1		SHEAR-DRILL-FOLD-WELD-FINISH					
CLASS NO.	EMC MEASURING TOOL			1		90-08-27	
	Part: BOX						
NAME NAAM	A. Coolen	VERF. VERV.		1-001			A3
DATE DATUM	90-08-20	Eindhoven van N.Y. PHILIPS GLOEI-LAMPEN-FABRIEKEN EINDHOVEN - NEDERLAND					

Annex 1. Drawing of the workbench Faraday cage.

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

Author: Th. v. Daele, Product Concept & Application Laboratory, Eindhoven, The Netherlands

1. INTRODUCTION

With an 83CL410 microcontroller and a PCF1252-x reset circuit, it is possible to make a software driven A-to-D converter. In this application note, an example is described where an 83CL410 measures its own supply voltage. The resolution of the measurement is 0.1V. The program example also refers to this application.

Chapter 2 describes the algorithm of the conversion. In chapter 3 the example with 83CL410/PCF1252 is described. Both hardware and software of this example are explained.

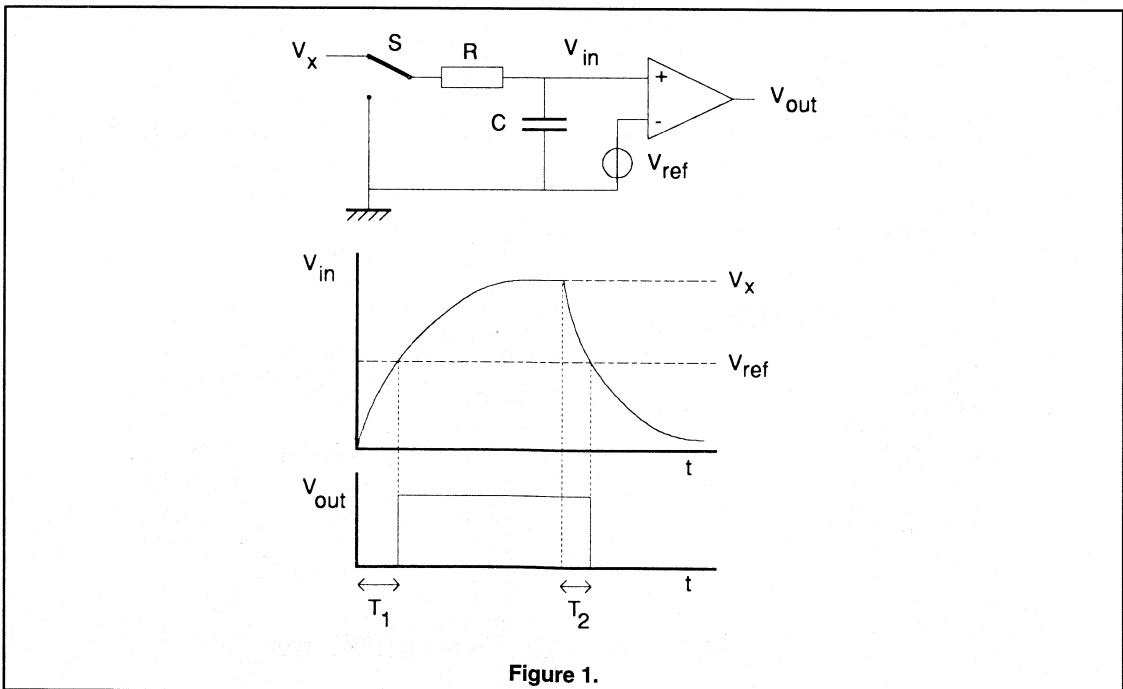
References:

- 80C51-based 8-bit microcontrollers; Data Handbook IC20
- PCF1252-x data sheet

2. A-TO-D CONVERSION PRINCIPLE

The basic principle of the conversion is to convert the voltage to a time measurement. A microcontroller without an on-chip A-to-D converter cannot measure voltages directly, but if converted to a time measurement, this can be done by software or with the help of an on-chip timer/count.

Figure 1 shows the basic circuit to do the conversion. The circuit consists of an integrator circuit, a voltage reference and an analog input switch S. The analog input switch is controlled by the microcontroller. The integrator is built around a comparator whose output is connected to a microcontroller input.



Before the measurement is started, the analog switch connects the integrator input to ground, so that the integration capacitor is fully discharged. The measurement is started by connecting the integrator input to the unknown voltage. The integration capacitor will charge up. When the non-inverting input exceeds the reference voltage, the comparator output will switch from LOW to HIGH.

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

The time between starting the measurement and the moment that the comparator output becomes HIGH, is measured by the microcontroller and is:

$$T_1 = -RC \cdot \ln \frac{V_x - V_{ref}}{V_x}$$

The charging continues until the capacitor is fully charged. Then the integrator input is grounded via the analog switch. The integration capacitor discharges while the comparator output is HIGH. When the input voltage becomes lower than the reference voltage, the comparator output becomes LOW again. The time between the start of the discharging and the moment that the comparator output becomes LOW, is measured by the microcontroller and is:

$$T_2 = -RC \cdot \ln \frac{V_{ref}}{V_x}$$

When the microcontroller uses the ratio between T_1 and T_2 , the result becomes independent of the values R and C. The resulting ratio can then be used as a pointer to a look-up table that may contain an indication for the measured parameter or display data.

3. EXAMPLE OF CONVERSION PRINCIPLE

3.1 Hardware

In the example, an application is used where a microcontroller measures its own supply voltage. The supply voltage is generated by solar cells, so power consumption should be minimized. The measured voltage is shown on an LCD display.

An 83CL410 is used as microcontroller. This controller is an 80C51 family member. Compared with a standard 80C51, it has the following extra features:

- Wide supply voltage range of 1.8V to 6V.
- Wide operating frequency range of 32kHz to 20MHz with internal oscillator. With external oscillator there are no limitations on the lower frequency limit.
- Byte I²C interface instead of UART.
- 8 extra external interrupt inputs on P1. The interrupt level is programmable. These interrupts can terminate the power-down mode.
- 3 mask programmable I/O port configurations.
These configurations are:
 - Standard quasi-bidirectional I/O
 - Open-drain output, standard input
 - Push-pull output, no input
- I/O port levels after RESET are mask programmable.

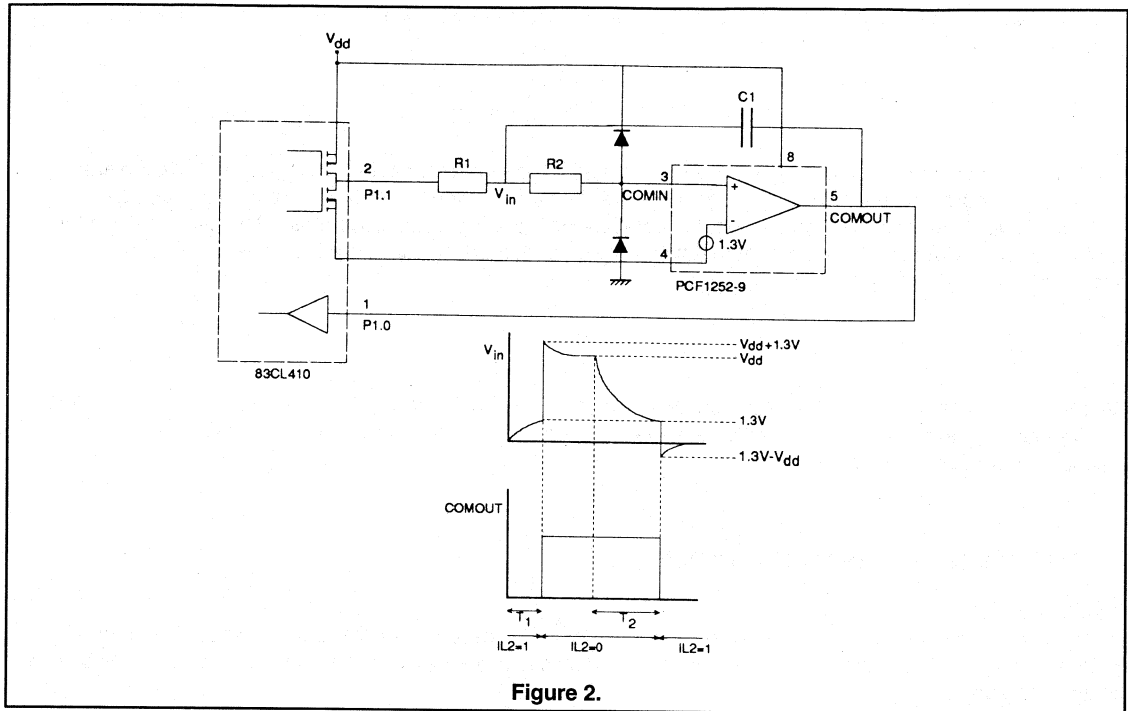
Of these features, only the byte I²C interface is not used.

The PCF1252 is used for monitoring the power supply voltage and generating a reset pulse when the supply drops too much. Several versions of PCF1252 are available, every one with its own trip voltage. The PCF1252 also has an unused comparator. This comparator is used for the integrator circuit. The inverting input is internally connected to a 1.3V reference.

Figure 2 shows the A-to-D conversion part of the circuit.

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006



R1 and C1 determine the time-constant of the circuit. The input of the integrator is connected to P1.1 of the microcontroller. This port line has a push-pull configuration. It is used as the analog switch shown in Figure 1. When this output line is made HIGH, the integrator is connected to the supply voltage. This is the voltage to be measured. The voltages on the non-inverting input and the output are shown in Figure 2. The output of the comparator is LOW, and the capacitor will be charged. When the voltage at the non-inverting input becomes higher than 1.3V, the comparator output will become HIGH. This HIGH level will cause an interrupt on P1.0 (IL2=1). The voltage on the inverting input will want to rise to $V_{CC} + 1.3V$. This voltage is limited by 2 external diodes. R2 is a current limit resistor.

Next step is that P1.1 becomes LOW, so that the capacitor will be discharged. The P1.0 input is now programmed to generate an interrupt on a LOW level (IL2=0). This will happen when the voltage on the inverting input has dropped below 1.3V.

At this point, both charge and discharge times are known, and the supply voltage can be calculated. In this circuit, the PCF1252-9 is used, which will generate a reset pulse when the supply voltage is lower than 2.55V. This limits the lower end of the supply voltage range. However, the 83CL410 is able to go as low as 1.8V.

To minimize power consumption, the lowest frequency is used where the internal oscillator can still be sued. this is 32.768kHz. This parameter, together with the resolution that must be met, determine the minimum value of R1.C1. In this example, a resolution of 0.1V is achieved. The software is made to handle voltage measurements from 1.8V to 5V.

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

To determine the minimum value of $R1C1$, the measurement of 5.0V and 4.95V is important because here the distance between measurements of T_1 and T_2 is minimal.

$$\begin{aligned} V_x = 5.0V: \quad T_1 &= 0.301RC \\ T_2 &= 1.347RC \\ T_2/T_1 &= 4.437 \end{aligned}$$

$$\begin{aligned} V_x = 4.95V: \quad T_1 &= 0.304RC \\ T_2 &= 1.337RC \\ T_2/T_1 &= 4.398 \end{aligned}$$

$$\begin{aligned} V_x = 4.9V: \quad T_1 &= 0.308RC \\ T_2 &= 1.327RC \\ T_2/T_1 &= 4.308 \end{aligned}$$

Because of latency in the interrupt routine, a deviation Δt may occur in T_1 and T_2 . The following conditions must be met:

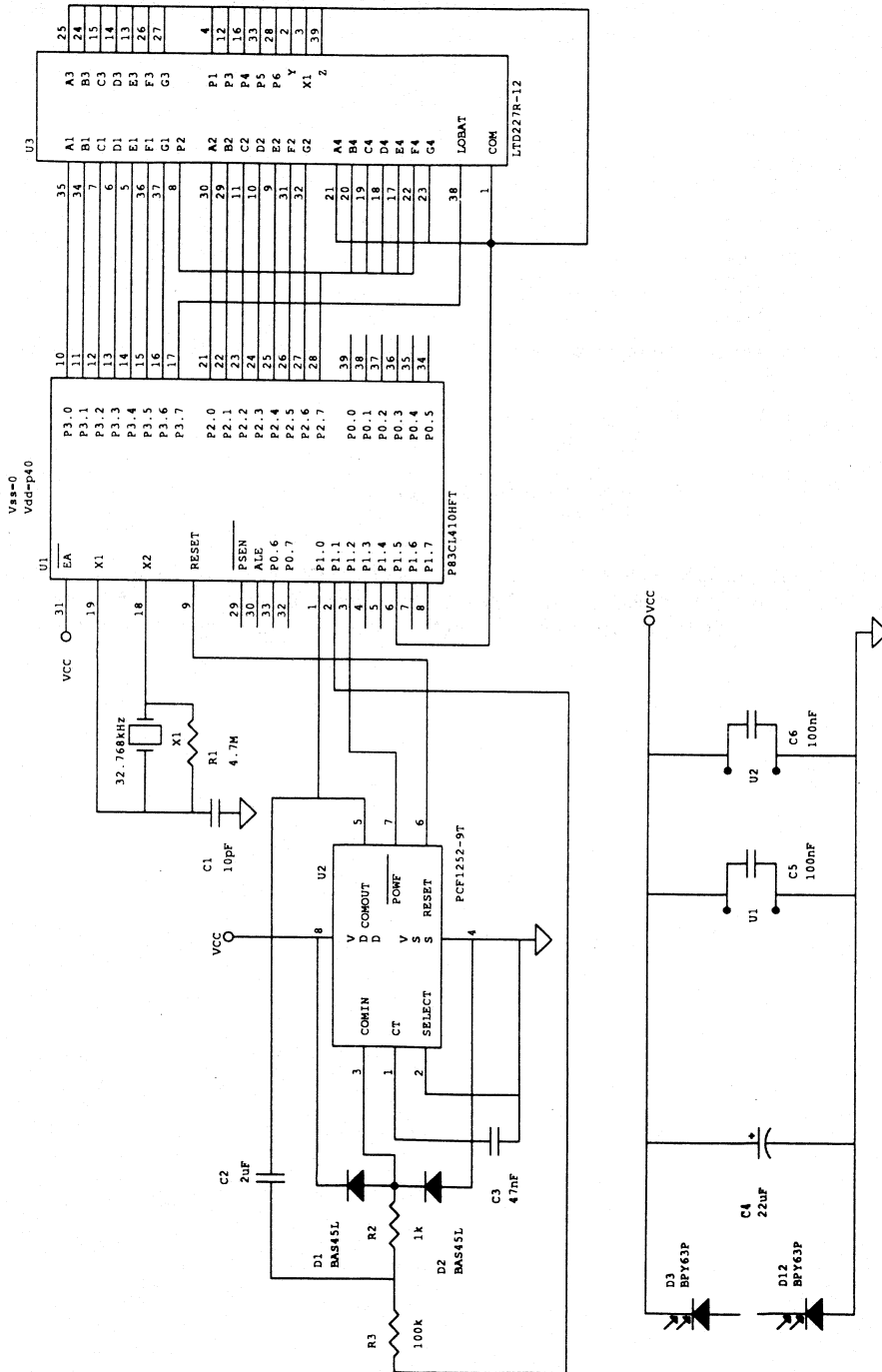
$$V_x = 5.0V: \quad \frac{T_2}{T_1} \frac{1.347RC \pm \Delta t}{0.301RC \pm \Delta t} \geq 4.398$$

$$V_x = 4.9V: \quad \frac{T_2}{T_1} \frac{1.327RC \pm \Delta t}{0.308RC \pm \Delta t} \geq 4.398$$

In this example, there is an uncertainty of 2 machine cycles when measuring T_1 and T_2 . Given an XTAL frequency of 32.768kHz, the minimum value of RC is 0.172. The complete circuit diagram is shown of the example with the LCD connections. Since the LCD connections are only outputs, the push-pull configuration is used for this.

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006



A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

3.2 Software

In chapter 4 the listing of the program is shown.

After RESET the initialization takes place from L.45 to L.70.

The following timers 83CL410 are used:

- Timer_0: Used to measure the T1 and T2 (mode_1).
- Timer_1: Used to generate a timer interrupt for LCD polarity reverse (mode_2).
This is the auto-reload mode. Every 10ms an interrupt is generated.

3 interrupts are used:

- INT2: External interrupt for the A-to-D conversion. Priority level_1.
- INT4: Interrupt from PCD1252-9 when supply voltage is smaller than 2.55V. Priority level_1.
- Timer_1: Interrupt for LCD polarity reverse. Priority level_0.

The measurement is started at L74. The interrupt level of INT2 is set to '1' and time_0 is started to measure T1. P1.1 is set, so that the supply-voltage is connected to the integrator input. While the integration capacitor is charging, the 83CL410 is in IDLE-mode to reduce power consumption. This part of the measurement is identified by ADC_Status=1. When INT2 is generated, the interrupt routine is entered (from L.181). Timer_0 is read and its value is stored in R4 (MSB) and R5 (LSB). The interrupt level of INT2 is cleared to '0' for the second part of the measurement. After clearing the interrupt flag, the main program is entered again at L.85. a delay is entered at this point so that the integration capacitor can be discharged sufficiently from $V_x + 1.3V$ to V_x . The delay is 32 time-outs of timer_1. During this delay, the controller is in IDLE mode. After the delay, P1.0 is cleared to discharge the integration capacitor. Timer_0 is started to measure T2. Again the microcontroller enters the IDLE mode until INT2 is generated. In the interrupt routine, the interrupt level is now set to '1' again for the next A-to-D conversion. In the main program, T2 is copied to R6 (MSB) and R7 (LSB) (L.96 and L97).

Now that T1 and T2 are known, the measured voltage can be determined. This is done by calculating the ratio between T1 and T2, and converting this to a pointer which points to a table with the segment data of the LCD display. In the calculation, the ratio between T1 and T2 should be equal or greater than 1. If $T2 < T1$ (L.107) then the flag 'gr_2V6' will be cleared (L1.65) and [R4.R5] and [R6.R7] are exchanged. This point is met when the input voltage is 2.55V.

Normally, voltages < 2.55V will not be measured, because by then the PCF1252-9 has generated an interrupt on INT4. This means a power-failure and the only way to start the measurement again, is that the PCF1252-9 resets the microcontroller. The INT4 routine is from L.227 .. L.237). The program, however, contains the conversion table for input voltages as low as 1.8V.

The pointer to the table (stored in [R6.R7]) is calculated by: $[R6.R7] = 16 * [R6.R7] / [R4.R5]$. The division is done by the subroutine '_sdivi'. This is a library function contained in the C-library of the BSO/Tasking C-compiler package (type number OM41326). This library must be linked to this program.

Of the result, only R7 is relevant. The segment data contains 2 bytes, so pointer R7 is multiplied by 2 (L.134). There are 2 segment tables: one for voltages > 2.55V (L.247 .. L.274) and one for voltages < 2.55V (L.284 .. L.304). DPTR is used as pointer for this table and is calculated by adding the table base (L.129 .. L.132) and R7 (L.134 .. L.137). The following table shows the relation between the measured voltage, $16 * T1 / T2$ or $16 * T2 / T1$, the pointer and the segment data.

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

Input voltage > 2.55V				
Input Voltage	16*T2/T1	Table addresses	Segment data	Display
4.95	70.4	140 .. 147	0x6d,0xbf	5.0V
4.85	67.5	136 .. 139	0x66,0xef	4.9V
4.75	64.83	130 .. 135	0x66,0xff	4.8V
4.65	62.18	124 .. 129	0x66,0x87	4.7V
4.55	59.57	120 .. 123	0x66,0xfd	4.6V
4.45	56.98	114 .. 119	0x66,0xed	4.5V
4.35	54.43	108 .. 113	0x66,0xeb	4.4V
4.25	51.91	104 .. 107	0x66,0xcf	4.3V
4.15	49.42	98 .. 103	0x66,0xdb	4.2V
4.05	46.96	94 .. 97	0x66,0x86	4.1V
3.95	44.54	90 .. 93	0x66,0xbf	4.0V
3.85	42.14	84 .. 89	0x4f,0xef	3.9V
3.75	39.82	80 .. 83	0x4f,0xff	3.8V
3.65	37.51	76 .. 79	0x4f,0x87	3.7V
3.55	35.24	70 .. 75	0x4f,0xfd	3.6V
3.45	33.02	66 .. 69	0x4f,0xed	3.5V
3.35	30.83	62 .. 65	0x4f,0xeb	3.4V
3.25	28.7	58 .. 61	0x4f,0xcf	3.3V
3.15	26.6	54 .. 57	0x4f,0xdb	3.2V
3.05	24.56	48 .. 53	0x4f,0x86	3.1V
2.95	22.56	46 .. 47	0x4f,0xbf	3.0V
2.85	20.62	42 .. 45	0xdb,0xef	2.9V LOBAT
2.75	18.73	38 .. 41	0xdb,0xff	2.8V LOBAT
2.65	16.89	34 .. 37	0xdb,0x87	2.7V LOBAT
2.55	15.12	32 .. 34	0xdb,0xfd	2.6V LOBAT

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

Input voltage < 2.55V				
Input Voltage	16*T1/T2	Table addresses	Segment data	Display
2.55	16.92	32 .. 37	0xdb,0xed	2.5V LOBAT
2.45	19.09	38 .. 43	0xdb,0xeb	2.4V LOBAT
2.35	21.77	44 .. 49	0xdb,0xcf	2.3V LOBAT
2.25	25.15	50 .. 57	0xdb,0xdb	2.2V LOBAT
2.15	29.51	58 .. 69	0xdb,0x86	2.1V LOBAT
2.05	25.32	70 .. 85	0xdb,0xbf	2.0V LOBAT
1.95	43.35	86 .. 109	0x86,0xef	1.9V LOBAT
1.85	54.96	110 .. 119	0x86,0xff	1.8V LOBAT

Now the data can be displayed on the LCD display. When writing data to the LCD, the timer_1 interrupt is disabled. The value of output LCD_COM (controlled by timer_1 interrupt routine) determines whether the data must be written inverted to the LCD or not. The data is written non-inverted from L.147 .. L.150, inverted from L.153 .. L.158. When the data is written, timer_1 interrupt is enabled again, and a new A-to-D conversion is started at label Start_ADC.

From L.210 .. L.219 the timer_1 interrupt routine is shown. In this routine the segment lines to the LCD display are inverted (on P2 and P3). Also LCD_COM is inverted, which is the COMMON pin of the display.

On L.218 the value Dis_tim is incremented. This is for the delay between the measurement of T1 and T2 (see L.85).

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

4: SOFTWARE LISTING

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 1

```

LOC   OBJ          LINE  SOURCE
                                     1  $TITLE(Main program)
                                     2  $DEBUG
                                     3  $
                                     4  # 1 "C:\Tools\Tasking\ASM51\Include\8051\Reg410.Inc"
                                     5
;*****
                                     6  ;Timer_0 is used for A/D conversion in combination with PCF1252
                                     7  ;Timer_1 is used as time base for LCD switching
                                     8  ;INT2 is used as input for A/D conversion
                                     9  ;INT4 is used as power-failure input from PCF1252
                                    10  ;All timing related to Xtal=32.768kHz
                                    11
;*****
                                    12
0091                                     13          Anal_in  EQU  91h    ;P1.1 is voltage switch
0095                                     14          LCD_COM EQU  95h    ;P1.5 is COM of LCD display
00B0                                     15          Digit_1 EQU  0B0h  ;P3 is first digit of LCD
                                     ;display
00A0                                     16          Digit_2 EQU  0A0h  ;P2 is second digit of LCD
                                     ;display
                                    17
                                    18
;*****
                                    19
                                    20
                                    21          Flags  segment Bit
                                    22          Stack  segment Data
                                    23          Main   segment Code Inblock
                                    24          Table_1 segment Code Page
                                    25          Table_2 segment Code Page
                                    26          ADC    segment Code
                                    27          LCD    segment Code
                                    28          Power_F segment Code
                                    29
-----
                                    30          RSEG Flags
0000:      R   31  ADC_Status: DBIT 1      ;Status flag AD converter
0001:      32  Gr_2v6:      DBIT 1      ;Flag indicating that V<2.55V
                                    33
-----
                                    34          RSEG Stack
0000:      R   35  stk_st:   DS 20      ;Define stack
                                    36
REG END      37          Dis_tim set r2 ;Timer for cap. discharge from Vdd+1.3
                                     ;to Vdd
                                    38
                                    39

```

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 2

```

LOC   OBJ          LINE  SOURCE
;*****
-----
40
----
41          CSEG AT 00      ;Reset vector
0000: 020000  R  42          ljmp Start
43
----
44          RSEG Main
0000: C291    R  45 Start:   clr Anal_in      ;Ground analog input
0002: 7581FF  R  46          mov sp,#Stk_St-1 ;Initialise stack pointer
0005: C200    R  47          clr ADC_Status  ;Initialise ADC status
48
0007: 75F805  49          mov ip1,#05h     ;INT2 is priority level 1 (Analog
;measurement.)
50          ;INT4 is priority level 1 (Power fail)
51          ;Timer_1 interrupt level 0 (LCD switch)
000A: 53E9FB  52          anl ix1,#0fbh   ;INT4 on low level
000D: D2EA    53          setb ex4        ;Enable INT4
000F: D2AF    54          setb ea         ;Global enable interrupts
55
0011: 758921  56          mov tmod,#21h   ;Timer_1 mode (LCD switch)
57          ; 8 bit auto reload mode
58          ;Timer_0 mode (A/D conversion)
59          ; 16 bit timer
0014: 758DF5  60          mov th1,#0f5h  ;Delay for discharging cap's (+/- 1sec)
0017: 758B50  61          mov t11,#050h
001A: D28E    62          setb tr1
001C: 308FFD  63          jnb tf1,$
001F: C28F    64          clr tf1
0021: C28E    65          clr tr1
66
0023: 758BE1  67          mov t11,#0e1h  ;LCD switch rate: 10ms
0026: 758DE1  68          mov th1,#0e1h
0029: D28E    69          setb tr1       ;Start timer_1
002B: D2AB    70          setb et1       ;Enable timer_1 interrupt
71
72          ;Measurement of supply voltage
002D: 43E901  73          orl ix1,#01H   ;INT2 on '1' input level
0030:          74 Start_ADC:
0030: 758CFF  75          mov th0,#0FFh  ;Load timer_0
0033: 758AFB  76          mov t10,#0FBh
0036: D28C    77          setb tr0       ;Start timer_0
0038: D291    78          setb Anal_in   ;Start raising part of measurement
003A: D2E8    79          setb ex2       ;Enable INT2
003C: 438701  80 Id_1:      orl pcon,#01   ;Go to idle mode
003F: 3000FA  R  81          jnb ADC_Status,Id_1;Wait till first part of measurement
;is handled
82          ;Exit from idle mode could be from LCD
;interrupt

```

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 3

LOC	OBJ	LINE	SOURCE
0042:	7A00	83	mov Dis_tim,#00 ;Discharge delay: Vdd+1.3...Vdd
0044:	438701	84	Id_2: orl pcon,#01 ;Go to idle mode
0047:	BA20FA	85	cjne Dis_tim,#20h,Id_2 ;Delay: 32 LCD_time_outs
		86	;If not equal: idle
		87	
004A:	758CFE	88	mov th0,#0FFh ;Load timer_0
004D:	758AFB	89	mov tl0,#0FBh
0050:	D28C	90	setb tr0 ;Start timer_0
0052:	C291	91	clr Anal_in ;Start second part of measurement is
			;handled
		92	
0054:	438701	93	Id_3: orl pcon,#01 ;Go to idle mode
0057:	2000FA	R 94	jb ADC_Status,Id_3 ;Wait till measurement is finished
005A:	C2E8	95	clr ex2 ;Disable INT2
005C:	AE8C	96	mov r6,th0 ;Get timer data of second part of
			;measurement
005E:	AF8A	97	mov r7,tl0 ;T2
		98	;Process data
		99	
		100	;If T1>T2 then [R6.R7] and [R4.R5] must
		101	;be exchanged. Also other segment table
		102	;must be used.
0060:	D201	R 103	setb gr_2v6 ;Set gr_2v6 flag
0062:	C3	104	clr c ;Test T2-T1
0063:	EE	105	mov a,r6
0064:	9C	106	subb a,r4
0065:	404C	107	jc Small_2v6 ;<2.6V: clear flag, exchange registers
0067:	7004	108	jnz Calculate
0069:	EF	109	mov a,r7
006A:	9D	110	subb a,r5
006B:	4046	111	jc Small_2v6
		112	;Calculate T2/T1
		113	;[R6.R7]=[R6.R7]/[R4.R5]
		114	;1<Result<4.4
		115	;Before dividing: [R6.R7]=16*[R6.R7]
006D:	7806	116	Calculate: mov r0,#06 ;r0 points to r6
006F:	EF	117	mov a,r7
0070:	C4	118	swap a
0071:	54F0	119	anl a,#0f0h
0073:	CF	120	xch a,r7 ;r7 shifted 4 bits left
0074:	C4	121	swap a
0075:	CE	122	xch a,r6
0076:	C4	123	swap a ;nibbles r6 swapped
0077:	D6	124	xchd a,@r0
0078:	FE	125	mov r6,a ;r6 also shifted 4 bits to the left
		126	

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 4

LOC	OBJ	LINE	SOURCE
0079:	120000	R 127	lcall __sdivi ; [R6.R7]=16*T2/T1
		128	
007C:	300105	R 129	jnb Gr_2v6,Sec_tab
007F:	900000	R 130	mov dptr,#Seg_tab_1
0082:	8003	131	sjmp Calc_point
0084:	900000	R 132	Sec_tab: mov dptr,#Seg_tab_2
0087:	C3	133	Calc_point: clr c
0088:	EF	134	mov a,r7 ;Calculate address of segment info
0089:	33	135	rlc a ;2*R7
008A:	F582	136	mov dpl,a
008C:	C3	137	clr c
008D:	9494	138	subb a,#148 ;If a>148 then V>5.0: Display H.I V
008F:	4003	139	jc Display
0091:	758294	140	mov dpl,#148 ;V>5.0V
		141	
0094:		142	Display: ;Display result. DPTR points to segment data
0094:	C2AB	143	clr etl ;Disable LCD switch interrupt
0096:	E4	144	clr a
0097:	93	145	movc a,@a+dptr ;Get segment_1 data (digit + LOBAT)
0098:	209509	146	jnb LCD_COM,Com_1 ;If LCD_COM=1, then invert result
009B:	F5B0	147	mov Digit_1,a
009D:	7401	148	mov a,#01
009F:	93	149	movc a,@a+dptr ;Get segment_2 data (digit + point)
00A0:	F5A0	150	mov Digit_2,a
00A2:	01AF	R 151	ajmp New_meas
00A4:		152	Com_1:
00A4:	64FF	153	xrl a,#0ffh
00A6:	F5B0	154	mov Digit_1,a
00A8:	7401	155	mov a,#01
00AA:	93	156	movc a,@a+dptr ;Get segment_2 data (digit+point)
00AB:	64FF	157	xrl a,#0ffh
00AD:	F5A0	158	mov Digit_2,a
00AF:		159	New_meas:
00AF:	D2AB	160	setb etl ;Enable LCD switch interrupt
00B1:	0130	R 161	ajmp Start_ADC
		162	
		163	
00B3:		164	Small_2v6: ;V<2.55V. Exchange registers
00B3:	C201	R 165	clr Gr_2v6 ;Clear flag
00B5:	CC	166	xch a,r4 ;Exchange [R4.R5] and [R6.R7]
00B6:	CE	167	xch a,r6
00B7:	FC	168	mov r4,a
00B8:	CD	169	xch a,r5
00B9:	CF	170	xch a,r7
00BA:	FD	171	mov r5,a
00BB:	80B0	172	sjmp Calculate ;Start calculation

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 5

```

LOC  OBJ          LINE  SOURCE
-----
                                173
                                174          extrn code(__sdivi)
                                175  $EJECT
                                176
                                177
;*****
                                178  ;A/D measurement interrupt routine (INT2)
                                179
;*****
                                180
-----
                                181          CSEG AT 3Bh      ;INT2 vector (Voltage measurement)
003B: 020000  R  182          ljmp ADC_Int
                                183
-----
                                184          RSEG ADC
0000: C28C    R  185  ADC_Int:  clr tr0          ;Stop timer_0
0002: B200    R  186          cpl ADC_Status
0004: 300009  R  187          jnb ADC_Status,Stat_0
                                188
                                189  ;ADC_Status is '1'
0007: AC8C    R  190          mov r4,th0      ;Get timer_0 values
0009: AD8A    R  191          mov r5,t10     ;T1
000B: 53E9FE  R  192          anl ix1,#0FEh ;INT2 on '0' input level
                                193
                                194          ;Check discharge Vd+1.3V
                                195
000E: 8003    R  196          sjmp Leave_INT2
                                197
                                198  ;ADC_Status is '0'
0010: 43E901  R  199  Stat_0:  orl ix1,#01H   ;INT2 on '1' input level
                                200
0013: C2C0    R  201  Leave_INT2:  clr iq2        ;Clear interrupt flag
0015: 32      R  202          reti
                                203
                                204
                                205  $eject
                                206
                                207
;*****
                                208  ;Timer_1 interrupt for inverting LCD signals
                                209
;*****
                                210
-----
                                211          CSEG AT 1Bh      ;Timer_1 interrupt (LCD switch)
001B: 020000  R  212          ljmp LCD_int
                                213
-----
                                214          RSEG LCD
0000: 63B0FF  R  215  LCD_int:  xrl Digit_1,#0ffh ;Invert digit_1
0003: B295    R  216          cpl LCD_COM     ;Invert LCD_COM
0005: 63A0FF  R  217          xrl Digit_2,#0ffh ;Invert digit_2

```

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 6

```

LOC  OBJ          LINE  SOURCE
0008: 0A          218          inc Dis_tim      ;Update discharge timer
0009: 32          219          reti
                220
                221  $reject
                222
                223
;*****
                224  ;Power failure interrupt routine (INT4)
                225
;*****
                226
----          227          CSEG AT 04Bh    ;Power failure interrupt from PCF1252
004B: 020000  R  228          ljmp P_Fail
                229
----          230          RSEG Power_F
0000: C291      R  231  P_Fail:    clr Anal_in     ;Ground analog input
0002: E4          232          clr a
0003: F5B0          233          mov Digit_1,a  ;Clear LCD
0005: C295          234          clr LCD_COM
0007: F5A0          235          mov Digit_2,a
0009: 80FE          236          sjmp $        ;Exit only with RESET !!
                237
                238  $reject
                239
                240

```

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 7

```

LOC   OBJ          LINE  SOURCE
                                241 ;Table with LCD segment data for V>=2.55V
                                242
                                243
----
                                244          rseg table_1
                                245
0000:          R  246  Seg_Tab_1:
0000:          247          ds 32          ;V<2.55V. (Adr:0..31)
0020: DBFD        248          db 0dbh,0fdh          ;'2.6V LOBAT'
                                ;(Adr:32..33)
0022: DB87DB87    249          db 0dbh,87h,0dbh,87h          ;'2.7V LOBAT'
                                ;(Adr:34..37)
0026: DBFFDBFF    250          db 0dbh,0ffh,0dbh,0ffh          ;'2.8V LOBAT'
                                ;(Adr:38..41)
002A: DBEFDDEF    251          db 0dbh,0efh,0dbh,0efh          ;'2.9V LOBAT'
                                ;(Adr:42..45)
002E: 4FBF        252          db 04fh,0bfh          ;'3.0V' (Adr:46..47)
0030: 4FB64F86    253          db 04fh,86h,04fh,86h,04fh,86h          ;'3.1V' (Adr:48..53)
0034: 4F86
0036: 4FDB4FDB    254          db 4fh,0dbh,4fh,0dbh          ;'3.2V' (Adr:54..57)
003A: 4FCF4FCF    255          db 4fh,0cfh,4fh,0cfh          ;'3.3V' (Adr:58..61)
003E: 4FE64FE6    256          db 4fh,0e6h,4fh,0e6h          ;'3.4V' (Adr:62..65)
0042: 4FED4FED    257          db 4fh,0edh,4fh,0edh          ;'3.5V' (Adr:66..69)
0046: 4FFD4FFD    258          db 4fh,0fdh,4fh,0fdh,4fh,0fdh          ;'3.6V' (Adr:70..75)
004A: 4FFD
004C: 4FB74FB7    259          db 4fh,87h,4fh,87h          ;'3.7V' (Adr:76..79)
0050: 4FFF4FFF    260          db 4fh,0ffh,4fh,0ffh          ;'3.8V' (Adr:80..83)
0054: 4FEF4FEF    261          db 4fh,0efh,4fh,0efh,4fh,0efh          ;'3.9V' (Adr:84..89)
0058: 4FEF
005A: 66BF66BF    262          db 66h,0bfh,66h,0bfh          ;'4.0V' (Adr:90..93)
005E: 66866686    263          db 66h,86h,66h,86h          ;'4.1V' (Adr:94..97)
0062: 66DB66DB    264          db 66h,0dbh,66h,0dbh,66h,0dbh          ;'4.2V' (Adr:98..103)
0066: 66DB
0068: 66CF66CF    265          db 66h,0cfh,66h,0cfh          ;'4.3V' (Adr:104..107)
006C: 66E666E6    266          db 66h,0e6h,66h,0e6h,66h,0e6h          ;'4.4V' (Adr:108..113)
0070: 66E6
0072: 66ED66ED    267          db 66h,0edh,66h,0edh,66h,0edh          ;'4.5V' (Adr:114..119)
0076: 66ED
0078: 66FD66FD    268          db 66h,0fdh,66h,0fdh          ;'4.6V' (Adr:120..123)
007C: 66876687    269          db 66h,87h,66h,87h,66h,87h          ;'4.7V' (Adr:124..129)
0080: 6687
0082: 66FF66FF    270          db 66h,0ffh,66h,0ffh,66h,0ffh          ;'4.8V' (Adr:130..135)
0086: 66FF
0088: 66EF66EF    271          db 66h,0efh,66h,0efh          ;'4.9V' (Adr:136..139)
008C: 6DBF6DBF    272          db 06dh,0bfh,06dh,0bfh,06dh,0bfh ;'5.0V' (Adr:140..147)
0090: 6DBF
0092: 6DBF        273          db 06dh,Cbfh
0094: 76B0        274          db 76h,0b0h          ;'H. IV' (Adr:148..149)
                                275
                                276 $EJECT

```

A/D conversion with P83CL410 PCF1252-x

EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 8

```

LOC   OBJ          LINE  SOURCE
                                     277
                                     278
;*****
279   ;Table with LCD segment data for V<=2.55V
280
;*****
281
---- 282           rseg table_2
283
0000:   R 284   Seg_Tab_2:
0000:   285           ds 32           ;V>2.55V. (Adr:0..31)
0020: DBEDDBED 286           db 0dbh,0edh,0dbh,0edh,0dbh,0edh;'2.5V LOBAT'
0024: DBED           ; (Adr:32..37)
0026: DBE6DBE6 287           db 0dbh,0e6h,0dbh,0e6h,0dbh,0e6h;'2.4V LOBAT'
002A: DBE6           ; (Adr:38..43)
002C: DBCFDBCF 288           db 0dbh,0cfh,0dbh,0cfh,0dbh,0cfh;'2.3V LOBAT'
0030: DBCF           ; (Adr:44..49)
0032: DBDBDBDB 289           db 0dbh,0dbh,0dbh,0dbh,0dbh,0dbh;'2.2V LOBAT'
0036: DBDB           ; (Adr:50..57)
0038: DBDB 290           db 0dbh,0dbh
003A: DB86DB86 291           db 0dbh,86h,0dbh,86h,0dbh,86h ;'2.1V LOBAT'
003E: DB86           ; (Adr:58..69)
0040: DB86DB86 292           db 0dbh,86h,0dbh,86h,0dbh,86h
0044: DB86
0046: DBBFDBBF 293           db 0dbh,0bfh,0dbh,0bfh,0dbh,0bfh;'2.0V LOBAT'
004A: DBBF           ; (Adr:70..85)
004C: DBBFDBBF 294           db 0dbh,0bfh,0dbh,0bfh,0dbh,0bfh
0050: DBBF
0052: DBBFDBBF 295           db 0dbh,0bfh,0dbh,0bfh
0056: 86EF86EF 296           db 86h,0efh,86h,0efh,86h,0efh ;'1.9V LOBAT'
005A: 86EF           ; (Adr:86..109)
005C: 86EF86EF 297           db 86h,0efh,86h,0efh,86h,0efh
0060: 86EF
0062: 86EF86EF 298           db 86h,0efh,86h,0efh,86h,0efh
0066: 86EF
0068: 86EF86EF 299           db 86h,0efh,86h,0efh,86h,0efh
006C: 86EF
006E: 86FF86FF 300           db 86h,0ffh,86h,0ffh,86h,0ffh ;'1.8V LOBAT'
0072: 86FF           ; (Adr:110..121)
0074: 86FF86FF 301           db 86h,0ffh,86h,0ffh,86h,0ffh
0078: 86FF
007A: 38BF38BF 302           db 38h,0bfh,38h,0bfh,38h,0bfh ;'1.0V LOBAT'
007E: 38BF           ; (Adr:122..133)
0080: 38BF38BF 303           db 38h,0bfh,38h,0bfh,38h,0bfh
0084: 38BF
0086:           304           end
0086:           305

```

Driver for 8xC851 E2PROM

EIE/AN91009

1. INTRODUCTION

A set of software functions is given to access the E²PROM on the 8xC851 microcontrollers. These functions can be called from application programs written in assembly, PL/M-51 or C. The functions are found in the E2PROM.OBJ file that can be linked to the application program.

The driver is written and tested with the following software tools from BSO-Tasking:

Assembler:	ASM51 V3.2	(OM4142)
PL/M51:	PL/M51 V3.0A	(OM4144)
C Comp.:	C51 V2.0	(OM4136)
Debugger:	XRAY51 V1.4c	(OM4129)

Resources used by driver:

- Exclusive use of 1 register bank (default RB1)
- Accumulator
- PSW
- 1 static bit addressable RAM byte

2. FUNCTION DESCRIPTIONS

The functions that use write and/or erase actions are interrupt driven except for E2PROM_wr_byte_pol. The application can check the status of these actions by testing the flag E2PROM_BUSY. This flag is available via the function E2PROM_status.

In the 8xC851, the E²PROM interrupt is combined with the UART interrupt. To enable the E²PROM interrupt, EA (in the IE-register) must be set (should be done in application program), the combined UART/E²PROM enable bit must be set (ES in the IE-register, done with function E2PROM_int_en) and the E²PROM interrupt enable bit (EEINT in ECNTRL register) must be set. The E²PROM interrupt flag is automatically set by functions that use erase/write actions. This means that the UART interrupt enable cannot be disabled while the E²PROM interrupt is completely enabled. The E²PROM can be disabled separately with the E2PROM_int_dis function.

The priority level for UART and E²PROM interrupt are the same and are defined with the E2PROM_int_en function.

The E²PROM driver has a link to a UART interrupt handler. When a UART interrupt occurs, the status of the controller is pushed on the stack and then interrupt flags are tested to determine the source of the interrupt. When the source of the interrupt is the UART, then subroutine _UART_HDL is called. the implementation of the UART interrupt handler is done by the user. On the disk a file UART.SRC is available that contains this subroutine. This routine will only clear the trx-interrupt flag (TI) and rcv-interrupt flag (RI).

Driver for 8xC851 E2PROM

EIE/AN91009

2.1 E2PROM_init

Function description:

This function must be called before any of the other functions is called. The timing register for writing/erasing the E²PROM is initialized and the register bank that the E²PROM functions can use is defined.

The default registerbank is RB1; the ETIM register which determines the write/erase timing is default initialized with 0x7B (XTAL = 12MHz). If other values are required, the parameters REGISTERBANK and XTAL must be changed in the equate list of the source file (E2PROM.ASM).

The E²PROM/UART interrupt is enabled and set to priority level '0'.

Calling Sequence:

```
E2PROM_init();
```

Function prototype:

```
void E2PROM_init (void)
```

Parameters:

None

2.2 E2PROM_int_en

Function description:

This function will enable the E²PROM/UART interrupt. The global enable bit EA is not effected and must be controlled by the application program.

The priority level of the E²PROM/UART is controlled by the parameter 'Pr_Level'.

Calling Sequence:

```
E2PROM_int_en (Pr_Level);
```

Function prototype:

```
void E2PROM_int_en (data char Pr_Level)
```

Parameters:

Pr_Level: This parameter determines the priority level on which the E²PROM/UART interrupts are handled. Values greater than 0x01 will be interpreted as 0x01.

2.3 E2PROM_int_dis

Function description:

This function will disable the E²PROM/UART interrupt.

Calling Sequence:

```
E2PROM_Int_Dis;
```

Function prototype:

```
void E2PROM_Int_Dis (void)
```

Parameters:

None

Driver for 8xC851 E2PROM

EIE/AN91009

2.4 E2PROM_status

Function description:

This function will return the E2PROM_BUSY bit, which indicates whether a read/write transfer from/to the E2PROM, or an erase action is finished.

'1' indicates that a read/write transfer is still in progress.

'0' indicates that no read/write transfer is in progress.

If the application program calls an E2PROM function while another E2PROM function is still in progress, parameters may be overwritten and an erroneous result will be obtained. There are 2 exceptions on this rule. When the functions "E2PROM_rd_byte_pol" or "E2PROM_wr_byte_pol" are called, parameters are passed to different registers in the registerbank, the E2PROM status is stored and the transfer is done by polling.

Note that the E2PROM_BUSY bit is not the same as EWP-flag in the ECNTRL register. For write operations the EWP-flag indicates when the writing/erasing of a byte to the E2PROM is finished. The E2PROM_BUSY flag indicates when a complete block of data (e.g., from the E2PROM_wr_block function) has been written to the E2PROM.

Calling Sequence:

```
bit Status;  
Status = E2PROM_Status;
```

Function prototype:

```
bit E2PROM_Status (void)
```

Parameters:

None

2.5 E2PROM_wr_byte

Function description:

This function will write a byte to E2PROM.

If the source byte has the same value as the byte in the E2PROM, no write action will take place.

Byte transfer is done on interrupt basis. The status of the transfer can be checked with the "E2PROM_Status" function.

This function will automatically enable the E2PROM interrupt. The application program should take care of the E2PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

Calling Sequence:

```
E2PROM_wr_byte (Src_Byte, Dest_Ptr);
```

Function prototype:

```
void E2PROM_wr_byte (data char Src_Byte, data char Dest_Ptr)
```

Parameters:

```
Src_Byte:   Byte to be written to E2PROM  
Dest_Ptr:   Address of E2PROM
```


Driver for 8xC851 E2PROM

EIE/AN91009

2.6 E2PROM_rd_byte

Function description:

This function will read a byte from E²PROM. The status of the transfer can be checked with the "E2PROM_Status" function.

Calling Sequence:

```
data char Result;
Result = E2PROM_rd_byte (Src_Ptr);
```

Function prototype:

```
char E2PROM_rd_byte (data char Src_Ptr)
```

Parameters:

Src_Ptr: Address of E²PROM

2.7 E2PROM_wr_block

Function description:

This function will write a block of data from internal RAM to E²PROM.

Byte transfer is done on interrupt basis. The status of the transfer can be checked with the "E2PROM_Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

If the source bytes are the same as the bytes in the E²PROM, no write action will take place. This function will automatically generate ROW-erases, whenever this will reduce programming time. If during execution of this function, the destination address to the E²PROM is equal to the beginning of an E²PROM row (3 least significant address bits are '0') and at least 8 more bytes have to be programmed, a check will be done whether a ROW-erase will reduce programming time. If no ROW-erase is done, the time to program the ROW will be:

$$T_{\text{prog}} = A \cdot t_W + B \cdot (t_E + t_W)$$

A: Byte in E²PROM is 0x00; source byte in RAM is not 0x00
 B: Byte in E²PROM is not 0x00; source byte in RAM <> E²PROM byte

If a ROW-erase is done, programming the ROW will take:

$$T_{\text{prog}} = t_E + C \cdot t_W$$

C: Source byte in RAM <> '0'

Because the erase time (t_E) and the write time (t_W) are equal, the function will do a ROW-erase if $A + 2 \cdot B - C - 1 \geq 0$

Calling Sequence:

```
E2PROM_wr_block (Src_Ptr, Dest_Ptr, Nr_Bytes);
```

Function prototype:

```
void E2PROM_wr_block (data char *data Src_Ptr, data char Dest_Ptr, data char Nr_Bytes)
```

Parameters:

Src_Ptr: Address pointer to internal RAM
 Dest_Ptr: Address of first E²PROM byte
 Nr_Bytes: Number of bytes to write to E²PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.8 E2PROM_rd_block

Function description:

This function will read a block of data from E²PROM and store it in internal RAM. The status of the transfer can be checked with the "E2PROM_Status" function.

Calling Sequence:

E2PROM_rd_block (Src_Ptr, Dest_Ptr, Nr_Bytes);

Function prototype:

```
void E2PROM_rd_block (data char Src_Ptr, data char *data Dest_Ptr, data char Nr_Bytes)
```

Parameters:

Src_Ptr: Address of first E²PROM byte
Dest_Ptr: Address pointer to internal RAM
Nr_Bytes: Number of bytes to read from E²PROM

2.9 E2PROM_wr_byte_pol

Function description:

This function will write a byte from internal RAM to E²PROM.

If the source byte has the same value as the byte in the E²PROM, no write action will take place.

If an E²PROM function is in progress (except E2PROM_rd_byte_pol), this function will be interrupted but its status will be saved so that the interrupted function will be resumed when the E2PROM_wr_byte_pol function is finished.

Byte transfer is done by polling.

This function may be used, e.g., in interrupt service routines, where the possibility exists that the interrupted main program has already started an E²PROM transfer.

Calling Sequence:

E2PROM_wr_byte_pol (Src_Byte, Dest_Ptr);

Function prototype:

```
void E2PROM_wr_byte_pol (data char Src_Byte, data char Dest_Ptr)
```

Parameters:

Src_Byte: Byte to be written to E²PROM
Dest_Ptr: Address of E²PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.10 E2PROM_rd_byte_pol

Function description:

This function will read a byte from E²PROM.

If an E²PROM function is in progress (except E2PROM_wr_byte_pol), this function will be interrupted but its status will be saved so that the interrupted function will be resumed when the E2PROM_rd_byte_pol function is finished.

This function may be used, e.g., in interrupt service routines, where the possibility exists that the interrupted main program has already started an E²PROM transfer.

Calling Sequence:

```
data char Result;  
Result = E2PROM_rd_byte_pol (Src_Ptr);
```

Function prototype:

```
char E2PROM_Rd_Byte_Pol (data char Src_Ptr)
```

Parameters:

Src_Ptr: Address of E²PROM

2.11 E2PROM_block_erase

Function description:

This function will erase all 256 E²PROM bytes.

The erase function is done on interrupt basis. The status of the transfer can be checked with the "E2PROM_Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

Calling Sequence:

```
E2PROM_block_erase();
```

Function prototype:

```
void E2PROM_block_erase (void)
```

Parameters:

None

Driver for 8xC851 E2PROM**EIE/AN91009**

2.12 E2PROM_security_on**Function description:**

This function inhibits access to E²PROM from external program memory.

The following scheme gives the access possibilities when this function is executed.

EA pin	Address of E2PROM access instruction	Access to E2PROM
1	< 4096	YES
1	>= 4096	NO
0	< 4096	NO
0	>= 4096	NO

The write function is done on interrupt basis. The status of the transfer can be checked with the "E2PROM Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

Calling Sequence:

```
E2PROM_security_on();
```

Function prototype:

```
void E2PROM_security_on (void)
```

Parameters:

None

Driver for 8xC851 E2PROM

EIE/AN91009

2.13 E2PROM_security_off**Function description:**

This function will remove E²PROM protection. Access to E²PROM from external program memory is now possible if this function is executed from the right program memory.

The following scheme gives the possibilities when 'E²PROM_security_off' is executed after completion of the 'E²PROM_security_on' function.

The following table assumes that the address at which 'E²PROM_security_off' resides is smaller than 4096.

EĀ pin	Address of E2PROM access instruction	Mode	Access to E2PROM	E2PROM erased
1	< 4096	0	YES	NO
1	>= 4096	0	YES	NO
1	< 4096	1	YES	YES
1	>= 4096	1	YES	YES
0	< 4096	0	NO	NO
0	>= 4096	0	NO	NO
0	< 4096	1	YES	YES
0	>= 4096	1	YES	YES

The following table assumes that the address at which 'E²PROM_security_off' resides is greater than 4096.

EĀ pin	Address of E2PROM access instruction	Mode	Access to E2PROM	E2PROM erased
1	< 4096	0	YES	NO
1	>= 4096	0	NO	NO
1	< 4096	1	YES	YES
1	>= 4096	1	NO	YES
0	< 4096	0	NO	NO
0	>= 4096	0	NO	NO
0	< 4096	1	YES	YES
0	>= 4096	1	YES	YES

Calling Sequence:

```
E2PROM_Security_Off;
```

Function prototype:

```
void E2PROM_Security_Off (data char Mode)
```

Parameters:

Mode: If '0', then the protection will only be removed when this function is executed from internal program memory. When executed from external memory, the protection remains.
If '1', then the protection can also be removed when this function is executed from external memory, however, all E²PROM bytes will be erased.

Driver for 8xC851 E2PROM

EIE/AN91009

3. PROGRAM EXAMPLES

Three examples are given that show how to use these functions with C, PL/M51 and assembly programs. In the examples, a string of characters is written to and read from E²PROM. When reading back the string, spaces are replaced by underscores.

3.1 C example

The disk contains the file E2PROM.H that should be included in the C application program. E2PROM.H contains the function prototypes of the E²PROM functions. The example program can be found on the disk in file TEST_C.C.

When the application module is compiled and assembled, it should be linked to the E²PROM function module E2PROM.OBJ and the UART interrupt handler UART.OBJ.

3.2 PL/M51 example

The disk contains the file E2PROM.DCL that should be included in the PL/M51 application program. E2PROM.DCL contains the external function declarations for the E²PROM functions. The example program can be found on the disk in file TEST_PLM.PLM.

When the application module is compiled and assembled, it should be linked to the E²PROM function module E2PROM.OBJ and the UART interrupt handler UART.OBJ. When linking, the linking control 'NOCASE' must be used!

3.3 Assembly example

The disk contains the file E2PROM.MAC which contains the macro definitions of the functions. Including these macro's in the source file eases programming. For instance, the sequence:

```
MOV _E2PROM_rd_block_BYTE ,Src_Ptr      ;Pointer to E2PROM
MOV _E2PROM_rd_block_BYTE+1, Dest_Ptr   ;Pointer to RAM
MOV _E2PROM_rd_block_BYTE+2, #Nr_Bytes  ;Number of bytes to transfer
LCALL _E2PROM_rd_block                  ;Call function
```

can be replaced by

```
%E2PROM_rd_block(Src_Ptr, Dest_Ptr, #Nr_Bytes)
```

The file E2PROM.GLO contains the EXTRN-definitions of the functions and constants that are used by the driver. Only the definitions used by the source program should be included.

When the application module is compiled and assembled, it should be linked to the E²PROM function module E2PROM.OBJ and the UART interrupt handler UART.OBJ.

Driver for 8xC851 E2PROM

EIE/AN91009

3.4 Listing of examples

LISTING C EXAMPLE:

```

#include "E2PROM.h"
#include <string.h>
#define E2PROM_Base_Address 0x58

rom char txt_tab[] = "This is an E2PROM test for 8xC851"

void main(void)
{
    data char    Data_Buffer[35];
    data char    Count;

    E2PROM_init();          /* Initialize E2PROM */
    E2PROM_int_en(0x01);   /* E2PROM interrupt level 1 */
    EA=1;                  /* Global interrupt enable */

    romidmove(&Data_Buffer,&Txt_tab,sizeof(Txt_tab)-1); /* Copy string from ROM
                                                         to RAM */
    E2PROM_wr_block(&Data_Buffer,E2PROM_Base_Address,sizeof(Txt_tab)-1);
                                                         /* Copy setting to E2PROM */

    /* Time to do other useful things while data is being written to
       E2PROM on interrupt basis */

    while (E2PROM_status()); /* Wait till transfer to E2PROM is finished */

    /* Read string from E2PROM and replace spaces " " by underscores "_" */
    for (Count=0;Count != sizeof(Txt_tab)-1;Count++)
    {
        /* Read E2PROM byte */
        Data_Buffer[Count] = E2PROM_rd_byte(E2PROM_Base_Address+Count);
        if (Data_Buffer[Count] == ' ')
            Data_Buffer[Count] = '_';
    }

    E2PROM_block_erase(); /* Erase E2PROM */
    /* Time to do other things while erasing */

    while (E2PROM_status()); /* Wait till erasing is finished */
    EA=0;
}

```

Driver for 8xC851 E2PROM

EIE/AN91009

LISTING PL/M51 EXAMPLE:

```

$DEBUG
$CODE

E2PROM: Do;
$INCLUDE (E2PROM.DCL)
$INCLUDE (UTIL51.DCL)

    Test: Do;

    Declare E2PROM_Base_Address literally '58H';
    Declare Txt_tab(*) Byte Constant
        ('This is an E2PROM test for 8xC851');
    Declare Data_Buffer(35) Byte Main;
    Declare Count Byte Main;

    Call E2PROM_init;          /* Initialize E2PROM */
    Call E2PROM_int_en(01);    /* E2PROM interrupt level 1 */
    Enable;                    /* Global interrupt enable */

    /* Copy string from ROM to RAM */
    Call MOVCD1(.Txt_tab,.Data_Buffer,length(Txt_tab));

    /* Copy string to E2PROM */
    Call E2PROM_wr_block(.Data_Buffer,E2PROM_Base_Address,length(Txt_tab));

    /* Time to do other useful things while data is being written to
       E2PROM on interrupt basis */

    Do While E2PROM_status = 1; /* Wait till transfer to E2PROM is finished */
    End;

    /* Read string from E2PROM and replace spaces " " by underscores "_" */
    Do Count=0 To length(Txt_tab);
        /*Read E2PROM byte */
        Data_Buffer(Count) = E2PROM_rd_byte(E2PROM_Base_Address+Count);
        If (Data_Buffer(Count) = ' ') then Data_Buffer(Count) = '_';
    End;

    Call E2PROM_block_erase; /* Erase E2PROM */
    /* Time to do other things while erasing */

    Do While E2PROM_status = 1; /* Wait till erasing is finished */
    End;
    Disable;

    End Test;
End E2PROM;

```


Driver for 8xC851 E2PROM

EIE/AN91009

LISTING ASSEMBLY EXAMPLE:

```

$DEBUG
$CASE

; *=====*/
; *                                           */
; *           INCLUDE FILE : E2PROM.GLO       */
; *           PACKAGE      : E2PROM          */
; *                                           */
; *=====*/

EXTRN CODE    (_E2PROM_init)

EXTRN CODE    (_E2PROM_int_en)
EXTRN CODE    (_E2PROM_int_en_BYTE)

EXTRN CODE    (_E2PROM_status)

EXTRN CODE    (_E2PROM_rd_byte)
EXTRN NUMBER  (_E2PROM_rd_byte_BYTE)

EXTRN CODE    (_E2PROM_wr_block)
EXTRN NUMBER  (_E2PROM_wr_block_BYTE)

EXTRN CODE    (_E2PROM_block_erase)

; *=====*/
; *           Include macro definitions       */
; *=====*/
$INCLUDE(E2PROM.MAC)

        BUFFER SEGMENT DATA
        RSEG BUFFER
Data_Buffer:    ds 35
Count:         ds 1
Stack:         ds 15

        TABLE SEGMENT CODE
        RSEG TABLE
Txt_tab:       db    "This is an E2PROM test for 8xC851"

E2PROM_Base_Address    EQU 58H
Length_Txt             EQU 33

        CSEG AT 00                ;Reset vector
        LJMP MAIN

        TEST_ASM SEGMENT CODE
        RSEG TEST_ASM

MAIN:    MOV SP,#Stack-1          ;Initialize stack pointer
        %E2PROM_init             ;Initialize E2PROM
        %E2PROM_int_en(#10)      ;E2PROM interrupt level 1
        SETB EA                  ;Enable global interrupt

        MOV DPTR,#Txt_tab        ;Initialize pointers to copy Txt_tab to RAM
        MOV R0,#Data_Buffer
        MOV R2,#Length_Txt

```

Driver for 8xC851 E2PROM

EIE/AN91009

```

COPY_LOOP:                                ;Copy Txt_tab to RAM
    CLR A
    MOVC A,@A+DPTR                        ;Get byte from ROM
    MOV @R0,A                             ;Store in RAM
    INC DPTR                               ;Update pointers
    INC R0
    DJNZ R2,COPY_LOOP                     ;Check if all copied

                                        ;Write data to E2PROM
    %E2PROM_wr_block(#Data_Buffer,#E2PROM_Base_Address,#Length_Txt)
    ;
    ; Time to do other useful things while data is being written to
    ; E2PROM on interrupt basis
    ;
NEW_CHECK:
    %E2PROM_status                        ;Wait till transfer to E2PROM is finished
    JC NEW_CHECK

    ;Read string from E2PROM and replace spaces " " by underscores "_"
    MOV R0,#Data_Buffer                   ;Initialize pointers
    MOV R1,#E2PROM_Base_Address
    MOV R2,#Length_Txt
READ_LOOP:
    %E2PROM_rd_byte(R1)
    MOV @R0,A                             ;Store byte in RAM
    CJNE A,#" ",NEXT_READ                 ;Check if byte is " "
    MOV @R0,#"_"                           ;If yes, replace with "_"
NEXT_READ:
    INC R0                                 ;Update pointers
    INC R1
    DJNZ R2,READ_LOOP

    %E2PROM_block_erase                   ;Erase E2PROM

    ;Time to do other things while erasing */
NXT_CHECK:
    %E2PROM_status                        ;Wait till transfer to E2PROM is finished
    JC NXT_CHECK
    CLR EA                                 ;Disable interrupts
    JMP $                                  ;End of program

```

Driver for 8xC851 E2PROM

EIE/AN91009

4. DEBUG MACROS

The disk contains some debug macros that ease the debugging of programs that use the 8xC851 E²PROM. These macros can be executed by the XRAY51 High Level Language debugger. The user can read from and write to E²PROM bytes without programming the individual SFRs.

Before the macros can be executed, they must be loaded by XRAY51. This will be done automatically if the file 'E2PROM.INC' is included when invoking XRAY51 or during a debug session. E2PROM.INC will load the macros and define some symbols used by the macros. If not all macros are used, the file E2PROM.INC can be edited to prevent the loading of these macros. This may be necessary when there is insufficient memory to load the macros, because, for instance, other macros have been loaded. Another advantage of only loading the relevant macros is reduction of loading time.

When a macro is called from the debugger, the following SFRs will remain unchanged: ECNTRL, EADDRH, EADRL and ETIM. During macro execution, all interrupts will be disabled. Access to the E²PROM with the macros is independent of the state of the security bit. The execution and results of the macro are visible on the I/O screen of XRAY51 (VSCREEN 3).

Read(Start address, Stop address):

The value of E²PROM bytes from 'START ADDRESS' to 'STOP ADDRESS' will be shown.
If 'START ADDRESS' <= 'STOP ADDRESS' only the value of 'START ADDRESS' will be shown.

Write(Start address, Stop address, Value):

The E²PROM bytes from 'START ADDRESS' to 'STOP ADDRESS' will be programmed with 'VALUE'.
If 'START ADDRESS' > 'STOP ADDRESS', no E²PROM bytes will be programmed.
If the ETIM register contains the value 0x08, it is considered that ETIM is not initialized. The macro will give a warning, and return to the debug screen.

Copyto(Ram address, E2PROM address, Count):

Macro will copy 'COUNT' bytes, starting from internal RAM address 'RAM ADDRESS' to the E²PROM, starting at address 'E2PROM ADDRESS'.
If during copying, the RAM address becomes > 0x7F or the E²PROM address becomes > 0xFF, copying will be stopped and a warning is given that an address limit is reached.
If the ETIM register contains the value 0x08, it is considered that ETIM is not initialized. The macro will give a warning, and return to the debug screen.

Copyfrom(E2PROM address, Ram address, Count):

Macro will copy 'COUNT' bytes from E²PROM address 'E2PROM ADDRESS' to the internal RAM, starting at address 'RAM ADDRESS'.
If during copying, the RAM address becomes > 0x7F or the E²PROM address becomes > 0xFF, copying will be stopped and a warning is given that an address limit is reached.

Erase():

All E²PROM bytes will be erased.
If the ETIM register contains the value 0x08, it is considered that ETIM is not initialized. The macro will give a warning, and return to the debug screen.

Driver for 8xC851 E2PROM**EIE/AN91009**

5. CONTENTS OF DISK

The disk contains the following 3 directories:

1. \USER

This directory contains the files that may be included or linked to the source program.

E2PROM.ASM	:Source file of E ² PROM driver
E2PROM.OBJ	:Object file of E ² PROM driver
E2PROM.H	:Header file for C applications
E2PROM.DCL	:Declaration file for PL/M51
E2PROM.MAC	:Macro definitions for assembly applications
E2PROM.GLO	:Global definitions for assembly applications
UART.SRC	:UART interrupt handler (will only clear flags; user should customize it)
UART.OBJ	:Object file of UART interrupt handler

2. \DEBUG

This directory contains the macros and include file used with XRAY51 debugger.

E2PROM.INC	:Include file that reads macro files in XRAY51
*.MAC	:XRAY51 macros

3. \EXAMPLE

This directory contains the source files of the example programs described in the note

TEST_C.C	:C example
TEST_PLM.PLM	:PL/M51 example
TEST_ASM.ASM	:ASM51 example

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

1. Introduction

Quite recently customers asked for dedicated 8-bit microcontrollers at high clockrates which would not cause Radio Frequency (RF-) disturbances in consumer and communication applications, e.g. key-board control, tuning, etc...

Up to now either a slower 4-bit microcontroller was used or a more modern 8-bit microcontroller is in use with additional RF-shielding and filtering measures.

With a carradio manufacturer, a bargain was set for a new (EMC friendly) 8-bit microcontroller such that the product would maintain equal receiving performance with more control features. The new receiver would then only be modified for the microcontroller part.

To set the emission requirements, for a microcontroller in this application, the measurement technique described in the application note, EIE/AN91001 "Workbench EMC evaluation method", was used and applied to the existing receiver in which the new microcontroller has to fit in.

Later on, the same technique was used to verify the basic and modified samples which contained one or more measures to reduce RF-emission.

Each (mask) shrinking event will cause circuits to become faster and produce more RF-disturbances for these kind of appliances. As such, more EMC measures need to be taken after each shrinking event to maintain the above set emission requirements.

2. Measures to reduce RF-emission

Up to now, evident measures are known to reduce RF-emission. Within a few years time, new techniques will mature to keep pace with shrinking actions.

So far it is only interesting to take emission reducing measures on-chip for a microcontroller when there are no external ROM, RAM or (E)EPROM-busses. The externally required data- and address-busses will cause much more RF-radiation, when used in a non-shielded way.

We've restricted ourselves to stand-alone controllers, which can control most functions locally with, when required low speed serial busses e.g. I²C (100 kbit/s or 400 kbit/s with output-edge-control), or parallel communication to another controller.

To compare our results, reference is made to an existing standard microcontroller based on the INTEL 80C51-core mounted in a 40 pin DIL package.

Dedicated software has been used to allow true comparison between all microcontrollers in a defined application.

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

2.1 IC-package

Density problems demand smaller packages to allow further integration of functions within a certain product size. It was decided to take the Quad Flat Pack (QFP) 44, because it was the lowest pin number package available above the 40 pins required to apply the circuit as stated above.

The advantage of this package is the smaller loop area enclosed by the currents running through the circuit. As a result, direct radiation will be reduced. The worst-case area difference between the package sizes is:

DIL 40:	48,46 x 15,24 x 4,3 mm (l x w x h, h = PCB + die-path height)
	diagonal: 50,8 mm
	looparea: 218,5 mm ²
QFP 44:	11,4 x 8,8 x 1,2 mm (l x w x h, h = die-path height)
	diagonal: 14,4 mm
	looparea: 17,3 mm ²

When package radiation is considered only, the magnetic dipole moment produced by the package will be reduced by a factor of 12,6, which can give a decrease in electromagnetic emission of about 22 dB.

External supply decoupling capacitors can be placed more closely to the pins where needed. The latter will shorten the length of the current path between Vdd and Vss, thus resulting in a lower voltage drop appearing in-between reference point taken on the PCB. Considering this, the current path reduction will be a factor of 4,5 which would result in an RF-emission reduction of about 12 dB. One should consider that also I/O-pins will contribute to the RF-emission.

Practical radiation figures have shown a decrease of about 12 dB.

2.2 IC-pinning

As already given above, supply and I/O pinning will determine emission performance, due to the fact that radiation is determined by the way currents flow through a device. For some years it is known that each output pin should be embedded in-between a Vdd and a Vss pin. As such, the 3 one-byte wide busses would need $3 \times (2 \times 8 + 1)$ pins (O, Vdd, Vss), equals at least 51 pins. For such a device this seems unpractical.

The advantage of such a pinning will be that currents will always flow through adjacent leadframe fingers and as such, emission will be absolutely minimal. For this application, I/O will commonly occur at low frequencies, and as such, their contribution to RF-emission will be low. In CMOS applications these currents will mainly occur during transitions by charging and discharging the output load.

A more serious contribution will arise from the ground-bounce or supply-bounce which will occur between the PCB-reference and the IC's-substrate. This disturbance voltage will be superimposed to all I/O's which are either coupled to Vss or Vdd. When these lines are long, longer than the path involved for supply decoupling, their contribution to radiation

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

can be high. The disturbance source, being the supply- or ground-bounce voltage, has a negligible impedance (ωL , L = leadfinger + bondingwire inductance) and will be difficult to filter by using simple and cheap components such as capacitors. More often, radiation will increase by such measures due to the increased current through these I/O's.

The most effective action will be the use of several ground and supply pins. The ground pins must be spread around the circumference of the package while the supply pins need to be adjacent to these grounds to benefit the mutual coupling in-between. This mutual coupling reduces the 'effective' series inductance with the external decoupling capacitor.

With the DIL 40, pin 20 is the Vss-pin as pin 40 is the supply pin, Vdd. The I/O-pins are randomly located. With the QFP-44, for the Vss the following pins are selected: 6, 16, 28, 39 and the supply pins are: 17 (for I/O) and 38 (for the core).

Another cause for ground-bounce can be the X-tal oscillator's output with its external capacitance. Normally, this contribution can be reduced by adding some series impedance with the output and changing the capacitors values such that the X-tal circuit operation remains within its linear range with sufficient amplitude.

All together, the ground-bounce, which will be emitted by the I/O, can be reduced by a factor of 4 (4 gnd pins QFP versus 1 gnd pin DIL), assuming random I/O current distributions. Individual decoupling of the I/O and core supply will dampen the supply bounce even further.

As such these measures will reduce emissions further by some 12 dB. The expectation of even more drastic effects can be accounted for by the shorter leadframe finger lengths comparing DIL 40 to QFP 44.

2.3 On-chip decoupling measures

From the above, it will be clear that RF-emission will primarily come from the core and that the lower frequencies will be mainly caused by the I/O. For the latter output-edge-rate control can be considered, when the number of outputs are high compared to the number of Vss and Vdd pins.

When core decoupling is integrated, the high frequency low energy currents can close their loop on silicon. As a result these RF-currents will not flow through the leadframe any longer and will not add to any additional ground-bounce.

If the latter is implemented without further considerations, the effect can be quite negative. The original circuit design assumed that all charge (current) comes from the external decoupling capacitor. This charge (current) then, flows from the capacitor, through its leadwires, PCB traces, IC leadframe, interconnect to the circuit (that needed it) and then back. When on-chip decoupling is used, charge is there and switching will occur instantaneously.

In our case, most bus-driving circuits were re-designed such that waveforms maintain within their specifications under worst-case conditions (temperature, supply voltage, etc.).

The overall result is that on-chip decoupling will require little space due to the fact that all

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

empty areas can be used, even within the circuit-blocks, and that bus-driving circuits can be made much smaller with respect to the present dimensions.

2.4 PCB design measures

The QFP-package and the pinning of the controller will only reduce the RF-emission when the PCB is laid out following some constraints.

An example of a good supply and ground plane lay-out is given in Fig.1. The microcontroller is mounted on the component side of a double layer PCB. The supply pairs V_{ss1}/V_{dd1} for the I/O-supply and V_{ss3}/V_{dd2} for the core supply are connected directly to the ceramic chip capacitors C1 and C2, which are surface mounted on the solder side. A short connection of V_{ss3}/V_{dd2} to the capacitor C2 minimizes loop inductance. Thus, the external supply current drawn by the core will flow in this loop mainly. This current path can be ensured by the insertion of an inductor (L2) in series to the +5V general supply. An equal action is taken for the I/O-supply.

The implemented on-chip decoupling allows a separation between core and I/O-ports. The PCB lay-out shall use these V_{dd}/V_{ss} connections to minimize the loop areas in-between signal lines from each port pin to any load via the ground plane back to these V_{ss}/V_{dd} -pins.

By applying these hints, V_{ss2} may be used as the return pin for ALE, PSEN, port0 and port2 because V_{ss2} is connected very near to them. V_{ss3} shall be used for the core supply mainly. V_{ss1} is nearest to lower part of port2, upper part of port3 and the crystal oscillator. Even though V_{ss1} may be the best return for port2 and port3, this pin shall in any case be used as return for the external crystal oscillator capacitors (not shown here). V_{ss4} is located nearest to the lower part of port3 and the whole port1, being the best return for these ports.

3 Future developments

Existing products are shrunk, to cut costs and increase complexity. Recent developments (SAC3 → SAC2 → SAC1, C300 → C250 → C200 and others) have demonstrated an upwards tendency in the RF-emission following shrinking when EMC is not considered. This means that the end in taking measures to reduce RF-emission has not been reached. Further improvements are still possible an already considered for new products, such as:

- ☛ a PLL-circuit, to replace the high frequency X-tal oscillator,
- ☛ output-edge-control (application dependent),
- ☛ further circuit improvements on-chip e.g. coplanar supply, decoupling measures within cell-blocks, multi-phase clock systems to prevent simultaneous switching

4 Results

All measurements were carried out under the same conditions, using the same kind of PCB, with the same software and the same bus loadings, Fig.2.

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

Tested were the following applications:

- ☛ 1 87C51, DIL 40, 12 MHz X-tal → 0 dB(rel)
- ☛ 2 80C31, QFP 44, but with 2 V_{ss} and only one V_{dd} connection, 12 MHz X-tal → 13 dB
- ☛ 3 83CE654, QFP 44, with Address Latch Enable (ALE) active, 12 MHz X-tal → 32 dB
4 V_{ss} and 2 V_{dd} pins as given above.
- ☛ 4 83CE654, QFP 44, ALE off, 12 MHz X-tal → 50 dB
- ☛ 5 83CE654, QFP 44, ALE off, externally 12 MHz, 500 mV sinewave. → 54 dB

5 Conclusions

With the measures indicated in chapter 2, RF-emission can be reduced substantially, especially for stand-alone microcontroller applications. Up to now, a number of measures have been implemented which indicate a improvement of about 40 dB in the FM-region when changing from a DIL 40, annex 1 to a QFP 44 application, annex 4, taking into account minor additional supply measures. The inclusion of a PLL can make the improvement even further to about 50 dB, annex 5.

With these measures, some 12 to 22 dB can be accounted for by the choice of the package, same 12 dB due to the pinning and the rest due to the internal measures.

The advantage of these measures can be easily wasted by insufficient measures on the PCB. Constraints are given in chapter 2.4.

When considering the reducing effects PCB-layouts might have to RF-emission, the following relative information needs to be considered:

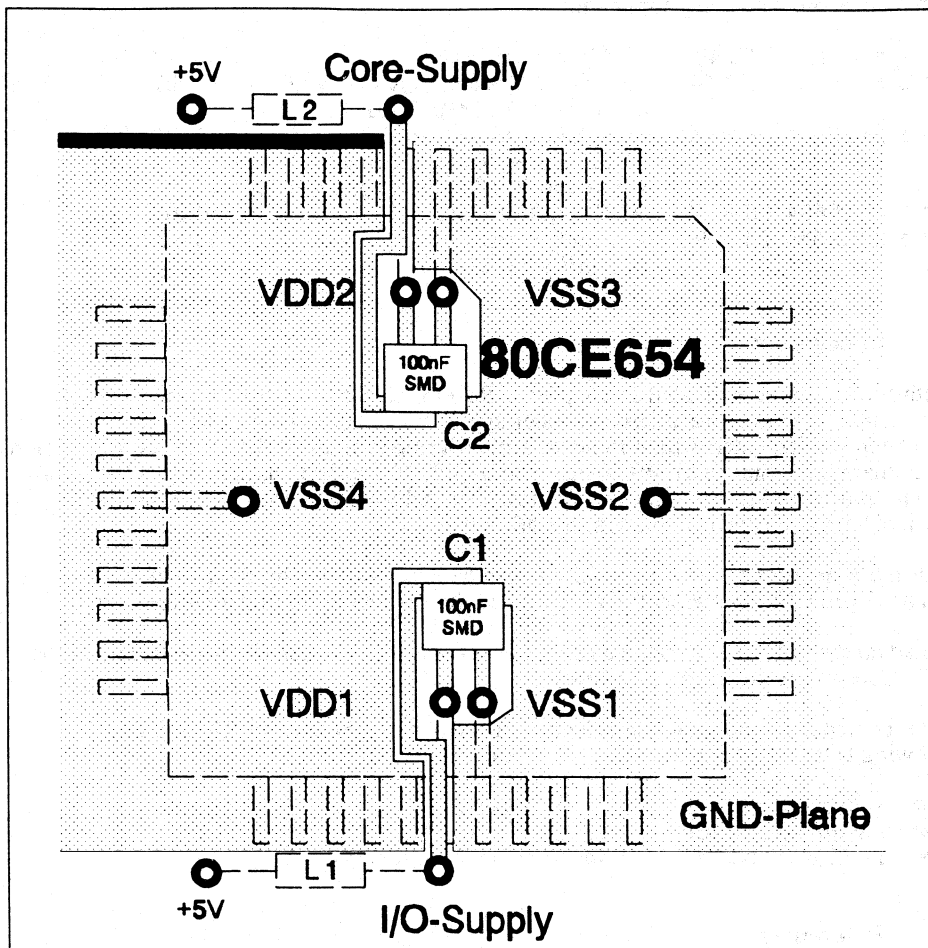
- ☛ single layer board → 0 dB (relative)
- ☛ double layer board → 26 dB (best case)
- ☛ multi layer board → 44 dB (4-layer, best case)

6 References

- [1] Syllabus of the design course "Fast digital and analog circuit design", Philips CTT, 1991, Eindhoven
- [2] IC package outlines, Philips Components, 1990, 12NC: 9398 175 80011.
- [3] Improvements in microcontrollers for a better EMC behaviour, H.Schutte, EIE/IN90039 version 2, 1991
- [4] Investigations on EME improvements for the microcontrollers 8xCE592 and 8xCE598, H-W Lütjens, HKI/IR 92001, 1992

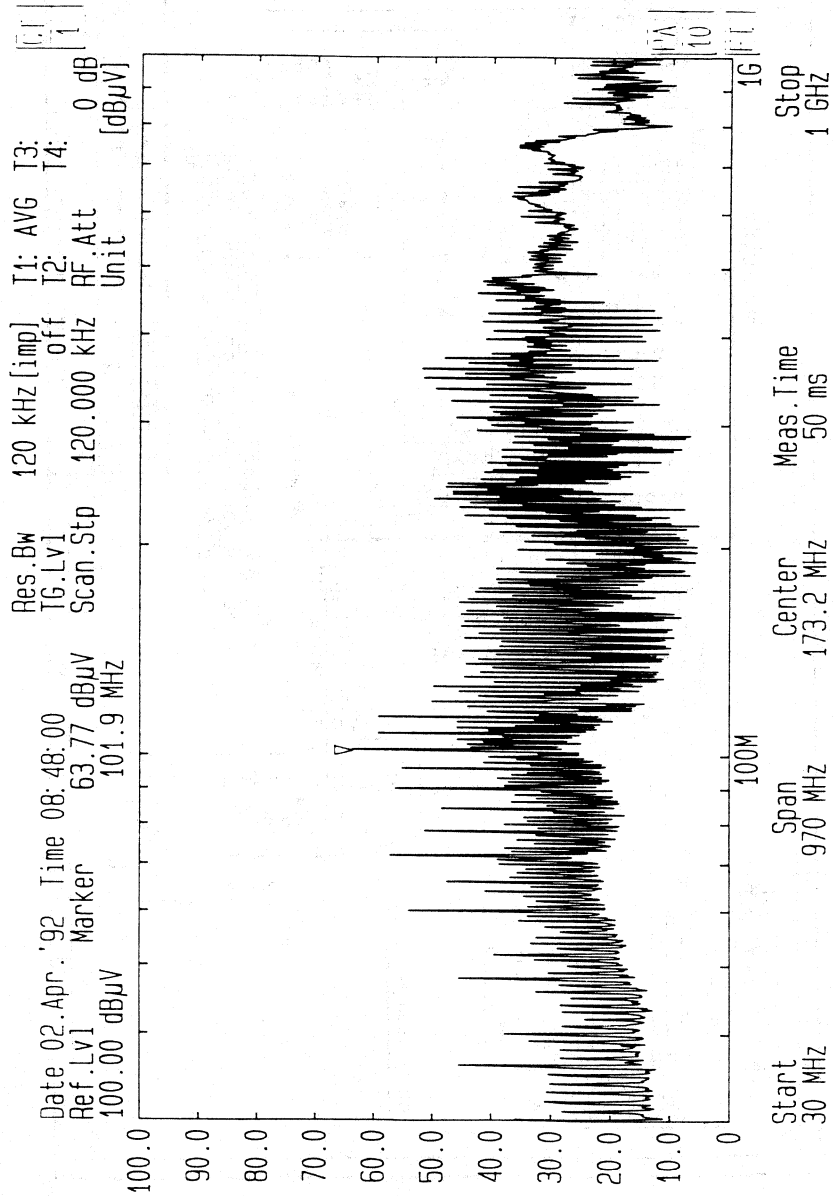
Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001



Low RF-emission applications with a P83CE654 microcontroller

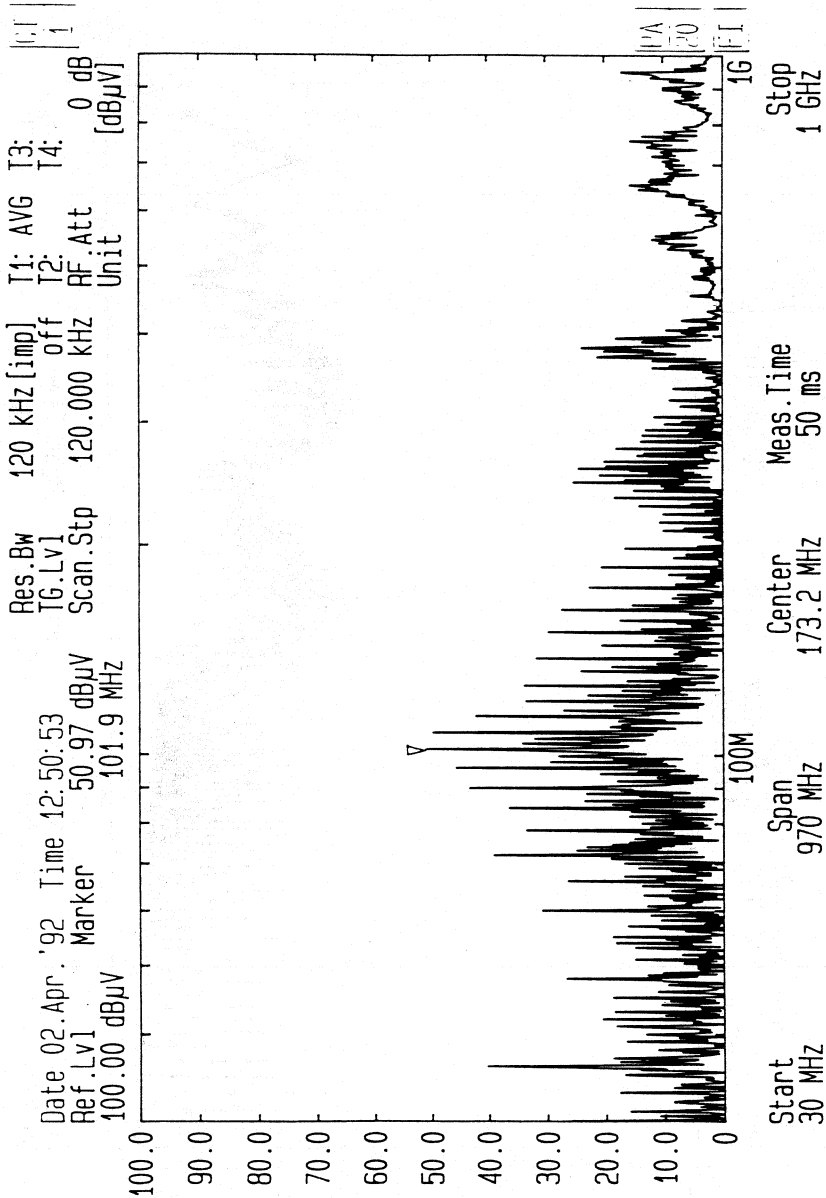
EIE/AN92001



Annex.1 RF-emission from a 87C51, DIL 40, 12 MHz X-tal.

Low RF-emission applications with a P83CE654 microcontroller

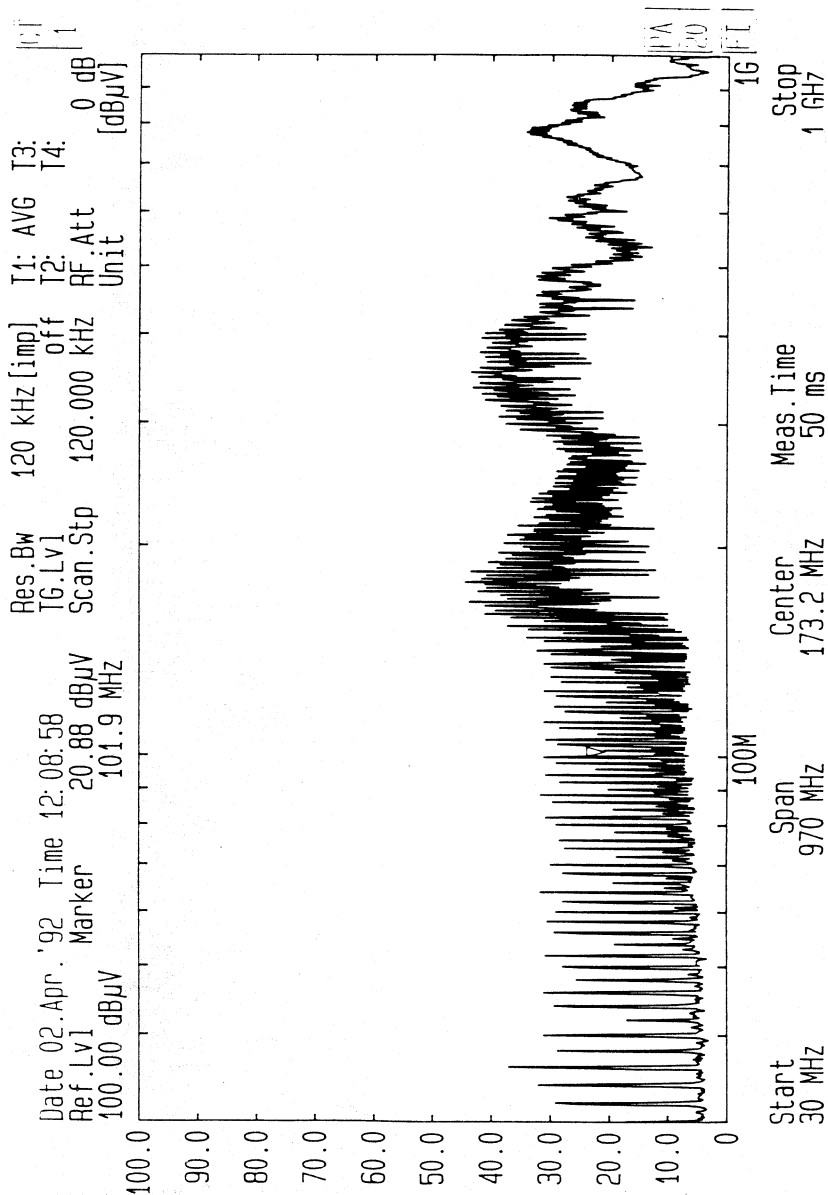
EIE/AN92001



Annex.2 RF-emission from a 80C31, QFP 44, but with 2 Vss and only one Vdd connection, 12 MHz X-tal.

Low RF-emission applications with a P83CE654 microcontroller

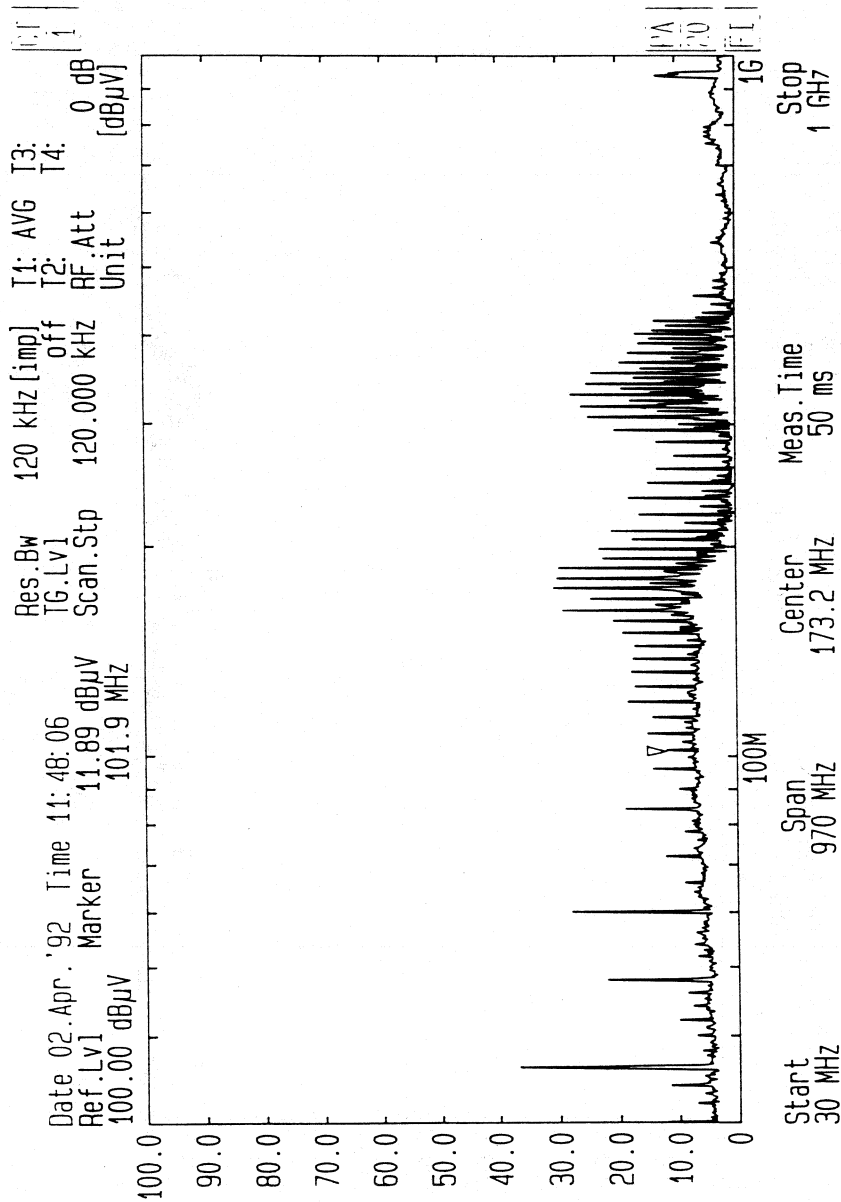
EIE/AN92001



Annex.3 RF-emission from a 83CE654, QFP 44, with Address Latch Enable (ALE) active, 12 MHz X-tal. 4 Vss and 2 Vdd pins as given above.

Low RF-emission applications with a P83CE654 microcontroller

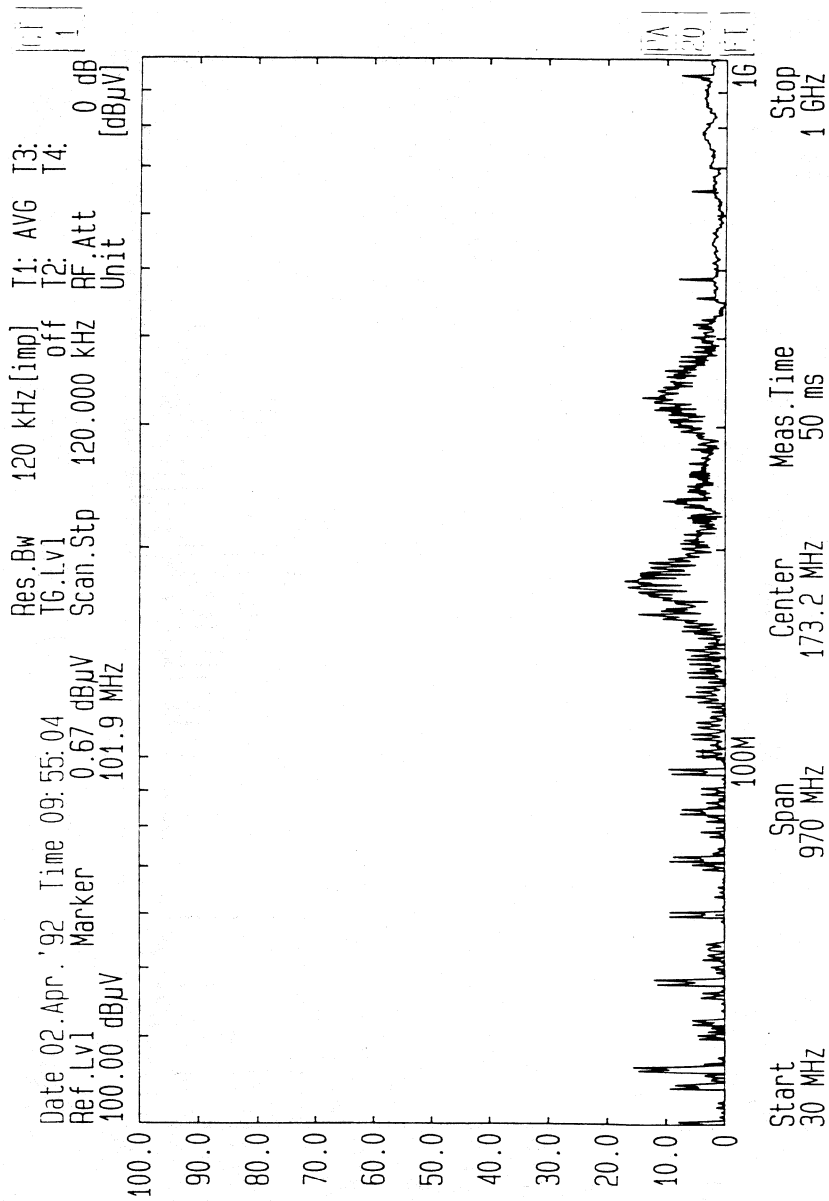
EIE/AN92001



Annex.4 RF-emission from a 83CE654, QFP 44, ALE off, 12 MHz X-tal.

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001



Annex.5 RF-emission from a 83CE654, QFP 44, ALE off, externally 12 MHz, 500 mV sinewave.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

Author: Theo van Daele, Philips Semiconductors Product Concept & Application Laboratory Eindhoven, the Netherlands

SUMMARY

On the 80C552 microcontroller, an 8-input 10-bit ADC is available. To get correct results from the ADC, the slew-rate of the input signal during sampling must be limited. 10 Bit accuracy will be obtained if the layout of the 80C552 application is done correctly. EMC measures must be taken into account. Some software examples are given on how to use the ADC.

1.0 INTRODUCTION

The 80C552 microcontroller has an on-chip ADC. The converter consists of an 8 input analog multiplexer, and a 10-bit binary successive approximation ADC. A conversion takes 50 machine cycles (is $20\mu\text{s}$ at 30MHz oscillator frequency). The ADC has dedicated analog supply and reference voltages to minimize influence from digital circuitry. The DAC of the successive approximation ADC is a resistor ladder network. This ensures that there are no missing codes.

To obtain the 10-bit accuracy, it is important to pay attention to the design of the application. First the operation of the ADC will be described. Then design and layout subjects are described that can influence the accuracy of the conversion result.

References:

4. 80C51-based 8-bit microcontrollers (Data Handbook IC20 1994)
5. Electro Magnetic Compatibility and Printed Circuit Board (PCB) Constraints (ESG89001)

2.0 INTERNAL OPERATION OF THE ADC

2.1 General Description

Figure 1 shows a general block diagram of the ADC.

The inputs of P5 are connected to a multiplexer and an input buffer with Schmitt-trigger inputs.

When the digital value on P5 must be read (e.g., with a MOV A,P5 instruction), the output of the Schmitt-trigger is taken. This output can be used for further processing.

An analog input signal on P5 that must be converted is selected by the input multiplexer. The bits ADCON.0 . . . ADCON.2 of the ADCON special function register select the input signal. The output of the multiplexer is connected to the input of a comparator. The sampling capacitor is included in the comparator. The ADC control block of the ADC controls the timing of the sampling and conversion.

After the input signal is sampled, the actual analog-to-digital conversion starts. The comparator compares the input signal V_{IN} with the output of the 10-bit DAC V_{DAC} . The output voltage of the DAC is determined by the output of the successive approximation register (SAR). The range of the DAC signal varies between AV_{REF-} and AV_{REF+} . These two signal levels also define the voltage range of the input signal.

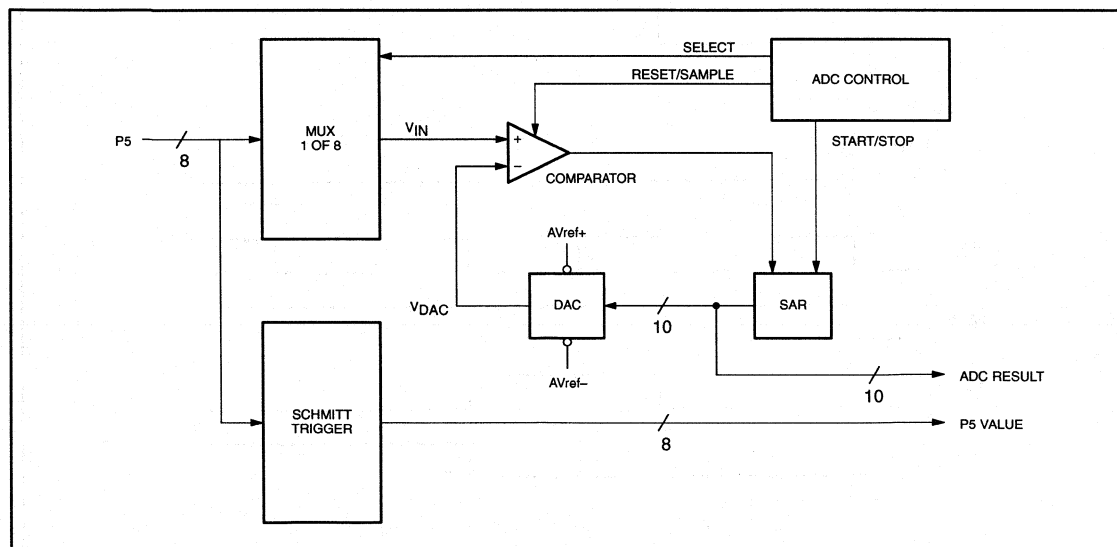


Figure 1.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

2.2 Conversion Process

Figure 2 shows an example of the conversion principle with 3 bit resolution.

The SAR will make its output bits SAR2 . . SAR0 successively high from MSB to LSB. Every time a SAR-line is made HIGH, a DA-conversion will take place. If the output of the DAC (V_{DAC}) is higher than the input voltage (V_{IN}), the SAR output bit that was made HIGH the last time will be made LOW. If V_{DAC} is smaller than V_{IN} , the SAR output bit will remain HIGH. The process will proceed

for the subsequent SAR output bits. At the end of the conversion, V_{DAC} has converged to a value of $V_{IN} \pm 1/2 \text{LSB}$.

Example: V_{IN} is $11/16 * V_{REF}$. The conversion sequence is shown in Table 1.

After STEP 3 the conversion is finished. The SAR register contains the result of the AD-conversion.

The ADC in the 80C552 has 10 bits resolution. The conversion in this ADC will take 10 conversion steps.

Table 1.

	SAR Value (SAR2.SAR1.SAR0)	V_{DAC} ($*V_{REF}$)	Output Comparator	Action by SAR
START	000	0	0	SAR2=1
STEP 1	100	4/8	0	SAR1=1
STEP 2	110	6/8	1	SAR1=0, SAR0=1
STEP 3	101	5/8	0	

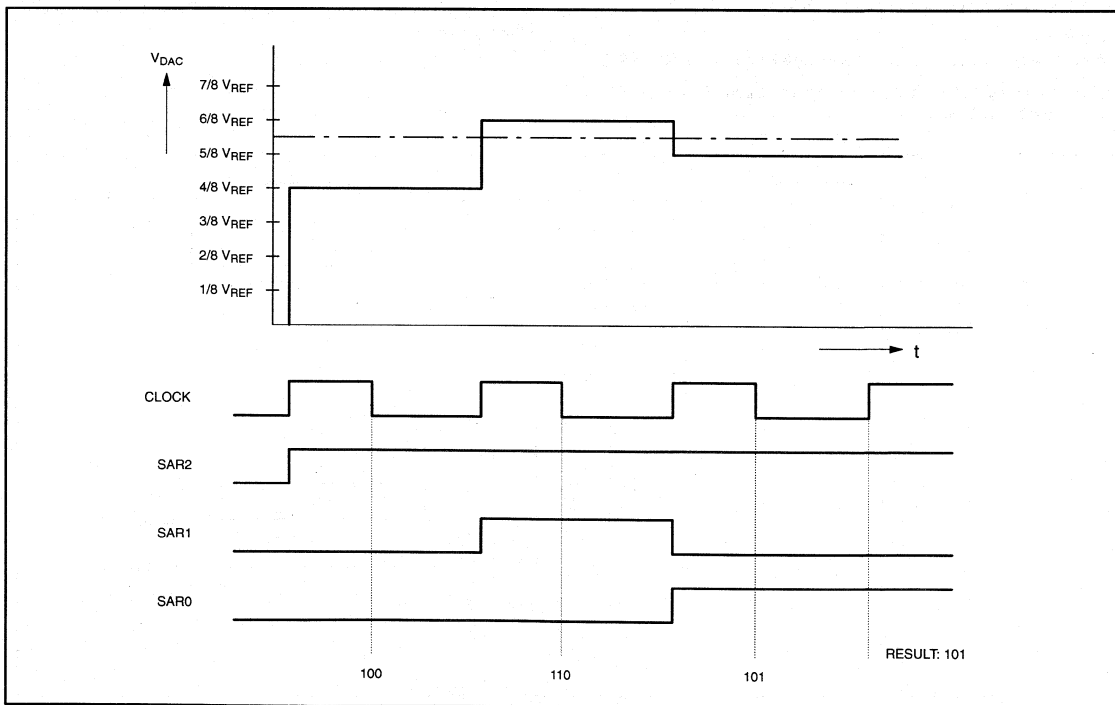


Figure 2.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

2.3 The ADC in the 80C552

Figure 3 shows a block diagram of the implementation of the ADC in an 80C552 microcontroller.

The analog input signal V_{IN} is connected to the non-inverting input of the comparator via switch S_1 during the sampling interval. Internally the comparator consists of 3 serially connected sampled-data-comparator stages $A_1 \dots A_3$. The stages are capacitively coupled. The coupling capacitor of the first comparator that is connected to S_1 will also act as sample capacitor for V_{IN} .

Sampled-data-comparators are used to minimize the effect of offsets and temperature drive. During the sampling interval, the value of the offset voltage of the comparator stages are stored on the coupling capacitors. This voltage will have the opposite sign of the comparators stages' offset, so it will cancel this offset voltage. This process is called auto-zeroing, and will be explained in 2.3.2.

The non-inverting input of the comparator is connected to $\frac{1}{2}V_{REF}$ via switch S_2 . S_2 consists of 2 parallel switches. There is always 1 switch closed, so the voltage on this input is always $\frac{1}{2}V_{REF}$. Although S_2 looks superfluous from a functional point of view, it assures that, for instance, switching glitches of S_1 and S_2 appear on both inputs of the comparator and will cancel each other.

When the sampling is finished, the actual conversion will start. S_1 will connect the inverting input of the comparator with the output of the DAC. At this moment, the output of the DAC is connected to the center tap of the resistor network. The voltage on the inverting input of the comparator will be $\frac{1}{2}V_{REF}$ (V_{REF} is defined as $\frac{1}{2}(V_{REF+} - V_{REF-})$). During conversion, the output of the SAR will determine which tap of the ladder-network will be connected to the

inverting input. Using a ladder-network guarantees a monotonic characteristic of the DAC. This in turn will result in an ADC-characteristic without missing codes. The relative deviations of the resistor values result in a non-linear transfer characteristic.

The following three phases in the ADC conversion can be determined and will be described in more detail:

- Start phase
- Sampling phase
- Conversion phase.

Timing of these phases is shown in Figure 4.

2.3.1 Start Detection Phase

An ADC conversion can be started by software or by a hardware trigger on the STADC pin.

Software start

When an ADC start is initiated by software (set ADCS in ADCON register), the internal start signal will immediately be active at $S6P2$ (for state timing, see [Reference]). The value of ADCS can be read by software. However, there is a delay of 2 machine cycles between the internal start signal and the ability of reading a '1' from ADCS.

Hardware start

A hardware start of an ADC conversion is initiated by a rising edge on STADC. The 80C552 samples STADC every machine cycle during $S6P2$. When a valid edge is detected, the internal start signal will be active at $S1P2$ in the subsequent machine cycle. To ensure that the edge is detected, the high and low time should be at least 1 machine cycle each. When a valid edge is detected, 'ADCS' is set.

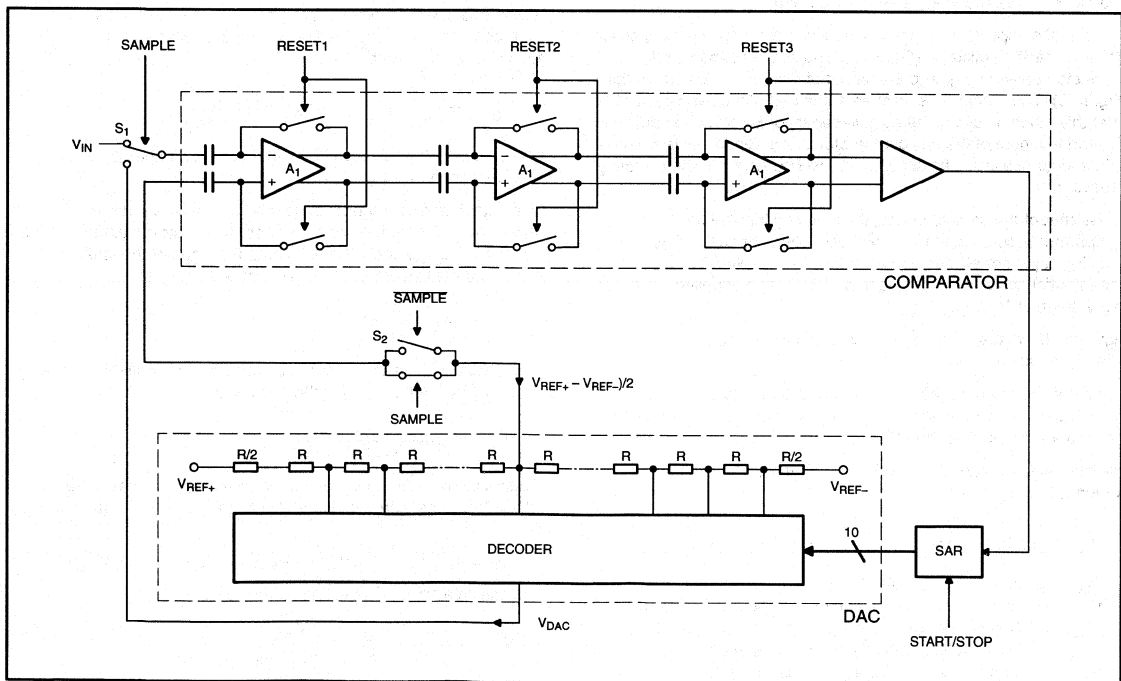


Figure 3.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

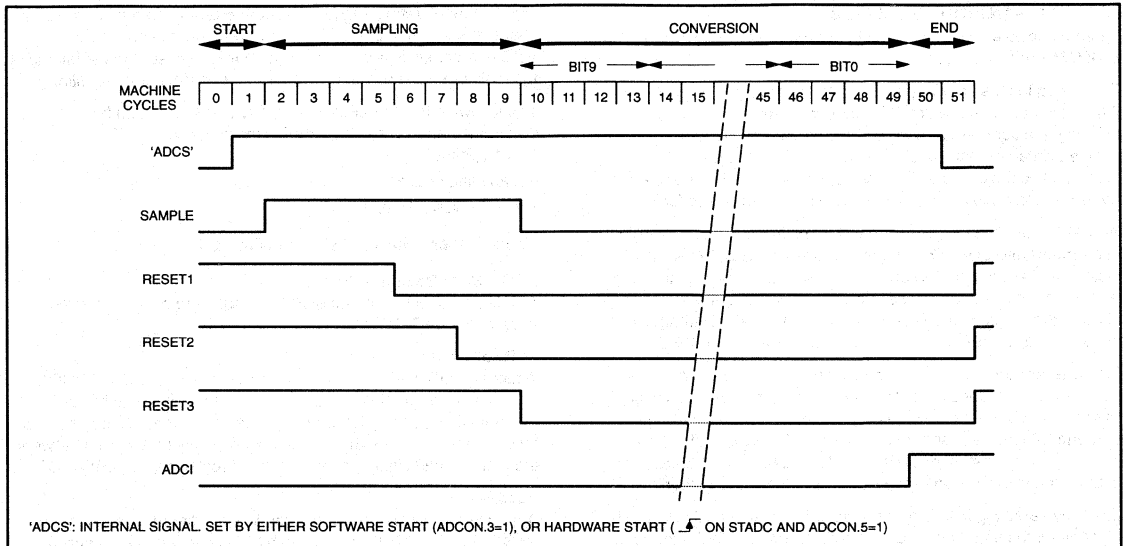


Figure 4.

2.3.2 Sample Phase

The 8 machine cycles following the start detection is the sample interval (Figure 4). In this time interval, the input signal is sampled and the 3 comparator stages are auto-zeroed.

The actual sampling of the analog input signal on the input capacitor starts at machine cycle '2' (Figure 4). The sample capacitor is connected between V_{IN} and the output of the first comparator stage (Figure 3). The sampling is finished at the end of machine cycle '5'. After this machine cycle, the sample capacitor is connected between V_{IN} and the input of the first comparator stage. Since this is a very high impedance input, no extra charge will be stored in the sampling capacitor.

At the start of the sample phase, the inverting input of the comparator is connected to V_{IN} via a coupling capacitor. This coupling capacitor also serves as sampling capacitor. The non-inverting input is connected to the DAC via a coupling capacitor to a voltage of $\frac{1}{2}V_{REF}$.

Figure 5 shows the sampling and auto-zeroing of the first comparator stage.

When the RESET1=1, switches will connect the outputs of the individual comparator stages to their inputs. The outputs will settle to the unity-gain output voltage V_{UG} .

The differential voltage (error voltage) on the inputs of the first comparator stage will be:

$$V_{IL} = V_{OL} = -V_{OS1} \times \frac{A_1}{A_1 + 1}$$

- V_{IN} = Input voltage of ADC
- V_{IL} = Differential input voltage of first comparator stage
- V_{OL} = Differential output voltage of first comparator stage
- V_{OS1} = Offset voltage of first comparator stage
- A_1 = Open loop gain of first comparator stage

The switches are opened when RESET1=0. The differential voltage on the comparator inputs is still V_{IL} because of the stored charge on the coupling capacitors.

The resulting offset voltage $V_{OS1,i}$ seen on the input of the comparator stage is obtained by adding this differential voltage V_{IL} to the input offset voltage V_{OS1} .

$$V_{OS1,i} = V_{OS1} + V_{IL} = V_{OS1} \times \left(\frac{A_1}{1 + A_1} \right)$$

The effective offset voltage at the input of the comparator stage is reduced with a factor $(1+A_1)$.

The auto-zeroing procedure described above will be repeated successively for the following 2 stages. After auto-zeroing the third comparator stage, the differential output voltage of the total comparator (all 3 comparators in series) will be:

$$V_{O,3} = A_3 \times \frac{V_{OS3}}{1 + A_3}$$

This voltage can be translated to an effective input offset voltage by dividing it by the total gain of the comparator:

$$V_{OS,i} = \frac{V_{OS3}}{A_1 \times A_2 \times (1 + A_3)}$$

If auto-zeroing was not used, and all comparator stages were DC-coupled, the differential output voltage of the comparator would be:

$$V_{O3} = A_1 \times A_2 \times A_3 \times V_{OS1} + A_2 \times A_3 \times V_{OS2} + A_3 \times V_{OS3}$$

The effective input offset voltage in this case is:

$$V_{OS,i} = V_{OS1} + \frac{V_{OS2}}{A_1} + \frac{V_{OS3}}{A_1 \times A_2}$$

As can be seen, the auto-zeroing reduces the effect of the individual comparator stages considerably.

Using the analog-to-digital converter of the 8XC52 microcontroller

EIE/AN93017

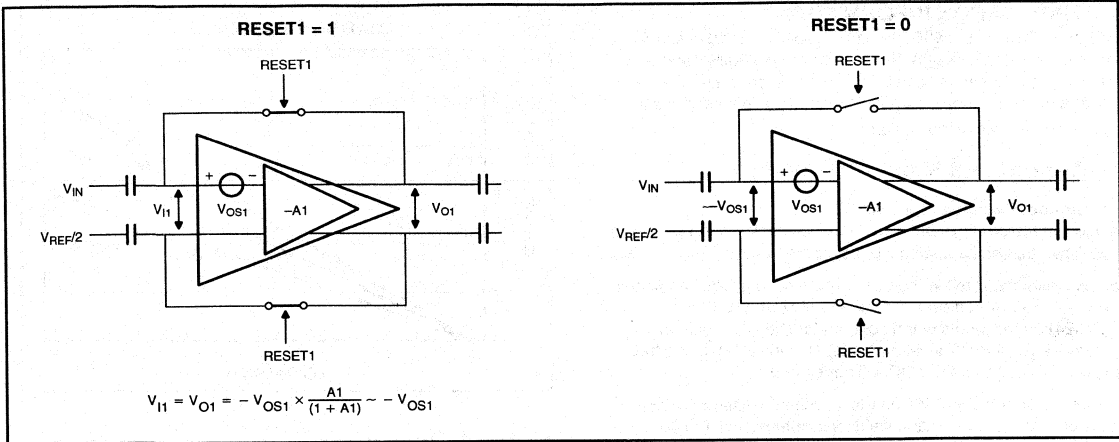


Figure 5.

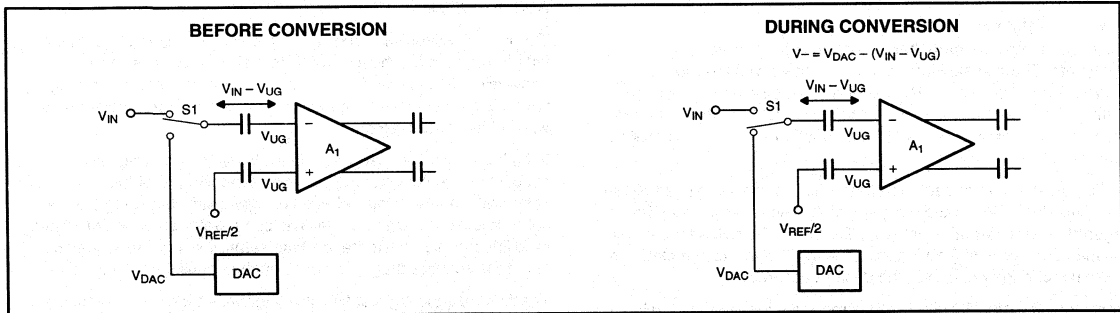


Figure 6.

2.3.3 Conversion Phase

Just before the sampling phase is finished, the following voltages are present over the coupling capacitors of the first comparator stage:

- Capacitor on inverting input: $V_{IN} - V_{UG}$
- Capacitor on non-inverting input: $\frac{1}{2}V_{REF} - V_{UG}$

For clarity, the offset voltages are neglected.

When the conversion phase is started, S1 (Figure 6) will connect the coupling capacitor of the inverting input to the output of the DAC. The effective voltage on the comparator input is the voltage applied to the coupling capacitor minus the voltage that was stored on the capacitor during the sampling phase.

The following voltages are present on the input of the first comparator stage:

- Inverting input: $V_{DAC} - V_{IN} + V_{UG}$
- Non-inverting input: V_{UG}

The comparator stage amplifies the differential voltage between its inputs. The output voltage of the first comparator stage will be:

$$V_{OL} = A_1 \times (V_{UG} - (V_{DAC} - V_{IN} + V_{UG})) = A_1 \times (V_{IN} - V_{DAC})$$

After amplification by the 3 comparator stages the input signal for the SAR is $|A_1 \times A_2 \times A_3 \times (V_{IN} - V_{DAC})|$. Depending on the sign of this signal, the SAR will set or clear the MSB. In the following cycles of the conversion, the other bits of the SAR will be updated. At the end of the conversion V_{DAC} will have a value of $V_{IN} \pm 0.5LSB$. The contents of the SAR that generates this V_{DAC} is the result of the AD-conversion.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

3.0 APPLICATION INFORMATION

Although the ADC in the 80C552 has a resolution of 10 bits, the user must be careful in the design of the application to really get this resolution. The constraints can be divided in 2 categories:

- Constraints on the analog input signal and the input signal source
- Layout constraints of the design.

3.1 Analog Input Signal Constraints

3.1.1 Range of Analog Input Signal

The value of the analog input signal must be between V_{REF+} and V_{REF-} . The span of the analog input signal is $V_{REF} = (V_{REF+} - V_{REF-})$.

There is a minimum limit to the span. This limit depends on the gain of the comparators. A differential voltage of 1LSB ($1LSB = V_{REF}/1024$) on the inputs of the comparator should be able to generate a logic '1' or '0' level on the input of the SAR. If not, the resolution of 10 bits for the ADC will not be met.

the comparator in the 80C552 needs a minimum differential input voltage of 0.3mV to generate a valid logic output level. For the 10-bits ADC in the 80C552, this means that V_{REF} should be at least $1024 \times 0.3mV = 0.31V$ to get 10-bit resolution. The absolute values of V_{REF+} and V_{REF-} that determine this span may not exceed AV_{SS} and AV_{DD} .

3.1.2 Slew Rate of Analog Input Signal

A distinction must be made between 2 different slew-rate constraints. The first slew-rate constraint deals with the required accuracy during sampling. The second constraint to prevent wrong readings deals with a limitation on the slew rate that may otherwise lead to a conversion result that has no relation at all with the analog input signal.

1: To obtain a stable reading from the ADC, the analog input signal should be stable during the sampling time. The sampling may be triggered by an external event (via ADEX pin). From this trigger point until machine cycle '5' (see Figure), the input signal is sampled and should not change more than the desired accuracy.

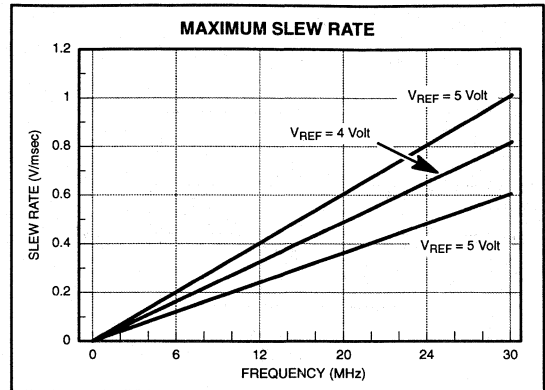
Example: If a stability of 0.5LSB is required, then the analog input signal should not change more than 0.5LSB in 6 machine cycles. In that situation the maximum slew-rate of the analog input signal is:

$$\frac{dV}{dt} = \frac{0.5LSB}{6T}$$

where T is the machine cycle time.

The following graph gives the maximum slew-rate as function of the operating frequency for various values of V_{REF} and a required stability of $1/2LSB$ for a 10-bit conversion.

When the slew-rate of the input signal is more than the maximum slew-rate as determined above, the read-out stability will decrease. The conversion result will be the digital value of the input signal somewhere between machine cycle '0' and machine cycle '5'. Consecutive conversions of a signal that consists of a DC-value with an AC-component that has high slew-rates as mentioned above, and that has the same amplitude between machine cycle '0' and machine cycle '1', may give different read-outs. However, the accuracy of the sampled signal will not be affected.



2: If the slew-rate exceeds a certain value, the accuracy of the conversion will decrease rapidly. The result of the conversion will not have any result anymore with the analog input signal. Tests have shown that the most probable conversion result is 0x3ff (result bits ADS.0 . . ADC.9 are '1').

This error situation will occur when the slew rate is too high in the time frame from machine cycle '2' to machine cycle '9' of the conversion. In this time frame, the comparators are auto-zeroing their offsets. For proper auto-zeroing, the comparator stages must work in their linear region.

If the input signal is changing rapidly, the voltage change may couple through the coupling capacitors to the input of the comparator stage. If this voltage change is sufficiently high, it may saturate the comparator stage. The comparator stage is not working in its linear region anymore, and the saturation voltage (equal to about the supply voltage) is stored on the coupling capacitors.

The ADC has the highest sensitivity to these high slew-rate signals in the time frame from machine cycle '8' to machine cycle '9'. In this time frame the RESET switches of comparator stage 1 and 2 are open; the RESET switch of comparator stage 3 is closed for auto-zeroing. An analog input signal with sufficient slew rate may couple through to comparator stage 3 via the coupling capacitors of stage 1 and 2. The high sensitivity comes from the fact that the signal is amplified by comparator stages 1 and 2 before it reaches the input of comparator stage 3.

When the saturation voltage is stored on the coupling capacitors, the following comparator stage is not useful anymore to determine the sign of $(V_{IN} - V_{DAC})$. Suppose the coupling capacitors on the input of the second comparator stage are charged to the saturation voltage V_{SAT} . The differential output voltage of the first comparator stage will be $A_1 \times (V_{IN} - V_{DAC})$. This signal is fed to the input of the second comparator stage whose output signal will be $A_2 \times [A_1 \times (V_{IN} - V_{DAC}) \pm V_{SAT}]$. Since the differential output voltage of the comparator stages can never be higher than $\pm V_{SAT}$, the output of this comparator stage will stay at its saturation level, independent of the value of $(V_{IN} - V_{DAC})$.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

The only way to avoid the error mentioned above is to limit the slew rate during the sampling interval (machine cycle 2 to 9). For the ADC in the 80C552 the slew rate of the analog input signal must be lower than 10V/ms.

From the discussion above, it becomes evident that it is essential that the slew-rate of the input signal is limited to 10V/ms. Although 'clean' DC signals may be applied to the ADC, noise spikes or crosstalk from neighboring signals may still result in signal components with a slew-rate >10V/ms on the DC signal during the sampling interval.

The following measures can be taken to reduce the slew rate on analog input signals:

- Supply the analog signal from a source with low output impedance. This will reduce the sensitivity to cross-talk.
- Keep the analog input signal lines away from digital signal lines. Analog signals may be screened from digital signal lines with a grounded guard ring on the PCB.
- Do not mix analog and digital signals on P5 pins.
- Connect an RC filter to the analog inputs. The time constant should be $\geq 500\mu\text{s}$.

For a 5V_{PP} input sine-signal, the slew rate constraint of 10V/ms results in a maximum input frequency of 637Hz. When we use the Nyquist criterion ($f_{\text{SAMPLE}} \geq 2 \cdot f_{\text{SIGNAL}}$), the maximum input signal frequency is 1kHz when the 80C552 runs on 1.2MHz (1 conversion every 500 μs).

This shows that the maximum input signal frequency for the ADC of the 80C552 is determined by the **slew-rate**. So, increasing the XTAL frequency on which the 80C552 is operating, does not automatically imply that the maximum input signal frequency also scales to a higher value. This scaling is only allowed for signals with a slew rate <10V/ms.

3.1.3 Analog Signal Drive

The output resistance of the analog signal source should be small enough not to add a significant error to the conversion result. The output impedance has two effects on the accuracy of the conversion, that is, the voltage drop over the source resistance and the time constant to charge the sampling capacitor.

1: The voltage drop over the output impedance due to the input current of the ADC. In the 80C552, the input current is a leakage current. this leakage current is specified as less than 1 μA . Practically, however, the leakage current will be less than 100nA.

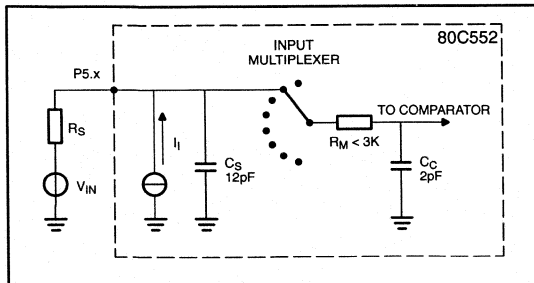


Figure 7.

Figure 7 shows the input circuit. The leakage current comes mainly from circuitry directly connected to the P5.x pin. Compared with this leakage current, the input current of the comparator can be neglected. C_S represents the contribution of the stray capacitances; C_C represents the sampling capacitor. Before the sampling capacitor, there is the series resistance R_M of the analog multiplexer.

The output resistance R_S of the input signal source will cause a voltage drop due to the input leakage current. The voltage that will be converted is the voltage on the sampling capacitor. This voltage is the input voltage V_{IN} minus the voltage drop over R_S . This voltage drop will give an error contribution in the conversion result of V_{IN} .

When an accuracy of $1/2\text{LSB}$ is required, the maximum source resistance R_S is:

$$R_S < \frac{0.5\text{LSB}}{I_I}$$

Example: If $V_{REF} = 5.12\text{V}$ and $I_I = 1\mu\text{A}$, the source resistor should be less than 2.5k Ω .

When this constraint on output resistance of the signal source cannot be met, the analog signal should be buffered with a buffer of sufficiently low output resistance. This buffer should be placed as close as possible to the analog source. the longer leads from buffer to the ADC input will be less sensitive to cross-talk (low impedance source resistance) than long leads from signal source to buffer (high impedance source resistance). Filtering may be included in the buffer stage to limit the slew-rate of the signal to <10V/ms.

2: The ability to charge the sampling capacitor within the sampling interval. Figure 7 also shows the dimensions of the capacitances as seen from the analog input. The capacitances consist of stray-capacitances and the actual sample capacitance. These capacitances must be charged within 4 machine cycles (machine cycle '2' until '5'), which will put a constraint on the maximum source resistance.

Example: An input signal with a slope of 10V/ms applied to the ADC input. For simplicity, assume that the capacitance that must be charged via R_S is 14pF and the 80C552 is running on 30MHz. At this frequency, the available charging time for the capacitor is 1.6 μs . With the given slew-rate and the charging time of the capacitor, the analog voltage V_{IN} has changed 16mV. The response of an RC-network on an input ramp signal is:

$$A \left[t - RC \left(1 - e^{-\frac{t}{RC}} \right) \right]$$

A is slew-rate of input signal; RC is time constant of input RC-network.

The term:

$$\text{ARC} \left(1 - e^{-\frac{t}{RC}} \right)$$

represents the deviation of the capacitor voltage from V_{IN} .

If this deviation must be less than $1/2\text{LSB}$ (is 2.5mV at $V_{REF} = 5.12\text{V}$) after 1.6 μs , then the RC-time must be less than 0.25 μs . Given that $R_S = 2.5\text{k}\Omega$ and $C = 124\text{pF}$, the RC time of the input circuit of the 8XC552 is 35ns. Hence, there will be no significant error contribution because of the charging time of the input capacitance.

The two constraints on R_S mentioned above show that the effect of the input leakage current determines the maximum value for R_S .

Conclusion: R_S should not exceed 2.5k Ω .

Using the analog-to-digital converter of the 8XC52 microcontroller

EIE/AN93017

3.2 Layout Considerations

Although this application note handles subjects related to the ADC, the following layout considerations are also valid for applications that do not use the ADC. For more general information on PCB layout design, see Reference 5.

3.2.1 Decoupling

The analog and digital circuit parts in the 80C552 have their own set of power supply pins. Mutual inductance between on-chip AV_{DD} and AV_{SS} signal lines will cause the analog AC current I_A to flow in the AV_{DD} and out of the AV_{SS} pin. The same is true for the digital AC current I_D in the V_{DD} and V_{SS} pins (Figure 8).

Because this mutual inductance is harder to realize off-chip, a low-impedance signal path must be created for both I_A and I_D . This

is realized with decoupling capacitors between AV_{DD} and AV_{SS} for the analog signal part; a decoupling capacitor between V_{DD} and V_{SS} will decouple the digital signal part.

To ensure a low impedance ground path, the use of a ground plane is recommended. The decoupling capacitors (for example, 100nF ceramic capacitors) must be placed as close as possible to the AV_{DD} and V_{DD} to minimize the loop area of the supply currents. Series inductors in the power supply lines may be used to improve decoupling (for example, 1..5μH). Using this decoupling scheme, both analog and digital supply connections can be connected together to a single (stable) +5V.

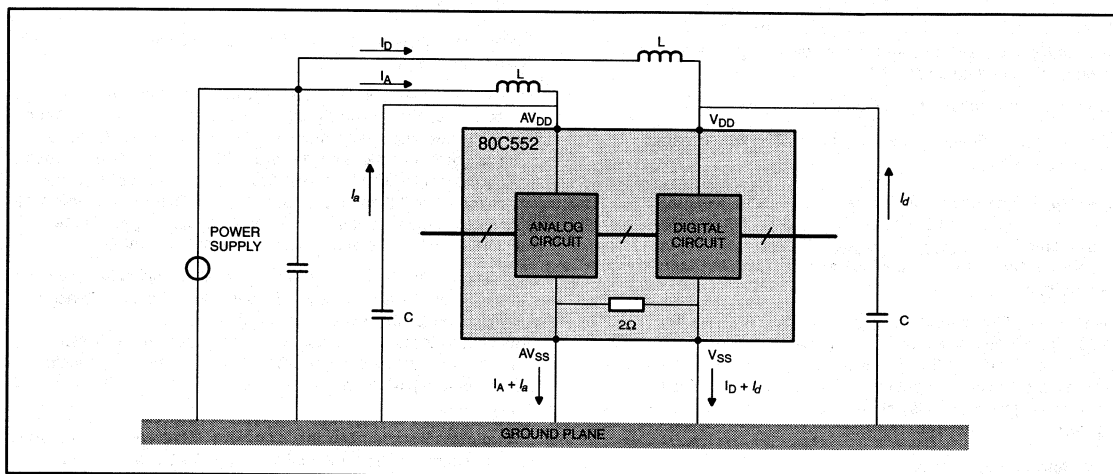


Figure 8.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

3.2.2 Grounding

When using both analog and digital circuits on the same PCB, it is common practice to isolate the analog power supply and ground as much as possible from the digital power supply and ground. This will reduce crosstalk via common ground impedances. Common impedances may result in noise in the (sensitive) analog circuitry caused by crosstalk of noise that originates from the digital circuitry.

At some point however, both analog and digital grounds must be tied together. When this takes place far from the microcontroller, it will increase impedance between the analog and digital ground connections. This can create a considerable differential voltage between the analog and digital ground lines. Analog and digital circuitry inside the microcontroller will operate at different ground levels and improper functioning may be the result. A second effect of large ground impedance is that the crosstalk between the analog and digital circuits does not take place via the common ground impedance anymore, but via internal parasitic capacitances and substrate contacts.

To prevent differences in ground levels, the analog and digital ground inside the 80C552 are connected together via an impedance of $\pm 2\Omega$ (Figure 8). This will keep both grounds at the same DC level. This impedance does not mean that now a common ground pin for analog and digital ground currents can be used. Impedance of the common bondwire will cause ground bounce, and thus crosstalk between digital and analog circuits.

When the supply lines are properly decoupled, mutual inductance between the on-chip supply traces will assure that the AC-part of the supply current that flows inside AV_{DD} and V_{DD} will leave via the AV_{SS} and V_{SS} pin even though AV_{SS} and V_{SS} are connected together via a 2Ω impedance. The best place on the PCB to connect AV_{SS} and V_{SS} to each other is directly outside the microcontroller. This is also the best place for the ground connection of the decoupling capacitors. If the ground connection is to a ground plane, a ground plane on the component side is preferred.

Two separate grounds are needed if the application has more analog ICs on the PCB, apart from the 80C552. In that case, the AV_{SS} and V_{SS} of the 80C552 should be connected to the analog ground. The digital ground may be the return path for, for instance, line drivers. This will result in a ground that is less 'clean' than the analog ground. For the digital circuits this less 'clean' ground is less of a problem because they always have a certain noise margin. An alternative is to create a 'star-point'. This is a connection between AV_{SS} and V_{SS} at the ADC area of the 80C552. Care should be taken to avoid ground loops in the other circuit parts up to the power supply.

If the 80C552 application also uses external digital circuits, noise margin may be lost due to possible different ground levels. This can be reduced by the connecting two anti-parallel diodes close to the ground pins of the 80C552 and the connected digital circuits (Figure 9).

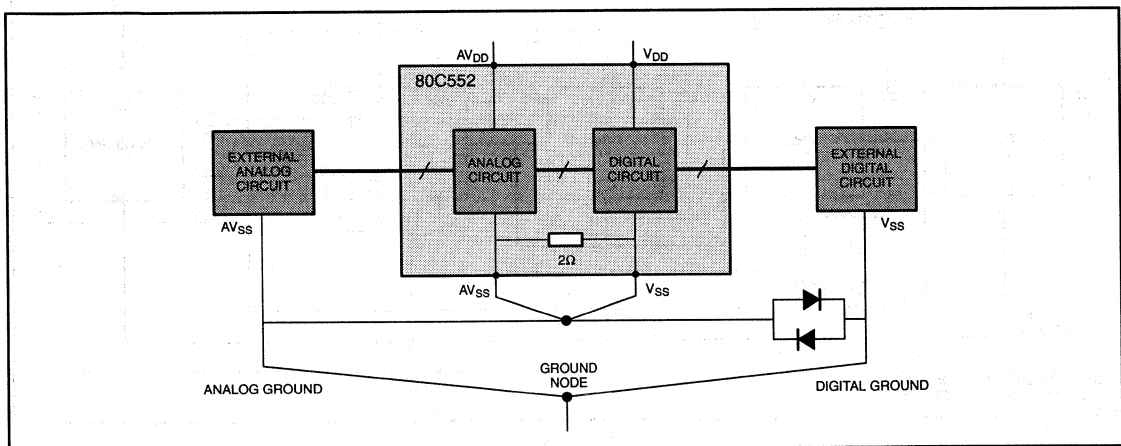


Figure 9.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

3.2.3 Placement of External Components

External circuits connected to the microcontroller should be placed as close as possible to the microcontroller. This will reduce signal loops and common impedances on which high frequent signals may occur. These signals may come from an external source or may be generated by the microcontroller itself. When the disturbing signals are coming from an external source, they may cause improper functioning of the microcontroller. If originating from the microcontroller, they will cause unwanted radiation.

When a common ground plane cannot be implemented, the microcontroller and circuits directly connected to the 80C552 must have a local groundplane (Figure 10). This local ground should have a connection to the main ground of the application at some point.

If, for layout reasons, it is not possible to place the external circuitry close to the microcontroller, an RC-filter may be placed between the microcontroller and the external circuit (Figure 10). When microcontroller lines are connected to Off-PCB circuits, it is also advised to connect an RC network to the I/O line. The reason is sensitivity of the microcontroller to short (4 .. 5ns) high-voltage (30 .. 40V) spikes. Although these pulses will not damage the

microcontroller, they may be responsible for incorrect functioning. With microcontrollers becoming faster, the on-chip I/O drivers will have increased bandwidths, hence become more sensitive for short spikes.

The resistor of the filter should be connected as close as possible to the output of the driving device. The capacitor of the filter should be connected as close as possible to the input of the receiving device. Also the ground connection of the capacitor must be connected as close as possible to the ground of the receiving device.

Values of the resistor of the filter depend on the drive capability of the outputs and the input impedances/levels of the inputs connected to the filter. The series resistor will reduce some noise margin. Typical capacitor values are 470pF. When connected to 80C552 inputs, the resistor value may be 1k; for 80C552 outputs 100Ω may be used.

NO (filter)-capacitors should be connected directly to outputs! This will cause (dis)charge currents to flow in the supply and ground lines. This can cause severe noise problems in the application.

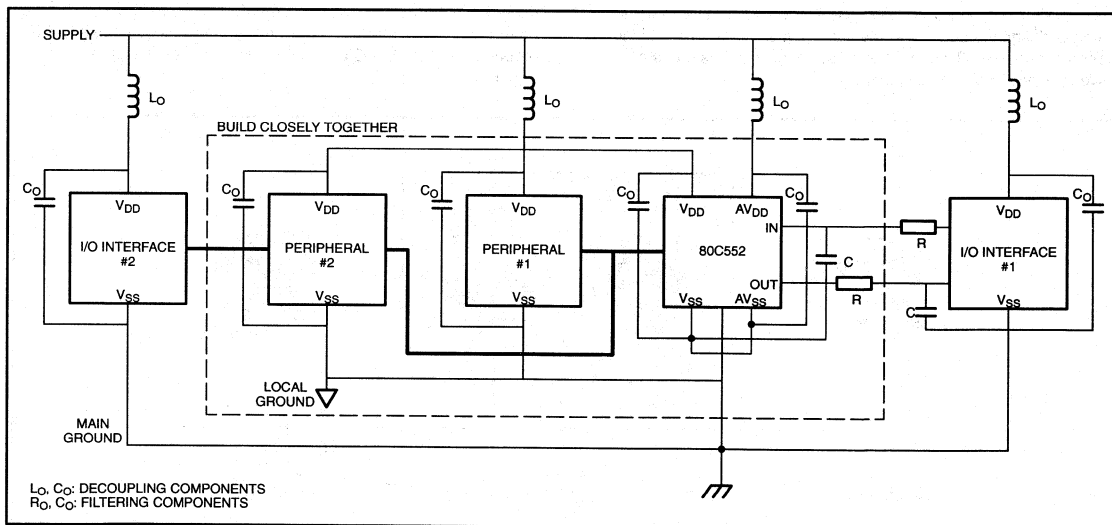


Figure 10.

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

4.0 PROGRAM EXAMPLES

Two program examples are given that show how to operate the ADC with software. The sources are written in 'C'. The program 'ADC_pol.c' works on polling basis; the program 'ADC_int.c' works on interrupts.

The programs will start scanning all ADC inputs when a rising edge appears on the STADC pin. This can be realized with a resistor from ground to STADC and a switch from STADC to V_{DD} . If STADC is connected to P4.0, a conversion of all channels will start every 1.14ms. Timer T2 controls the timing of P4.0. When all analog input signals on P5 are converted, the results will be sent to the UART and be made visible on a terminal. The communication part of the program is given in the file 'output.c'. After compiling and assembling 'output.c', it should be linked to 'ADC_pol' or 'ADC_int'.

The required terminal settings are:

- 8 data bits
- no parity
- 1 STOP-bit
- 19200 baud.

The 80C552 must run on 11.0592MHz.

The following points are important when writing code for the ADC:

- ADCS and ADCI must be cleared, before programming AADC0..AADR2.

- Channel selection bits AADR0..AADR2 must be programmed before setting ADCS (software start) or ADEX (enabling hardware start)
- When working in polling mode, use ADCI to test if the conversion is finished.
 - Do not test 'ADCS=0' for this purpose.
 - When an ADC conversion is initiated by software, there is a delay of 2 machine cycles between the moment of writing a '1' to ADCS and reading a '1' from the ADCS-bit in ADCON.
 - When an ADC conversion must be initiated by a rising edge on ADEX, it is not known when ADCS becomes '1'. This depends on the external signal that starts the conversion.

The following development tools were used on a PC (DOS 5.0):

TOOL	TYPE	PHILIPS NUMBER
C-compiler	BSO/Tasking V2.1	OM4136
Assembler	BSO/Tasking V3.3	OM4142
Emulator	SDS+80C552 probe	OM4120S +OM1092 +OM1095
Debugger	BSO/Tasking XRAY51 V1.4d	OM4129

ADC_pol.c

```

/*****
*
***
*0*  MODULE           : adc_pol.c
***
*0*  FILENAME        : adc_pol.c
***
*0*  APPLICAITON     : Demo code for ADC of 8xc552 polling
*0*                   mode
***
*0*  PROGRAMMER      : T. v. Daele
***
*0*  DESCRIPTION     : After rising edge on STADC-pin, all
*0*                   ADC channels are scanned.
*0*                   Rising edges are available on P4.7 at a
*0*                   repetition rate of 1.14ms. This timing is
*0*                   controlled by T2.
*0*                   Results are sent to UART.
***
*****/

#define ADEX      0x20
#define ADCI      0x10
#define ADCS      0x08

void write_UART (unsigned int *, unsigned int);

```

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

```

void main(void)
{
  unsigned int conversion,result_ADC[8];
  unsigned char ADC_Channel;

  SOCON=0x40;          /* 8 bits, no parity, 1 STOPbit */
  TH1+TL1=0xfd;       /* 19200 Baud @11.0592MHz */
  PCON=0x80;
  TMOD=0x20;
  TR1=1;

  TM2CON=0x0d;        /* Source T2: osc/96 */
  RTE=0x80;           /* Overflow rate: 0.569ms
                       P4.7 toggles every 0.569ms
                       ADC conversion on rising edge STADC
                       P4.7/STADC: 1.14ms conversion rate */

  /*
  conversion=0;

  while(1)
  {
    for (ADC_Channel=0;ADC_Channel<8;ADC_Channel++)
    {
      ADCON=0;          /* Make sure ADCI and ADCS are cleared
    */
      ADCON=ADC_Channel; /* before ADC channel is selected */

      if (ADC_Channel==0)
        ADCON=ADEX;     /* ADC0: External start */
      else
        ADCON+ADCON|ADCS; /* ADC1..ADC7: Software start */
      while((ADCON&ADCI)==0); /* Wait till conversion finished
                               by checking ADCI */
      result_ADC[ADC_Channel]=5*((256*ADCH+(ADCON&0xc0)>>6);
                               /* Calculate 10-bits binary result
                               relative to 5.12V ref */
    }
    write_UART(&result_ADC,conversion++); /*Output results to UART */
    if (conversion==10000)
      conversion=0;
  }
}

```

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

ADC_int.c

```

/*****
*
***
*0*   MODULE           : adc_int.c
***
*0*   FILENAME        : adc_int.c
***
*0*   APPLICATION     : Demo code for ADC of 8xC552 in interrupt
*0*                   mode
***
*0*   PROGRAMMER      : T. v. Daele
***
*0*   DESCRIPTION     : After rising edge on STADC-pin, all
*0*                   ADC channels are scanned.
*0*                   Rising edges are available on P4.7 at a
*0*                   repetition rate of 1.14ms. This timing is
*0*                   controlled by T2.
*0*                   Results are sent to UART.
***
*****/
/

#define ADEX      0x20
#define ADCI     0x10
#define ADCS     0x08
#define ADCIn    0xref
#define FALSE    0
#define TRUE     1

void write_UART(unsigned int *, unsigned int);
bit conversion_finished;

void main(void)
{
    unsigned int conversion,result_ADC[8];
    unsigned char ADC_channel;

    SOCON=0x40;                /* 8 bits, no parity, 1 STOPbit */
    TH1=TL1=0xfd;             /* 19200 Baud @11.0592MHz */
    PCON=0x80;
    TMOD=0x20;
    TR1=1;

    TM2CON=0x0d;              /* Source T2; osc/96 */
    RTE=0x80;                 /* Overflow rate: 0.569ms
                               P4.7 toggles every 0.569ms
                               ADC conversion on rising edge STQADC
                               P4.7/STADC: 1.14ms conversion rate
    */

    EAD=1;                    /* Enable ADC interrupt */
    EA=1;

    conversion_finished=FALSE;
    ADC_channel=conversion=0;

    ADCON=0;                  /* First conversion; external start */

    ADCON=ADEX;
    while(1)
    {
        if (conversion_finished==FALSE)
        {
            /* User code executed while conversion is in progress */
        }
    }
}

```

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

```
else
{
    result_ADC[ADC_channel]=5*((256*ADCH+(ADCON&0xc0))>>6); /* Store
result */
    if (ADC_channel!=7)
    {
        /* Prepare conversion of next channel */
        ADCON=++ADC_channel;
        ADCON+ADCON|ADCS;
    }
    else
    {
        /* ADC0..ADC7 is converted. Send results to UART */
        write_UART(&result_ADC,conversion++);
        if (conversion==10000)
            conversion=0;
        ADC_channel=0;
        ADCON=0; /* Prepare next scan */
        ADCON=ADEX;
    }
    conversion_finished=FALSE;
}
}

interrupt 10 using 1 void ADC(void)
{
    ADCON=ADCON&ADCIn; /* Clear ADCI flag */
    conversion_finished=TRUE;
}
```

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

output.c

```

/*****
***
*0*   MODULE           : output.c
***
*0*   FILENAME        : output.c
***
*0*   APPLICATION     : Example program for 80C552 ADC
***
*0*   PROGRAMMER      : T. van Daele
***
*0*   DESCRIPTION     : The results of the conversion are written
*0*                   to the 80C552 UART.
***
*0*   FUNCTIONS       : <name>           <description>
*0*                   write_UART       entry point
*0*                   send_byte        trx byte
*0*                   decode           trx binary nibble
*0*                   send_bin_byte     trx binary byte
*0*                   send_dec_int      trx decimal integer
*0*                   send_string       trx aSCII string
***
*****/

rom char string_0[] = "Conversion #";
rom char string_1[] = ": (Ref is 5.12V)";
rom char string_2[] = "ADC_Channel # ";
rom char string_3[] = "mV";
rom char string_4[] = ": ";
rom char new_line[] = "\r\n";

/*****
***
*1*   FUNCTION        : send_byte
***
*2*   SYNOPSIS       : send_byte(src_byte)
***
*3*   ARGUMENTS     : type           name
*3*                   char           src_byte
***
*4*   RETURNS       : nothing
***
*5*   MODIFIES      : nothing
***
*6*   DESCRIPTION   : Send byte to terminal via UART
*6*                   Wait till transmission is finished
***
*7*   HISTORY       : data           who           description
*7*                   05-02-93      tvd           initial
***
*****/

void send_byte(char src_byte)
{
  S0BUF = src_byte; /* Byte to transmit */
  while (TI == 0); /* Wait till byte is transmitted */
  TI = 0;          /* Clear transmit flag */
}

```

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

```

/*****
***
*1*   FUNCTION       : decode
***
*2*   SYNOPSIS      : decode(char src_nibble)
***
*3*   ARGUMENTS     : type           name
*3*                   char           src_nibble
***
*4*   RETURNS       : nothing
***
*5*   MODIFIES      : nothing
***
*6*   DESCRIPTION   : Decode least significant nibble to
*6*                   ASCII and transmit
***
*7*   HISTORY       : data           who           description
*7*                   05-02-93      tvd           initial
***
*****/
void decode(char src_nibble)
{
    if ( src_nibble < 0x0a)
        send_byte(src_nibble + 0x30);
    else
        send_byte(src_nibble + 0x41 - 0x0a);
}

/*****
***
*1*   FUNCTION       : send_bin_byte
***
*2*   SYNOPSIS      : send_bin_byte(char src_byte)
***
*3*   ARGUMENTS     : type           name
*3*                   char           src_byte
***
*4*   RETURNS       : nothing
***
*5*   MODIFIES      : nothing
***
*6*   DESCRIPTION   : Split a binary byte in nibbles, decode
*6*                   to ASCII and transmit
***
*7*   HISTORY       : data           who           description
*7*                   05-02-93      tvd           initial
***
*****/
void send_bin_byte(char src_byte)
{
    decode((src_byte>>4) & 0x0f);    /* Get ms_nibble */
    decode(src_byte & 0x0f);        /* Get ls_nibble */
}

```


Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

```

/*****
***
*1*   FUNCTION           : send_dec_int
***
*2*   SYNOPSIS          : send_dec_int(unsigned int src_wrd)
***
*3*   ARGUMENTS        : type           name
*3*                       unsigned      src_wrd
***
*4*   RETURNS          : nothing
***
*5*   MODIFIES         : nothing
***
*6*   DESCRIPTION      : Decode binary integer to decimal and
*6*                       transmit
***
*7*   HISTORY          : data           who           description
*7*                       07-07-93      tvd           initial
***
*****/
void send_dec_int(unsigned int src_wrd)
{
    unsigned char a,b,c,d,e;

    a=src_wrd/1000;           /* a='thousands' */
    b=((src_wrd%1000)/100);  /* b='hundreds' */
    c=((src_wrd%100)/10);    /* c='tens' */
    d=src_wrd%10;           /* d='units' */
    e=16*c+d;               /* Print value for tens and units */

    /* Print integer without leading zero's */
    if (a==0)
    {
        send_byte(0x20);
        if (b==0)
        {
            send_byte(0x20);
            if (c==0)
            {
                send_byte(0x20);
                decode(d);
            }
            else
                send_bin_byte(e);
        }
        else
        {
            decode(b);
            send_bin_byte(e);
        }
    }
    else
    {
        send_bin_byte((16*a)+b);
        send_bin_byte(e);
    }
}

```

Using the analog-to-digital converter of the 8XC552 microcontroller

EIE/AN93017

```

/*****
***
*1*   FUNCTION           : send_string
***
*2*   SYNOPSIS          : send_string(rom char *str_ptr)
***
*3*   ARGUMENTS         : type           name
*3*                       rom char *       str_ptr
***
*4*   RETURNS           : nothing
***
*5*   MODIFIES          : nothing
***
*6*   DESCRIPTION       : Send a string of characters from ROM to
*6*                       terminal.
***
*7*   HISTORY           : data           who           description
*7*                       05-02-93      tvd           initial
***
*****/
void send_string(rom char *str_ptr)
{
    while (*str_ptr != 0)
        send_byte(*(str_ptr++));    /* Send byte */
}

/*****
***
*1*   FUNCTION           : write_UART
***
*2*   SYNOPSIS          : write_UART(unsigned int *ADC_result,
***                       unsigned int conversion_cnt)
***
*3*   ARGUMENTS         : type           name
*3*                       unsigned int *   src_ptr
*3*                       unsigned int     msg_ptr
***
*4*   RETURNS           : nothing
***
*5*   MODIFIES          : nothing
***
*6*   DESCRIPTION       : Decode results to correct format and send
*6*                       to UART
***
*7*   HISTORY           : data           who           description
*7*                       30-06-93      tvd           initial
***
*****/
void write_UART(unsigned int *result_ptr, unsigned int conversion_cnt)
{
    unsigned char cnt;

    send_string(new_line);
    send_string(new_line);
    send_string(string_0);           /* Send message number */
    send_dec_int(conversion_cnt);
    send_string(string_1);
    for (cnt=0;cnt<8;cnt++)
    {
        send_string(new_line);
        send_string(string_2);
        decode(cnt);                 /* Send channel number */
        send_string(string_4);
        send_dec_int(*(result_ptr++)); /* Send result to UART */
        send_string(string_3);
    }
}

```

Chips push CAN bus into embedded world

Electronic Engineering TIMES

Monday
August 24, 1992
Issue 707

A CMP Publication

THE INDUSTRY NEWSPAPER FOR ENGINEERS AND TECHNICAL MANAGEMENT

Chips push CAN bus into embedded world

This week:
Special Report
Surface-mount technology gets bigger as packages get smaller

Sunnyvale, Calif. — The Controller Area Network (CAN) serial bus is being thrust into a key role in embedded systems, boosted by a series of recent silicon announcements. Delta-t GmbH, Intel Corp., Motorola Semiconductor and Philips/Signetics Co. have all prepared new chips that should slash the cost of a CAN bus connection, increase bus functionality and accelerate the bus's migration from a little-known automotive standard to a widely used industrial interconnect scheme.

Developed by Intel with Robert Bosch GmbH in the mid-1980s to solve cabling and control problems in vehicles, the CAN bus combines an inexpensive, one-wire or two-wire medium with multimeter, error-correcting protocol and very high resistance to electromagnetic interference. Unlike most multi-master buses, however, CAN also guarantees a maximum latency, often in the neighborhood of 1 ms, for high-priority messages while making room for lower-priority traffic as well.

Thus, although the bus was originally conceived as a way to reduce the literally miles of wire in a modern automobile, it is in many ways ideally suited for a wide range of industrial control applications as well.

General-purpose bus

CAN has since emerged as a general-purpose sensor/actuator bus system for distributed real-time control applications that extend beyond the automotive realm. "CAN was spotted by the industry as a very promising field-bus technology in the area of industrial automation," said Tom Suters, systems architect at Philips Medical Systems, in Da Best, the Netherlands.

"We are beginning to see CAN as a widely used standard in trucks and farm equipment, industrial automation and even some medical equipment," agreed Signetics marketing manager Mike Thomson.

But CAN's opponents have often criticized its high implementation cost as well as the lack of controller chips and supporting tools. Proponents have argued that volume production for the automotive market would inevitably bring down the silicon prices, but that has yet to happen. Now vendors are pointing to a new generation of low-cost or highly integrated controllers as the solution.

Motorola is sampling a controller, dubbed the 68HC705X4, that will be supported by an evaluation board and an emulator. Other implementations based on the 68HC11 and the 68HC16/683XX are in the pipeline.

Intel Corp., working in conjunction with design partner Bosch, has sampled a next-generation CAN chip, the 82527, that offers support for two sizes of message identifiers—11 bit and 29 bit—as specified in CAN Spec. 2.0, released in September.

Craig Szydlowski, product marketing engineer for CAN at Intel, said the original CAN 1.2 standard allowed for only 11-bit message identifiers. Supporters of J1850—the competing automotive bus favored by the Society of Automotive Engineers—fought for an optional larger message field so 1850-style 27-bit commands could be easily mapped into CAN protocols.

"This gives CAN the ability to broadcast commands using what had been essentially an address field," explained Signetics' Thomson. "For example, a CPU might send out a command telling all of the

lamps in a vehicle to test themselves for conductivity and report any burn-outs."

The new format does not imply that CAN and J1850 physical buses can be easily interfaced, Szydlowski emphasized, only that the message structure would be similar.

Just the next step

The Intel chip "looks like a smart RAM," Szydlowski said, with RAM space on-chip to store 14 8-byte receive/transmit message objects (with a fifteenth area for received message objects). The message objects are stored at fixed RAM locations. One on-chip filter is dedicated to the receive message object, with a global acceptance filter mask used for the other 14 message objects.

Intel's new chip is merely the next step in integration, to be followed by efforts to embed 527 functionality into a standard Intel 16-bit controller architecture.

Philips/Signetics has added to its product line as well but from a different direction. The company has integrated a CAN bus controller into a heavily configured 8051-type MCU to come out with the 8XC592. The chip combines large ROM and RAM space with five 8-bit I/O ports, a 10-bit A/D, two 8-bit-resolution PWM outputs, and the usual counter/timers and UART.

The announcement puts Philips and Intel in a confrontation over processing power. Intel will pursue a strategy of integrating CAN controllers into 16-bit MCUs and driving up node performance. The Dutch giant is going in the opposite direction, starting with a high-end commodity 8-bit part and heading downward in cost.

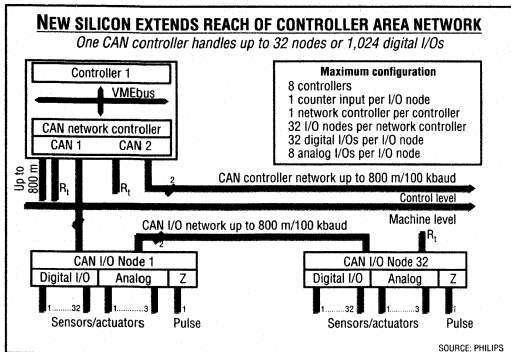
"Customers will want a part for a real simple node, without even a CPU on it," claimed Thomson. "You don't hang an 8051 on a light bulb. We are working on a controller solution for under \$1." Philips has also announced a high-speed CAN transceiver that can be hooked directly to its controllers.

The IX controller from Delta-t GmbH, meanwhile, should prove useful to manufacturers of sub-systems for multiprotocol environments when it debuts sometime next year. The protocol used for each application will be able to be programmed into a flash-memory section of the controller.

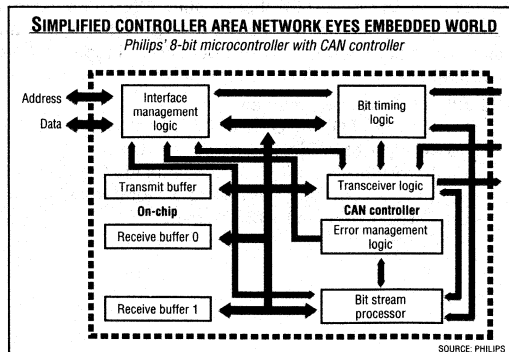
The design activity promises to both lower the cost of a simple CAN-bus node and increase the computing power that can be integrated into an expensive node. Both moves should expand the market deeper into industrial control and instrumentation applications, although CAN partisans see little hope of displacing J1850 from the big three domestic markets.

CAN is a multibus topology network that connects several stations. An International Standards Organization draft from 1990 specifies the first two layers of the Open Systems Interconnect (OSI) model: the physical layer and the data-link layer. Philips Medical Systems developed an application layer, using CAN to control X-ray diagnostic systems, control real-time image acquisition and connect user-interface devices. Today, the three layers together form the CAN Reference Model.

—Additional reporting
by Ron Wilson



Handling CAN-controller complexity has been a barrier to acceptance for this control-oriented bus.



Chips push CAN bus into embedded world

NEWS

Controller Area Network (CAN) Products From Philips Semiconductors

8XC592 CAN Microcontroller - The 8XC592 is a stand-alone high-performance microcontroller based on the 80C51 architecture. Its on-chip CAN interface makes it ideal for applications with a harsh, noisy environment.

The **8XC592 Features** include:

- 16K x 8 EPROM (87C592)
- 16K x 8 ROM (83C592)
- ROMless (80C592)
- 512 x 8 RAM, expandable externally to 64k bytes
- Three Standard 16-bit timer/counters
- A 10-bit ADC with 8 Multiplexed Analog Inputs
- Two 8-bit Resolution, Pulse Width Modulation Outputs
- Five 8-bit I/O Ports Plus One 8-bit Input Port Shared with Analog Inputs
- CAN Controller with DMA Transfer between Internal Data RAM and CAN Registers
- Standard 80C51 UART
- On-Chip Watchdog Timer

82C200 CAN Interface - The 82C200 is a highly integrated stand-alone controller for CAN. The 82C200 with a simple bus line connection performs all the functions of the physical and data-link layers. The application layer of an electronic control unit (ECU) is provided by a microcontroller, to which the 82C200 provides versatile interface.

The **82C200 Features** include:

- Multi-Master Architecture
- Interfaces with a Large Variety of Microcontrollers
- 2032 Message Identifiers
- Powerful Error Handling Capability
- Configurable Bus Interface

82C150 Serial Linked I/O CAN Interface - The 82C150 Serial Linked I/O CAN is a single-chip 16-bit I/O device with an on-chip CAN-controller. 82C150 is a very cost-effective way of increasing the I/O-capability of a microcontroller as well as reducing the amount and complexity of wiring. Advanced safety provided by the CAN protocol combined with low-cost makes the 82C150 a very attractive product for a wide variety of applications.

The **82C150 Features** include:

- Single-Chip I/O Device with CAN Protocol Controller
- 16 Configurable Digital or Analog I/O Ports
- Each Port Individually Configurable via the CAN bus
- 10-bit A/D converter with up to 6 Multiplexed Inputs
- Bit Rate 20 kbit/sec to 125 kbit/sec

Add Text Overlay to Any Video Display

CIRCUIT CELLAR **I N K**®

THE COMPUTER APPLICATIONS JOURNAL

October/November, 1992 — Issue #29

MEASUREMENT & CONTROL

*SPECIAL SECTION:
Embedded
Graphics & Video*

Time Domain Reflectometer
Overlay Text on Video

X-10 Interfacing



Add Text Overlay to Any Video Display

Add Text Overlay to Any Video Display

All the latest VCRs have on-screen programming in an effort to simplify the notorious task. Now you can have your HCS II or any other computer send you messages while you're watching your favorite TV show.

SPECIAL SECTION

Bill Houghton

S

Suppose, for the moment, you've built and installed the HCS II home control system described primarily in issues 25 and 26 (February/March and April/May '92) of the *Computer Applications Journal*. In issue 27 (June/July '92), Ed Nisley described an add-on LCD output device as a way to obtain information about the status of the system and its various nodes. It's a nice addition to the network, but useful only when you're near to the display module. What do you do if you're across the room watching TV settled into your favorite armchair? You could get up and venture across the room. Or, if you build the interface described in this article, you could hit a button on your HCS II IR remote and

see network information displayed on your TV set overlaid onto the program you are watching.

This article describes an On-Screen Display (OSD) terminal for HCS II (we'll call it "TV-Link" to match the other HCS II modules) that allows color text characters to be displayed on top of a background color video signal. The terminal is built around the Philips/Signetics 87C054 OSD microcontroller.

FEATURES OF 87C054

The 87C054 is an 80C51-based microcontroller designed to provide an advanced OSD for TV and video applications. It can produce characters in eight foreground and eight background colors. In addition, the background color can be removed, showing through the original video. It also has nine pulse-width modulator outputs for controlling analog functions. Similar to a standard 80C51, it has 28 digital I/O pins, two external interrupts, and two timer/counters. RAM and ROM spaces on the 87C054 are larger than the 80C51: 192 bytes of RAM and 16K bytes of EPROM. (The OSD has *additional* RAM and EPROM areas that are not part of the normal 80C51 memory map.)

One unique feature of the 87C054 is what Signetics describes as a "soft-

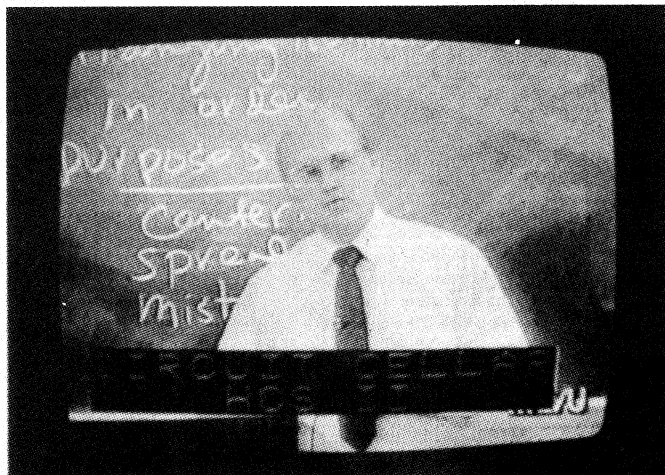


Photo 1—The TV-Link can be used by your computer to overlay messages on any video signal.

Add Text Overlay to Any Video Display

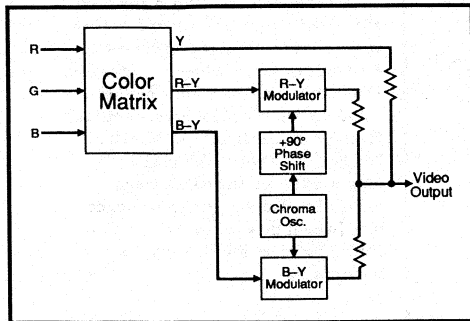


Figure 1—Converting from RGB to NTSC involves modulating and adding difference signals.

ware ADC." This ADC consists of an internal 4-bit DAC that feeds one input of a comparator. The other comparator input can be connected to one of four I/O pins. The output of the comparator is tied to a status bit in a register that is testable by software. A TV set often uses this logic for measuring the AGC voltage during tuning. A real-time clock and other low-precision analog measurements can also use it as a zero-crossing detector.

The OSD of the 87C054 consists of a 128-character RAM array (OSDRAM), a 64-character font EPROM, a video clock oscillator, and the OSD logic. The OSD logic accepts horizontal sync (*HSYNC*) and vertical sync (*VSYNC*) signals and provides three digital video outputs for character information. In the datasheet for this part, these outputs are called *VID0*, *VID1*, and *VID2*, but they can also (and perhaps better) be thought of as *RED*, *GREEN*, and *BLUE*. A multiplexer control output is also present to indicate when to display character data or original video information.

The video clock oscillator provides timing for the character dots. In most applications, this oscillator is simply an LC tank circuit connected to the *VCLK* pins. The frequency controls the character width. One nice feature is that the video clock is killed at the leading edge of *HSYNC* and restarted at the trailing edge of *HSYNC*, which causes the video clock to start in the same phase on every line, ensuring the dots align vertically from one scan line to the next.

The character font stores the binary pattern for the individual characters. Characters are 14 dots wide and 18 scan lines high.

The OSDRAM stores the characters to be displayed on the screen along with certain attribute data pertaining to those characters. Once a character has been written

to the OSD, no further CPU intervention is required to "refresh" the screen.

Many OSD architectures have been developed over the years for use in the consumer television market. Almost all of them have required fixed character row formats, limiting the designer's flexibility in designing video menus and screens.

The 87C054 was designed to avoid such constraints, and there are no architectural limits on the number of characters in a row of text or the number of rows of text to a screen. (There are physical limits imposed by the dot clock frequency and the scan rate, of course.)

The *HSTART* and *VSTART* parameters in the *OSORG* (on-screen origin) register define the initial position of the start of the OSD. Once the initial vertical and horizontal positions have been found, the 87C054 will "fetch" characters from the OSDRAM and place them sequentially on the screen. In order to have multiple rows of text, a special character has been

defined and is referred to as *NEWLINE*. The *NEWLINE* character is much like a carriage return/line feed sequence on a computer in that it terminates the current row of characters and starts a new row of text. One advantage of this architecture is that it eliminates the need to pad display memory with space characters. The fetching and painting of rows of text will continue until either a new vertical sync pulse is detected or until an *END* attribute is fetched along with a *NEWLINE* character.

NOW FOR THE DETAILS...

Now that you understand the concepts of an OSD operation and the capabilities of the 87C054, focus your attention on the details required to overlay characters onto live video.

The 87C054 OSD has a multiplexer output for switching video sources. Simply switching between the input video signal and the OSD character data would be nice. Unfortunately, you can't because the input video (from our home entertainment center) is in NTSC format and the character data is in RGB format. (Keep in mind that the goal is to input live video, add on-screen text, and present the result as a video signal at the output of our circuit.)

One solution is to decode the input video into separate red, green, blue, *HSYNC*, and *VSYNC* signals. Then you could perform the multiplexing between video information and character data in RGB format. The resultant signals could then be encoded back into baseband video. If there were other reasons for the conversion into RGB, such a conversion

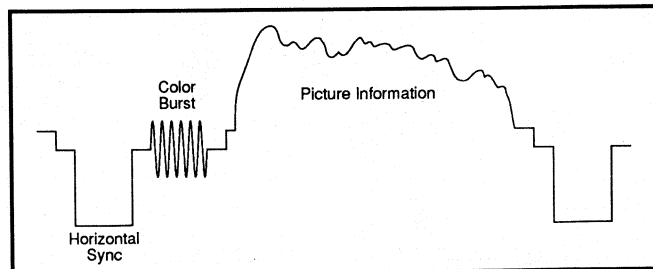


Figure 2—A typical line of NTSC video consists of an initial horizontal sync pulse, followed by a short color burst signal, then the actual picture information.

Add Text Overlay to Any Video Display

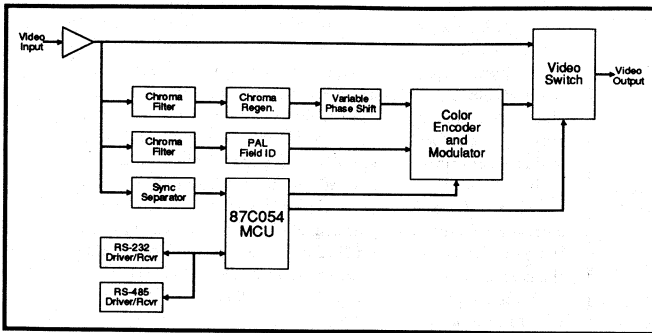


Figure 3—At the core of the TV-Link is the Signetics 87C054 microcontroller. The unit supports both RS-232 and RS-485 communications with a host computer.

would be the way to go. However, the process of converting to RGB and then converting back to video introduces some distortion that could be visible on the screen.

Another solution is to find a way to encode the RGB data from the OSD microcontroller into video. Then you can simply switch between the two sources. Sound simple? The situation gets a little more complex when you consider the issues of making the characters appear with the proper color in NTSC. Reviewing how color is

encoded in NTSC is in order.

COLOR TELEVISION

When you first look at video, you often wonder why in the world it was done the way it was. A long time ago, before Americans had ever heard the names of Japanese TV makers, RCA research labs were developing color television. One of the requirements imposed on designers by the FCC was that the broadcast signal needed to be compatible with existing black-and-white TV sets and had to be contained

within the same bandwidth as a B/W signal. Such requirements meant that some component of the signal had to contain overall brightness information, which is the main reason why they could not simply transmit separate R, G, and B channels within the video bandwidth allowed. To make a very long story short, the engineers involved decided that the scene brightness (which they called "Y" or luminance) could be described by the relationship

$$Y = 0.59G + 0.3R + 0.11B$$

Someone observed that if they took two copies of the luminance signal and subtracted one copy from one of the colors (say, RED) and did the same with a different color (say, BLUE), the result would be two signals that contained all of the information needed to represent color. These resultant signals, R-Y and B-Y, are the color difference signals.

Now that you have two signals, how can they be transmitted on one RF carrier? The answer they came up with was to modulate one of the signals (B-Y) with an RF carrier. The

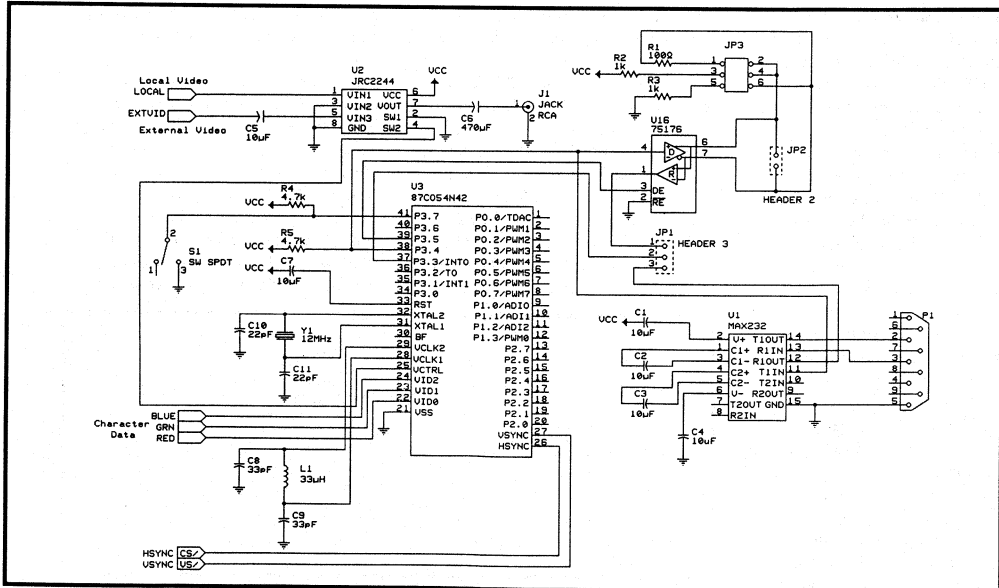


Figure 4a—The JRC2244 switches between the incoming video signal and the on-screen characters under control of the 87C054 MCU.

Add Text Overlay to Any Video Display

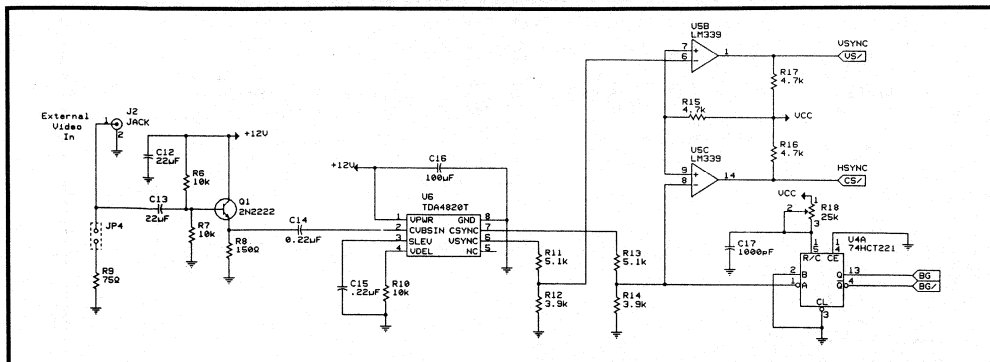


Figure 4b—The TDA4820T sync separator provides the processor with horizontal and vertical sync signals.

other signal [R-Y] was to be modulated with the same RF carrier, but the carrier would be shifted by 90°. When the outputs of the two modulators are added together, the result is the *vector sum* of the two signals, containing both an amplitude and a direction (phase angle). See Figure 1.

We now have a single signal that contains all of the color information. The TV receiver needs just one more piece of information to demodulate this signal. It needs a reference for the carrier used for the modulation, that is, the receiver needs to know where 0° of the color carrier is [in video, this color carrier is referred to as the *chroma subcarrier*]. In order to give this reference to the receiver, a small number, or "burst," of cycles of the color subcarrier (hence, the term color burst) are transmitted on the back porch of the horizontal sync pulse. In most NTSC systems, this chroma subcarrier has a frequency of approximately 3.58 MHz. A typical line of color NTSC video is shown in Figure 2.

In order to convert the character data from the OSD into NTSC, you will need to sum the data into R-Y and B-Y components. Then you will need to modulate these components with a chroma subcarrier at 0° for B-Y and +90° for R-Y.

One more item to consider. If you have an output video signal composed of a video source with characters overlaid onto it, the chroma subcarrier reference (i.e., color burst) present on

the output video signal is the color burst provided in the original input video. In order for the receiver/monitor to interpret the color of the OSD characters correctly, the chroma subcarrier used to modulate the OSD's R-Y and B-Y components must have exactly the same frequency and phase as the color burst on the original video input signal.

Once you get the OSD information in the form just described, you can switch between this "OSD video" and the original input video to produce the final output.

THE TV-LINK HARDWARE SOLUTION

Figure 3 shows a block diagram of the TV-Link, while Figure 4 shows the schematic.

Referring to the schematic, the original input video connects to J2 and is AC coupled into buffer amplifier, Q1. This amplifier provides load isolation between the video signal source and the circuits on the TV-Link board. JP4 is a jumper allowing for a 75-ohm termination resistor to be connected to J2. The output of the Q1 buffer amplifier feeds the sync separator, the video switch, and the chroma subcarrier regenerator circuits.

SYNC SEPARATION

The sync separator consists of U6, a TDA4820T Philips sync separator. The video signal is coupled into the TDA4820T through capacitor C14, where it is amplified with a gain of 15.

The black level clamping voltage is stored in capacitor C14. From the stored black level voltage and the peak sync voltage, the 50% value of the peak sync voltage is generated and stored in capacitor C15. A slicing level control circuit ensures a constant 50% peak sync value regardless of the picture content amplitude provided the sync pulse amplitude is between 50 mV and 500 mV. A comparator in the composite sync slicing stage compares the amplified video signal with the DC voltage derived from the 50% peak sync voltage, producing the composite sync output. Vertical slicing circuits compare the composite sync signal with a DC level equal to 40% of the peak sync signal, producing the vertical sync output. The reduced vertical slicing level ensures more energy for the vertical pulse integration. The slope is double integrated to eliminate the effects of interference caused by noise or line reflections. The value of resistor R10 sets the vertical integration delay time.

The outputs of the sync separator are positive-going signals with peak amplitudes exceeding 10 V. Resistor pairs R11/R12 and R13/R14 serve as voltage dividers for the VSYNC and CSYNC outputs, respectively. An LM339 comparator, U5, serves as an inverter for the sync signals because the modulator circuits require active low sync signals.

There is a great tendency with video circuits to make the coupling capacitors very large to pass the low-

Add Text Overlay to Any Video Display

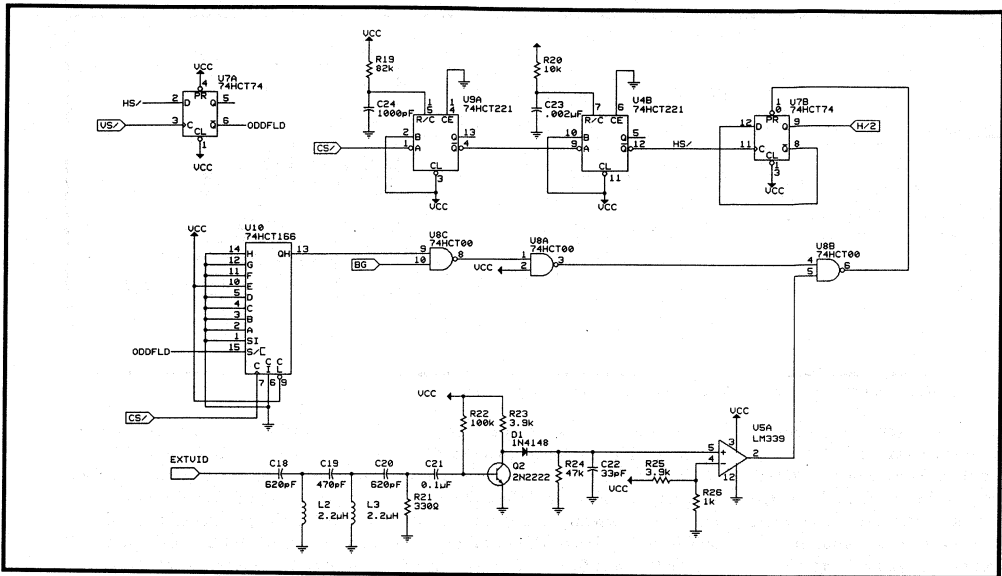


Figure 4c—In order to properly overlay colors onto a PAL signal, you must know whether you're on an odd or an even field, so extra circuitry must be included on the board to support PAL.

frequency sync components (60/50 Hz, typically) into low-impedance nodes. The TDA4820T has a moderately high input impedance on pin 2. Because the black level is stored in C14, the value of C14 should be kept close to 0.22 μ F.

THE 87C054 MCU WITH OSD

The 87C054 microcontroller, U3, accepts composite sync and vertical sync signals from the sync separator and provides RGB digital outputs for character data. The multiplexer control output, VCTRL, connects to the video switch, U2, a JRC2244.

Inductor L1 and capacitors C8 and C9 form a video clock oscillator that determines the width of a character font dot. The values of these components are not critical but are typically chosen such that a video dot width is equal to the spacing between scan lines. This oscillator is killed at the leading edge of the HSYNC signal and allowed to startup at the trailing edge. Such synchronization causes the oscillator to start at exactly the same point on one scan line to the next, causing character dots to appear in exactly the same spot on each line.

In addition to the OSD functions, the 87C054 also performs network interfacing and protocol tasks. This microcontroller has plenty of performance bandwidth because the OSD logic is self-refreshing and independent of the MCU core.

VIDEO SIGNAL SWITCHING

The JRC2244 video switch, U2, contains three video inputs, two of which are used in this application. One of these inputs, VIN1, is capacitively coupled to the OSD video signal. This signal is the 87C054's RGB data after encoding into baseband video. The other input, VIN3, is capacitively coupled to the original video input signal. The JRC2244 provides internal bias sources to provide DC restoration to its video inputs. The JRC2244 accepts a switching control signal from the 87C054 and switches its output between the original video input and the OSD video signal. The video switch also has an internal 75-ohm line driver in its output stage.

The JRC2244 has a moderate input impedance of about 15k ohms, allowing 10- μ F coupling capacitors to

be used. The output coupling capacitor is large because this signal can be used to drive 75-ohm loads.

RGB ENCODING

The LM1886, U13, and the LM1889, U14, encode the RGB data from the 87C054 into baseband video. The LM1886 has three DACs, one for each color. Each of these DACs has 3-bit inputs, but because the 87C054 data is digital, the inputs to the LM1886 DACs are tied together yielding an output for each DAC that is either full-scale or zero. The outputs of the three DACs are internally summed to produce the luminance, R-Y, and B-Y amplitudes. The LM1889 accepts the regenerated chroma subcarrier, modulates the R-Y and B-Y signals, and produces baseband video on pin 13. Transistor Q5 is used as a buffer amplifier with voltage dividers R49 and R50 producing proper levels for the video switch. Note that the LM1889 accepts an external subcarrier signal at the junction of R46 and C52, but this subcarrier undergoes a phase shift caused by the resistor and capacitor networks associated with pins 1

Add Text Overlay to Any Video Display

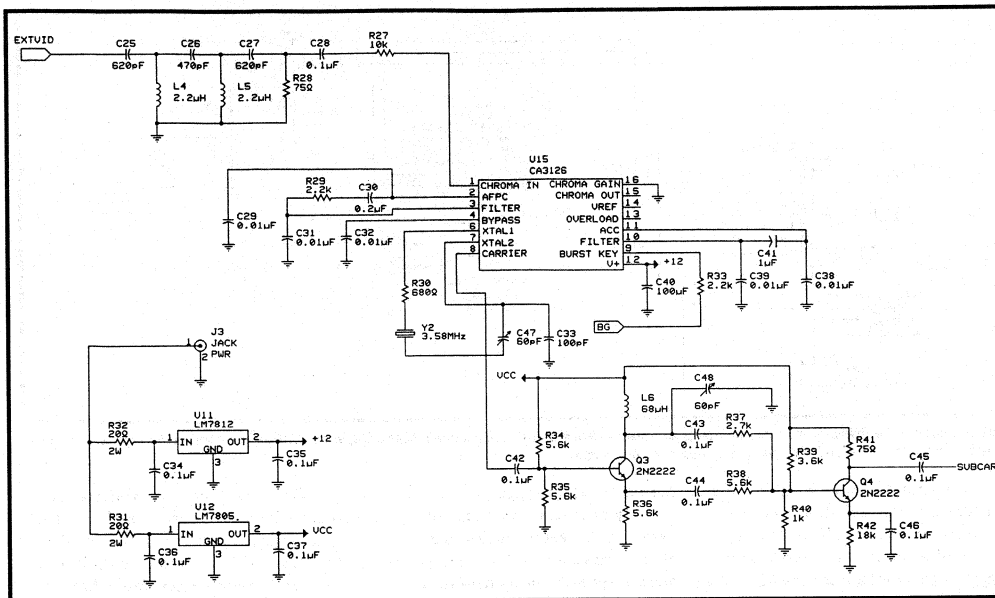


Figure 4d—The CA3126 TV chroma processor is designed specifically for regenerating chroma subcarriers.

and 18 of the LM1889. This phase shift will need to be considered when the subcarrier is regenerated.

CHROMA SUBCARRIER REGENERATION

The circuits that reproduce a chroma subcarrier in the same frequency and phase as the color burst consist of a high-pass filter, a sample-and-hold phase-locked loop (PLL), and a phase shift network and amplifier.

The passive high-pass filter consists of inductors L4 and L5, resistor R28, and capacitors C25, C26, and C27. The filter starts passing signals at about 3.2 MHz, allowing the chroma subcarrier to pass through to the CA3126, U15.

The CA3126 is a TV Chroma Processor IC designed specifically for regenerating chroma subcarriers. This IC contains a VCO and a PLL with sample-and-hold circuits in the error correction loop. As a result, the VCO-generated carrier is compared with the chroma signal from the high-pass filter during the time that color burst is present, indicated by the burst gate pulse (which I will describe later).

The regenerated carrier output is present on pin 8 of the CA3126. Even though this carrier is phase locked to the color burst, it is not at exactly the same phase as the color burst. The nature of a PLL is such that the output will be locked but will always have some constant fixed phase delay relative to the input. Also, recall that the input circuits of the LM1889 added an additional constant phase shift to the injected carrier.

The phase shift network and amplifier consisting of the Q3 and Q4 stages compensate for these fixed phase delays. This circuit provides an output whose phase is adjustable by means of variable capacitor C48, and has a tuning range of approximately 0° to 160° of phase shift. For a given input signal amplitude, the output signal amplitude is constant, regardless of the phase shift introduced. The output of this circuit is the signal injected into the LM1889 circuits.

CONNECTING TO THE HCS II

Now that you have a working terminal circuit for overlaying text onto live video, you still need to

connect it to the HCS II network. In order to do this, you will need a serial interface compatible with the network, some software that handles network message formats, and software that interprets network messages and creates responses or actions (or both) to those HCS II network messages.

This particular design includes both an RS-232 and an RS-485 interface. U1, a MAX232, provides the RS-232-to-TTL conversion for both the transmitter and receiver. U16, a 75176, provides the RS-485-to-TTL conversion. JP1 connects the receiver pin of the 87C054 to either the RS-232 or RS-485 interfaces. The transmitter pin of the 87C054 connects to both the RS-232 and RS-485 interfaces. One pin of the 87C054, P3.5, controls the driver enable of the 75176, allowing for selective talking on the HCS II network. JP3 provides for termination of the network.

"But, wait a minute! The 87C054 doesn't have a UART," you say. True. There is no built-in UART on the 87C054 and the part does not have a transmitter pin or receiver pin.

Add Text Overlay to Any Video Display

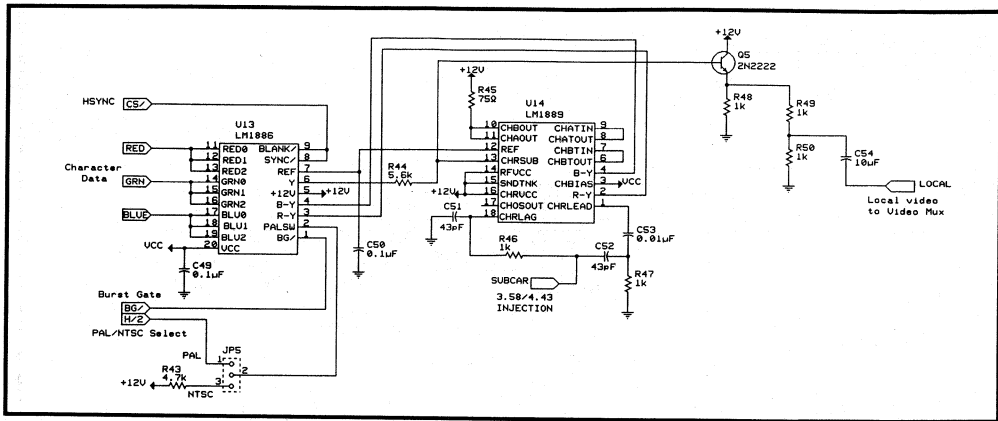


Figure 4e—Encoding the RGB data into baseband video is accomplished with an LM1886, which contains three DACs, and an LM1889, which accepts the regenerated chroma subcarrier, modulates the R-Y and B-Y signals, and produces baseband video.

In this application, the serial data transmission and reception has been performed in software. The routine that handles serial transmission and reception was taken from the Signetics BBS ([800] 451-6644). It was originally designed for the 87C751 and had to be slightly modified to operate with one of the 87C054's timer/counters. The technique is often called "bit banging" and has the advantage of saving some hardware if you can afford the necessary time required of the software.

NETWORK PROTOCOL PROCESSING

As I indicated earlier, in addition to the serial interface software, you need code that handles network message formats. The code starts by waiting until either a "#" or an "!" is received, either of which indicates the start of a network message, then the entire message is stored in a buffer.

Once the carriage return has been stored, the beginning character of the message is checked to see whether the message includes a checksum. If the message does not contain a checksum, the packet is assumed to be valid and the contents of the packet are processed. If a checksum is included, then the VERIFY routine is called to perform a checksum calculation on the packet. If the checksum matches, the packet is processed; otherwise, it is ignored and I return to waiting for the

next network message.

My original plans for handling network checksums included a checksum generator for sending network responses and a checksum checker for received messages. However, when I flowcharted the needs of both routines, I found that an awful lot of the logic was common to both. I went back and looked at the suggestions that Ed Nisley had provided for handling the checksums and understand now why he made those suggestions. My VERIFY routine's logic is based on Ed's previous work.

The VERIFY routine performs two functions. First, it takes the checksum digits in the packet, converts them to binary numbers, and stores them in temporary variables. Next, the checksum digits are replaced with ASCII zeros and the checksum of the string is calculated. If the checksum matches, the error flag, CHKERR, is cleared; otherwise, it is set. The checksum that was calculated is converted to ASCII and stuffed into the checksum digits position, replacing the ASCII zeros.

To prepare a string for transmission, all that is necessary is to stuff the message in the buffer with the checksum digits set to ASCII zeros and call the VERIFY routine. To check a message for correct checksum, simply call the VERIFY routine and check the CHKERR flag on return.

Once the checksum verification (if required) has been performed, you still need to process the packet to see if it belongs to this terminal, and if it does, then you need to determine what action the network controller is asking you to take.

The PROCESS routine first scans the packet, converting characters into upper case until the end of the packet has been reached. Next, the first character is examined to determine if the packet has checksums or not and a pointer is set to the NODEID position of the packet. The NODEID in the packet is compared with the NODEID variable. If there is no match, the packet is ignored and you wait for the next network message. If it does belong do this terminal, you can process the body of the network message.

NETWORK COMMANDS AND SYNTAX

The real essence of a network message is to carry a command from the network controller to the terminal or carry a response from the terminal back to the network controller. Table 1 shows the syntax of the commands available for operating the TV-Link terminal. These commands allow the HCS II Supervisory Controller to manipulate ports on the 87C054, format text for display, implement special built-in display functions such as color bars, and to read and write

Add Text Overlay to Any Video Display

A = string	Set HCS II network address to string
Fx	Execute special function <ul style="list-style-type: none"> 0 Initialize screen 1 Display on 2 Display off 3 Display color bars 4 Wipe on 5 Wipe off
Hxy	Set Px.y high
Lxy	Clear Px.y low
Nn	Network response mode <ul style="list-style-type: none"> N0 = normal network interface, no auto error or acknowledge responses N1 = auto error and acknowledge response
Px	Query port x <ul style="list-style-type: none"> 0 = Port 0 1 = Port 1 2 = Port 2 3 = Port 3
Px= nn	Write to port x where nn= two-digit hex value <ul style="list-style-type: none"> 0 = Port 0 1 = Port 1 2 = Port 2 3 = Port 3
Rx	Query register x; returns two-digit hex number <ul style="list-style-type: none"> 0 = OSDT (contents undefined) 1 = OSAT 2 = OSCON 3 = OSORG 4 = OSMOD 5 = Default char. attribute 6 = Default background space attr 7 = Default NEWLINE attribute
Rx = nn	Write to register x; for use from outside of a string of text; writes to these registers from within a string; should use the Wxnn command <ul style="list-style-type: none"> 0 = OSDT 1 = OSAT 2 = OSCON 3 = OSORG 4 = OSMOD 5 = Default char. attribute 6 = Default background space attr 7 = Default NEWLINE attribute
S= string	String for OSD display; can include escape sequences for text formatting, color, selection, etc.
Wxnn	Write to register x; for use within a string; functionally equiv. to the Rx = nn command
Special characters for use within a string of text	
\E	End of Display at current position
\B	Background Space
\S	Split Background Space
\N	NEWLINE

Table 1—The set of supported commands resembles that of most of the other HCS II network modules.

OSD registers directly, giving full control of the OSD to the HCS II.

CONCLUSIONS

Developing this application was interesting and enjoyable. It also

presented some challenges.

The 87C054 proved well suited to this application in large part because of the 80C51 core and that the OSD is independent of the CPU. Once characters have been written to the OSD, you

can forget the OSD until you want to change the display, and the CPU is free to pursue other tasks.

The on-screen display and the microcontroller operations are primarily digital functions. The question of how to combine this technology with an analog video signal can be perplexing to most system designers whose professional experiences have been mostly digital circuits. One of the most perplexing issues during this project was how to re-create the chroma subcarrier. I knew that every color TV set had to perform this function, but finding out solutions took some searching before I discovered the CA3126. I'm hopeful you can profit from my experiences on this project. □

My thanks to Herb Knies and George Ellis of Signetics for their help. Thanks also to Greg Goodhue from Signetics, who wrote the software-based UART code for the 87C751 that I modified for this project.

Bill Houghton is an Applications Engineer at Signetics specializing in 80C51-based microcontrollers.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

SOURCES

Requests for literature on Signetics/Philips microcontrollers including the "80C51-Based 8-Bit Microcontroller Data Handbook" may be directed to Sharon Baker at (408) 991-3518.

Contact Bill Houghton at (408) 991-3560 with technical questions specific to the 87C054 and for information on the availability of a PC board and components for this project.

©Circuit Cellar INK, The Computer Applications Journal. Reprinted by permission

Section 9

Development Support Tools

Application Notes and Development Tools for 80C51 Microcontrollers

CONTENTS

Development support tools	759
Ashling CTS51 Microprocessor development systems for Philips microcontrollers	766
BSO/Tasking: The total development solution for the 8051 family	771
CEIBO DB-51 Development Board	778
CEIBO DS-51 In-Circuit Emulator	780
CEIBO DS-300 Peripheral Development Tools	788
CEIBO DS-750 Microcontroller Development Tool	790
CEIBO DS-752 Microcontroller Development Tool	792
CEIBO EB-51 Emulation Board	794
CEIBO MP-51 Programmer	796
MetaLink iceMASTER-PE 8051 Family In-Circuit Emulator	798
MetaLink iceMASTER 8051 Family In-Circuit Emulators	803
NOHAU EMUL51-PC – PC-based in-circuit emulator	816
PDS51 Development System for 8XC51 Microcontroller Derivatives	827
LCP Programmers for 87C51 and Derivatives	831
S87C00KSD 8XC51 and I ² C Bus Evaluation Board	834
OM4130 CAN evaluation board with the 8XC552 and 82C200 CAN controller	835
OM4239 CAN evaluation board with the 8XC592 microcontroller	836
OM4240 Evaluation board for the 8XCE598 microcontroller	837
OM4272 SLIO evaluation board with the 82C150 and 82C250 CAN ICs	838
OM4280 P83C852 Smart Card crypto-controller demonstration kit	839
PEB552 Evaluation board	840

Development support tools

DEVELOPMENT SUPPORT TOOLS

Philips Semiconductors manufactures support tools and also works closely with many "third-party" vendors who provide support tools for our wide variety of 80C51-based microcontroller derivatives.

Development Systems

In most cases, development systems are available in two versions for ROM and ROMless applications. The ROM emulation products are capable of supporting all versions of a given device type, including EPROM, ROM, and ROMless devices. In contrast, a ROMless emulator can only support applications designed for a ROMless microcontroller. Most development systems are designed to connect to an IBM-PC or compatible personal computer.

EPROM Programming Support

Philips Semiconductors works closely with major suppliers of EPROM programming equipment to support our family of EPROM microcontrollers. As a result, EPROM programming support is available within the programming facilities of many major distributors.

The following is a list of vendors that offer support for Philips Semiconductors 80C51 microcontroller family.

DEVELOPMENT SYSTEM CONTACTS

COMPANY	ADDRESS	TELEPHONE
Ashling Microsystems Limited	Plassey Technological Park Limerick, Ireland	(353) 61 334 466
	Eastern Systems Inc. 160 East Main Street Westboro, MA 01581	(508) 366-3220
BSO Tasking	Norfolk Place 333 Elm Street Dedham, MA 02026-4530	(800) 458-8276
Ceibo Ltd.	105 Gleason Rd. Lexington, MA 02173	(617) 863-9927
	Merkazim Building, Industrial Zone P.O. Box 2106 Herzeliya 46120, ISRAEL	972-52-555387
Lauterbach Datentechnik GmbH	Fichtenstrasse 27 85649 Hofolding Germany	49 8104 894 328
	945 Concord Street Framingham, MA 01701	(508) 620-4521
MetaLink Corp.	325 E. Elliot Road, Suite 23 Chandler, AZ 85225	(602) 926-0797
Nohau Corp.	51 E. Campbell Ave. Campbell, CA 95008-2053	(408) 866-1820
Philips Semiconductors	Corporate Centre Building BAE-2 P.O. Box 218 5600 MD Eindhoven The Netherlands	31-40-724223
SIGNUM Systems	171 E. Thousand Oaks Blvd., #202 Thousand Oaks, CA 91360	(805) 371-4608

EPROM PROGRAMMING SUPPORT CONTACTS

Advin Systems 1050-L East Duane Ave. Sunnyvale, CA 94086 (408) 736-2503	Logical Devices, Inc. 1201 Northwest 65th Place Ft. Lauderdale, FL 33309 (305) 974-0967	North Valley Products P.O. Box 32899 San Jose, CA 95152 (408) 929-5345
BP Microsystems 10681 Haddington #190 Houston, TX 77043 (800) 225-2102, (713) 461-9430	Logical Systems P. O. Box 6184 Syracuse, NY 13217-6184 (315) 478-0722	Strebor Data Communications 1008 N. Nob Hill American Fork, UT 84003 (801) 756-3605
Data I/O Corp. 10525 Willows Road N.E. P.O. Box 97046 Redmond, WA 98073-9746 (206) 881-6444	Needham's Electronics 4535 Orange Grove Ave. Sacramento, CA 95841 (916) 924-8037	

Development support tools

SOFTWARE SUPPORT CONTACTS

COMPANY	ADDRESS	TELEPHONE
Archimedes Software, Inc.	2159 Union St. San Francisco, CA 94123	(415) 567-4010
BSO/Tasking	Tasking Software BV P.O. Box 899 3800 AW Amersfoort The Netherlands	31-33-55-85-84 (Telephone) 31-33-55-00-33 (Fax)
	Norfolk Place 333 Elm Street Dedham, MA 02026-4530	(617) 320-9400 (Telephone) (617) 320-9212 (Fax) (800) 458-8276 (Toll Free)
Franklin Software, Inc.	888 Saratoga Ave. #2 San Jose, CA 95129	(408) 296-8051
Keil Software	Bretonischer Ring 15 85630 Grasbrunn Germany	49-89-46-50-57 (Telephone) 49-89-46-81-62 (Fax)

MICROCONTROLLER DEVELOPMENT SYSTEMS

PRODUCT	DEVICES SUPPORTED
NOHAU CORPORATION	
EMUL51-PC/E32	12MHz Emulator, 32k emulation memory
EMUL51-PC/E128	12MHz Emulator, 128k emulation memory
EMUL51-PC/E32-16	16MHz Emulator, 32k emulation memory
EMUL51-PC/E128-16	16MHz Emulator, 128k emulation memory
EMUL51-PC/E128-20	20MHz Emulator, 128k emulation memory
EMUL51-PC/E128-24	24MHz Emulator, 128k emulation memory
EMUL51-PC/E128-30	30MHz Emulator, 128k emulation memory
EMUL51-PC/E128-33	33MHz Emulator, 128k emulation memory
EMUL51-PC/E128-BSW	12MHz Emulator, 128k bankswitched CODE memory
EMUL51-PC/E128-BSW-16	16MHz Emulator, 128k bankswitched CODE memory
EMUL51-PC/E128-BSW-24	24MHz Emulator, 128k bankswitched CODE memory
EMUL51-PC/E256-BSW	12MHz Emulator, 256k bankswitched CODE memory
EMUL51-PC/E256-BSW-16	16MHz Emulator, 256k bankswitched CODE memory
EMUL51-PC/E256-BSW-24	24MHz Emulator, 256k bankswitched CODE memory
POD-C054	12MHz 83C053, 83C054, 87C054, 87C055 pod
POD-31	12MHz 8031 pod
POD-C31	12MHz 80C31 pod
POD-C31-1	16MHz 80C31 pod
POD-C31-20	20MHz 80C31 pod
POD-C31-24	24MHz 80C31 pod
POD-C31-30	30MHz 80C31 pod
POD-C31-33	33MHz 80C31 pod
POD-32	12MHz 8032 pod
POD-C32	12MHz 80C32 pod
POD-C32-16	16MHz 80C32 pod
POD-C652	12MHz 80C652 pod
POD-C652-16	16MHz 80C652 pod
POD-C51B	12MHz bondout pod for 8051, 80C51, 83C552, 83C652, 83C654, 83C851, and EPROM or ROMless versions of the above
POD-C51B-16	16MHz bondout pod for 8051, 80C51, 83C552, 83C652, 83C654, 83C851, and EPROM or ROMless versions of the above
POD-C51B-24	24MHz version of the above
POD-C52	12MHz 80C32, 80C52, 87C52
POD-C52-16	16MHz 80C32, 80C52, 87C52
POD-CL410	12MHz 80CL31, 80CL51, 83CL410, 83CL610
POD-C451-DIP	12MHz 80C451 DIP pod
POD-C451-DIP-16	16MHz 80C451 DIP pod
POD-C451-PGA	12MHz 80C451 PLCC pod (PGA from pod)
POD-C451-PGA-16	16MHz 80C451 PLCC pod (PGA from pod)
POD-C451B-PGA	12MHz bondout pod for 83C451, 87C451, 80C451 PLCC (PGA from pod)
POD-C451B-PGA-16	16MHz bondout pod for 83C451, 87C451, 80C451 PLCC (PGA from pod)

Development support tools

MICROCONTROLLER DEVELOPMENT SYSTEMS (Continued)

PRODUCT	DEVICES SUPPORTED
NOHAU CORPORATION (Continued)	
POD-C528	12MHz 83C524, 87C524, 83C528, 87C528
POD-C528-16	16MHz 83C524, 87C524, 83C528, 87C528
POD-C550-PGA	12MHz 80C550, 83C550, 87C550
POD-C550-PGA-16	16MHz 80C550, 83C550, 87C550
POD-C552-PGA	12MHz 80C552 PLCC (PGA from pod)
POD-C552B-PGA	12MHz bondout pod for 83C552, 87C552, 80C552 PLCC (PGA from pod), 80C562, 83C562
POD-C552B-PGA-16	16MHz bondout pod for 83C552, 87C552, 80C552 PLCC (PGA from pod), 80C562, 83C562
POD-C552B-PGA-24	24MHz pod for 83552, 87C552, 80C552
POD-C575	12MHz 87C575
POD-C558-16	16MHz 83CE558, 89CE558 pod
POD-CL580	12MHz bondout POD for 83CL580
POD-C592-PGA	12MHz 80C592, 83C592, 87C592 pod
POD-C592-PGA-16	16MHz 80C592, 83C592, 87C592 pod
POD-C652B	Order as POD-C51B
POD-C851B	Order as POD-C51B
POD-C751	12MHz 83C750, 87C750, 83C751, 87C751 pod
POD-C751-16	16MHz 83C750, 87C750, 83C751, 87C751 pod
POD-C752	12MHz 83C752, 87C752 pod
POD-C752-16	16MHz 83C752, 87C752 pod
POD-CL782	12MHz 83CL781, 83CL782, 83CL52 pod
EMUL51-PC/TR4	12MHz 4k trace buffer option
EMUL51-PC/TR16	12MHz 16k trace buffer option
EMUL51-PC/TR4-16	16MHz 4k trace buffer option
EMUL51-PC/TR16-16	16MHz 16k trace buffer option
EMUL51-PC/TR16-20	20MHz 16k trace buffer option
EMUL51-PC/TR16-24	24MHz 16k trace buffer option
EMUL51-PC/TR16-30	30MHz 16k trace buffer option
EMUL51-PC/TR16-33	33MHz 16k trace buffer option
EMUL51-PC/ATR64-16	16MHz 64k Advanced Trace Option
EMUL51-PC/ATR256-16	16MHz 256k Advanced Trace Option
EMUL51-PC/ATR64-24	24MHz 64k Advanced Trace Option
EMUL51-PC/ATR256-24	24MHz 256k Advanced Trace Option
EMUL51-PC/ATR64-33	33MHz 64k Advanced Trace Option
EMUL51-PC/ATR256-30	30MHz 256k Advanced Trace Option
EMUL51-PC/BOX-CS	Serial box with emulator (E128-16) and trace (TR16-16)
EMUL51-PC/BOX-CS-20	Serial box with emulator (E128-20) and trace (TR16-20)
EMUL51-PC/BOX-CS-24	Serial box with emulator (E128-24) and trace (TR16-24)
EMUL51-PC/BOX-CS-30	Serial box with emulator (E128-30) and trace (TR16-30)
EMUL51-PC/BOX-S	Box with serial port and cable. Box allows operation of emulator external to PC.
METALINK CORPORATION	
IM-8051/200-20	iceMASTER-8051 emulator Model 200, 32K emulation memory, 20MHz
IM-8051/400-20	iceMASTER-8051 emulator Model 400, 32K emulation memory, 4K trace buffer, 2 performance analyzers, 20MHz
IM-8051/400-24	iceMASTER-8051 emulator Model 400, 128K emulation memory, 4K trace buffer, 2 performance analyzers, 24MHz
128KUP	128K memory expansion option for iceMASTER-8051
752/1 PGMPC	Programmer accessory for iceMASTER to program 87C751, 87C752
8031-12PC	0.5 to 12MHz 8031, 80C31
8031-16PC	0.5 to 16MHz 8031, 80C31
8031-20PC	0.5 to 20MHz 8031, 80C31
8031-24PC	0.5 to 24MHz 8031, 80C31
8032-12PC	0.5 to 12MHz 8031, 80C31, 8032, 80C32
8032-16PC	0.5 to 16MHz 8031, 80C31, 8032, 80C32
8032-20PC	0.5 to 20MHz 8031, 80C31, 8032, 80C32

Development support tools

MICROCONTROLLER DEVELOPMENT SYSTEMS (Continued)

PRODUCT	DEVICES SUPPORTED
METALINK CORPORATION (Continued)	
8032-24PC	0.5 to 24MHz 8032, 80C32, 8031, 80C31
8052-12PC	0.5 to 12MHz 8031, 80C31, 8032, 80C32, 8051, 8751, 80C51, 87C51, 8052, 8752, 80C52, 87C52
8052-16PC	0.5 to 16MHz 8031, 80C31, 8032, 80C32, 8051, 8751, 80C51, 87C51, 8052, 8752, 80C52, 87C52
80410-12PC	0.5 to 12MHz 80CL410
80451-12PC	1.2 to 12MHz 80C451
80451-16PC	1.2 to 16MHz 80C451
80528-12PC	1.2 to 12MHz 80C528
80528-16PC	1.2 to 16MHz 80C528
80552-12PC	1.2 to 12MHz 80C552, 80C562
80552-16PC	1.2 to 16MHz 80C552, 80C562
80652-12PC	1.2 to 12MHz 8031, 80C31, 80C652
80652-16PC	1.2 to 16MHz 8031, 80C31, 80C652
80851-12PC	1.2 to 12MHz 8031, 80C31, 80C851
83053-12PC	6 to 12MHz 83C053, 83C054, 87C054
83451-12PC	1.2 to 12MHz 80C451, 83C451, 87C451
83528-12PC	1.2 to 12MHz 80C528, 83C528, 87C528, 83C524, 87C524
83528-16PC	1.2 to 16MHz 80C528, 83C528, 87C528, 83C524, 87C524
83550-10PC	1.2 to 10MHz 80C550, 83C550, 87C550
83552-12PC	1.2 to 12MHz 80C552, 83C552, 87C552, 80C562, 83C562
83552-16PC	1.2 to 16MHz 80C552, 83C552, 87C552, 80C562, 83C562
83652-12PC	1.2 to 12MHz 80C652, 83C652, 87C652, 80C552, 83C552, 87C552, 80C562, 83C562
83652-16PC	1.2 to 16MHz 80C652, 83C652, 87C652, 80C552, 83C552, 87C552, 80C562, 83C562
83654-12PC	1.2 to 12MHz 80C652, 83C652, 87C652, 83C654, 87C654, 80C552, 83C552, 87C552, 80C562, 83C562
83654-16PC	1.2 to 16MHz 80C652, 83C652, 87C652, 83C654, 87C654, 80C552, 83C552, 87C552, 80C562, 83C562
83751-12PC	0.5 to 12MHz 83C751, 87C751
83751-16PC	0.5 to 16MHz 83C751, 87C751
83752-12PC	0.5 to 12MHz 83C752, 87C752

Development support tools

EPROM MICROCOMPUTER PROGRAMMING SUPPORT

DEVICE	MANUFACTURER/MODEL	MODULE/ADAPTOR	SOFTWARE VERSION
87C054 SDIP	N. Valley Products Philips, Ceibo MP-51	SAM-054SD PPA-054SD	
87C51 DIP	Advin Sailor-PAL/SA, /SB BP Microsystems EP-1140 Ceibo MP-51 (PMP51SD) Data I/O Unisite 40 Data I/O Unipak 2b Data I/O Series 1000 Logical Devices ALLPRO N. Valley Products SPGM-100 Philips LCPX5X40 (P8051LCP40) Strebtor PLP-S1A	Adaptor-8751 PPA-51XSD 351B103 SR40 SAM-51SD MC4851DIP	V2.2 V16 V05 (Use Intel 87C51 menu) V1.47 V1.0
87C51 PLCC	Ceibo MP-51 (PMP51SD) Data I/O Unisite 40 Data I/O Unipak 2b Logical Devices ALLPRO N. Valley Products SPGM-100 Philips LCPX5X40 (P8051LCP40)	PPA-51XSD Chipsite 351B103P Required SAM-51ASD	V2.3 V16 V1.47 V1.0
87C52 DIP	BP Microsystems EP-1140 Ceibo MP-51 (PMP51SD) Data I/O 29B, Unipak 2b Data I/O Unisite 40 N. Valley Products SPGM-100 Philips LCPX5X40 (P8051LCP40)	PPA-XSD 351B103 SAM-52SD	V23 V3.1
87C451 DIP	Ceibo MP-51 (PMP51SD) Logical Devices ALLPRO N. Valley Products SPGM-100	PPA-451SD PPRequired SAM-451SD	V1.47 V1.0
87C451 PLCC	Advin Sailor-PAL/SA, /SB Ceibo MP-51 (PMP51SD) N. Valley Products SPGM-100 Data I/O Unisite Philips LCPX5X (P8051LCPX)	Adaptor-87451 PPA-451ASD SAM-451ASD Chipsite	V1.0 V2.8
87C528 DIP	BP Microsystems EP-1140 Ceibo MP-51 N. Valley Products SPGM-100 Philips LCPX5X40 (P8051LCP40)	PPA-XSD SAM-528	
87C528 PLCC	Ceibo MP-51 (PMP51SD) N. Valley Products SPGM-100 Philips LCPX5X40 (P8051LCP40)	PPA-51XSD SAM-528A	
87C552	Ceibo MP-51 (PMP51SD) N. Valley Products SPGM-100 Data I/O Unisite Philips LCPX5X (P8051LCPX)	PPA-552ASD SAM-552ASD Chipsite	V2.2 V3.1
87C652 DIP	BP Microsystems EP-1140 Ceibo MP-51 N. Valley Products Philips LCPX5X40 (P8051LCP40)	PPA-51XSD SAM-52SD	
87C652 PLCC	Ceibo MP-51 (PMP51SD) Philips LCPX5X40 (P8051LCP40)	PPA-51XSD	
87C654 DIP	BP Microsystems EP-1140 Ceibo MP-51 (PMP51SD) N. Valley Products SPGM-100 Philips LCPX5X40 (P8051LCP40)	PPA-51XSD SAM-654SD	
87C654 PLCC	Ceibo MP-51 (PMP51SD) Philips LCPX5X40 (P8051LCP40)	PPA-51XSD	

Development support tools

EPROM MICROCOMPUTER PROGRAMMING SUPPORT (Continued)

DEVICE	MANUFACTURER/MODEL	MODULE/ADAPTOR	SOFTWARE VERSION
87C751 DIP	Advin Sailor-PAL/SA, /SB BP Microsystems EP-1140 Ceibo MP-51 (PMP51SD) Data I/O Unisite 40 Data I/O 29B, Unipak 2b Logical Devices ALLPRO N. Valley Products SPGM-100 Needham's Electronics MetaLink Logical Systems (Sunshine EW-901) Philips LCPX5X (P8051LCPX) Strebtor PLP-S1A	EM-751 HEAD-40A PPA-751SD 351B113D OPTAPC-751 SAM-751SD 752/1 PGMPC PA751 MC7512DIP	V2.3 29B V6, Unipak 2B V20 V1.47 V1.0 V2.6a (use with MicroICE+)
87C751 PLCC	Ceibo MP-51 (PMP51SD) Data I/O Unisite 40 Logical Devices ALLPRO N. Valley Products SPGM-100	PPA-51XSD Chipsite Required SAM-751ASD	V2.6 V1.47 V1.0
87C752 DIP	Advin Sailor-PAL/SA, /SB BP Microsystems EP-1140 Ceibo MP-51 (PMP51SD) Data I/O Unisite 40 N. Valley Products SPGM-100 Needham's Electronics MetaLink Logical Systems (Sunshine EW-901) Logical Devices ALLPRO Philips LCPX5X (P8051LCPX) Strebtor PLP-S1A	EM-751 HEAD-40A PPA-752SD SAM-752SD 752/1 PGMPC PA751 OPTAPC-752 MC7512DIP	V2.6 V1.0 (Use Type 751) V2.6a (use with MicroICE+)
87C752 PLCC	Ceibo MP-51 (PMP51SD) Data I/O Unisite 40 N. Valley Products SPGM-100	PPA-752ASD Chipsite SAM-752ASD	V2.8 V1.0 (Use Type 751)

NOTE:

Philips programmers are available in the U.S.A. through Philips Semiconductors distributors.

Development support tools

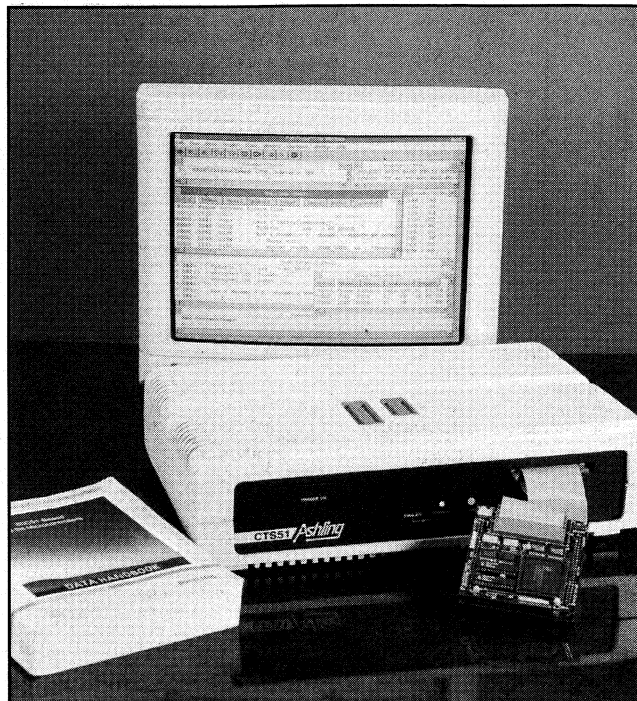
ADDITIONAL PROGRAMMING SUPPORT

DEVICE	MANUFACTURER	MODULE/ADAPTOR	COMMENTS
87C51/52/652/ 654/528 PLCC	Logical Systems	PA51-44	Use with any 40-pin microcontroller programming site that supports the appropriate EPROM size.
87C51/52/652/ 654/528 QFP		PA52-QFP	Use with any 40-pin microcontroller programming site that supports the appropriate EPROM size.
87C451 DIP		PA451-64	Use with any 87C51 40-pin programming site.
87C451 PLCC		PA451-68	Use with any 87C51 40-pin programming site.
87C550 DIP		550BASE	Use with any 87C51 40-pin programming site.
87C550 PLCC		PA550-44	Use with any 87C51 40-pin programming site.
87C552 PLCC		PA552-68	Use with any 8752/C52/C252/C51FA 40-pin programming site.
87C752 PLCC		PA28-28	Use with any 87C752 28-pin DIP programmer.
87C751 PLCC	Philips	No part number assigned	This adapter allows programming the 87C751 PLCC part in conjunction with any programmer that can already program the DIP version of the part.

MICROCONTROLLER SUPPORT

PRODUCT	DEVICES SUPPORTED	MANUFACTURER	DESCRIPTION
8051 C Compiler 8051 C Compiler 80C51 C Compiler	8051 and derivatives 8051 and derivatives 8051 and derivatives	Franklin Software Archimedes Software BSO/Tasking	C Compiler for 8051 family C Compiler for 8051 family C Compiler for 8051 family
P8051DB	8051 and derivatives	Ceibo	80C51 Family Development Board
S87C00KSD	--	Philips	I ² C Demonstration Board. 87C751 controls various I ² C peripherals. Board has sockets for 87C752, 87C652, and 87C552 also.
--	--	Philips	The Philips computer Bulletin Board system has available a microcontroller newsletter, application and demonstration programs for download, and the ability to send messages to microcontroller applications engineers. Access by modem at 2400, 1200, or 300 baud. The telephone numbers are: (800) 451-6644 (in the U.S.) or (408) 991-2406.
SMI-CNV451SD	80/83/87C451	Philips	Philips product adapts a PLCC emulator plug for the 80C451 to the DIP pinout.

CTS51 Microprocessor Development Systems for Philips Microcontrollers



Ashling's CTS51 Universal Microprocessor Development System for Philips Microcontrollers

Features of Ashling's Microprocessor Development systems:

- ◆ Integrated Development Environment for Philips 8051 under Windows
- ◆ Real-time in-circuit emulator for all Philips 8051 derivatives; full-speed, non-intrusive emulation
- ◆ Real-time, DSP-based Performance Analysis and Code Coverage systems for Software Quality Assurance
- ◆ Source-Level Debugging for 8051 programs, under Windows and DOS hosts
- ◆ Emulation and probe support for Philips low-voltage microcontrollers and Smart-Card microcontrollers
- ◆ Hardware break-before-execute breakpoints at every code and data address
- ◆ Stand-alone system, with built-in power supply; interchangeable probe cards for all derivatives and packages
- ◆ Programmer for all Philips 8051-family EPROM and EEPROM microcontrollers
- ◆ ISO9001-Certified Supplier; Ashling Microsystems Ltd. is certified to EN ISO9001/ASQC Q91

The Development Systems Company

Ashling

Product Information

Ashling's CTS51 Universal Microprocessor Development System provides a complete hardware and software development environment for the Philips 8051 microcontroller family and all of its derivatives.

In-Circuit Emulator

The CTS51 In-Circuit Emulator provides real-time emulation in both Single-Chip and Expanded modes. Target voltages in the range 2.7 - 5 Volts are supported; the emulation voltage can be set at 3.0, 3.3V or 5.0V, or can track the target system voltage throughout the range.

ICE Probes

In-Circuit emulation probes are available for all device-packages, including DIP-24, DIP-28, DIP-40, PLCC-44, QFP-44, SDIP-42, SDIP-52, PLCC68, QFP-80, QFP-100, and Philips Smart-Card Microcontrollers in ISO7816 or SIM (GSM) card-format.

Devices Supported

Using field-upgradeable personality probes, the CTS51 Universal Development System supports all Philips 8051 derivatives, including:

80C51B/80C31B	80C31/80C52	80CL51	83C550/83C550	80C51FA/FB/FC
83C552/80C552	83C562/80C562		83C053/54 83C654	PCD509x
83C750 83C751 83C752	83C851		83CE598/80CE598	SAA5290
83C592/80C592	83C528/80C528	83C575	83CE558/80CE558	83CL434
83CL410 83CL580	83CL782		83C852/855	83CL168

Support for new Philips microcontrollers

Ashling's close technical collaboration with Philips Semiconductors ensures that new Philips microcontroller devices and device-architectures are supported at, or soon after, their introduction by Philips. New Ashling personality probes are regularly introduced for new standard microcontrollers, and microcontroller-based ASICs from Philips.

Philips Smart Card development support

Developed in co-operation with Philips, Ashling's CTS51 Microprocessor Development System provides a complete development-environment for Philips' 8CC852, 83C852 family of Smart Card Microcontrollers. Probes are available for ISO7816 and GSM (SIM) card formats. Ashling also supplies the SCPC4281 Smart Card Verification kit for stand-alone smart-card program emulation.

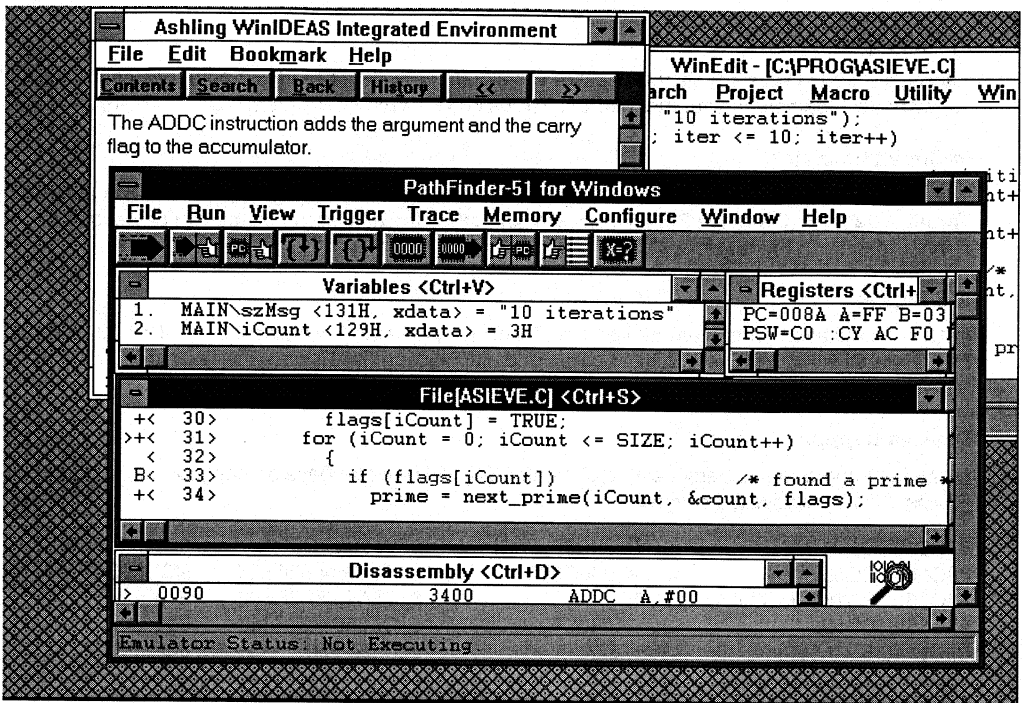
ISO9001 Certification; Quality Management System

Ashling Microsystems Ltd. operates a company-wide Quality Management System, formally certified to the ISO9001 international standard (NSAI Registration No. M619). This certification applies to all of Ashling's product development, manufacturing and customer-support activities.



The Development Systems Company

The Ashling logo is a stylized, italicized script font with a horizontal line underneath the letters.



Ashling WinIDEAS: With the Windows Integrated Development Environment for Ashling Systems supports the full Philips Microcontroller family. A single keystroke moves you between the edit, help-lookup, compile, debug and emulate stages.

Integrated Development Environment

WinIDEAS, the Windows Integrated Development Environment for Ashling Systems, allows you to edit, compile, assemble, simulate, debug, download and execute code on your Philips microcontroller target system in the Windows environment throughout. WinIDEAS provides a uniform, flexible and extensible windows interface for Editing, C Compiling, Assembling, Fuzzy-logic design and compilation, Linking, In-circuit emulation, Performance Analysis, Code Coverage Measurement, Software Validation reporting, and EPROM/EEPROM programming on Philips Microcontroller projects.

System Execution Analyser

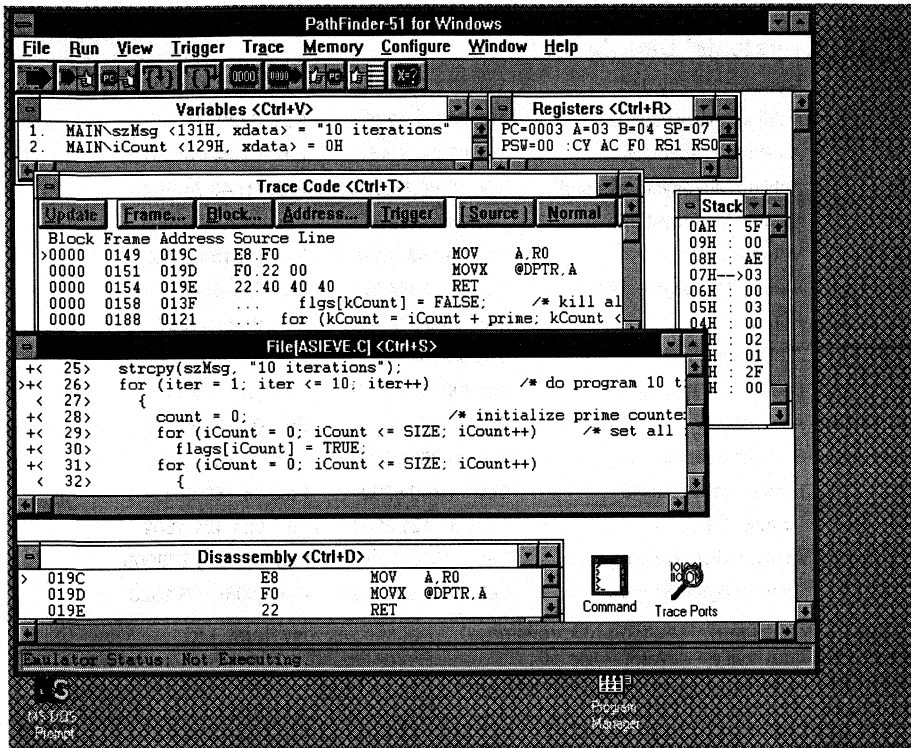
The System Execution Analyser (SEA) is a built-in, DSP-based option for Ashling's Universal Microprocessor Development System. It provides real-time, non-intrusive, non-statistical Performance Analysis, Code Coverage and Report Generation. The SEA also provides symbolic function-trace, time-stamping, timing analysis and software verification. You can measure maximum, minimum and average execution times, execution counts and percentage code execution at Source or Assembly level.

The Development Systems Company



Code Coverage System

The Code Test Management (CTMS) option for the Ashling's Universal Microcontroller Development System measures 8051 code execution in real-time throughout a test session. It maps tested and untested code; identifies all tested, partially-tested or untested C source statements and assembler instructions; generates formal, annotated, test reports; identifies redundant code; and provides a formal measurement of test completeness.



This Pathfinder for Windows screen shows the Source program, Disassembly, Real-Time Trace (source-level, disassembly and opcodes), Variables-watch, On-chip Registers and Stack contents for an 8051 program.

Windows Source-Level Debugger

Ashling's "PathFinder for Windows" Source-Level debugger supports all popular microcontroller C and PL/M Compilers and Assemblers for the Philips microcontroller device family. PathFinder provides up to 20 display-windows, controlled by mouse, menu-bar, command-line, accelerator keys or button-bar. Assembly-language source level debugging is a unique feature of PathFinder for Windows.

The Development Systems Company

Ashling

Microcontroller EPROM/EEPROM Programming

Ashling's CTP51 Prom Programming System is used with the Universal Microprocessor Development System, to program the entire Philips 8751 EPROM/EEPROM microcontroller family. Features include:

- ◆ Programming support for all Philips 8751-family EPROM microcontrollers, including 87C51, 87C51FC, 87C52, 87C528, 87C550, 87C552, 87C592, 87C751 and 87C752; plus all Philips 89C51-family EEPROM microcontrollers, including 89CE528, 89CE558, 89CE598 and SAA5290NV.
- ◆ Software-driven programming data files allow easy upgrade for new Philips devices.
- ◆ Programming Adapters are available for all Philips EPROM/EEPROM microcontroller packages.

Ashling Microsystems' Distributors:

AUSTRALIA:	Metromatics Pty. Ltd.	Tel: 07-3585155	Fax: 07-2541440
AUSTRIA:	Ashling Mikrosysteme	Tel: 08202 1276	Fax: 08202 8745
BELGIUM:	Air Parts Electronics	Tel: 02 241 64 60	Fax: 02 241 81 30
FRANCE:	Ashling Microsystèmes sarl	Tél: (1) 46.66.27.50	Fax: (1) 46.74.99.88
GERMANY:	Ashling Mikrosysteme	Tel: 08202 1276	Fax: 08202 8745
HUNGARY:	Vanguard Kft.	Tel: (1) 156-9000	Fax: (1) 156-8982
IRELAND:	Ashling Microsystems Ltd	Tel: 061-334466	Fax: 061-334477
ISRAEL:	RDT Ltd.	Tel: 03-6450745	Fax: 03-6478908
ITALY:	All-Data s.r.l.	Tel: 02-66015566	Fax: 02-66015577
KOREA:	DaSan Technology	Tel: (02) 501 8277	Fax: (02) 501 8276
NETHERLANDS:	Air Parts b.v.	Tel: 01720-43221	Fax: 01720-20651
SPAIN:	Sistemas Jasper s.l.	Tel: (1) 803 8526	Fax: (1) 803 8526
SWEDEN:	Ferner Elektronik AB	Tel: 08-760 8360	Fax: 08-760 8341
SWITZERLAND:	Litronic AG	Tel: 061 421 3201	Fax: 061 421 1802
U.K.:	Ashling Microsystems Ltd.	Tel: (01628) 773070	Fax: (01628) 773009
U.S.A.:	Eastern Systems Inc	Tel: (508) 366 3220	Fax: (508) 366 1520

Ashling Microsystèmes sarl 2, rue Alexis de Tocqueville Parc d'Activités Antony 2 92183 ANTONY - France. Tél: (1) 46.66.27.50 Télécopieur: (1) 46.74.99.88	Ashling Mikrosysteme Waldstraße 18 86510 Ried-Baindlkirch Germany. Tel: 08202-1276 Fax: 08202-8745	Ashling Microsystems Ltd. Butler House, Market St. Maidenhead Berks. SL6 8AA, U.K Tel: (01628) 773070 Fax: (01628) 773009
---	---	--

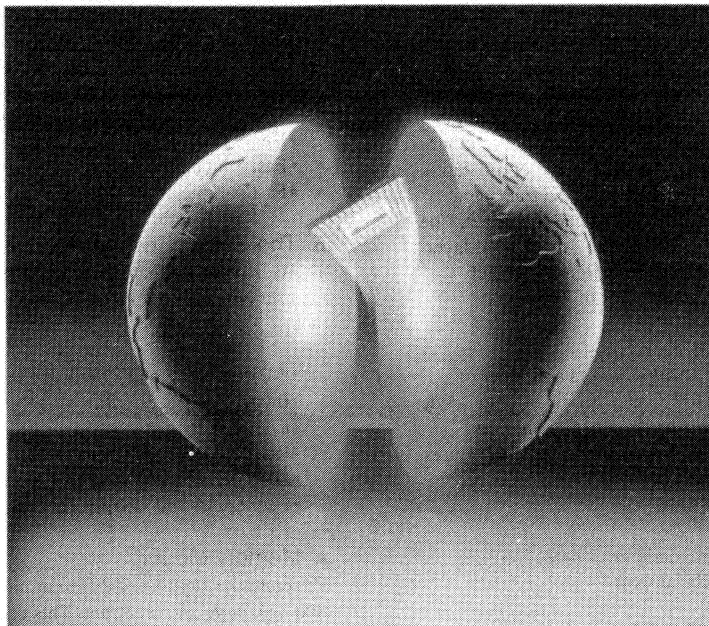
Ashling Microsystems Ltd Plassey Technological Park Limerick Ireland Tel: +353-61-334466 Fax: +353-61-334477	Eastern Systems Inc. 160 East Main Street Westboro MA 01581, USA Tel: (508) 366 3220 Fax: (508) 366 1520
---	---

The Development Systems Company



doc:philipdb.doc/cb Oct 94 ver. 1.0

The Total Development Solution for the 8051 Family



The BSO/TASKING C-51 cross-compiler offers a new approach to high-level language programming for the 8051 family. It combines our 15+ years of experience in producing software tools for the embedded marketplace and our considerable expertise with the 8051 microcontroller.

Although intended as a general purpose programming language, C is perhaps most powerful in the area of real-time programming for embedded microcontrollers. Since the operators and type definitions match so well with the instruction sets and word lengths of microcontrollers, C is very efficient in terms of code size and execution speed for these devices.

C Compiler

Efficient ANSI C compiler with powerful 8051 extensions for all derivatives

Assembler

Intel compatible macro assembler and linker/locator with libraries supplied in source

CrossView Debugger

C and assembly level debugging with a full Windows interface

the C Compiler

The BSO/TASKING 8051 C compiler is built using the latest compiler technology, including optimization and multi-layering. It is essentially a one pass compiler, translating on a function by function basis, achieving very fast compilation and a large span for the optimization process. Significant effort was made to obtain a well-defined, layered product.

The top layer basically consists of a lexical and grammar analyzer with co-routine structures to serve both the preprocessing and compilation needs. The grammar analyzer is produced by an LALR(1) parser generator, eliminating a potential source of error. A second layer separates the actual code-generation layer from the top layer and simplifies retargeting of the compiler to other target processors. This is where most of the optimizations are done and it is designed to represent the translated program independent of the language and the target. The code-generation layer is driven both by the second layer and by a set of tables and rules reflecting the target assembly language and the behavior of individual instructions.

The 8051 family ANSI C compiler uses a number of sophisticated techniques to produce highly efficient code. The features that ANSI has brought to C compilers, together with the BSO/TASKING 8051 specific features and extensions brings a new dimension to programming the 8051 microcontroller family:

- Full ANSI C to ensure early error detection
- Complete 8051 family support
- In-line assembly
- Single precision floating point
- Memory keywords: data, idat, pdat, xdat and rom
- C level access to special function registers
- C level bit, interrupt and bit addressable 'chars' & 'ints'
- Automatic register bank switching
- Data overlay mechanism
- Powerful extensions including:
 - bit-type to use the on-chip bit addressable area
 - interrupt <number> and using <registerbank> for interrupt servicing at C-level
- Generates Intel compatible assembly source
- Outputs symbol information for source level debugging
- 4 memory models to best meet application requirements
- Complete C and run-time library support,
- Generates reentrant and relocatable code and data
- Full Intel OMF51 and IEEE695 v4.1 object formats
- Full calling interface with parameter passing to our PL/M51 compiler
- Intelligent configuration of startup code
- Keyword *_at* to connect a global or static variable to a logical address

▲ Efficiency

The C compiler has a number of features which make code production very efficient. The compiler also makes use of 8051 memory very efficiently.

- Function prototyping gives the user control over the size of parameters
- Parameter passing between functions is done using static 'fast internal RAM'
- Pointer sizes are automatically determined
- The compiler recognizes automatic variables and register variables that are used repeatedly and stores them in internal RAM
- In-line expansion of predefined functions such as 'rol', 'ror', 'da', 'testclear', 'setbit', 'putbit'
- Character 'switch' statement produces fast inline code
- Very efficient pointer arithmetic
- Fast 8 bit char arithmetic
- Fast parameter passing by avoiding the stack.

▲ The Target Environment

There are specific features in the 8051 C compiler that let the user tailor the output to the specific target environment

- 4 Different memory models - see next page
- Interrupt service routines can be written in C or assembly language
- ROMable code
- 8051 derivative is switch selectable
- User controlled mapping of code and data

▲ Memory Models

The compiler supports 4 different memory models: small, auxpage, large and reentrant. This enables the compiler to generate the best possible code for the particular hardware implementation. See the Memory Model table.

▲ Data Types

All ANSI types are supported. In addition to these types, bit, sfrbyte, sfrbit and bitbyte are added. The keywords sfrbyte and sfrbit are available to access special function registers which deal with I/O. Sfr's are treated like variables declared with the volatile type qualifier.

▲ Data Sizes

<u>type</u>	<u>size (in bytes)</u>
bit	1 bit
char	1
short int	2
int	2
long int	4
float, (long) double	4 (IEEE single precision)
pointer type	1 or 2

Pointers to data, idata and pdat have a size of 1 byte, but pointers to rom, xdat and functions have a size of 2 bytes.

▲ Code Optimizations

The compiler intermediate layer performs general code optimizations and the code generator does some final 8051 specific optimizations.

General optimizations include:

- Register allocation
- Branch optimization
- Dead code elimination
- Constant folding
- Arithmetic simplification
- Strength reduction
- Jump chaining
- Common subexpression elimination
- Loop rotation
- Induction variable elimination
- Sharing of string literals

Code generator optimizations:

- Store-copy optimization
- Peephole optimization

▲ Pragmas

The #pragma directive supplies target dependent data to the compiler without violating the ANSI C language, making the resultant program portable. In the BSO/TASKING C 8051 compiler pragmas are used to merge C lines with generated assembly, to control code generation for interrupt service routines and to support inline assembly programming.

▲ Libraries

The compiler is delivered with libraries for all combinations of the 8051 family. The libraries include ANSI C libraries, run time libraries including I/O calls (+ printf), memory management, arithmetic functions and floating point. Special libraries are included for floating point using internal RAM only. There is also a separate library to support the 80C751 derivative.

Source code is provided for most of the library routines allowing the user to tailor the libraries to their specific application.

▲ Compiler Switches

- ? Display invocation syntax
- Ccpu Use special function register definitions for cpu
- Dmc[=df] Define preprocessor macro

- E[m] Preprocess only or emit dependencies
- Hfile Include file before starting compilation
- Idirectory Look in directory for include files
- Ms Compile using small memory model
- Ma Compile using auxpage memory model
- Ml Compile using large memory model
- Mr Compile using reentrant memory model
- N Check for use of new style prototyping
- Oa Relax alias checking
- OA Strict alias checking
- Oc Enable common subexpression elimination (CSE)
- OC Disable CSE
- Od Enable constant and copy propagation
- OD Disable constant and copy propagation
- Of Produce fast code
- OF Produce small code
- Og Enable global CSE
- OG Disable global CSE
- Oo Overlay data
- OO Do not overlay data
- Ov Enable loop variable detection
- OV Disable variable detection
- Rm[=nm] Change segment name
- S Put strings in ROM only
- _Umac Remove preprocessor macro
- asize Change allocated space for variable argument list
- bnumber Specify default register bank number
- e Remove output file if compilation errors occur
- err Send diagnostics to error list file
- f file Read options from file
- g Enable symbolic debug information
- gt Enable symbolic debug information with function prototypes
- go Enable symbolic debug information for older debugger versions
- l Generate listfile
- li Generate listfile with expanded include files
- mm=size Specify memory size
- n Send output to standard output
- o file Specify output file name
- rs Select small ROM model
- rm Select medium ROM model
- rl Select large ROM model
- s Merge C source code with assembly output
- t Display module summary
- u Treat 'char' variables as unsigned
- v Don't generate interrupt vectors
- w[num] Suppress warning messages, or just one indicated by its number
- xsize Extend amount of internal RAM for automatics

▲ Memory Models

<u>model</u>	<u>data allocation</u>
small	static, in on-chip direct addressable RAM (data)
auxpage	static, in first page of external RAM (pdat)
large	static, in external RAM (xdat)
reentrant	dynamic + static, in external RAM (xdat)

<u>application</u>
fast programs in small environments
derivatives with 256 bytes on-chip "external" RAM
fast, non reentrant with large external RAM
large, reentrant programs in large environments

the Assembler

The BSO/TASKING assembler is an integral part of the toolset and delivers features that enable it to be used on its own. It is supplied complete with linker/locator, librarian and object format utilities.

The 8051 assembler translates 8051 assembly language into relocatable object code. The assembler accepts Intel compatible assembler source programs and produces relocatable (.obj) object files. An absolute or executable load image is then obtained by using the linker/locator. Features of the assembler include:

- Produces relocatable object code and listing files
- Accepts Intel compatible source programs
- Supports all members of the 8051 family
- Compatible with BSO/TASKING C and PL/M 8051 compilers
- Compatible with high level and assembly level debuggers
- Optimizes jmp/call instructions from the compiler
- Supports segment overlay at the assembly level
- Intel compatible macro preprocessor
- Number of symbols only limited by available host memory
- Extensive section directives
- Error file with textual error reporting

the Linker/Locator

The linker and locator is an essential part of the software building process that enables you to configure the code to match your target environment. The linker/locator brings together all the necessary relocatable objects, including library modules, resolves external references and then locates the modules in memory according to your specification. Features include:

- Intel compatible linker controls
- Accepts object files and object libraries in the Intel OMF-51 file format.
- Supports absolute Intel OMF-51 and IEEE695 object formats
- Automatic segment overlaying using information from the compiler and assembler
- Absolute map files and diagnostic messages
- Incremental linking
- Automatic inclusion of library modules
- Map listing to help with debugging

Utilities

▲ Librarian

Listing utility with numerous options for showing:

- section records
- symbol table
- code bytes and much more

Format conversion utilities

▲ EDE

BSO/TASKING's new embedded development environment for the PC which includes:

- make utility
- C interface generator
- line searcher and sorter
- interface to INTERSOLV PVCS

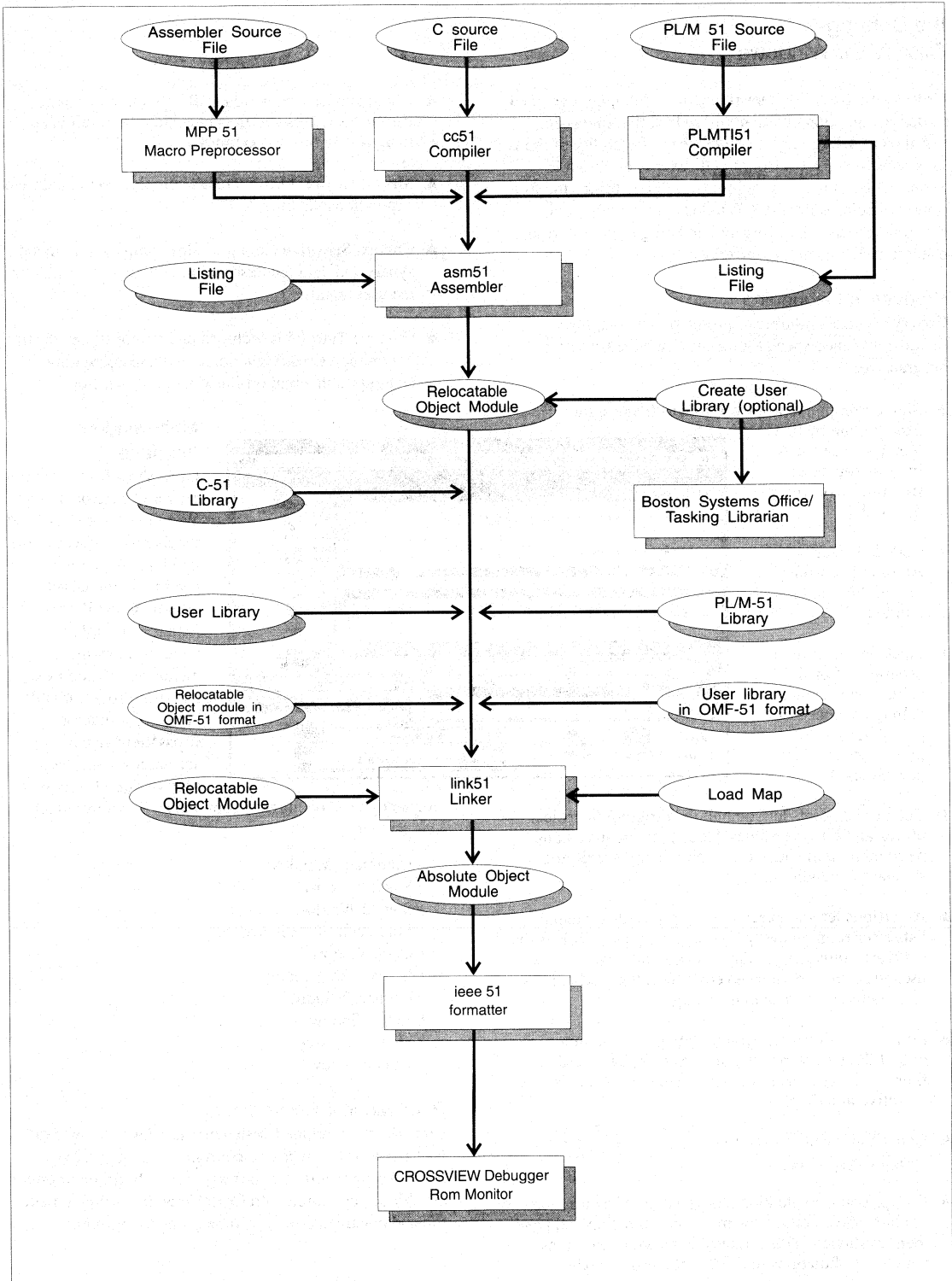
Availability

C compiler and Assembler are available now to run on:

PC DOS	SUN SPARC SUNOS
HP9000 HPUX	SUN SPARC Solaris
VAX VMS	IBM RS6000 AIX

Available from Philips as:

OM4144	80C51 Cross-Assembler / PL/M-Compiler package for PC/DOS
OM4285	80C51 Cross-Assembler / C-Compiler package for PC/DOS
OM4286	Cross View PC/DOS – MS/Windows Debugger for SDS80C51 (OM4120S)



8051 Tool-chain overview

the Debugger CrossView Windows

CrossView is our high level language debugger designed to deliver functionality that will reduce the time spent testing and debugging. It combines the flexibility of the C language with the control of code execution found in assembly language debugging. CrossView brings the full power of Microsoft and X Windows to the debugging environment by displaying and updating the most critical execution data in an organized way.

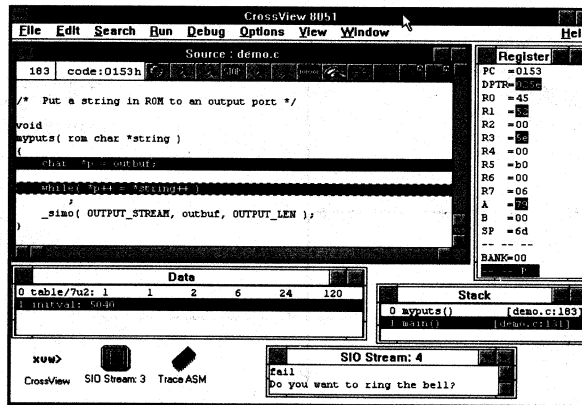
Productive Debugging

CrossView has a number of productive debugging features designed specially for the embedded systems programmer:

- ▲ Single Stepping allows you to watch a program execute line by line. You can step single lines of source or, step in or over procedure calls.

- ▲ Code in C Debug in C. Automatic correlation of variables and line numbers

- ▲ Stack Trace shows the program's function calling path. Function calling sequence and parameter values passed are displayed at each level.



- ▲ Code breakpoints let you halt the program in critical places and observe values. Code breakpoints can be permanent, temporary or conditional by specifying standard C conditions
- ▲ Assertions let you execute user specified command lists after running every line of source code. This is the software equivalent of data breakpoints, they can be used to set up sophisticated error checking mechanisms that uncover the most elusive bug.
- ▲ Simulated I/O enables you to debug without input/output devices being available. Screen, keyboard or files can be used as I/O devices and eight streams can be active at a time.
- ▲ Data Monitoring is used to monitor any expression or variable continually.
- ▲ C Expression Evaluation lets you enter C expressions or CrossView commands in any combination and have them evaluated. You can even evaluate expressions which call functions defined in the source code.

- ▲ Command Line Function Calls used to pass sample parameter values and then checking the result is an ideal way to test subroutines.
- ▲ Macros let you store and recall complex commands and expressions via buttons.
- ▲ Context Sensitive Help provides complete command syntax and detailed description with hypertext links to the user manual.
- ▲ On-Line Tutorial is included and demonstrates the use of common CrossView command and debugging techniques in a real on-line debugging session.

Multi-window Interface

CrossView brings the full power of Microsoft and X Windows to organize and display every facet of the application as it is running in the target system. CrossView features multiple child windows, extensive menus and dialog boxes and user programmable buttons for macros. CrossView also has an accelerator bar which provides quick access to

frequently used debugging commands. Windows available include:

- Command Window
- Stack Window
- Source Window
- Simulated I/O Window
- Data Window
- Breakpoint Window
- Register Window
- Help Window
- Macro Window
- Trace window

ROM Monitor Environment

CrossView is supplied with a monitor for loading into RAM or ROM and can be configured for any 8051 family target board. All files required to build the monitor are shipped as source with CrossView, including a make file and documentation for retargeting the monitor .

The monitor program is approximately 3Kb. The internal data requirements are kept to a minimum: it only uses register bank 3, 20 bytes of stack space and 12 bits in bit addressable memory. All other monitor data (74 bytes) is stored in external memory

CrossView communicates with the monitor on the target board via an RS232 interface. It is possible to communicate directly with the ROM monitor with a terminal only, or from within CrossView via transparency mode.

Transparency Mode

CrossView enables direct communication with the target to give you access to the low level hardware features of the target system. There are 35 monitor commands which manipulate the registers, memory control execution.

Availability

CrossView Windows for 8051 will be available to run on:

PC DOS	SUN SPARC	SUNOS
HP9000 HPUX	SUN SPARC	Solaris
VAX VMS	IBM RS6000	AIX

Customer Support

Purchasing BSO/TASKING products marks the beginning of a long term relationship. BSO/TASKING is dedicated to providing quality products and support worldwide. This support includes program quality control, product update service and support personnel to answer questions by telephone or fax.

BSO/TASKING product support begins before the purchase of any of our products with extensive testing in order to ensure our high standard of quality assurance.

A 90 day maintenance plan is included with the purchase of BSO/TASKING products and entitles the customer to enhancements and improvements as well as individual response to problems. Annual maintenance contracts are available at the end of the 90 day maintenance plan. This extremely valuable service, in return for a small annual fee, provides the user with all program enhancements released during the period of the program maintenance agreement, and assures response to all problem reports submitted by the user.

BSO/TASKING

BSO/TASKING is an international software manufacturing company with headquarters in Massachusetts. Founded in 1974, BSO/TASKING is privately held with 100+ employees and revenues of \$19M. BSO/TASKING has offices in the United States, the Netherlands, Italy, Japan, Germany and the United Kingdom.

BSO/TASKING develops, manufactures, and supports software development tools for DOS, Unix and VMS programming environments. In 1974, the company pioneered the concept of 'cross development' on DEC platforms.

Today, BSO/TASKING is the leading supplier of software development tools for embedded and system applications across industry standard computing platforms. Software Products from BSO/TASKING are developed with great care at our innovative "Software Factory". A fully integrated environment for software development, modeled after and compatible with the standards defined in the ESPRIT project PCTE (Portable Common Tool Environment). This paradigm facilitates the creation of tools that execute in multiple environments (DOS, Unix, VMS) from a single master source file that is then verified with the Plum Hall Test Suite.

To complement the software development tools BSO/TASKING delivers cost effective hardware systems - in-circuit emulators and evaluation boards - integrated with our software to provide you with a Total Solution for all your embedded development needs. BSO/TASKING also provides consulting and training services to complement each product area.

BSO/TASKING supports products on PC/DOS, Sun SPARC, DEC VMS, DECStation/Ultrix, IBM RS6000, HP9000/300,400,700.

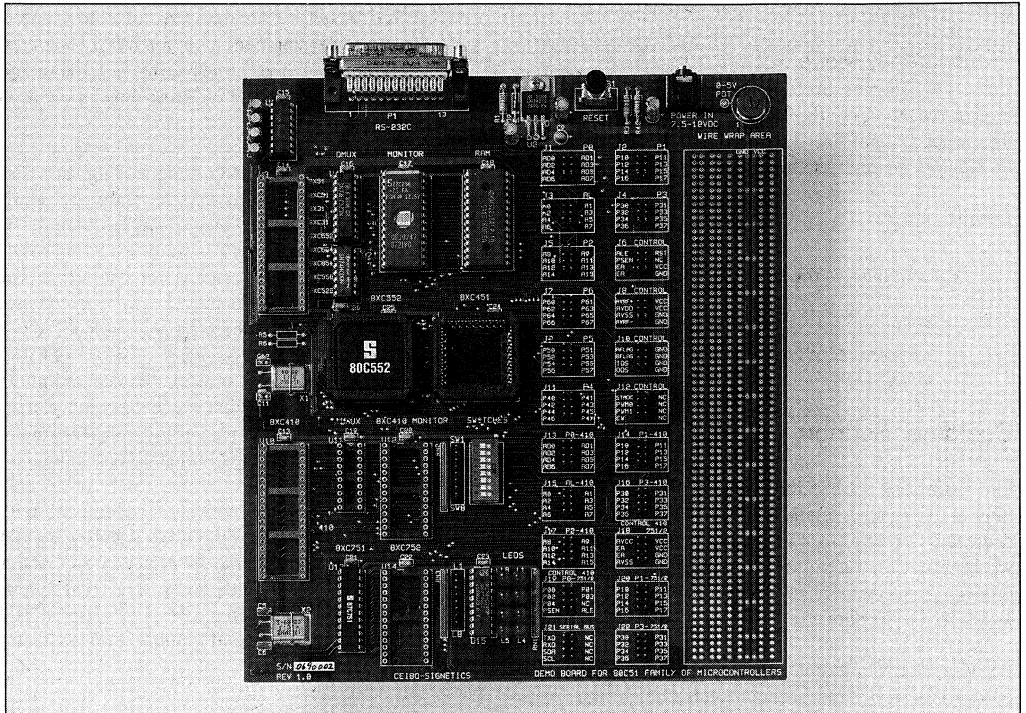
The company mission is to provide customers with highly integrated, leading-edge cross software development tools across industry standard computing platforms, coupled with the most comprehensive support and training services available.

Boston Systems Office, BSO, Boston Systems Office/TASKING, BSO/TASKING and CrossView are registered trademarks of Boston Systems Office Inc and TASKING B.V.

BSO/Tasking assumes no responsibility for any errors which may appear in this document.

BSO/TASKING retains the right to make changes to the specification at any time, without notice. Contact your local sales office to obtain the latest information.

CEIBO DB-51 Development Board



Development Board for Philips 80C51 Microcontrollers

FEATURES

- Supports Most of the Philips 80C51 Microcontrollers
- Serially linked to IBM PC or Compatible Hosts
- 32K of User Code Memory
- Software Breakpoints
- Examine and Alter Chip Registers, RAM and Ports
- Source Level Debugger
- Upload and Download of Object and Hex Files
- Special Wire-Wrap Area for Prototyping
- User's Manual with Examples and Applications

CEIBO DB-51 Development Board

DB-51 is a high-performance system design board dedicated to the Philips 80C51 family of microcontrollers. It provides an easy-to-use flexible instrument which enables the user to build a primary prototype, analyze and debug it, make changes and continue debugging. You can improve your design decisions by using the DB-51 to check and test the advantages of several different microcontrollers. The DB-51 is also a great training and tutorial aid for becoming familiar with designs using the 80C51 architecture. Note that the DB-51 is not intended to replace a full emulator system in complex microcontroller designs.

SPECIFICATIONS

SYSTEM MEMORY

DB-51 provides 32K of user code memory. This RAM memory permits downloading and modifying of user's programs.

BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address.

CONDITIONAL BREAKPOINTS

A complete set of conditional breakpoints permits halting program emulation on code addresses, source code lines, access to on-chip memory, ports and register contents.

USER SOFTWARE

The board is provided with a very easy-to-use menu-driven software program as well as command orientated user interface software. On-line Assembler and Disassembler are provided together with upload and download capabilities of hexadecimal and object files. Also, DB-51 includes the following functions: Source Level Debugger for PLM and C, Software Trace, Conditional Breakpoints, Performance Analyzer and a unique Assembler Debugger.

SOURCE LEVEL DEBUGGER

DB-51 gives full support for debugging directly in PLM and C source code. From your code source you can specify a breakpoint, execute a line step or an assembly instruction, open a flexible-in-size watch window to display any variable, use the function keys to display the trace memory, registers and data, redefine the Program Counter and reset the microcontroller.

SOFTWARE ANALYZER

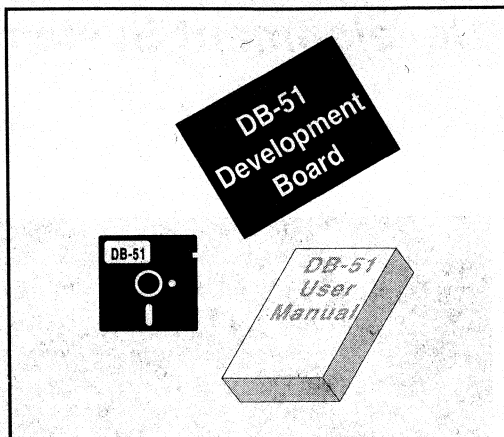
A 64 KByte buffer is used to record any software and hardware events of your program, such as executed code, memory accesses, port and internal register states, on-chip data memory and others. The trace buffer can be viewed in disassembled symbolics or high-level language source code.

LANGUAGES AND FILE FORMATS

DB-51 accepts files generated by Intel software (Assembler, PLM) or compatibles in hex or object formats. Other assemblers and high-level languages such as C with Intel OMF51 compatible format (Franklin, Archimedes, IAR, etc.) are also supported.

SUPPORTED MICROCONTROLLERS

8x31/51, 8x32/52, 8xC31/51, 8xC32/52, 8xC652, 8xC654, 8xC851, 8xC550, 8xC552, 8xC562, 8xC451, 8xC528 and others with external memory addressing and a UART are fully supported. 8xCL410, 8xC751 and 8xC752 have very limited support.



LIMITATIONS

"Fully supported" microcontrollers are self-debugging on the DB-51. Thus, some of the chip resources are used by the board: the monitor program uses the bottom 32K of program memory; chips are always operated in the external memory mode; the UART is used to communicate with the PC and is not normally available to the user program; interrupt response is slowed slightly by re-vectoring from the monitor program to the user program; use of watchdog timers and power-down and idle modes of operation are limited due to interaction with the monitor program. "Limited support" microcontrollers do not have on-chip UARTs and most do not support external program memory. Thus, download of programs to these parts is not supported on the DB-51. The 87C751 provided with the board is pre-programmed with a "micro" monitor program and some predefined experiments described in the user manual. Also, these parts use the I²C bus to communicate with the PC, limiting the use of I²C for other purposes.

INPUT POWER

7.5 VDC to 12.0 VDC (9 VDC wall transformer supplied).

MECHANICAL DIMENSIONS

20cm x 25cm.

ITEMS SUPPLIED AS STANDARD

DB-51 board, 80C552 and 87C751 microcontrollers, monitor EPROM, power supply, RS-232 cable. User software including Symbolic Debugger, On-line Assembler and Disassembler. User's Manual and Operating Instructions.

Ordering Information

Part No. P8051DB SD/OM4238
Available through Philips distributors.
12NC:9350-360-40112

Product and Company names are trademarks of their respective organizations.

CEIBO DS-51 Microprocessor Development System



In-Circuit Emulator for 8051 Family of Microcontrollers

FEATURES

- Real-Time and Transparent In-Circuit Emulator
- Supports Most of the 8051 Derivatives
- Emulates 1.5V to 6V Microcontrollers
- Maximum Frequency of 42MHz
- 128K of Internal Memory
- 32K Trace Memory and Logic Analyzer
- 64K Hardware and Conditional Breakpoints
- Source Level Debugger for Assembler, PLM and C
- On-line Assembler and Disassembler
- Performance Analyzer
- Serially linked to IBM PC at 115 KBaud

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

EIBO DS-51 In-Circuit Emulator

DESCRIPTION

S-51 is a real-time in-circuit emulator dedicated to the 8051 family of microcontrollers. It is serially linked to a D/XT/AT or compatible systems and carries out a transparent emulation on the target microcontroller.

The system emulates almost every 8051 derivative in the complete voltage and frequency range specified by the microcontroller manufacturer.

S-51 also supports the new low-power and low-voltage 8051 microcontrollers and derivatives and can emulate the microcontrollers using either the built-in 5V power supply or any voltage applied to the target circuitry. The permitted voltage range is from 1.5V to 6V or higher.

The software includes Source Level Debugger for PLM and C, a unique Assembler Debugger, Performance analyzer, On-line Assembler and Disassembler, Conditional Breakpoints and many other features.

Files generated by the most common 8051 Assemblers and High-Level Language Compilers are accepted by S-51.

Standard systems are supplied with 128 KBytes of internal memory, 64K hardware breakpoints, 32K real-time trace memory and logic analyzer with external test points, and personality probe C51 supporting most of the 40-pin DIP or 44-pin PLCC/QFP microcontrollers.

SUPPORTED DEVICES

ROBE MICROCONTROLLER

51	8051, 8751, 8031, 8052, 8752, 8032, 80C51, 87C51, 80C31, 80C52, 87C52, 80C32, 87C51FA, 87C51FB, 87C51FC, 87C54, 87C58, 83C504, 87C504, 83C524, 87C524, 80C528, 83C528, 87C528, 80C550, 83C550, 87C550, 80C652, 83C652, 87C652, 83C654, 87C654.
168	83CL167, 83CL168, 83CL267, 83CL268.
410	80CL31, 80CL51, 80CL410, 83CL410, from 1.5V to 6V.
434	83CL434, 83CL834, 87CL134.
451	80C451, 83C451, 87C451, 87C453.
552	80C552, 83C552, 87C552, 80C562, 83C562.
556	80C558, 83C558, 89C558, 83C559, 89C559.
575	80C575, 83C575, 87C575, 83C576, 87C576.
580	80CL580, 83CL580 from 1.5V to 6V.
592	80C592, 83C592, 87C592.
598	80C598, 83C598, 87C598.
752	83C748, 87C748, 83C749, 87C749, 83C750, 87C750, 83C751, 87C751, 83C752, 87C752.
781	83CL781, 83CL782, 80CL32, 80CL52 from 1.5V to 6V.
055	83C054, 87C054, 83C055, 87C055

Consult Ceibo for other Supported microcontrollers.

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

CEIBO DS-51 In-Circuit Emulator

SPECIFICATIONS

EMULATOR MEMORY

DS-51 provides 128 KBytes of code memory with software mapping and banking capabilities

HARDWARE BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address. Breakpoints on data read or write and an AND/OR combination of two external signals are also implemented

CONDITIONAL BREAKPOINTS

A complete set of conditional breakpoints permit halting program emulation on code addresses, source code lines, access to external and on-chip memory, port and register contents.

SOFTWARE ANALYZER

A 64 KByte buffer is used to record any software and hardware events of your program, such as executed code memory accesses, port and internal register states, external or on-chip data memory contents and more

LANGUAGES AND FILE FORMATS

DS-51 accepts files with Intel OMF51 object or hex format. Assemblers and high-level languages such as C with OMF51 format (Intel, Franklin, Archimedes, IAR, MCC, BSO/Tasking, etc.) are also supported

SOURCE LEVEL DEBUGGER

The DS-51 Software includes a Source Level Debugger. This function may be used to debug code written in Assembler, PLM and C. The Source Level Debugger includes commands which allow the user to get all the information necessary for testing the programs and hardware in real-time. The commands permit setting breakpoints on high-level language lines, adding a watch window with the symbols and variables of interest, modifying variables, displaying floating point values, showing the trace buffer, executing assembly steps and many more useful functions.

PERSONALITY PROBES

DS-51 uses standard and bond-out microcontrollers for hardware and software emulation. The selection of a different microcontroller is made by replacing the microcontroller on the probe or changing the probe. The Personality Probes run at the frequency of the crystal on them or from the clock source supplied by the user hardware. Therefore, the same probe may be adapted to your frequency requirements. The minimum and maximum frequencies are determined by the emulated chip characteristics, while the emulator maximum frequency is 42MHz.

TRACE AND LOGIC ANALYZER

The 32 KByte trace memory is used to record the microprocessor activities. Eight lines are user selectable test points. Trigger inputs and conditions are available for starting and stopping the trace recording. The trace buffer can be viewed in disassembled instructions or high level language lines embedded with the related instructions.

ITEMS SUPPLIED AS STANDARD

In-circuit emulator with 64 KByte breakpoints, 128 KByte internal code memory. Personality probe C51 for 8051 microcontrollers. User software including Source Level Debugger, On-line Assembler and Disassembler. User's Manual and Operating Instructions. RS-232 cable. Power supply.

OPTIONS

Personality probes for the different microcontrollers. Memory bank setup. Adapter for 44-pin PLCC devices.

PERFORMANCE ANALYZER

This useful function checks the software trace buffer and provides time statistics on modules and procedures as a percentage of the total execution time.

COMMAND SET

The available functions include: FILE (load, save), VIEW (watches, variables, module, cpu, dump, registers, trace, file), RUN (run, go, trace into, step over, execute, animate, halt, reset), BREAKPOINTS (toggle, expression true global, hardware breakpoint, delete all), DATA (inspect, evaluate, add watch), OPTIONS (environment, path, communications, architecture, mode, save, load), WINDOW (zoom, next, size, move, close), HELP (index, previous topic).

INPUT POWER

5VDC/1.5A.

MECHANICAL DIMENSIONS

26mm x 151mm x 195mm (1" x 6" x 7").

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

CEIBO DS-51 In-Circuit Emulator

TRACE CAPABILITIES

External Trace Start/Stop Triggers

DS-51 has two External Trigger signals that allow starting and stopping of the trace recording upon external events.

Stop Trace when Full or Continuous Recording Mode

There are two trace recording modes: Cyclic and Trace Full. In the Cyclic Mode the trace is continuously filled with recorded data. In the Full Mode recording stops when trace is full.

Selectable Trace Trigger Levels

The trigger state permits selecting the way trigger signals behave. The active mode may be either level or edge for the external start and stop trigger signals.

Trace Status on the Fly

Trace Status allows viewing information without stopping emulation. The Trace Status includes buffer full, buffer empty, length, etc.

Trace Filtering on Address Ranges

Up to 10 different ranges are allowed to filter the recorded data into the trace memory. The start and stop addresses of modules and procedures can be entered using special prefixes.

The screenshot shows the 'Ceibo 51 Windows Debugger' interface. The main window displays a 'TRACE DUMP' of assembly code. A smaller 'Trace Dump' window is overlaid on top, showing register values for various EINIT instructions.

Main Trace Dump:

```
(32D7h) 2CB8h  cpl  ABORT_FLAG
EINIT#404  int_hmsec_flag = FALSE; /* Clear 100 mSec interrupt flag */
(32D8h) 2CBCh  clr  INT_HMSEC_FLAG
EINIT#405  int_msec_flag = FALSE; /* Clear 1 mSec interrupt flag */
(32D9h) 2CBEh  clr  INT_MSEC_FLAG
EINIT#406  sw_hmsec_flag = FALSE; /* Clear 100 mSec software loop */
(32DAh) 2CC0h  clr  SW_HMSEC_FLAG
EINIT#407  ds_state_powerup = FALSE;
(32DBh) 2CC2h  clr  00h
EINIT#408  status_flag = FALSE;
(32DCh) 2CC4h  clr  MON_ONCH
EINIT#409  status_flag = FALSE;
(32DDh) 2CC6h  clr  MON_ONCH
EINIT#410  ds_state_flag = FALSE;
(32DEh) 2CC8h  clr  0Eh
EINIT#411  status_flag = FALSE;
(32DFh) 2CCAh  clr  MON_ONCH
EINIT#412  error_no = 0;
(32E0h) 2CCCh  mov  ERROR_NO, 0
EINIT#413  time_base = 0;
(32E1h) 2CCFh  mov  TIME_BASE, 0
EINIT#414  time_base = 0;
(32E2h) 2CD2h  mov  TIME_BASE, 0
EINIT#415  Dis_CBus; /* Enable access to LCD */
(32E3h) 2CD5h  setb P1.3
```

Trace Dump Window:

```
(32CDh) EINIT#393  IP = 00001010b
(32CEh) EINIT#394  TMOD = 0010000b
(32CFh) EINIT#395  TCON = 0000010b
(32D0h) EINIT#397  T2CON = 0011000b
(32D1h) EINIT#398  TL0 = LOW(Timer)
(32D2h) EINIT#399  TH0 = HIGH(Timer)
(32D3h) EINIT#400  SCON = 0101000b
(32D4h) EINIT#401  PCON = 1000000b
(32D5h) EINIT#402  prob_time_out = 0
(32D6h) EINIT#403  abort_flag = FALSE
(32D7h) EINIT#404  int_hmsec_flag = FALSE
```

Debugger status bar: 155.1 | 80C32/52 | Sim | Ready

CEIBO DS-51 In-Circuit Emulator

BREAKPOINT OPTIONS

Breakpoint on Data Read/Write and Address

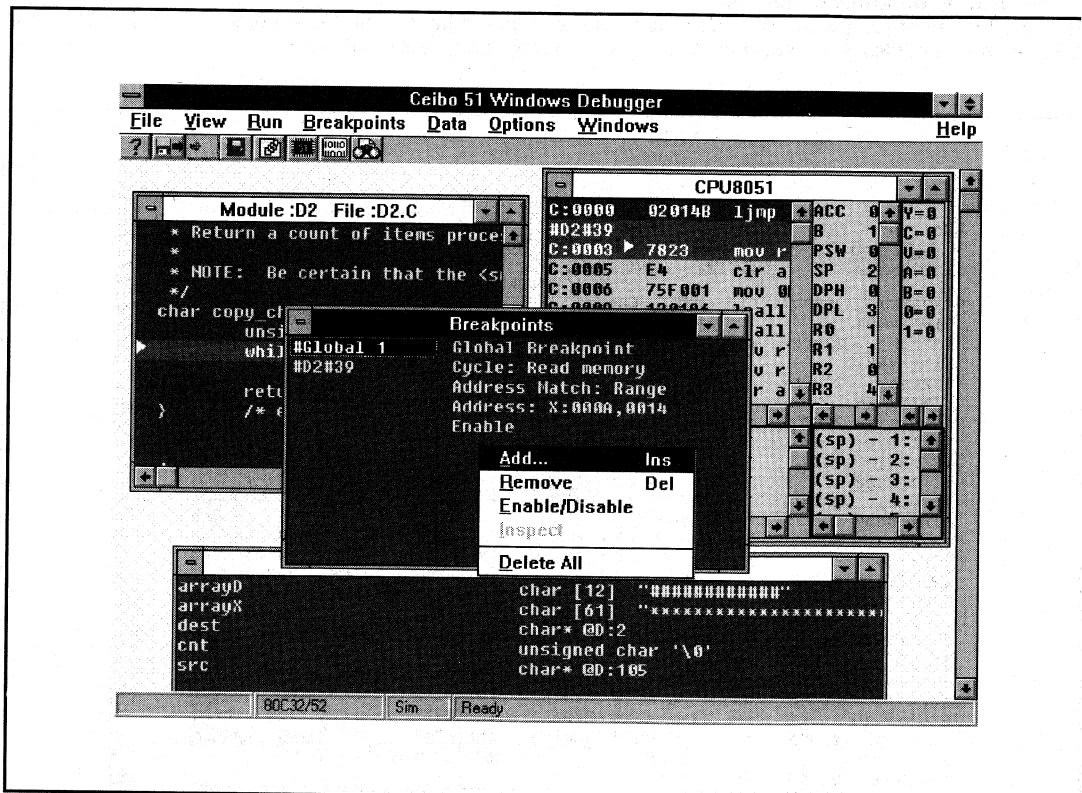
Real-time Hardware Breakpoints can be set on Data Read, Data Write, Data Read or Write and Opcode Fetch in any address range.

Stop Emulation on Pass Count

Pass Count on any event including opcode may be specified by the number of occurrences.

Breakpoint on External Events

Trace Testpoints may be used as external Breakpoints on Hardware events if enabled. The External Breakpoint triggers may also be the Trace Full condition, then program execution will stop once it is full. The Logic Operator commands allow selection of AND/OR combinations of the two external breakpoints. The options are both low, any one is low, both are high, only one is high, both are leading edge, etc



CEIBO DS-51 In-Circuit Emulator

ABOUT THE DEBUGGER

Environment

C51D is a menu-driven program supplied with DS-51. This debugger runs either under DOS or Windows. C51D Debugger is used to load a program, execute it in real-time or simulate the software environment and many other functions.

Tracing

A program may be executed one line at a time. You can trace a program using high-level language lines or assembly instructions.

Stepping

This is like tracing but program execution steps over CALL instructions without leaving the current procedure.

Viewing

C51D Debugger allows special windows to be opened showing your program state from various perspectives: variables and their values, breakpoints, a text file, a source file, CPU registers, memory, peripheral registers, etc.

Inspecting

The debugger can delve deeper into your program and show you the variable contents.

Changing

The current value of a variable can be replaced with your specified value.

Watching

Program variables can be isolated and their values kept track of while the program runs.

Global Menus

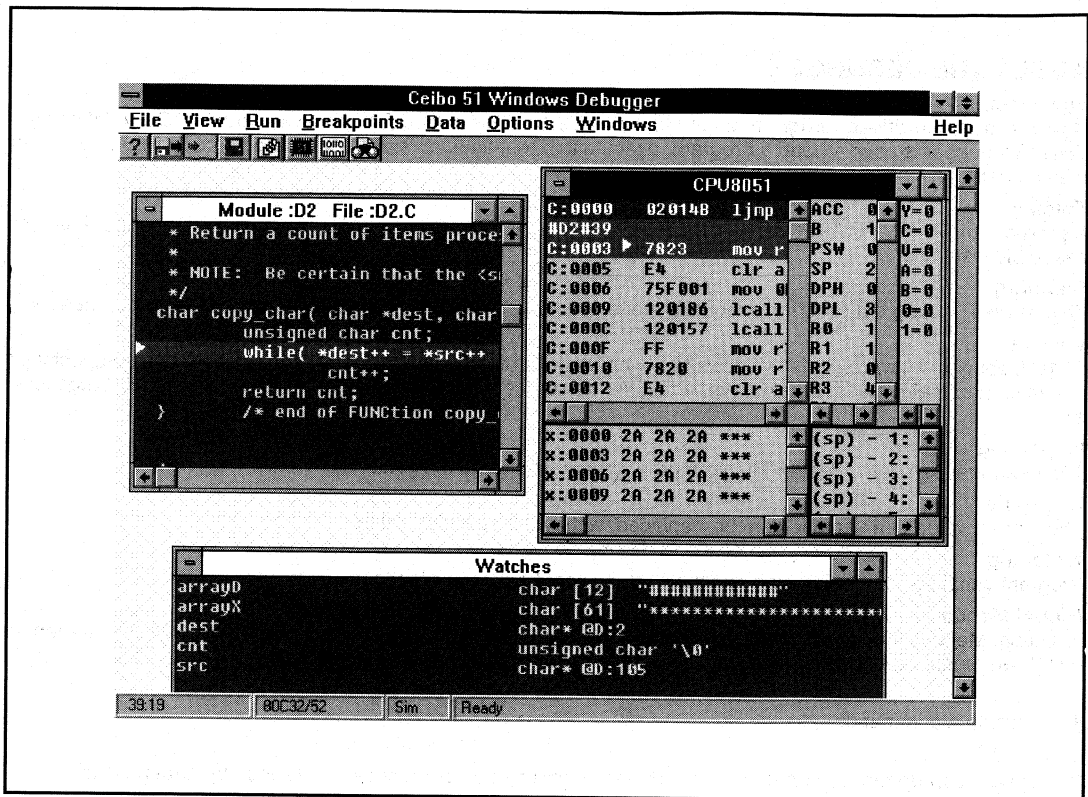
A Global Menu is the list of commands easily accessible from a bar which runs along the top of the screen. A pull-down menu is available for each item on the menu bar and allows the following:

- Execute a command.
- Open a pop-up menu. Pop-up menus appear when a menu item is chosen followed by a menu icon
- Open a dialog box. Dialog boxes appear when a menu item is chosen followed by a dialog box icon (...).

Global menus are accessed by pressing F10 and using the arrow keys or typing the first letter of the menu name. Their contents never change.

Some of the menu commands have hot key shortcuts that are available from any part of the C51D Debugger.

CEIBO DS-51 In-Circuit Emulator



Local Menus

C51D Debugger is context-sensitive and uses Local Menus specifying different windows. Local menus are tailored to the particular window you are in. It is important not to confuse them with global menus. To prompt a local menu press Alt-F10 or click the right button of your mouse. Menu placement and contents depends on which window or pane you are in and where your cursor is. Contents may vary from one local menu to another. Many local commands appear in almost all local menus. The results of these similarly-named commands may differ, depending on the context. Every command on a local menu has a hot key shortcut consisting of Ctrl plus the highlighted letter in the command. Because of this setup, a hot key, like Ctrl-S might mean one thing in one context but something quite different in another. The core commands are still consistent across local menus. For example, the Goto command and the Search command always do the same thing, even when they are invoked from different windows.

Input boxes

Many of the C51D Debugger command options are available in input boxes. An input box prompts you to type in a string, i.e. the name of a file.

Windows

C51D Debugger displays all information and data in both global and local menus, dialog boxes (where options are set and information entered) and windows. There are many window types depending on the kind of information it holds. Windows may be opened and closed using menu commands (or hot key shortcuts for those commands). Most of the Debugger windows come from the View menu. After a window has been opened, you can move, resize, close, and otherwise manage them with commands from the Window and System menus.

CEIBO DS-51 In-Circuit Emulator

WARRANTY

TWO-YEAR WARRANTY ON ALL CEIBO PRODUCTS.

ADDRESSES

For more information contact us today:

Toll Free (U.S.A. and Canada) 1-800-833-4084

CompuServe 100274,2131

CEIBO USA TEL: 314-830-4084 FAX: 314-830-4083 7 EDGESTONE CT. FLORISSANT, MO 63033	CEIBO ISRAEL TEL : 972-9-555387 FAX: 972-9-553297 MERKAZIM BLDG., 5 MASKIT ST. P.O. BOX 2106, HERZELIA 46120
CEIBO SPAIN TEL: 91-5774296 FAX: 91-5764966 HERMOSILLA 31 MADRID 28001	CEIBO DEUTSCHLAND TEL: 06151-27505 FAX: 06151-28540 RHEINSTRASSE 32 64283 DARMSTADT

FRANCE
HOLLAND
ITALY
KOREA
SINGAPORE
S.AFRICA
SWEDEN
TAIWAN
OTHER COUNTRIES

TEL: 062-072954
TEL: 05427-33333
TEL: 051-727252
TEL: 02-786-5456
TEL: 74-46873
TEL: 011-8877879
TEL: 0589-19250
TEL: 02-9171873
TEL: 972-9-561535

FAX: 062-072953
FAX: 05427-33888
FAX: 051-727515
FAX: 02-786-5458
FAX: 74-45971
FAX: 011-8872051
FAX: 0589-16153
FAX: 02-9126641
FAX: 972-9-553297

Ordering Information

Part No. : CEIBO DS-51

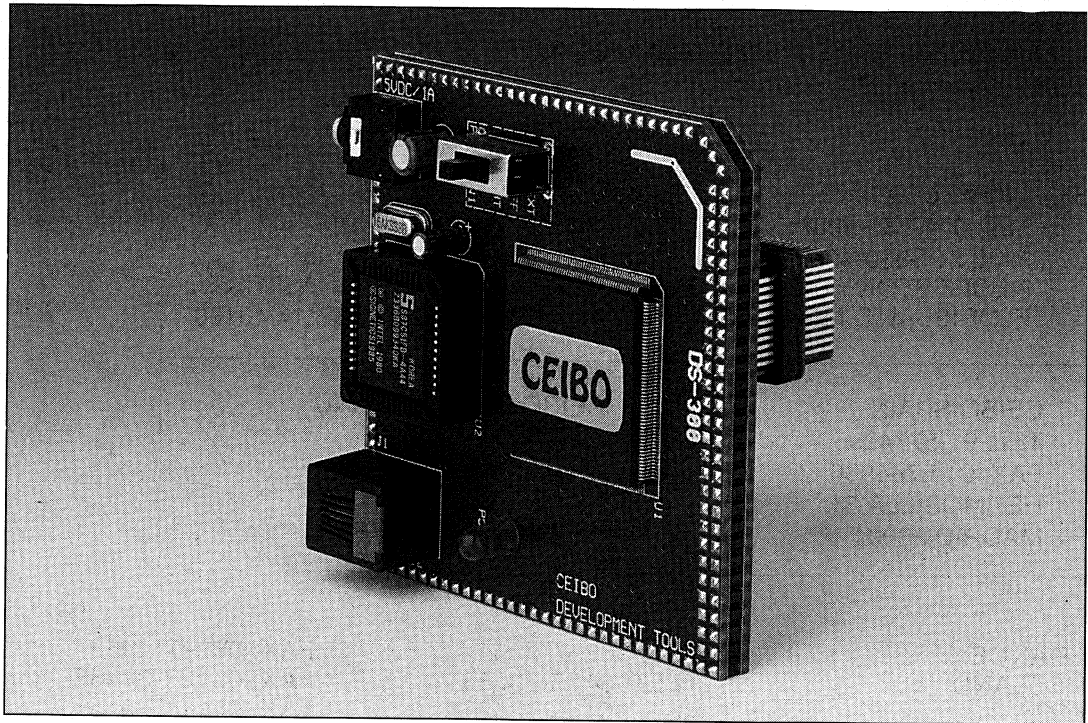
Product and Company names are trademarks of their respective organizations.

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

CEIBO DS-300 Peripheral Development Tools



Development Tools for PSD-3xx Devices

FEATURES

- Emulates PSD-3xx Devices
- Reads from and Writes to Memories
- 1024 Kbits of Emulated EPROM
- 16Kbits of Emulated SRAM
- 19 Programmable I/O ports
- Full DPLD Emulation
- Supports 8 and 16 Bus Bits
- Configuration Software for Windows
- Software for Programming PSD-3xx
- Serially Linked to IBM PC at 115KBaud

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

CEIBO DS-300 Peripheral Development Tool

DS-300 is a complete software and hardware development tool that allows file generation and emulation of the PSD-3xx devices. The configuration software provides all the elements necessary to set the device with minimum learning time. Memories, I/O ports, bus width and the DPLD are easily setup in a graphic environment that runs under the Windows Operating System. The emulator provides an immediate way to check that the devices are properly configured and allows examination and modification of the memories, and I/O lines. The system emulates PSD-3xx memories. The EPROM is up to 1024 Kbits and the SRAM 16KBits. As the EPROM is emulated by the SRAM device, memory contents may be examined and modified without generating a new file.

SPECIFICATIONS

SYSTEM MEMORY

DS-300 provides 16KBits of SRAM and 1024 Kbits of EPROM overlay memory. This memory may be configured according to all the possibilities of PSD-3xx devices. The EPROM is emulated by a RAM memory, thus allowing to download and modify its contents.

I/O PORTS

The emulator supports 19 I/O ports that may be individually used as a microcontroller I/O port expansion, programmable address decoder or latched address output. The outputs can be selected as open drain or CMOS.

BUS WIDTH

DS-300 allows configuration of memories as 8 or 16 bit wide. The emulator may be used with any microcontroller supported by the PSD-3xx devices.

PROGRAMMABLE ARRAYS

Two programmable arrays, PAD A and PAD B, are fully emulated by DS-300. A total of 40 product terms and up to 16 inputs and 24 outputs are set by the downloaded file and may be modified by the integrated software environment.

BUS SIGNALS

DS-300 includes address latches for multiplexed address/data bus, supports non-multiplexed address/data bus mode, allows definition of ALE and Reset polarity and permits the selection of read and write signals as RD/WR or R/W/E.

TRACK MODE

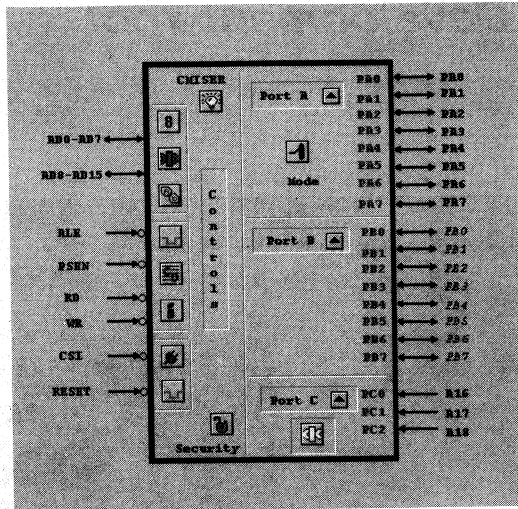
The emulator supports the address/data track mode, that enables an easy interface to shared resources with other microcontrollers or a host processor.

CONFIGURATION SOFTWARE

DS-300 includes a configuration software that runs under Windows Operating Systems. The software integrates the configuration of the device, communications with the DS-300 emulator and with the CEIBO MP-51 programmer. The software menus are: FILE, VIEW, COMPILER, OPTIONS, WINDOW, EMULATOR, PROGRAMMER and HELP.

FILE MENU

The file menu allows file downloading and saving and also defines the printer setup. The format is compatible to other existing file generators and programs. The file may be displayed in graphics mode showing the chip configuration and memory contents in hex format.



VIEW MENU

The view menu is used for a graphic representation of the chip and allows the setup change just by placing the cursor on the selected variables and clicking the options. The additional memory map screen may be used to define and display the chip selects graphically.

COMPILE MENU

The selected chip options may be compiled and the resulting file can be saved for future debugging purposes or for programming the device.

EMULATOR MENU

This menu is used to interact with the hardware through the DS-300 emulator. The different emulation options are used for downloading a file, resetting the device and selecting many hardware possibilities.

PROGRAMMER MENU

The software is prepared to communicate with the CEIBO MP-51 programmer. The integrated environment makes it possible to program the devices after finishing the debugging and during production.

INPUT POWER

5VDC from external power supply or from the target circuitry.

ITEMS SUPPLIED AS STANDARD

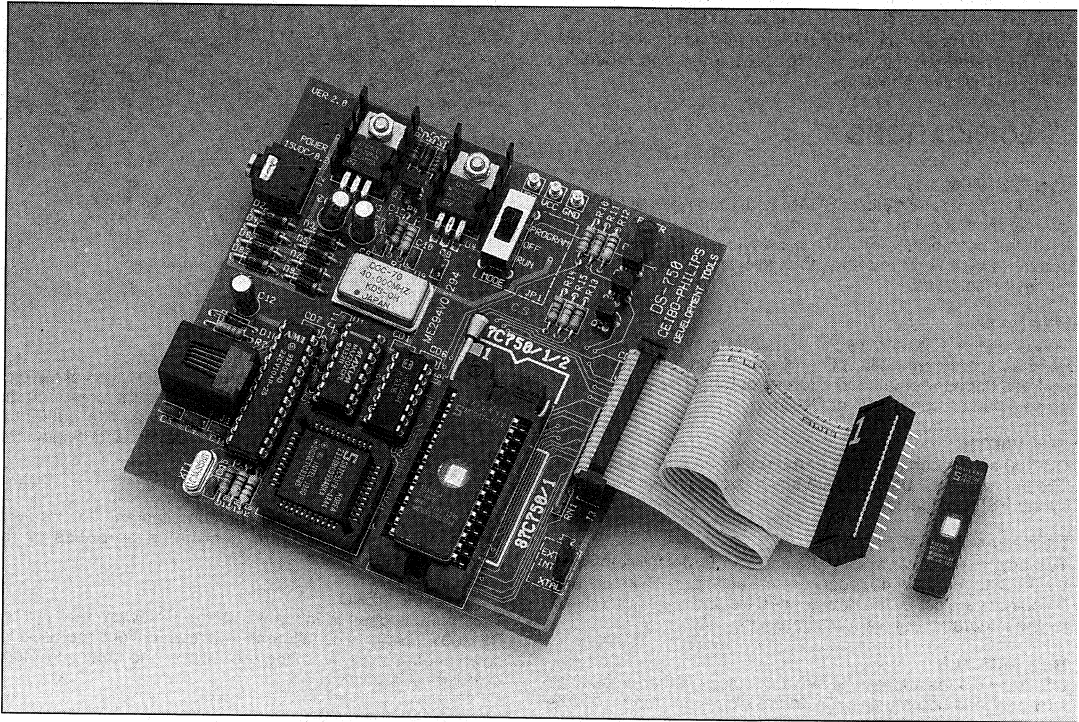
Emulation header, 44-pin PLCC male connector, user and configuration software, User's Manual and RS-232 interface cable.

Ordering Information

Part No. CEIBO DS-300

Product and Company names are trademarks of their respective organizations.

CEIBO DS-750 Microcontroller Development Tool



Development Tool for 87C750 Microcontrollers

FEATURES

- Emulates 87C750 Microcontrollers in Real-Time
- Programmable Clock up to 40MHz
- Built-in Programmer for 87C750/1/2
- Simulator Debug Mode
- Source-Level Debugger for C, PLM and Assembler
- DOS and Windows Software
- 24-pin DIP Emulation Header
- Serially linked to IBM PC at 115 Kbaud

CEIBO DS-750 Microcontroller Development Tool

DS-750 is a development tool that supports Philips 87C750 microcontrollers at any frequency allowed by the devices. It is serially linked to a PC/XT/AT or compatible system and can emulate the microcontroller using either the built-in clock oscillator or any other clock source connected to the microcontroller. The clock oscillator generates 40MHz, 20MHz, 16MHz, 10MHz and 5MHz. Emulation is carried out by programming an 87C752 microcontroller with the user software and an embedded monitor program. DS-750 provides the on-board programming capabilities and locates the monitor in the upper 1K that is not available for the 87C750. Two working modes are available: real-time and simulator debug mode. In the real time mode the user software is executed transparently and without interfering with the microcontroller speed. Breakpoints can be added to stop program execution at a specific address. In the simulator debug mode, an additional microcontroller is used to take control of the 87C750 lines and to simulate its operation but not in real-time. This operating mode allows access to all the microcontroller functions (I/O, timers, interrupts, etc.) and interacts with the hardware according to the user software execution or directly by means of emulator commands sent from the host computer. The combination of the two available working modes allows an easy way to debug hardware and software functions. The software includes C and PLM and Assembler Source Level Debugger, On-line Assembler and Disassembler, Software Trace, Conditional Breakpoints and many other features. Two versions of the software are available: DOS and Windows. The system is supplied with a User's Manual, microcontroller documentation, two samples of the 87C752 and one of the 87C750 (all windowed EPROM microcontrollers) and a power supply.

SPECIFICATIONS

MONITOR PROGRAM

The Monitor Program links the microcontroller to the host computer and is used to control the emulation of the application software in real time. The monitor code is transparent for the 87C750 and does not reduce the 1KByte of available code for the application.

BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address. Breakpoints are set when an EPROM device is programmed with a user's code, but can be disabled although disabling a breakpoint implies adding a few cycles to the program execution.

USER SOFTWARE

DS-750 program has two software environments: DOS and Windows 3.X. The DOS program is based on pull-down menus. The additional Windows program carries out the same functions of the DOS program with all the features and benefits of the new operating system.

SOURCE LEVEL DEBUGGER

The DS-750 software includes a source level debugger for Assembler and High-Level Languages (PLM, C and others) with the capability of executing lines of the program while displaying the state of any variable.

SOFTWARE TRACE

Program execution can be recorded in a 64K buffer. Conditional breakpoints may be defined to stop program execution. The user can define events and variables to be added to the software trace. The software trace is not a real-time function and is performed by slowing down the emulation speed.

FREQUENCY

The system includes a crystal oscillator able to supply clock frequencies of 5MHz, 10MHz, 16MHz, 20MHz and 40 MHz. Additionally, the user may select any other frequency by connecting an external clock source through the application hardware. Frequency selection is done by means of jumpers.

SIMULATOR DEBUG MODE

The simulator allows breakpoints to be set at any address and condition even though the user software is actually programmed in the 87C752 EPROM. The simulator debug mode will be automatically activated in case a breakpoint is enabled and not programmed in the device.

BUILT-IN PROGRAMMER

The built-in programmer may be used to program the following devices: 87C750, 87C751 and 87C752. All the programming features like security and encryption are fully supported.

EMULATION RESTRICTIONS

The following restrictions are valid for DS-750:

1. The system uses some of the microcontroller resources to emulate it: one interrupt (either INTO or INT1 according to software selection) and 5 bytes of the internal stack.
2. If you specify a breakpoint not programmed in the device, program execution will be slowed down by the simulator.
3. Disabling a programmed breakpoint will add a few cycles to the program execution. This only happens while reaching the specified condition, otherwise the real time is not affected.

INPUT POWER

15V to 18VDC (15 VDC/120 VAC wall transformer supplied).

MECHANICAL DIMENSIONS

10cm x 10cm.

ITEMS SUPPLIED AS STANDARD

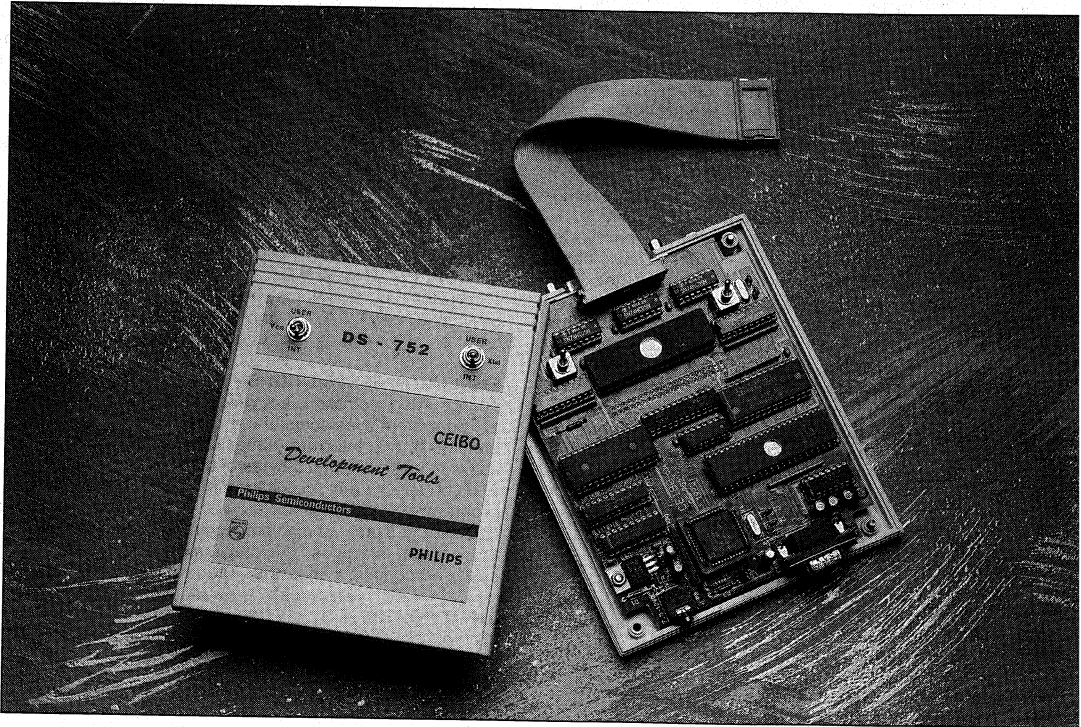
Development System with built-in programmer. 24-pin emulation header. User software including source level debugger, on-line assembler and disassembler. User's Manual and Operating Instructions. RS-232 interface cable. Power supply.

Ordering Information

Part No. P750EM SD
Available through Philips distributors.
110V- 12NC:9351-504-20112 220V- 12NC:9351-622-50112

Product and Company names are trademarks of their respective organizations.

CEIBO DS-752 Microcontroller Development Tool



Development Tool for 8XC750/1/2 Microcontrollers

FEATURES

- Real-time and Transparent In-Circuit Emulator
- Supports Philips 83C750/1/2 and 87C750/1/2 Microcontrollers
- Symbolic Debugger Compatible with Intel Object Files
- Source-Level Debugger for C, PLM and Assembler
- 2K Hardware Breakpoints and Conditional Breakpoints
- 2K of Internal Memory
- 64K Software Trace
- Serially Linked to PC and Compatible Hosts
- On-line Assembler and Disassembler
- Easy to Follow Pull-Down Menus and Windows

CEIBO DS-752 Microcontroller Development Tool

Ds-752 is a real time, high performance microcontroller development system dedicated to the 8XC750/1/2 single-chip microcontrollers. It provides an easy to use and flexible instrument which reduces the development and debugging cycle and enables the user to solve hardware and software problems quickly and efficiently. It operates with a PC or compatible computer and carries out complete real-time and transparent emulation of the target CPU.

SPECIFICATIONS

EMULATION MEMORY

DS-752 provides 2 KBytes of code memory.

HARDWARE BREAKPOINTS

Up to 2,048 hardware breakpoints allow real-time program execution until an opcode is executed at a specified address or line of your source code.

CONDITIONAL BREAKPOINTS

A complete set of conditional breakpoints permits halting program emulation on code addresses, source code lines, access to on-chip memory, port and register contents.

SOFTWARE ANALYZER

A 64 KByte buffer is used to record any software and hardware events of your program, such as executed code, memory accesses, port and internal register states, on-chip data memory and others. The trace buffer can be viewed in disassembled symbolics or high level language source code.

SYMBOLIC DEBUGGER

DS-752 allows symbolic debugging of assembler or high-level languages. The symbolic debugger uses predefined port and register names, and the symbols contained in your software, like labels, variable names line numbers and others.

SOURCE LEVEL DEBUGGER

DS-752 gives full support for debugging directly in PLM and C source code. From your source code screen you can specify a breakpoint, execute a line step or an assembly instruction, open a flexible-in-size watch window to display any variable, use the function keys to display the trace memory, registers and data, redefine the Program Counter, and reset the microprocessor.

LANGUAGES AND FILE FORMATS

DS-752 accepts files generated by Intel software (Assembler, PLM) or compatibles in hex or object format. Other assemblers and high-level languages such as C with Intel compatible format (Franklin, IAR, etc.) are also supported.

PERSONALITY ADAPTERS

DS-752 uses standard microcontrollers for hardware and software emulation. The selection of a different microcontroller is made by software commands and using a supplied socket adapter. The systems run at the frequency of the crystal on them or from the clock source supplied by the user hardware. Therefore, the same system may be adapted to your frequency requirements. The minimum frequency is determined by the emulated chip characteristics, while maximum frequency for standard systems is 16MHz.

SUPPORTED DEVICES

83C748,	87C748,
83C749,	87C749,
83C750,	87C750,
83C751,	87C751,
83C752,	87C752.

COMMAND SET

The available functions include: FILE (load, save), DEBUG (go, halt, reset, source level debugger, breakpoints), MODIFY (code, byte, registers, ports, assembler, disassembler), VIEW (watch window, publics, modules, procedures, lines, symbols, file, dir), ANALYZER (conditional breakpoints, software trace, event watch), SETUP (default, chip, base, rs-232 port).

INPUT POWER

7.5VDC to 12VDC or 5VDC from the user circuit.

MECHANICAL DEMENSIONS

1" x 5" x 6" (2.4 cm x 13 cm x 15 cm).

ITEMS SUPPLIED AS STANDARD

In-Circuit Emulator with 2 KByte Breakpoints, 2 KByte Internal Code Memory. Personality adapter for 24 pin DIP Microcontrollers. User Software including Source Level Debugger, On-line Assembler and Disassembler. User's Manual and Operating Instructions. RS-232 Interface Cable. 9 VDC wall transformer is not included.

Ordering Information

Part No. P752EM SD
Available through Philips distributors.
12NC: 9350-554-10602

Product and Company names are trademarks of their respective organizations.

CEIBO EB-51 Emulation Board

EB-51 is an emulation board dedicated to all Philips 80C51 microcontroller derivatives with a 40 or higher pin count. It is serially linked to a PC/XT/AT or compatible systems and can emulate the microcontroller using either the built-in clock oscillator or any other clock source connected to the microcontroller. The clock oscillator generates 24MHz, 16MHz, 12MHz and 6MHz. The system emulates the microcontroller in both ROMless and ROMed mode. A special emulation technology is used to recreate port 0 and 2 with standard devices, releasing most of the microcontroller resources to the user. A two microcontroller architecture leaves the serial port for user applications. The software includes a Source Level Debugger for C, PLM and Assembler, On-line Assembler and Disassembler, Software Trace, Conditional Breakpoints and many other features. All the Debugger functions run under DOS and Windows operating systems. The code memory permits downloading and modifying of user's programs. Mapping the 64KByte data memory to a target circuit or to the system is possible. Breakpoints allow real time execution until an opcode is executed at a specified address or line of the source code. All I/O lines are easily accessed and may be connected to the on-board switches and LEDs when trying out a specific idea. The system is supplied with a User's Manual, software, emulation cable and a power supply.

SPECIFICATIONS

SYSTEM MEMORY

EB-51 provides 64K of user code memory and 64K of user data memory. This RAM memory permits downloading and modifying of user's programs and variables.

CODE MEMORY

The system includes 64KBytes of RAM to be used as code memory. The upper 2KBytes of this memory are not available for programs and they are automatically loaded with a monitor program which links the microcontroller to the host computer and controls the emulation of the application software in real-time.

DATA MEMORY

Data memory is accessed by MOVX instructions and can be mapped as belonging to the system or to the target circuitry.

BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address.

SOFTWARE TRACE

Program execution can be recorded in a 64K buffer. Conditional breakpoints may be defined to stop program execution. The user can define events and variables to be added to the software trace. The software trace is not a real-time function and is performed by slowing down the emulation speed.

USER SOFTWARE

EB-51 software has two software environments: DOS and Windows 3.X. The DOS is based on pull-down menus. The additional Windows program carries out the same functions as the DOS program with all the features and benefits of the new operating system.

SOURCE LEVEL DEBUGGER

The EB-51 software includes a source level debugger for assembler and high level languages (PLM, C and others) with the capability of executing lines of the program while displaying the state of any variable.

FREQUENCY

The system includes a crystal oscillator able to support clock frequencies of 24MHz, 16MHz, 12MHz and 6MHz. The operating frequency range is from the microcontroller fmin to fmax in ROMless mode. The fmax in ROMed mode is 20MHz.

SIMULATION DEBUG MODE

The simulation debug mode allows the software to be tested without any hardware. All the emulation functions are supported by this powerful simulation debugger.

VOLTAGE

The system has a built-in voltage regulator that permits selection of either a 5V or a 3.3V voltage supply. Selection is done by setting the position of the power switch to the desired voltage.

PERSONALITY ADAPTERS

EB-51 uses standard microcontrollers for hardware and software emulation. The selection of a different microcontroller is made by replacing the microcontroller on the board with the appropriate microcontroller or a daughter board.

SUPPORTED MICROCONTROLLERS

The supported microcontrollers are most of the Philips 80C51 derivatives in both ROMless and ROMed versions. The required microcontroller on the board and supported devices are:

MICROCONTROLLER

80C32 or 87C52

87C51FB

87C528

87C504

87C550

87C654

SUPPORTED DEVICES

8031/2, 80C51/2/4/8,

87C51/2/4/8

8xC51FA/FB/FC

8xC524, 8xC528

8xC504

8xC550

8xC652, 8xC654

Daughter boards and emulation cables are available for other devices: 8xC552, 8xC562, 8xC575, 8xC576, 8xC592, 8xC598, etc.

Consult Ceibo for other supported microcontrollers.

EMULATION RESTRICTIONS

- The following emulation restrictions are valid for EB-51:
- RD and WR lines may not be used as I/O ports. There are no restrictions while RD and WR are activated by MOVX instructions.
 - The system uses some of the microcontroller resources to emulate it: one interrupt (either INTO or INT1 according to software selection) and 5 Bytes of the internal stack.

MECHANICAL DIMENSIONS

13cm x 13cm.

ITEMS SUPPLIED AS STANDARD

Emulation board, 40-pin emulation header, user software including Source Level Debugger, Simulator, Assembler, User's Manual, RS-232 interface cable and power supply.

Ordering Information

Part No. CEIBO EB-51

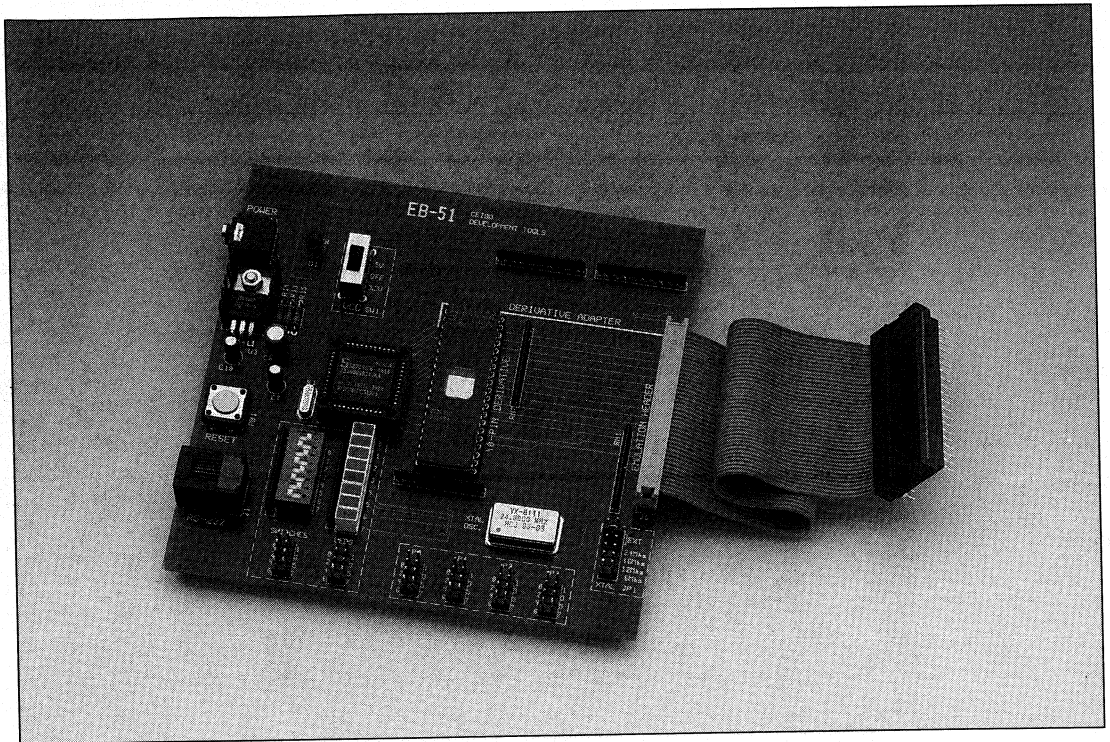
Product and Company names are trademarks of their respective organizations.

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

CEIBO EB-51 Emulation Board



Development Tool for 80C51 Microcontrollers

FEATURES

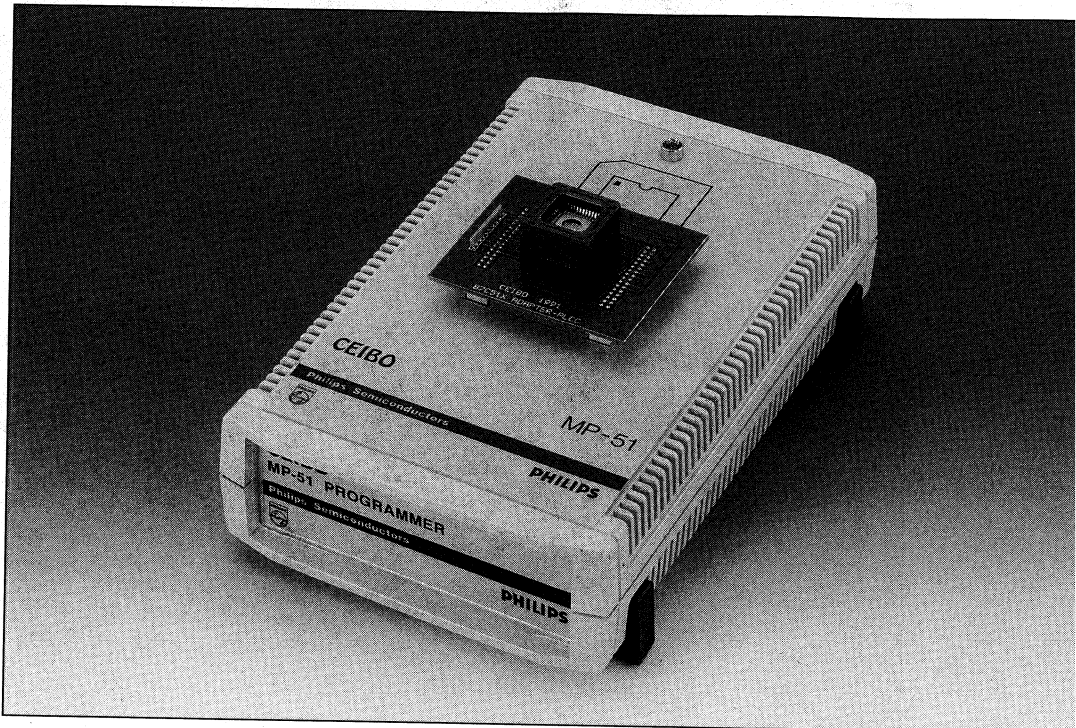
- Emulates 80C51 Microcontrollers and Derivatives
- Real-Time Operation up to 40 MHz
- 3.3V or 5V Voltage Operation
- Simulation Debug Mode
- Source-Level Debugger for C, PLM and Assembler
- Runs under DOS and Windows Software
- Support for ROMless and ROMed Microcontrollers
- 64K of Code and 64K of Data Memory
- Data Memory with Mapping Capabilities
- Performance Analyzer
- Real-Time and Conditional Breakpoints
- Emulation Header and Signal Testpoints
- Serially Linked to IBM PC at 115K Baud

CEIBO

TEL: 314-830-4084

FAX: 314-8304083

CEIBO MP-51 Programmer



EPROM, Flash, PLD and Microcontroller Programmer

FEATURES

- EPROM, Flash, PLD and Microcontroller Programmer
- Serially Linked to PC or Compatible Host
- Programs all the 8051 Microcontrollers
- Programs 16-bit Microcontrollers
- Supports 24 to 32-pin EPROMs
- Programs 32-pin Flash Memories
- Programs PLD and PSD Devices
- Easy to Follow Windows and Pull-Down Menus
- Handles Hex, Binary, Object and JEDEC Files
- Programs DIP, QFP, LCC and PLCC Devices
- Supports Lock Bits, Encryption Table and Security Bits

EIBO MP-51 Programmer

eibo MP-51 is an EPROM, Flash Memory, PLD and microcontroller Programmer dedicated to standard 24 to 32-pin EPROMs, all of the microcontrollers belonging to the 8051 family, 16-bit microcontrollers, high density PLDs, PSD devices and flash memories. Its modern design provides a high performance instrument, which is easy to use and conveniently sized. MP-51 operates with an IBM PC/XT/AT or compatible personal computer and carries out a set of powerful functions on the selected device. An RS-232 interface is used to link MP-51 to a PC. The unit consists of the instrument and adapters. The adapters may be replaced to suit the user's requirements. Adapters are available for all the possible packages such as DIP, LCC, PLCC and QFP. MP-51 software handles a PC Memory Buffer where code is loaded from a disk or filled with the contents of a device. Furthermore, this buffer may be saved on a disk file, parts of the buffer can be moved from one location to another, filled with a constant, or modified by the user. The Memory Buffer can be displayed, and finding values or strings in it is possible. Before programming, MP-51 checks if the installed adapter is compatible with the device type selected by the user. This test is done before programming any device. MP-51 has the capability to check if the device is totally erased, and can also compare if the contents of the plugged device are equal to the contents of the Memory Buffer. Address range can be specified or both operations. MP-51 allows the PLD or Microcontroller security capabilities to be enabled or disabled and handles the Lock Bit 1, Lock Bit 2, Lock Bit 3 and the Encryption Table available in several Microcontrollers.

SPECIFICATIONS

SUPPORTED DEVICES

Following is a partial list of supported devices:

EPROMs: 2716, 2732, 2764, 27128, 27256, 27512, 27010, 27020, 27040, both NMOS and CMOS versions for all the available programming voltages.

FLASH MEMORIES: 28F256, 28F512, 28F010, 28F020.

8-BIT MICROCONTROLLERS: 3755A, 8751H, 8751BH, 87C51, 87C51FA, 87C51FB, 87C51FC, 87L51FA, 87L51FB, 87C52, 87C54, 87C58, 87CL134, 87C451, 87C504, 87C524, 87C528, 87C550, 87C552, 87C575, 87C576, 87C592, 87C598, 87C652, 87C654, 87C748, 87C749, 87C750, 87C751, 87C752, 87C054-MTV, 87C055-MTV, 89C558, 89C559.

PLDs: AT22V10, ATV750, ATV2500, ATV5000, ATV5100.

PSDs: PSD301, PSD301L, PSD302, PSD302L, PSD303, PSD303L, PSD311, PSD311L, PSD312, PSD312L, PSD313, PSD313L.

FILE FORMATS

MP-51 loads different file formats: Intel Hex and Motorola S-records, Binary files, Object files, JEDEC files. It saves portions of memory in Intel, Motorola, Binary and JEDEC formats.

USER SOFTWARE

The MP-51 commands are organized in pull-down menus and windows. This software is powerful yet extremely user friendly.

COMMAND SET

The available functions include: TYPE, BLANK CHECK, SECURITY, PROGRAM, LOAD, SAVE, READ, VIEW, COMPARE, CHECKSUM, FILE, MOVE, MODIFY, DIRECTORY, TEXT FILE, DUMP, QUIT.

INPUT POWER

85VAC to 265VAC, 50Hz to 60Hz. That makes this unit suitable for any country outlet.

MECHANICAL DIMENSIONS

MP-51 is 155mm long, 60mm high and 250mm wide.

ITEMS SUPPLIED AS STANDARD

MP-51 Programmer, User software, User's Manual, RS-232 cable. PMP-51 Includes PPA-51X (87C51D-40 pin DIP adapter). Power cord not included.

ADAPTERS AND SUPPORTED DEVICES

The following list shows which adapter should be used for the different supported devices.

ADAPTER	SUPPORTED DEVICES
EPROMs and Flash Memories:	
27512D - 28 PIN DIP	2716 to 27512 NMOS and CMOS
27512P - 32 PIN PLCC	2764 to 27512 NMOS and CMOS.
27040D - 32 PIN DIP	27010 to 27040, 28F256, 28F512, 28F010, 28F020.
27040P - 32 PIN PLCC	27010 to 27040, 28F256, 28F512, 28F010, 28F020.
8-Bit Microcontrollers:	
87C51D - 40 PIN DIP	8751H, 8751BH, 87C51, 87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52, 87C54, 87C58, 87C504, 87C524, 87C528, 87C550, 87C652, 87C654.
87C51P - 44 PIN PLCC	8751H, 8751BH, 87C51, 87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52, 87C54, 87C58, 87C504, 87C524, 87C528, 87C652, 87C654, AT89C51.
87C51Q - 44 PIN QFP	87C51, 87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52, 87C54, 87C58, 87C504, 87C524, 87C528, 87C652, 87C654, 87CL134, 87CL134.
87CL134D - 42 PIN SHRINK DIP	87CL134.
87CL134Q - 44 PIN QFP	87C451.
87C451D - 64 PIN DIP	87C451, 87C453.
87C451P - 68 PIN PLCC	87C550.
87C550P - 44 PIN PLCC	87C552.
87C552P - 68 PIN PLCC	87C575, 87C576, 87C51, 87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52, 87C54, 87C58, 87C504, 87C524, 87C528, 87C652, 87C654.
87C575D - 40 PIN DIP	87C592.
87C592P - 68 PIN PLCC	87C598, 89C558, 89C559.
87C598Q - 80 PIN QFP	87C748, 87C750, 87C751.
87C751D - 24 PIN SKINNY DIP	87C748, 87C750, 87C751.
87C751P - 28 PIN PLCC	87C749, 87C752.
87C752D - 28 PIN DIP	87C749, 87C752.
87C752P - 28 PIN PLCC	87C054-MTV, 87C055-MTV.
87C054D - 42 PIN SHRINK DIP	PCD3755A.
3755AD - 28 PIN DIP	PCD3755A.
3755AS - 28 PIN SO	PCD3755A.
3755AQ - 32 PIN QFP	PCD3755A.
PSDs:	
PSD301P - 44 PIN PLCC	PSD301, PSD301L, PSD302, PSD302L, PSD303, PSD303L, PSD311, PSD311L, PSD312, PSD312L, PSD313, PSD313L.

Consult Ceibo or Philips for ordering information code (12NC or PPA number in US).

Ordering Information

Part No. PMP-51 SD/OM4260

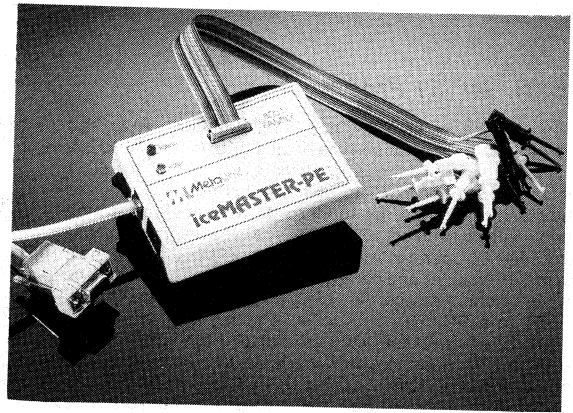
Available through Philips distributors.

12NC: 9350-410-00112

Product and Company names are trademarks of their respective organizations.

iceMASTER-PE™
8051 Family
In-Circuit Emulator

NOW
8051, 8052,
8xC51FX
SUPPORT



PRODUCT DESCRIPTION

The unique **iceMASTER-PE** packs an advanced feature set into a tiny, palm-sized package that any engineer can afford. Designed for demanding projects, **iceMASTER-PE** supports frequencies up to 42 MHz with a full complement of emulation memory, external data memory, and a transparent trace buffer 16K frames deep with advanced searching capabilities. The entire emulator plugs directly into the target application or operates in a stand-alone mode.

The **iceMASTER-PE** is the world's most portable emulator because the emulator and probe electronics are both integrated into a pocket-able package about the size of a PC mouse. To achieve this dramatic breakthrough in the size and cost of high-performance emulation, MetaLink invented a new emulation system architecture, Advanced Emulator Technology (AET, patent pending).

The **iceMASTER-PE**'s windowed user interface delivers the highest development productivity and is easy to learn and easy to use, including a context-sensitive hypertext and hyperlinked help system. This powerful, productive interface gives the user total control and flexibility in the configuration of the size, position, content, and color of each window.

The **iceMASTER-PE** includes a full symbolic and source-level debugger for Assemblers and Compilers. The emulator supports all 9 of the most popular 8051 Assemblers and Compilers.

The **iceMASTER-PE** sets the new standard for excellence, portability and value in 8051 family emulation.

KEY CHARACTERISTICS

- Supports 8051 family devices up to 42MHz
- Support for 8031, 8051, 8x32, 8x52, 8xC54, 8xC58, 8xC51FA, 8xC51FB, 8xC51FC, 8xC751, 8xC752, 8xC750, NMOS, CMOS, ROM, EPROM/OTP Devices.
- Full-Featured, Real-time & Transparent Emulator
- Emulator and Probe electronics integrated into a single package only 3" x 4" x 1"
- Plugs directly into target applications or operates in a stand-alone mode
- Patent-pending AET system architecture

HARDWARE CAPABILITIES

- Up to 64K Program & 64K External Data Memory
- 16K frame trace buffer
- View trace while executing
- Up to 128K hardware breakpoints
- Up to 64K Trace ON & 64K Trace OFF triggers
- Integrated diagnostic self-test capability

SYSTEM FEATURES

- PC-hosted via RS-232 serial link
- Efficient, powerful, easy to learn
- User control of window size, content & color
- Supports third party Assemblers & Compilers
- Full Symbolic & Source-Level debug
- Complete system includes Emulator, 8051 Macro Cross Assembler, RS-232 cable & power supply

MetaLink Corporation
25 E. Elliot Road
 Chandler, AZ 85225
 Phone: (602) 926-0797
 Phone: (800) 638-2423
 Fax: (602) 926-1198

MetaLink Europe GmbH
8011 Kirchseeon-Eglharting
 Germany
 Phone: 08091 2046
 Fax: 08091 2386

©1993 MetaLink Corporation

1042-001/692/TP

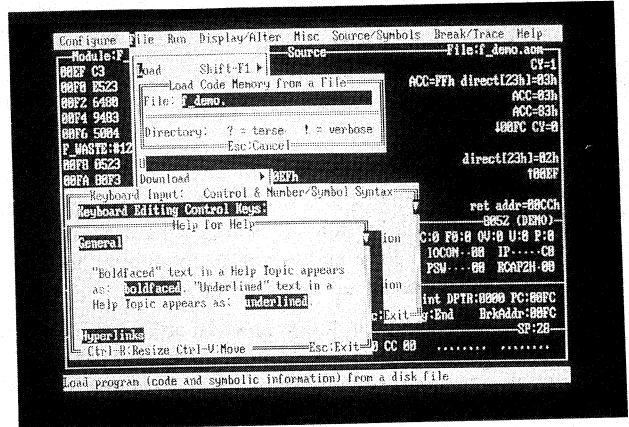
MetaLink®
Corporation

FLEXIBLE, EASY-TO-USE INTERFACE

iceMASTER has an advanced, windowed user interface that emphasizes ease of use. Each window can be sized, moved, highlighted, scrolled, color-controlled, added, or removed completely. iceMASTER provides pull-down and pop-up menus, function keys, and context-sensitive help. The contents of any memory space may be perused and altered directly from the appropriate window using the keyboard or a mouse.

You have immediate access to the hypertext/hyperlinked, context-sensitive, on-line help system which clearly explains what your options are (at any detail level you choose), keeping you productive. There is even a HELP-FOR-HELP feature. Whether you are beginning your first design project, or are a veteran designer searching for the fastest possible debugging method, you will appreciate the EASE-OF-USE features designed into iceMASTER.

Novices can navigate smoothly through a debugging session by accessing the commands and menus as standard pull-downs. Experienced designers can instantaneously pop-up their menu of choice by using redefinable hot keys.

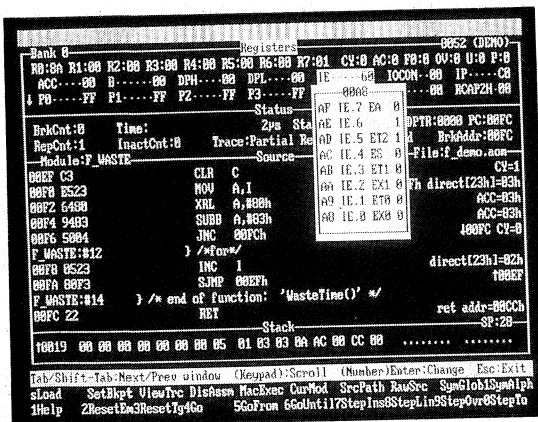


DECREASED DEVELOPMENT TIME

Your iceMASTER source window accelerates the debugging process with Dynamically Annotated Code. When single-stepping, instruction execution information is displayed and retained next to each instruction. You can clearly see the data behind your program's flow, including contents of all accessed (read or write) memory locations and registers, as well as flow-of-control direction change markers. A moving color bar indicates the current position in the program as it executes.

At your option, the iceMASTER source window allows operations on three different views of code memory: disassembled instructions, instructions mixed with High Level Language (HLL) source statements, or HLL source only.

HLL source statements and symbolic disassembly information are also displayed when you disassemble the program or view the trace buffer.



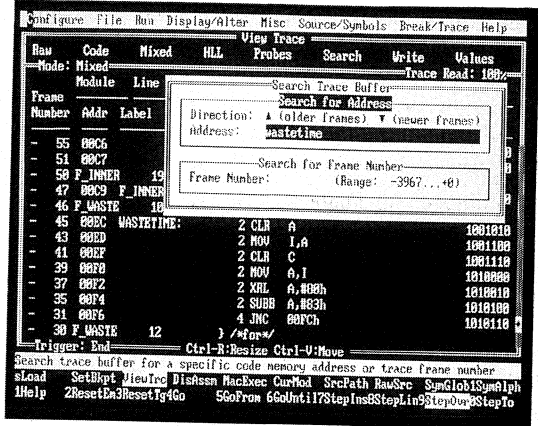
THE COMPLETE DEBUGGER

The trace buffer captures data in real-time. Trace information consists of address and data bus values and user-selectable probe clips. You can view the trace buffer data through several display filters: raw hex, disassembled instructions, instructions mixed with HLL source statements, or HLL source only. You can display the probe clip bit values in binary, hex, or digital waveform formats.

You can trigger the trace to begin capturing data on all instructions leading up to a breakpoint, around (before and after) a breakpoint, or following a breakpoint. Capture filtering allows you to focus attention only on areas of interest, eliminating clutter.

To further speed your design process, an integrated search mechanism allows you to locate any label, HLL source line number, or address in the trace buffer in either the backward or forward direction. The days of manually scanning or post-processing a large buffer of trace data are gone!

If you WRITE your program in a HLL, you should be able to DEBUG it that way - iceMASTER lets you do just that!

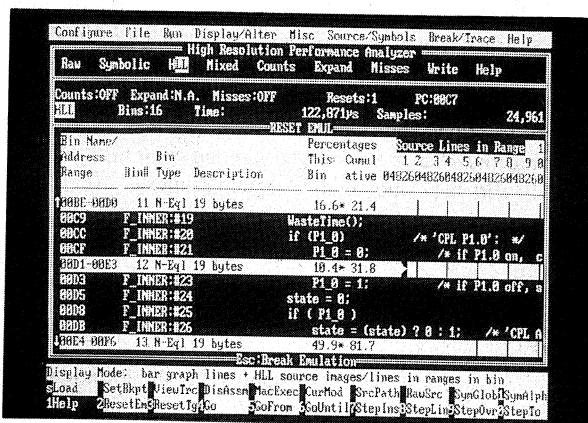


IMPROVING THE QUALITY OF YOUR PRODUCT: FINDING THE HOT (OR NOT SO HOT) SPOTS

iceMASTER emulators (Model 400 & PE) provide a PERFORMANCE ANALYZER that allows you to monitor the time spent executing specific portions of your program to find "hot spots" or "dead code." You can define and analyze memory areas based on code address, module, line number or label ranges.

You can view results dynamically during emulation or later for a more detailed analysis. You can toggle the display from bar graph format to actual frequency counts, and you can filter the level of detail in the displayed results to include raw data, code labels, and/or HLL source statements.

iceMASTER Model 400 provides an additional Performance Analyzer that is the absolute best in the industry for tuning and testing your code. It is very flexible and far more accurate than most other emulators, with a resolution of less than six microseconds. You can define and analyze up to 15 memory areas and each of these 15 memory areas can consist of non-contiguous, logically related subranges, (e.g. groupings of related functions which are not necessarily contiguous in memory).

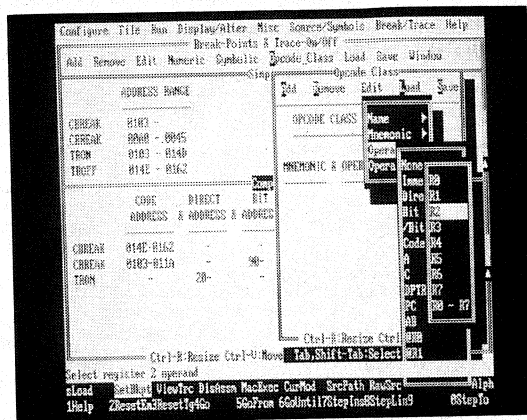


FINDING THAT BUG

You have access to as many as 128K breakpoints and 64K tracetriggers. These triggers can be enabled, disabled, set or cleared. Simple triggers are based on code or external data addresses or address ranges. Simple breakpoints can be set or cleared directly in the source window at disassembled instructions or HLL source statements.

Breakpoints can also be complex triggers, based on code address, direct address, bit address, opcode value, opcode class or immediate operand. Complex triggers can be ANDed and ORed together.

Finding that elusive bug is now much easier!



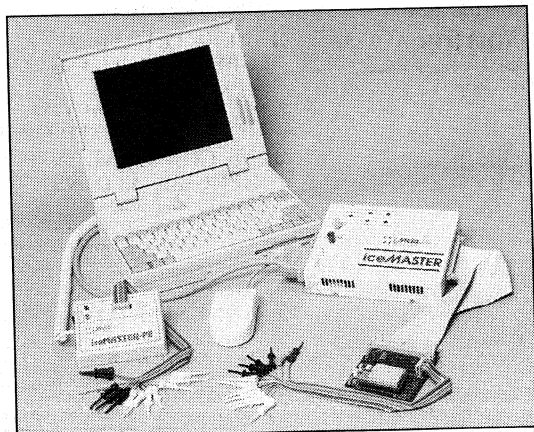
SMALL AND POWERFUL

Some companies design emulators that use one or two full size PC expansion board slots. These companies take it for granted that you have the "right" kind of PC, that you don't need the expansion slots for something else, and that you don't mind taking apart your computer to install their emulator!

At MetaLink, we don't take our customers for granted.

The iceMASTER family is the culmination of over 7 years of focused engineering by MetaLink to bring advanced semiconductor technologies to emulator design. Using state-of-the-art PALs (PEELs), LCAs, VLSI memories and microprocessors, MetaLink designed world-class emulation capability into the smallest emulator footprints in the industry, combining powerful and complex emulation features with all-around ease-of-use.

The high speed serial link connects easily to the back of your computer with a standard RS-232 cable. The emulators are smaller than the size of a VCR tape, fit neatly in crowded work spaces and are easy to move around on the bench or to other computers. **PORTABLE PRODUCTIVITY!**



iceMASTER-PE Development System

Metalink specializes in enhancing the emulation technology required by EMBEDDED SYSTEMS DESIGNERS. Metalink has consistently led the industry in emulation technology: The first PC-based 8051 emulator; the first 8052 emulator; the first to offer support for all the unique features of the 8051 family components, such as watchdog-timer, idle modes, power down mode, DMA and A/D.

Metalink is a full-service emulation company. We support our customers over the long term with services such as repair, discounted upgrades, rental units, 10-day trial purchase periods and free technical support for applications problems. A network of world-wide sales and service representatives is augmented by a well-trained telemarketing staff at headquarters.

❑ HOST SPECIFICATIONS

Host IBM PC, XT, AT, 386, 486, PS/2, Laptop, Notebook or compatible system, 640K bytes of RAM, Hard Disk Drive, Monochrome or Color Display, with a Standard RS-232 Serial Port. Operating System DOS 2.0 or greater.

❑ EMULATOR SOFTWARE SPECIFICATIONS

File Formats Supported with Symbolic or Source-Level Debugging:

2500AD, Archimedes, Avocet, BSO/Tasking, Franklin, IAR Systems, Intel OMF, Keil, MCC, Microtec Research, Motorola 'S' Record, Systronix and Intel HEX.

Source/Symbol Support:

Assembler, BASIC, C and PL/M Languages

Symbolic or Source-Level Debug:

Setting of Breakpoints
Setting of Trace ON/OFF
Viewing of Trace Buffer
Viewing of Source Window Display
Viewing of Performance Analyzer
Assembly/Disassembly of Code

HLL Structure/Content Display of:

Modules Line Numbers
Scopes Program Variables

❑ EMULATOR HARDWARE SPECIFICATIONS

iceMASTER-PE MODELS:

8031: Supports 8031 and 80C31

Operates up to 33MHz
40-Lead DIP or optional 44-Lead PLCC

8032: Supports 8031, 80C31, 8032, 80C32 and C501

Operates up to 42MHz

40-Lead DIP or optional 44-Lead PLCC

83752: Supports 83C751, 87C751, 83C752, 87C752 and 87C750

Operates up to 16MHz

24- & 28-Lead DIP or optional 28-Lead PLCC

8351FX: Supports 8031, 80C31, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8052, 80C52, 8752, 87C52, 80C51FA, 83C51FA, 83C51FB, 87C51FB, 83C51FC, 87C51FC, 80C54, 87C54, 80C58 and 87C58

Operates up to 16MHz

40-Lead DIP or optional 44-PLCC

❑ OPERATING CHARACTERISTICS

Clock frequency user-selectable between external target crystal/clock or internal clock source

Real-time, Electrically and Operationally Transparent

❑ USER INTERFACE

Keyboard or Mouse Control

Pull-down & Pop-up menus with fill-in boxes

Available Main Screen Windows:

Registers and PSW bits

Bit Memory

Stack data displayed in HEX &/or ASCII

Up to 5 Internal Data Memory displayed in HEX &/or ASCII

Up to 5 External Data Memory displayed in HEX &/or ASCII

Up to 5 Code Memory displayed in HEX &/or ASCII

Source Program displayed in Code, HLL Source or Mixed

Watch window for variable data

System Status data

Main Screen Window Display Controls:

Movable

Selectable (On/Off)

Sizable

Color selection

Scrollable

Highlighting of key data

Function/Hot Key Access:

User-assignable for commands

User-displayable for quick reference

❑ MEMORY OPERATIONS

Emulation Memory:

iceMaster-PE Models: 8031 8032 8351FX 83752

Program Memory: 64K 64K 64K 2K

External Data Memory: 64K 64K 64K N/A

Mapping Resolution:

Program: Down to 1-byte

External Data: Down to 1-byte

Program Memory:

Single Line Assembler (full instruction set support)

Disassemble in Code or Source/Code mode

Disassembly may be written to a disk file

Data Memory:

Internal or External Memory

Fill a block of memory with data

Copy a block of data to another area

Change a single address data content

Compare any two blocks of addressed data

Displayed data may be written to a file

Registers/SFRs/Bit Memory:

Examine or Modify

Program Variables:

Examine or Modify

❑ EMULATION CONTROLS

Reset from Emulator and Go

Reset from Target and Go

Reset Processor

Go from current Program Counter

Go From a new Program Counter

Go Until a Program Counter/Label

Slow Motion (Repetitive Step commands)

Step by machine instruction

Step by Line Number

Step Over calls

Step To next function or procedure

❑ HARDWARE BREAKPOINTS

Up to 64K real-time Program Addresses

Up to 64K real-time External Data Addresses

❑ TRIGGER CONDITIONS

Set directly in Source window or Pull-down menu

PC address & range of addresses

Opcode Value

Opcode Class

SFRs/Registers

Direct byte address & range of addresses

Direct bit address & range of addresses

Immediate operand value

Read/Write to bit address

Register address modes

Read/Write to Register address

Logical AND/OR of any of the above

External Data address & range of addresses

Break Count Overflow

External (CLIP) Break Input

External (CLIP) Trigger Output

❑ TRACE

Real-time trace with view while executing code

16K-Frame Trace Buffer

Start, Center, End and Variable Trace Trigger settings

Up to 64K Trace ON/OFF triggers for trace filtering

Trace Contents consist of:

16-bits Address Bus

8-bits Data Bus

7-bits External Clips

Trace Display Modes:

Raw Hex

Symbolic

HLL Source

Mixed

External Clips display format:

Binary data

HEX data

Digital waveform

Trace Buffer Operations:

Write trace buffer to a disk file

Search trace buffer for labels & addresses

❑ PERFORMANCE ANALYZER

Program profiling capability

7 year duration

Display options:

Bar Graph

Frequency Count

Display Modes:

Raw

HLL Source Lines

Symbolic

Mixed

Up to 999 Bin capacity

User-controlled Bin set-up:

By Address By Symbol

By Module By Line Number

Automatic

❑ HELP

On-Line

Context sensitive

Hypertext/Hyperlinked

❑ DIAGNOSTIC SELF-TEST

Determines Emulation Hardware Status

❑ MACRO

Repetitive routines

User-created and callable

❑ ELECTRICAL SPECIFICATIONS

Input Power (maximum):

0.75 A @ +5 VDC +/-5%

Power Source: Target system or power supply

❑ MECHANICAL SPECIFICATIONS

Emulator Dimensions:

4.4" x 3.25" x 0.9"

11.18cm x 8.25cm x 2.29cm

Emulator weight:

0.5lbs 0.3kg

❑ ANNUNCIATORS

Power and Active LEDs

Power ON/OFF switch

❑ WARRANTY

One (1) year limited warranty, parts and labor

Call Today

1(800) 638-2423

1(800) METAICE

or contact your local distributor

Rental plans are available.

Product names are used for purposes of identification only and may be trademarks or registered trademarks of their respective companies.

iceMASTER™ 8051 In-Circuit Emulators

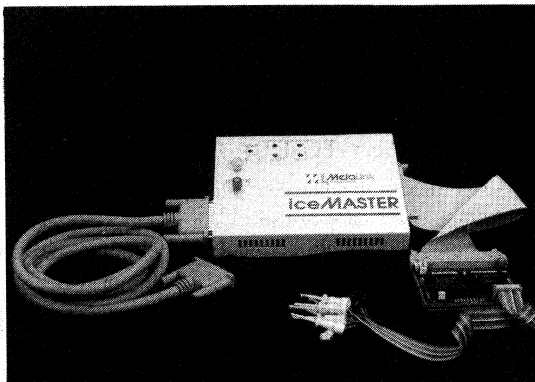
PRODUCT DESCRIPTION

The **iceMASTER-8051** Model 200/400 emulator represents a culmination of 7 years of focused engineering to create the world's most affordable and portable full-featured emulators.

The **iceMASTER-8051** emulators offer real-time and transparent emulation at up to 24MHz for a broad number of derivative devices through the use of interchangeable probe cards. Powerful breakpoint systems allow an engineer to stop a program at any time and examine all states and conditions. Trace memory provides a complete history of each event that has occurred, including source-level information, address, data, status, searching and external logic events. The best performance analyzer capability in the industry allows a thorough evaluation of the program to decide what areas are taking the most time and simplify those areas requiring improved performance.

The **iceMASTER-8051** emulators support symbolic and source-level debugging for the most popular 8051 Cross Assemblers, 'C' Compilers, and PL/M Compilers. These capabilities allow the designer to debug their system the way it was designed, at the symbolic or source-level. This methodology increases productivity while decreasing cost and time-to-market.

The **iceMASTER-8051** emulators provide pull-down and pop-up menus, mouse support, function/hot keys, and context-sensitive hyperlinked help. The advanced windowed user interface emphasizes ease of use. Each window can be sized, moved, scrolled, highlighted, color-controlled, added or removed completely. The contents of any memory space may be perused and altered directly from the appropriate window, with multiple memory spaces displayable simultaneously.



KEY CHARACTERISTICS

- Full-Featured, Real-time & Transparent Emulator
- Supports 8051 family devices up to 24MHz
- Interchangeable Probe Cards
- Hosted on any PC or compatible, including Laptops, Notebooks, or Micro Channel
- 115K baud serial link using a standard comm port
- Unlimited user support

HARDWARE CAPABILITIES

- 64K Program & 64K External Data Memory
- 4K frame trace buffer
- Advanced trace search ability
- 128K hardware breakpoints
- 64K Trace ON/OFF triggers
- Broad 8051 derivative device support (more than 65 devices)
- Dual Performance Analyzers

SYSTEM FEATURES

- Efficient, powerful, easy to learn
- User control of window size, content & color
- Supports third party Assemblers & Compilers
- Full Symbolic & Source-Level debug
- Complete system - includes Emulator, 8051 Macro Cross Assembler, RS-232 cable & power supply

MetaLink Corporation Phone: (602) 926-0797
325 E. Elliot Road Phone: (800) 638-2423
Chandler, AZ 85225 Fax: (602) 926-1198

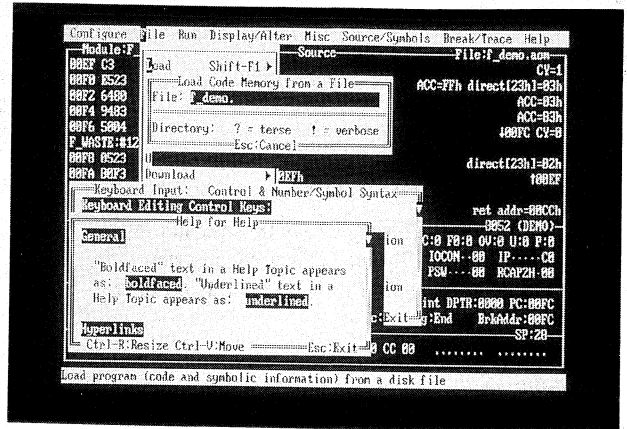
MetaLink Europe GmbH Phone: 08091 2046
8011 Kirchseeon-Eglharting Fax: 08091 2386
Germany

FLEXIBLE, EASY-TO-USE INTERFACE

iceMASTER has an advanced, windowed user interface that emphasizes ease of use. Each window can be sized, moved, highlighted, scrolled, color-controlled, added, or removed completely. iceMASTER provides pull-down and pop-up menus, function keys, and context-sensitive help. The contents of any memory space may be perused and altered directly from the appropriate window using the keyboard or a mouse.

You have immediate access to the hypertext/hyperlinked, context-sensitive, on-line help system which clearly explains what your options are (at any detail level you choose), keeping you productive. There is even a HELP-FOR-HELP feature. Whether you are beginning your first design project, or are a veteran designer searching for the fastest possible debugging method, you will appreciate the EASE-OF-USE features designed into iceMASTER.

Novices can navigate smoothly through a debugging session by accessing the commands and menus as standard pull-downs. Experienced designers can instantaneously pop-up their menu of choice by using redefinable hot keys.

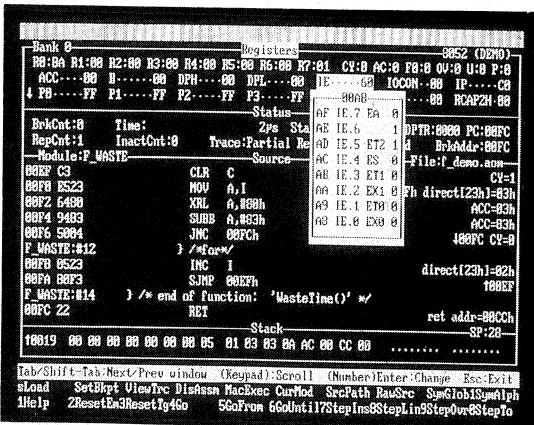


DECREASED DEVELOPMENT TIME

Your iceMASTER source window accelerates the debugging process with Dynamically Annotated Code. When single-stepping, instruction execution information is displayed and retained next to each instruction. You can clearly see the data behind your program's flow, including contents of all accessed (read or write) memory locations and registers, as well as flow-of-control direction change markers. A moving color bar indicates the current position in the program as it executes.

At your option, the iceMASTER source window allows operations on three different views of code memory: disassembled instructions, instructions mixed with High Level Language (HLL) source statements, or HLL source only.

HLL source statements and symbolic disassembly information are also displayed when you disassemble the program or view the trace buffer.



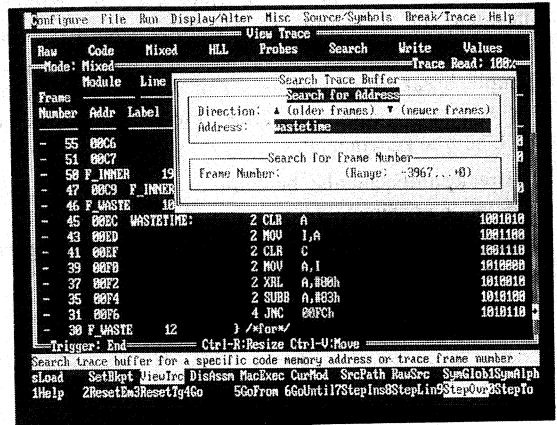
THE COMPLETE DEBUGGER

The trace buffer captures data in real-time. Trace information consists of address and data bus values and user-selectable probe clips. You can view the trace buffer data through several display filters: raw hex, disassembled instructions, instructions mixed with HLL source statements, or HLL source only. You can display the probe clip bit values in binary, hex, or digital waveform formats.

You can trigger the trace to begin capturing data on all instructions leading up to a breakpoint, around (before and after) a breakpoint, or following a breakpoint. Capture filtering allows you to focus attention only on areas of interest, eliminating clutter.

To further speed your design process, an integrated search mechanism allows you to locate any label, HLL source line number, or address in the trace buffer in either the backward or forward direction. The days of manually scanning or post-processing a large buffer of trace data are gone!

If you WRITE your program in a HLL, you should be able to DEBUG it that way - iceMASTER lets you do just that!

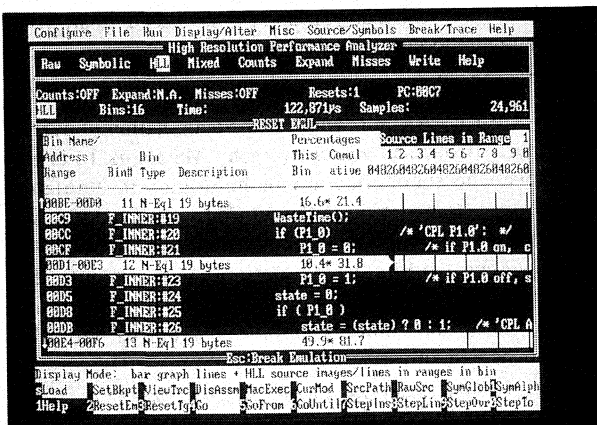


IMPROVING THE QUALITY OF YOUR PRODUCT: FINDING THE HOT (OR NOT SO HOT) SPOTS

iceMASTER emulators (Model 400 & PE) provide a PERFORMANCE ANALYZER that allows you to monitor the time spent executing specific portions of your program to find "hot spots" or "dead code." You can define and analyze memory areas based on code address, module, line number or label ranges.

You can view results dynamically during emulation or later for a more detailed analysis. You can toggle the display from bar graph format to actual frequency counts, and you can filter the level of detail in the displayed results to include raw data, code labels, and/or HLL source statements.

iceMASTER Model 400 provides an additional Performance Analyzer that is the absolute best in the industry for tuning and testing your code. It is very flexible and far more accurate than most other emulators, with a resolution of less than six microseconds. You can define and analyze up to 15 memory areas and each of these 15 memory areas can consist of non-contiguous, logically related subranges, (e.g. groupings of related functions which are not necessarily contiguous in memory).

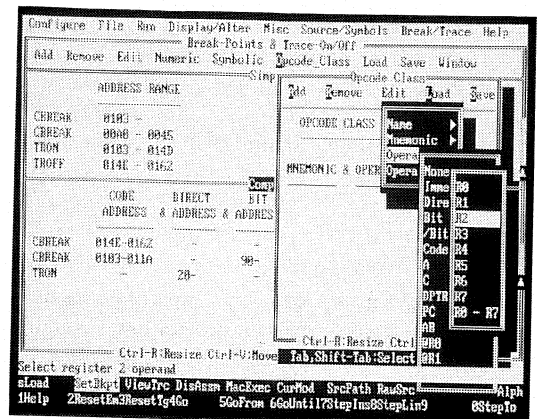


FINDING THAT BUG

You have access to as many as 128K breakpoints and 64K tracetriggers. These triggers can be enabled, disabled, set or cleared. Simple triggers are based on code or external data addresses or address ranges. Simple breakpoints can be set or cleared directly in the source window at disassembled instructions or HLL source statements.

Breakpoints can also be complex triggers, based on code address, direct address, bit address, opcode value, opcode class or immediate operand. Complex triggers can be ANDed and ORed together.

Finding that elusive bug is now much easier!



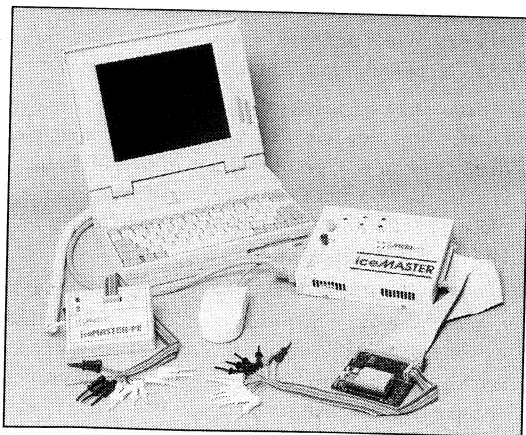
SMALL AND POWERFUL

Some companies design emulators that use one or two full size PC expansion board slots. These companies take it for granted that you have the "right" kind of PC, that you don't need the expansion slots for something else, and that you don't mind taking apart your computer to install their emulator!

At MetaLink, we don't take our customers for granted.

The iceMASTER family is the culmination of over 7 years of focused engineering by MetaLink to bring advanced semiconductor technologies to emulator design. Using state-of-the-art PALs (PEELs), LCAs, VLSI memories and microprocessors, MetaLink designed world-class emulation capability into the smallest emulator footprints in the industry, combining powerful and complex emulation features with all-around ease-of-use.

The high speed serial link connects easily to the back of your computer with a standard RS-232 cable. The emulators are smaller than the size of a VCR tape, fit neatly in crowded work spaces and are easy to move around on the bench or to other computers. **PORTABLE PRODUCTIVITY!**



ceMASTER-8051 Model 200/400 Development Systems

MetaLink specializes in enhancing the emulation technology required by EMBEDDED SYSTEMS DESIGNERS. MetaLink has consistently led the industry in emulation technology: the first PC-based 8051 emulator; the first 8052 emulator; the first to offer support for all the unique features of the 8051 family components, such as watchdog-timer, idle modes, power down mode, DMA and A/D.

MetaLink is a full-service emulation company. We support our customers over the long term with services such as repair, discounted upgrades, rental units, 10-day trial purchase periods, and free technical support for applications problems. A network of world-wide sales and service representatives is augmented by a well-trained telemarketing staff at headquarters.

HOST SPECIFICATIONS

Host IBM PC, XT, AT, 386, 486, PS/2, Laptop, Notebook or compatible system, 640K bytes of RAM, Hard Disk Drive, Monochrome or Color/Graphic Display, with a Standard RS-232 Serial Port. Operating System DOS 2.0 or greater.

EMULATOR SOFTWARE SPECIFICATIONS

File Formats Supported with Symbolic or Source-Level Debugging:

Archimedes, Avocet, BSO/Tasking, Franklin, IAR Systems, Intel OMF, Keil, MCC, Microtec Research, Motorola 'S' records and Intel HEX.

Source/Symbol Support:

Assembler, C and PL/M Languages

Symbolic or Source-Level Debug:

Setting of Breakpoints
Setting of Trace ON/OFF
Viewing the Trace Buffer
Viewing of Source Window Display
Viewing of Performance Analyzer
Assembly/Disassembly of Code

HLL Structure/Content Display of:

Modules Line Numbers
Scopes Program Variables

EMULATOR HARDWARE SPECIFICATIONS

ceMASTER-8051 Models:

200 Basic Emulator
400 Enhanced Emulator with:

4K Trace Buffer
2 Performance Analyzers
Full Watchdog Timer Support

Operating Modes:

Single-Chip ROM
Romless

Interchangeable Probe Card

Used to support the functional derivatives of each micro controller family as well as the full range of NMOS, CMOS, EPROM, and OTP technology variations. Contact MetaLink for the latest information on any device supported.

Device Unique Support

A-to-D Converter
FIFO
Multiple-Divide Unit
Multiple DPTRs

OPERATING CHARACTERISTICS

Clock frequency user-selectable between external target crystal/clock or internal crystal source

Electrically and Operationally Transparent

Real-time: 0.5-24MHz

USER INTERFACE

Keyboard or Mouse Control

Pull-down & Pop-up menus with fill-in boxes

Available Main Screen Windows:

Registers and PSW bits
Bit Memory
Stack data displayed in HEX &/or ASCII
Up to 5 Internal Data Memory displayed in HEX &/or ASCII
Up to 5 External Data Memory displayed in HEX &/or ASCII
Up to 5 Code Memory displayed in HEX &/or ASCII
Source Program displayed in Code, HLL Source or Mixed

Watch window for variable data
System Status data displayed in HEX
Main Screen Window Display Controls:
Moveable Selectable (On/Off)
Sizable Color selection
Scrollable Highlighting of key data

Function/Hot Key Access:

User-assignable for commands
User-displayable by reference

MEMORY OPERATIONS

Emulation Memory:

64K Program Memory
64K External Data Memory

Mapping Resolution:

Program: 16-bytes/block
External Data: 16-bytes/block

Program Memory:

Single Line Assembler (full instruction set support)
Disassemble in Code or Source/Code mode
Disassembly may be written to a disk file

Data Memory:

Internal or External Memory
Fill of a block of memory with data
Copy a block of data to another area
Change a single address data content
Compare any two blocks of addressed data
Displayed data may be written to a file

Registers/SFRs/Bit Memory:

Examine or Modify

Program Variables:

Examine or Modify

EMULATION CONTROLS

Reset from Emulator and Go

Reset from Target and Go

Reset Processor

Go from current Program Counter

Go From a new Program Counter

Go Until a Program Counter/Label

Slow Motion (Repetitive Step commands)

Step by machine instruction

Step by Line Number

Step Over calls

Step To next function or procedure

HARDWARE BREAKPOINTS

64K real-time Program Addresses

64K real-time External Data Addresses

TRIGGER CONDITIONS

Set directly in Source window or Pull-down menu
PC address & range of addresses

Opcode Value

Opcode Class

SFRs/Registers

Direct byte address & range of addresses

Direct bit address & range of addresses

Immediate operand value

Read/Write to bit address

Register address modes

Read/Write to Register address

Logical AND/OR of any of the above

External Data address & range of addresses

Break Count Overflow

External (CLIP) Break Input

External (CLIP) Trigger Output

TRACE (Model 400)

4K-Frame Trace Buffer

Start, Center, End & Variable Trace Trigger settings

64K Trace ON/OFF triggers for trace filtering

Trace Contents consist of:

16-bits Address Bus
8-bits Data Bus
7-bits External Clips
DMA Activity

Trace Display Modes:

Raw Hex
Symbolic
HLL Source
Mixed

External Clips display format:

Binary data
HEX data
Digital waveform

Trace Buffer Operations:

Write trace buffer to a disk file
Search trace buffer for labels & addresses

PERFORMANCE ANALYZERS (Model 400)

Real-time Program profiling capability

5.4µ-sec sampling period

7 year duration

Display options:

Bar Graph
Frequency Count

Display Modes:

Raw Symbolic
Mixed HLL Source Lines

Up to 999 Bin capacity

User-controlled Bin set-up:

By Address By Symbol
By Module By Line Number
Automatic

HELP

On-Line

Context sensitive

Hypertext/Hyperlinked

MACRO

Repetitive routines

User-created and callable

ELECTRICAL SPECIFICATIONS

Input Power (maximum):

1.5 A @ +5 VDC +/-5%

MECHANICAL SPECIFICATIONS

Emulator Dimensions:

7.0" x 5.5" x 1.0"
17.8cm x 14cm x 2.54cm

Emulator weight:

2.0lbs 0.9kg

ANNUNCIATORS

Power, Reset, Break, and Active LEDs

Power ON/OFF, Reset, Break switches

WARRANTY

One (1) year limited warranty, parts and labor



Call Today

1(800) 638-2423

1(800) METAICE or contact your local distributor

Rental plans are available.

Product names are used to purposes of identification only and may be trademarks or registered trademarks of their respective companies.

Ordering Information**Philips Semiconductors 8051 Family Microcontroller Support****iceMASTER-PE Emulators**

The iceMASTER-PE emulators for the 8051 family are self-contained units, each dedicated to a particular member of the 8051 family. The emulator and probe card electronics are combined into a single, small package. Every iceMASTER-PE emulator comes with 64K of Code memory, 64K of External Data memory, a 16K frame Trace Buffer (which can be viewed without breaking emulation), power supply, RS-232 cable and an 8051 Macro Cross Assembler. All iceMASTER-PE emulators support an optional Break-Input signal clip, a Trigger-Output signal clip and seven user-placeable Trace-Capture-Input signal clips.

The following iceMASTER-PE emulators are currently available:

PE-8031-33

This emulator supports device operation from DC to 33MHz and has a 40-lead DIP footprint. This emulator supports the 8031 and 80C31 devices, regardless of process variation.

PW-8032-30

This emulator supports device operation from DC to 30MHz and has a 40-lead DIP footprint. This emulator supports the 8031, 80C31, 8032 and 80C32 devices, regardless of process variation.

PE-83752-16

This emulator supports device operation from DC to 16MHz and has a 28-lead DIP footprint. This emulator supports the 83C750, 87C750, 83C751, 87C751, 83C752 and 87C752 devices, regardless of process variation. This emulator is supplied with a 28-lead DIP to 24-lead DIP converter (CONV11) for 8xC751 support.

PE-8351FX

This emulator supports device operation from DC to 16MHz and has a 40-lead DIP footprint. This emulator supports the 8031, 80C31, 8032, 80C32, 8051, 80C51, 8751, 8052, 80C52, 8752, 87C52, 8xC51FX, 80C54, 87C54, 80C58 and 87C58, regardless of process variation.

The following iceMASTER-PE emulators will be available in Q2'94.

PE-87L51FB

This emulator supports device operation from 3.5 to 16MHz and has a 40-lead DIP footprint. This emulator supports the 87L51FB device from 3.0 to 5.5 volts.

PE-8XCL410

This emulator supports device operation from 0.5 to 16MHz and has a 40-lead DIP footprint. This emulator supports the 83CL410 and 80CL410 devices from 1.8 to 6 volts.

PE-80CL51

This emulator supports device operation from 32kHz to 16MHz and has a 40-lead DIP footprint. This emulator supports the 80CL51 and 80CL31 devices from 1.8 to 6 volts.

iceMASTER Model 200 & Model 400 Emulators

All iceMASTER Model 200 & Model 400 emulators support full probe card interchangeability. To emulate several different devices, you need purchase only a single emulator base unit, the iceMASTER-8051, and then one probe card for each unique device to be emulated.

Both the iceMASTER-8051 Model 200 and the iceMASTER-8051 Model 400 emulators come with 64K of Code memory and 64K of External Data memory, power supply, RS-232 cable and an 8051 Macro Cross Assembler.

The IceMASTER Model 400 emulator has the following additional features/capabilities:

7. 4K-Frame Trace Buffer
8. Performance Analyzers
 - a. High Resolution
 - b. High Bin Count
9. Full Watchdog Timer (WDT) Support in all emulation modes

Philips Semiconductors 8051 Family Probe Cards

All MetaLink probe cards include a Break-Input signal clip, a Trigger-Output signal clip and seven user-placeable Trace-Capture-Input signal clips. In addition, there is a user-selectable jumper for XTAL source. Most probe cards have other user-selectable jumpers to control/configure the unique capabilities of each particular device. Some probe cards also contain a user-selectable jumper for V_{CC} source. iceMASTER-8051 emulators fully supports:

All I/O ports, with no restrictions.

The CMOS Idle and Power Down modes of operation, in those devices having such capabilities.

The Watchdog Timer (WDT), in those devices having such a capability. (iceMASTER-8051 Model 400 only)

Correct device interconnection footprints.

Following is a list of all Philips Semiconductors 8051 Family Probe Cards which MetaLink currently offers. Since MetaLink is constantly adding to its list of supported devices, please contact MetaLink for information on any device not listed below.

If support for a particular device is available at different maximum frequencies (MHz), the description of the Probe Card appears only once, with the speed variations noted in the Probe Card names above the description. Example: 8031-24 for a 24MHz 8031 Probe Card and 8031-20 for a 20MHz 8031 Probe Card.

MHW-8031-20

MHW-8031-24

The 8031 Probe Card supports device operation from 0.5 to 24MHz and has a 40-lead DIP package interface. This probe card supports 8031 and 80C31 devices, regardless of process variation.

MHW-8032-20

The 8032 Probe Card supports device operation from 0.5 to 20MHz and has a 40-lead DIP package interface. This probe card supports 8031, 80C31, 8032, 80C32 and 80C32 devices, regardless of process variation.

MHW-80C51FA-16

The 80C51FA Probe Card supports device operation from 0.5 to 16MHz and has a 40-lead DIP package interface. This probe card supports 8031, 80C31, 8032, 80C51FA and 80C32 devices, regardless of process variation.

MHW-83C51FX-16

The 83C51FC Probe Card supports device operation from 0.5 to 16MHz and has either a 40-lead DIP or a 44-lead PLCC package interface. This probe card supports 8031, 80C31, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8x51FA, 8x51FB, 8052, 80C52, 8752 and 87C52 devices, regardless of process variation.

MHW-8052-16

The 8052 Probe Card supports device operation from 0.5 to 16MHz and has a 40-lead DIP package interface. This probe card supports 8031, 80C31, 8032, 80C32, 8051, 80C51, 8052, 80C52, 8751, 87C51, 8752, and 87C52 devices, regardless of process variation.

MHW-83C053-12

The 83C053 Probe Card supports device operation from 6 to 12MHz and has a 42-lead Shrink DIP package interface. This probe card supports 83C053, 83C054, 87C054, 83C055 and 87C055 devices, regardless of process variation.

MHW-80CL410-12

The 80CL410 Probe Card supports device operation from 0.5 to 12MHz and has a 40-lead DIP package interface. This probe card supports the 80CL410 device (4.5V–5.5V only).

MHW-80C451-16

The 80C451 Probe Card supports device operation from 1.2 to 16MHZ and has a 68-lead PLCC package interface. This probe card supports the 80C451 device, regardless of process variation. Converter #5 is available to convert the probe card to a 64-lead DIP footprint.

MHW-83C451-12

The 83C451 Probe Card supports device operation from 1.2 to 12MHz and has a 68-lead PLCC package interface. This probe card supports the 80C451, 83C451 and 87C451 devices, regardless of process variation. Converter #5 is available to convert the probe card to a 64-lead DIP footprint.

MHW-80C528-16

The 80C528 Probe Card supports device operation from 1.2 to 16MHz and has a 40-lead DIP package interface. This probe card supports the 80C528 device.

MHW-83C528-12**MHW-83C528-16**

The 83C528 Probe Card supports device operation from 1.2 to 16MHz and has either a 40-lead DIP or 44-lead PLCC package interface. This probe card supports 80C528, 83C528, 87C528, 83C524 and 87C524 devices, regardless of process variation.

MHW-83C550-12

The 83C550 Probe Card supports device operation from 1.2 to 12MHz and has a 44-lead PLCC package interface. This probe card supports the 83C550, 80C550 and 87C550 devices, regardless of process variation.

MHW-80C552-16

The 80C552 Probe Card supports device operation from 1.2 to 16MHz and has a 68-lead PLCC package interface. This probe card supports the 80C552 and 80C562 devices, regardless of process variation. Converter #7 is available to convert the probe card interface to a 40-lead DIP footprint for operation as an 8031, 8032, 80C31, 80C32 or 80C652.

MHW-83C552-16

The 83C552 Probe Card supports device operation from 1.2 to 16MHz and has a 68-lead PLCC package interface. This probe card supports the 80C552, 83C552, 87C552, 80C562, 83C562 and 87C562 devices, regardless of process variation. Converter #7 is available to convert the probe card interface to a 40-lead DIP footprint for operation as an 8031, 8032, 80C31, 80C32, 8x51, 8xC51, 8x52, 8xC52, 8xC652 or 8xC654.

MHW-83C575-16

The 83C575 Probe Card supports device operation from 0.5 to 16MHz and has either a 40-lead DIP or a 44-lead PLCC package interface. This Probe Card supports 80C575, 83C575 and 87C575 devices, regardless of process variation.

MHW-83C592-16

The 83C592 Probe Card supports device operation from 1.2 to 16MHz and has a 68-lead PLCC package interface. This probe card supports the 80C592, 83C592 and 87C592 devices, regardless of process variation.

MHW-80C652-16

The 80C652 Probe Card supports device operation from 1.2 to 16MHz and has a 40-lead DIP package interface. This probe card supports 8031, 80C31 and 80C652 devices from any IC manufacturer, regardless of process variation.

MHW-83C652-16

The 83C652 Probe Card supports device operation from 1.2 to 16MHz and has a 40-lead DIP package interface. This probe card supports 80C652, 83C652 and 87C652 devices from any IC manufacturer, regardless of process variation.

MHW-83C654-16

The 83C654 Probe Card supports device operation from 1.2 to 16MHz and has a 40-lead DIP package interface. This probe card supports 80C652, 83C654 and 87C654 devices from any IC manufacturer, regardless of process variation.

MHW-83C751-16

The 83C751 Probe Card supports device operation from 0.5 to 16MHz and has a 24-lead Skinny DIP package interface. This probe card supports 83C751 and 87C751 devices, regardless of process variation.

MHW-83C752-12

The 83C752 Probe Card supports device operation from 0.5 to 12MHz and has a 28-lead DIP package interface. This probe card supports 83C752 and 87C752 devices, regardless of process variation.

MHW-80C851-12

The 80C851 Probe Card supports device operation from 1.2 to 12MHz and has a 40-lead DIP package interface. This probe card supports 8031, 80C31, and 80C851 devices, regardless of process variation.

MHW-83C851-12

The 83C851 Probe Card supports device operation from 1.2 to 12MHz and has a 40-lead DIP package interface. This probe card supports 80C851, and 83C851 devices, regardless of process variation.

iceMASTER Converters

iceMASTER converters are used with the iceMASTER-8051 Probe Card to allow the user to change the device footprint supported by the Probe Card to new device footprint or to allow the iceMASTER-8051 Probe Card to support a new device with the existing Probe Card.

MHW-CONV5

A 68-lead PLCC to 64-lead DIP converter for 83C451 or 80C451 Probe Cards to 80C451, 83C451 and 87C451 devices.

MHW-CONV7

A 68-lead PLCC to 40-lead DIP converter for 83C552 or 80C552 Probe Cards to 8031, 8032*, 80C31, 80C32*, 8051, 8751, 80C51, 87C51, 8052*, 8752*, 80C52*, 80C652, 83C652, 87C652, 87C654, 83C654 and 87C52* devices.

*Not all modes of Timer 2 supported.

MHW-CONV11

A 28-lead DIP to 24-lead DIP converter for the PE-83752 or 83C752 Probe Card to 83C751 and 87C751 devices.

MHW-CONV12

A 24-lead Skinny DIP to 28-lead PLCC converter for 83C751 Probe Card to 83C751 and 87C751 devices.

MHW-CONV13

A 28-lead DIP to 28-lead PLCC converter for PE-83752 and 83C752 Probe Card to 83C752 and 87C752 devices.

MHW-CONV14

A 40-lead DIP to 44-lead PLCC converter for PE-8031, 8031 Probe Card to 8031, 80C31, 8031 and 80C31 devices.

A 40-lead DIP to 44-lead PLCC converter for PE-8032, 8032 Probe Card to 8031, 80C31, 8031, 80C31, 80C32, 8032 and 80C32 devices.

A 40-lead DIP to 44-lead PLCC converter for 8052 Probe Card to 8031, 80C31, 8031, 80C31, 80C32, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8052, 80C52, 8752 and 87C52 devices.

A 40-lead DIP to 44-lead PLCC converter for 8351FB Probe Card to 8031, 80C31, 8031, 80C31, 80C32, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8052, 80C52, 8752, 87C52, 8XC51FA and 8XC51FB devices.

A 40-lead DIP to 44-lead PLCC converter for 80410 Probe Card to 80CL410 devices.

A 40-lead DIP to 44-lead PLCC converter for 80652 and 83652 Probe Cards to 80C652, 87C652, 83C652, 87C654 and 83C654 devices.

A 40-lead DIP to 44-lead PLCC converter for 80528 and 83528 Probe Cards to 80C528, 87C528, 83C528, 87C524 and 83C524 devices.

A 40-lead DIP to 44-lead PLCC converter for 80851 and 83851 Probe Cards to 80C851 and 83C851 devices.

MHW-CONV19

A 68-lead PLCC to 44-lead PLCC converter for 83552 and 80552 Probe Cards to 8031, 8032*, 80C31, 80C32*, 8051, 8751, 80C51, 87C51, 8052*, 8752*, 80C52*, 80C652, 83C652, 87C652, 87C654, 83C654 and 87C52* devices.

MHW-CONV23

A 44-lead PLCC to 40-lead DIP converter for 80550 Probe Card to 80C550, 87C550 and 83C550 devices.

iceMASTER-8051 and PE Hardware Miscellaneous Products**MHW-EXT24DIP**

A 24-lead Skinny DIP, 6 inch target interface extension cable for 83C751 and 87C751 devices.

MHW-EXT28DIP

A 28-lead DIP, 6 inch target interface extension cable for 83C752 and 87C752 devices.

MHW-EXT40DIP

A 40-lead DIP, 6 inch target interface extension cable for 8031, 80C31, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8051FA, 83C51FA, 87C51FA, 8051FB, 83C51FB, 87C51FB, 8052, 80C52, 8752, 87C52, 80C528, 83C528, 87C528, 83C524, 87C524, 80C652, 83C652, 87C652, 83C654, 87C654, 80CL410, 80C851 and 83C851 devices.

MHW-EXT28PLCC

A 28-lead PLCC, 6 inch target interface extension cable for 83C751, 87C751, 83C752 and 87C752 devices.

MHW-EXT44PLCC

A 44-lead PLCC, 6 inch target interface extension cable for 8031, 80C31, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8051FA, 83C51FA, 87C51FA, 8051FB, 83C51FB, 87C51FB, 8052, 80C52, 8752, 87C52, 80C528, 83C528, 87C528, 83C524, 80C550, 83C550, 87C550, 80C652, 83C652, 87C652, 83C654, 87C654, 80CL410, 80C851 and 83C851 devices.

MHW-EXT68PLCC

A 68-lead PLCC 6 inch target interface extension cable for 83C552, 87C552, 83C562, 87C562, 80C552, 83C451, 87C451, 80C451, 87C592, 80C592 and 83C592 devices.

MHW-DIPSE24/3

A 24-lead Skinny DIP pin isolator for 83C751 and 87C751 devices.

MHW-DIPSE28/6

A 28-lead DIP pin isolator for 83C752 and 87C752 devices.

MHW-DIPSE40/6

A 40-lead DIP pin isolator for 8031, 80C31, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8051FA, 83C51FA, 87C51FA, 8051FB, 83C51FB, 87C51FB, 8052, 80C52, 8752, 87C52, 80C528, 83C528, 87C528, 83C524, 87C524, 80C550, 83C550, 87C550, 80C652, 83C652, 87C652, 83C654, 87C654, 80CL410, 80C851 and 83C851 devices.

MHW-DIPSE42S/6

A 42-lead Shrink DIP pin isolator for 87C054, 83C054 and 83C053 devices.

MHW-PLCSE28

A 28-lead PLCC pin isolator for 83C751, 87C751, 83C752 and 87C752 devices.

MHW-PLCSE44

A 44-lead PLCC pin isolator for 8031, 8032, 80C32, 8051, 80C51, 8751, 87C51, 8051FA, 83C51FA, 87C51FA, 8051FB, 83C51FB, 87C51FB, 8052, 80C52, 8752, 87C52, 80C528, 83C528, 87C528, 83C524, 87C524, 80C550, 83C550, 87C550, 80C652, 83C652, 87C652, 83C654, 87C654, 80CL410, 80C851 and 83C851 devices.

MHW-PLCSE68

A 68-lead PLCC pin isolator for 83C552, 87C552, 83C562, 87C562, 80C552, 83C451, 87C451, 80C451, 87C592, 80C592 and 83C592 devices.

iceMASTER-8051 and PE Software Miscellaneous Products**MSW-ASM51**

MetaLink 8051 Macro Cross Assembler

MSW-SIM51

Archimedes Software SimCASE 8051 for PC-hosted systems

MSW-SIMI/O51

Archimedes Software Sim I/O 8051 for PC-hosted systems

MSW-A51

Archimedes Software Assembler 8051 for PC-hosted systems

MSW-C51

Archimedes Software 8051 Family C Compiler for PC-hosted systems

MSW-AVA51

Avocet Systems, Inc. AvCase Assembler for the 8051

MSW-AVC51

Avocet Systems, Inc. AvCase C-Compiler and Assembler for the 8051

MSW-AVS51

Avocet Systems, Inc. AvCase Simulator for the 8051

MSW-BSO/A51

BSO/Tasking Assembler 8051 for PC-hosted systems

MSW-BSO/C51

BSO/Tasking C-Compiler and Assembler for 8051 for PC-hosted systems

MSW-BSO/PLM51

BSO/Tasking PL/M-Compiler and Assembler for 8051 for PC-hosted systems

MSW-F/4010

Franklin Software A51 Assembler for PC-Hosted Systems

MSW-F/5020

Franklin Software C-51 Compiler and A51 Assembler for PC-hosted systems

MSW-F/8220

Franklin Software 8051 Developer's Kit with C-51 Compiler, A51 Assembler, Simulator/Debugger for PC-hosted systems

MSW-F/8310

Franklin Software 8051 Professional Developer's Kit with C-51 Compiler, A51 Assembler, Bank Linker 51, Tiny Real Time Operating System 8051, Simulator/Debugger for PC-hosted systems

NOHAU EMUL51-PC – PC-based in-circuit emulator

1.0 System Architecture

Features of the Nohau EMUL51-PC in-circuit emulator include:

- Low-cost full real-time emulation
- IBM PC-bus plug-in boards, or stand alone Box version with 115K baud RS232-C connection to IBM PC
- Easy-to-learn user interfaces with windows and pull-down menus
- Choice of interfaces: Windows, DOS, ChipView (Borland keypress compatible), other third-party interfaces
- Source-level debugging in C, PL/M or Pascal with full support for typed symbols
- 256k frames by 64 bit real-time trace option with time stamp
- Program Performance Analyzer

The Nohau EMUL51-PC consists of a board which plugs directly into the IBM PC/XT/AT bus for fast file transfer. An optional external box with a serial link is also available. The LanICE option supports network workstations. The optional Trace board features an advanced trace function with many trigger capabilities.

The POD, which plugs into the target system, is connected with a 5 ft (1.5 m) ribbon cable to the Emulator board to provide a useful operating range.

The EMUL51-PC uses no wait states and does not intrude on memory, stack, I/O or interrupt pins.

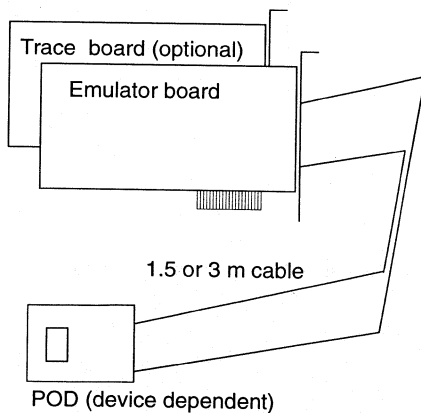


fig 1 EMUL51-PC plug-in board version

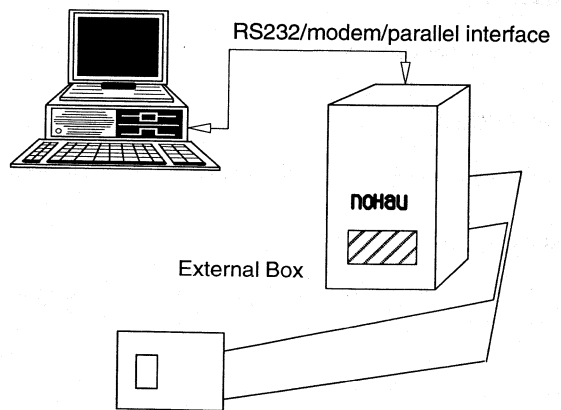


fig 2 External Box version

2.0 User Interfaces

2.1 DOS Interface

The user interface is based on MS-DOS software. It incorporates a variety of techniques. The user can select to operate either with short typed commands or using pull-down menus. On-screen features include windows to monitor or alter:

- Assembly or high level language source code
- CPU registers
- Internal data and Special Function Registers
- External data
- Watch variables in C, PL/M or assembly
- Trace setup and display

All commands are supported with context-sensitive help and full on-line manual.

2.2 Windows Interface

- Pull-down menus
- Speed bar (point and click)
- Source Window
- Watch Window for high-level variables

All the ease and flexibility of Microsoft Windows.

2.3 ChipView-51 Interface

- High Level / Low Level Debugger
- Keypress-compatible with Borland's Turbo Debugger
- Resizable moveable windows
- C call stack
- Support all emulator variations

3.0 Specific Features

3.1 Source Level Debugging

Using high level language for code generation is a way to cut development time. Therefore the EMUL51-PC gives full support for debugging directly in C or PL/M source code. This eliminates the need for paper listings. Breakpoints can be marked directly in the source code window. The user can single step through the program line by line and follow the program execution on the screen listing.

The trace permits the user to trace his source code in real-time.

All variables can be displayed and altered. This includes full support for typed symbols which permits the user to address such variables as floating-point, arrays and structures — both global and local.

3.2 Emulation Memory

The EMUL51-PC features 64 K of code memory and 64 K of external data memory. The addressable memory can be mapped either to target or to the emulator in 4K pages.

The emulator can load all common file formats and the user can view and alter all registers and memory areas of the microcontroller.

3.3 Breakpoints

The EMUL51-PC permits the user to generate program breakpoints in a number of ways:

- 64K program breakpoints
- 64K data read and 64K data write breakpoints
- Break on external signal
- Break on direct access to internal bit or byte memory
- Break on contents of internal register or memory
- Break on program access out-of-boundary

Using the Trace board it is possible to break on combinations of address, data, control signals, port signals and external signals.

3.4 Macros and debug session logging

Test session automation is made possible using the Macro commands. This permits the user to define his own command sequences using structures like IF/ELSE and REPEAT/WHILE. Such command sequences can be stored to a file which can be loaded automatically when the emulator is invoked.

A complete debug session and all setups can also be recorded to a file.

3.5 Trace Memory

The optional trace board features a trace buffer capable of storing up to 256k frames of 64 bit data each. The 64 bits consist of address, data, control signals, port signals, external signals and a time stamp.

The trace memory can be displayed, reprogrammed and restarted during emulation.

3.5.1 Trace Filter

The trace filter makes it possible to select what events are stored in the trace buffer. The qualifiers permit the user to define the criteria for which bus cycles are stored. The qualifiers can specify address, data, control signals, port signals and external signals.

3.5.2 Trace Trigger

The trace works much like a logic analyzer. It is therefore possible to trigger the trace on an event and display what happened before or after that event.

The trigger event can be defined using any of the qualifiers in up to eight levels. It is possible to trigger on boolean combinations of the qualifiers, or sequential combinations including a loop counter.

The trigger point can be selected anywhere within the 256k trace buffer to give full choice of pre/post trigger alignment.

3.5.3 Trace Display

The trace information can be displayed in high-level language statements, in disassembled form or in binary/hex form. It can also be stored to a file.

3.5.4 Program Performance Analyzer

With the PPA, information can be generated showing which addresses the user program spends its time on. The information can be displayed as statistics or in histogram form.

4.0 General Specifications

- Host:** IBM PC/XT/AT/386/486, PS/2 or compatible with 640K of RAM. Monochrome, color or enhanced graphics display.
- External Box:** The emulator boards can be installed in an external box with serial communication to the PC.
- Processors supported:** 8051, 8052, 8031, 8032, 80C51, 80CL51, 80C52, 80CL52, 80C31, 80CL31, 80C32, 8XC053/054/055, 83/80CL410/610, 83/80C451, 8XC524, 8XC528, 8XC550, 83/80C552, 8XC558, 80C562, 8XC575, 80CL580, 8XC592, 8XC598, 83/80C652, 8XC748, 8XC749, 83/87C750, 83/87C751, 83/87C752, 8XCL781, 8XCL782, 83/80C851, 8XC51FA/FB (For more details, please refer to the following pages.)
A switch from one microcontroller to another is made by changing the low cost POD.
- File formats supported:** Intel HEX/OBJ/SYM/OMF, Avocet, Archimedes/IAR, BSO/Tasking, Franklin/Keil, Intermetrics/Whitesmiths/Comac, and many more.
- Clock speed:** Allows operation up to 33 MHz in real-time.
- Power supply:** The boards are powered from the PC-bus. The Emulator board requires 1.7A/5V and the Trace board 1.3A/5V.

5.0 Ordering information

PHILIPS SEMICONDUCTORS Microcontroller Support

EMULATOR UNITS

Includes software and ribbon cable. Must be connected to a POD to operate. Each step up in frequency rating covers all lower frequency steps. Includes DIP isolator if ordered with 40-pin POD. Emulator software has high-level debug and all options.

Order Number	Description
EMUL51-PC/E32	12 MHz Emulator, 32 kB (kiloByte) Emulation Memory.
EMUL51-PC/E32-16	16 MHz Emulator, 32 kB Emulation Memory.
EMUL51-PC/E128	12 MHz Emulator, 128 kB Emulation Memory.
EMUL51-PC/E128-16	16 MHz Emulator, 128 kB Emulation Memory.
EMUL51-PC/E128-20	20 MHz Emulator, 128 kB Emulation Memory.
EMUL51-PC/E128-24	24 MHz Emulator, 128 kB Emulation Memory.
EMUL51-PC/E128-30	30 MHz Emulator, 128 kB Emulation Memory.
EMUL51-PG/E128-33	33 MHz Emulator, 128 kB Emulation Memory.

TRACE BOARD OPTIONS

Optional second PC plug-in board. Emulator board contains no trace capability. Each frequency step covers all lower steps.

Order Number	Description
EMUL51-PC/TR4	12 MHz 4 kiloframe (4096 frames) Trace Buffer.
EMUL51-PC/TR4-16	16 MHz 4 k Trace Buffer.
EMUL51-PC/TR16	12 MHz 16 k Trace Buffer.
EMUL51-PC/TR16-16	16 MHz 16 k Trace Buffer.
EMUL51-PC/TR16-20	20 MHz 16 k Trace Buffer.
EMUL51-PC/TR16-24	24 MHz 16 k Trace Buffer.
EMUL51-PC/TR16-30	30 MHz 16 k Trace Buffer.
EMUL51-PC/TR16-33	33 MHz 16 k Trace Buffer.

Advanced Trace Boards feature 32-bit timestamping with 16-bit prescaler, eight-level triggers, state and counter functions, search.

Order Number	Description
EMUL51-PC/ATR64-16	16 MHz 64 k Advanced Trace Buffer.
EMUL51-PC/ATR256-16	16 MHz 256 k Advanced Trace Buffer.
EMUL51-PC/ATR64-24	24 MHz 64 k Advanced Trace Buffer.
EMUL51-PC/ATR256-24	24 MHz 256 k Advanced Trace Buffer.
EMUL51-PC/ATR64-33	33 MHz 64 k Advanced Trace Buffer.
EMUL51-PC/ATR256-30	30 MHz 256 k Advanced Trace Buffer.

BONDOUT PODS, 24-PIN, 40-PIN, 68-PIN, 84-PIN

Allow full emulation of internal, external and mixed modes of bus or port input/output. On-board POD crystal is 12 MHz on all PODs of any frequency specification. Each step up in frequency rating covers all lower frequency steps.

Order Number	Description
POD-C51B	12 MHz Bondout POD for 80C51, 8051, 87C51, 8751, 80C31, 8031, 80C/83C652, 83C654, 80C/83C662, 80C/83C851 single-chip or external mode.
POD-C51B-16	16 MHz Bondout POD for 80C/87C51-1, 80C/87C51, 80/8751, 80C31-1, 80C31, 8031, 80C/83C652, 83C654, 80C/83C662, 80C/83C851 single-chip or external mode.
POD-C51B-24	24 MHz POD for 80C/87C51-24, 80/8751, 80C31-24, 80C/83C652/654, 80C/83C662, 80C/83C851 single-chip or external mode. Special Requirement: Due to bondout chip timing, this POD requires a 30 MHz emulator board and trace board must be 30 MHz if used.
POD-CL410	POD for 83CL410, 83CL610, 80CL31, 80CL51. Target voltage range: 1.5–5.0V. Maximum frequency: 12 MHz at 5.0V. This POD is intended for emulating internal code. External bus operation is limited.
POD-C451B-PGA	12 MHz Bondout POD for 83C451, 87C451, 80C451 PLCC, single-chip or external mode. PGA from POD. Use optional adapter for PLCC target.
POD-C451B-PGA-16	16 MHz Bondout POD for 83C451, 87C451, 80C451 PLCC, single-chip or external mode. PGA from POD. Use optional adapter for PLCC target.
POD-C552B-PGA	12 MHz Bondout POD for 83C552, 87C552, 80C552 PLCC, single-chip or external mode. PGA from POD. Use optional adapter for PLCC target.
POD-C552B-PGA-16	16 MHz Bondout POD for 83C552, 87C552, 80C552 16 MHz, 12 MHz single-chip or external mode. PGA from POD. Use optional adapter for PLCC target.
POD-C552B-PGA-24	24 MHz Bondout POD for 8XC552 or 8XC562 PLCC single-chip or external mode. PGA from POD. Use optional adapter (etc). Special Requirement: Due to bondout chip timing, this POD requires a 30MHz emulator board and trace board must be 30 MHz if used.
POD-CL580	12MHz Bondout POD for 83CL580.
POD-C652B	Order as POD-C51B.
POD-C751	12 MHz 83C750, 87C750, 83C751, 87C751, DIP POD. Includes EXT-DIP24.
POD-C751-16	16 MHz 83C750, 87C750, 83C751, 87C751, DIP POD. Includes EXT-DIP24.
POD-CL782	12 MHz POD for 83CL781, 83CL782, 83CL52

“HOOKS” MODE PODS, 28-PIN, 40-PIN, 42-PIN, 44-PIN, 68-PIN

Standard chip operated in special “hooks” emulation mode for single-chip or external modes. Some electrical characteristics are different from microcontroller’s. On-board POD crystal is 12 MHz on all PODs of any frequency specification.

Order Number	Description
POD-C52	12 MHz POD for 80C31, 80C32, 80C51, 80C52, 87C51, 87C52.
POD-C52-16	16 MHz POD for 80C31, 80C32, 80C51, 80C52, 87C51, 87C52.
POD-L51P-16	16 MHz POD for 8XC51FA/FB.
POD-C528	12 MHz POD for 80C528, 83C528, 87C528, 83C524, 83C528.
POD-C528-16	16 MHz POD for 80C528, 83C528, 87C528, 83C524, 83C528.
POD-C550-PGA	12 MHz POD for 80C550, 83C550, 87C550. PGA from POD. Use optional adapter for PLCC target.
POD-C550-PGA-16	16 MHz POD for 80C550, 83C550, 87C550. PGA from POD. Use optional adapter for PLCC target.
POD-C558-16	16 MHz POD for 83CE558/89CE558. Use optional adapter.
POD-C575	12 MHz 87C575, 87C575 chip may have to be supplied by user.
POD-C592-PGA	12 MHz POD for 8XC592. 8XC592 chip may have to be supplied by user. PGA from POD. Use optional adapter for PLCC target.
POD-C592-PGA-16	16 MHz POD for 8XC592. 8XC592 chip may have to be supplied by user. PGA from POD. Use optional adapter for PLCC target.
POD-C752	12 MHz 83C752, 87C752 DIP POD. Includes EXT-DIP28.
POD-C752-16	16 MHz 83C752, 87C752 DIP POD. Includes EXT-DIP28.
POD-C054	12 MHz POD for 8XC053, 8XC054, 8XC055 (Microcontroller-for-Television-Video).
POD-C575	16 MHz POD for 8XC575.

POD BOARDS, 40 PIN EXTERNAL MODE

Port 2 is upper address bus only. Port 0 is address/data bus. P3.6 is WRITE, P3.7 is READ. On-board POD crystal is 12 MHz on all PODs of any frequency specification. Each step up in frequency rating covers all lower frequency steps.

Order Number	Description
POD-31	12 MHz 8031 POD.
POD-C31	12 MHz 80C31 POD.
POD-32	12 MHz 8032 POD.
POD-C32	12 MHz 80C32 POD.
POD-C652	12 MHz 80C652 POD.
POD-C31-1	16 MHz 80C31-1 POD.
POD-C32-16	16 MHz 80C32 POD.
POD-C31-20	20 MHz 80C31 POD.
POD-C31-24	24 MHz 80C31 POD.
POD-C31-30	30 MHz 80C31 POD.
POD-C31-33	33 MHz 80C31 POD.

-S version of the above boards allows P3.6, P3.7 to be used either as unidirectional I/O or write and read. (Example: POD-31-S; POD-C31-S-1; POD-C32-S-16.)

EXTERNAL MODE PODS, 64-PIN, 68-PIN

Port 2 is upper address bus only. Port 0 is address/data bus. P3.6 is WRITE, P3.7 is READ. On-board POD crystal is 12 MHz on all PODs of any frequency specification. Each step up in frequency rating covers all lower frequency steps.

Order Number	Description
POD-C451-DIP	12 MHz 80C451 DIP POD.
POD-C451-DIP-16	16 MHz 80C451 DIP POD.
POD-C451-PGA	12 MHz 80C451 PLCC POD. PGA from POD.
POD-C451-PGA-16	16 MHz 80C451 PLCC POD. PGA from POD.
POD-C552-PGA	12 MHz 80C552 POD. PGA from POD. Use optional adapter for PLCC target.
POD-C552-PGA-16	16 MHz 80C552 POD. PGA from POD.
POD-C552-PGA-24	24 MHz 80C552 POD. PGA from POD.
POD-C552-PGA-30	30 MHz 80C552 POD. PGA from POD.
POD-C562-PGA-16	16 MHz 80C562 POD. PGA from POD.

LanICE LOCAL AREA NETWORK OPTION FOR XWindows (Sun, HP, and other workstations)

Order Number	Description
LanICE	XWindows Sun or HP workstation option for TCP/IP networks. Order this LanICE plus any emulator and POD unit. Trace buffer units are optional. The LanICE can accommodate multiple emulators.

BOX OPTIONS

Box units are AC line-powered. Emulator software runs under DOS on PC and uses a COM port at 110 baud to 115 kilobaud. Serial cable included.

Order Number	Description
EMUL51-PC/BOX-S	Serial Box. Select emulator separately; trace optional. POD not included.
EMUL51-PC/BOX-CS	Serial Box with E128-16 Emulator and TR16-16 Trace. POD not included.
EMUL51-PC/BOX-CS-20	Serial Box with E128-20 Emulator and TR16-20 Trace. POD not included.
EMUL51-PC/BOX-CS-24	Serial Box with E128-24 Emulator and TR16-24 Trace. POD not included.
EMUL51-PC/BOX-CS-30	Serial Box with E128-30 Emulator and TR16-30 Trace. POD not included.

MISCELLANEOUS OPTIONS

Order Number	Description
EMUL51-PC/PRG	Universal Programmer (EPROM, 8751, 87C51, PAL).
EMUL51-PC/PRG751	Programmer for 87C751, 87C752.
PHILIPS/LCPX5X	Single board programmer for DIP parts: 87C750, 'C751 and 'C752; PLCC parts: 87C451 and 'C552.
PHILIPS/LCPX5X40	Single board programmer for Philips 87C51, 'C51FA, 'C51FB, 'L51FB, 'C52, 'C524, 'C528, 'C550, 'C575, 'C652 and 'C654 DIP or PLCC.
EMUL51-PC/EXT-DIP24	Additional extender cable for 24 pin DIP.
EMUL51-PC/DIP24-ISO	24 pin DIP Isolator.
EMUL51-PC/DIP24-PLCC28	24 pin DIP to 28 pin PLCC.
EMUL51-PC/EXT-DIP24-PLCC28	Extender cable, 24 pin DIP to 28 pin PLCC.
EMUL51-PC/EXT-DIP28	Extender cable, 28 pin DIP.
EMUL51-PC/DIP28-PLCC28	28 pin DIP to 28 pin PLCC.
EMUL51-PC/DIP28-ISO	Additional 28 pin DIP Isolator.
EMUL51-PC/DIP28-DIP24-ADAP	28-pin (0.600-In) to 24-pin (0.300-In) adapter to plug POD-C752 into 8XC751 target. The user is responsible for port and register compatibility.
EMUL51-PC/EXT-DIP40	Extender cable for 40 pin DIP.
EMUL51-PC/DIP40-ISO	Additional 40 pin DIP Isolator.
EMUL51-PC/DIP40-PLCC44	40 pin DIP to 44 pin PLCC.
EMUL51-PC/DIP40-ONCE-DIP40	Clips over target microcontroller. Disables target micro to allow emulation without removing target chip. Works only on chips with on-chip emulation (ONCE) disable feature.
EMUL51-PC/DIP40-ONCE-PLCC44	Clips over target microcontroller. Disables target micro to allow emulation without removing target chip. Works only on chips with on-chip emulation (ONCE) disable feature.
EMUL51-PC/PGA44-PLCC44	PGA to PLCC adapter, 44 pin.
EMUL51-PC/PGA44-PLCC44-EL2	PGA to PLCC "elevator" or "tower" rigid 2-inch extender-adapter.
EMUL51-PC/PGA44-DIP40-C550	Adapter to use POD-C550-PGA in DIP target.
EMUL51-PC/EXT-DIP48	Extender cable for 48 pin DIP.
EMUL51-PC/DIP48-ISO	48 pin DIP Isolator.
EMUL51-PC/PGA68-PLCC68	PGA to PLCC adapter unit, 68 pin.
EMUL51-PC/PGA68-DIP84	PGA to DIP adapter unit for 451 PGA PODs to plug into DIP target.
EMUL51-PC/PGA68-ISO	68 pin PGA Isolator.
QILEXT-1	Extractor tool for PLCC parts.
EMUL51-PC/EZ	E-Z-Hook® wires for trace.
	<small>E-Z-Hook is a registered trademark of Tektest, Inc.</small>
EMUL51-PC/CBL10-S	10 foot substitute for 5 foot POD cable (not for bondout PODs).
EMUL51-PC/CBL10-A	Additional 10 foot POD cable (not for bondout PODs)
EMUL51-PC/CBL5-A	Replacement 5 foot POD cable

BANKSWITCH EMULATOR BOARDS

User selectable as two banks of 64 kBytes, or as three switchable banks in upper half (8000 – FFFF) with lower half (0000 – 7FFF) not switchable. MOVX read-write data memory is only separate from code if data is mapped to target.

Order Number	Description
EMUL51-PC/E128-BSW	12 MHz Bankswitch Emulator, 128 kB Emulation Memory. Requires Bankswitch POD.
EMUL51-PC/E128-BSW-16	16 MHz Bankswitch Emulator, 128 kB Emulation Memory. Requires Bankswitch POD.
EMUL51-PC/E128-BSW-24	24 MHz Bankswitch Emulator, 128 kB Emulation Memory. Requires Bankswitch POD.
EMUL51-PC/E128-BSW-33	33 MHz Bankswitch Emulator, 128 kB Emulation Memory. Requires Bankswitch POD.
EMUL51-PC/E256-BSW	12 MHz Bankswitch Emulator, 256 kB Emulation Memory. Requires Bankswitch POD.
EMUL51-PC/E256-BSW-16	16 MHz Bankswitch Emulator, 256 kB Emulation Memory. Requires Bankswitch POD.
EMUL51-PC/E256-BSW-24	24 MHz Bankswitch Emulator, 256 kB Emulation Memory. Requires Bankswitch POD.
-BSW option to any POD	
Examples:	
POD-31-BSW	12 MHz 8031 Bankswitch POD.
POD-C31-BSW-1	16 MHz 80C31-1 Bankswitch POD.
POD-31-S-BSW	12 MHz 8031 -S option Bankswitch POD.

SOFTWARE PACKAGES

Order Number Description

NOHAU Corporation
EMUL51-PC/SWSUB

EMUL51-PC emulator software update service, one year.

Avocet Systems, Inc.
AVOCET/AVA51

Avocet Systems AvCase51 8051 Assembler

Avocet is a registered trademark of Avocet Systems, Inc.

BSO/Tasking
BSOTSK/C51PKG
BSOTSK/PLM51PKG

BSO/Tasking 8051 Family C-Compiler and Assembler Package.
BSO/Tasking 8051 Family PL/M Compiler and Assembler Package.

Chip Tools, Inc.
CV51-NOH
CV51-S
CV51-SNOH

Chip Tools Chip View-51 High-Level/Low-Level Debugger Emulator interface for EMUL51-PC.
Chip Tools Chip View-51 High-Speed Simulator.
Chip Tools Chip View-51 CV51-NOH + CV51-S Combo package.

Chip Tools and Chip View are trademarks of Chip Tools, Inc.

Franklin Software, Inc.
FRANKLIN/4020
FRANKLIN/5020
FRANKLIN/7020
FRANKLIN/8220

Franklin 8051 Macro Assembler with Linker, Librarian, Object-to-Hex utility.
Franklin V4 Very High Performance 8051 Compiler & Assembler.
Franklin V5 Simulator/Debugger, Windowed Interface.
Franklin Developers Kit with 5020 Compiler, 4020 Assembler and 7020 Simulator/Debugger.

Franklin is a trademark of Franklin Software, Inc.

Intermetrics Microsystems Software, Inc./Whitesmiths, Ltd.

INTERMET/C851HX
INTERMET/D851HXNOH
INTERMET/D851HXSIM

Intermetrics Whitesmiths C-Compiler and Assembler, Extended.
Intermetrics Whitesmiths C Source-Level Debugger/Emulator Version (Interface for EMUL51-PC).
Intermetrics Whitesmiths C Source-Level Debugger/Simulator Version.

Whitesmiths and CXDB are registered trademarks of Intermetrics, Inc.

The EMUL51-PC Emulator, Trace, POD, and Box hardware is sold with a one-year warranty. The EMUL51-PC Emulation software is sold with no warranty, but upgrades will be distributed to all customers up to one year from the date of purchase. Nohau Corporation makes no warranties, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. In no event will Nohau Corporation be liable for consequential damages. The SIMUL51-PC Simulator software includes updates for a period of one year. Third-party software and programmers sold by Nohau carry manufacturers' warranties.

CHOOSING PODS for the EMUL51-PC

People sometimes ask: "How do I choose which POD to use for the 8051-family chip I want to emulate?"

Which POD you pick for your application depends on several things. What micro you are using, the frequency, the mode you are using it in, your target configuration, and price all play a part. The **EMUL51-PC ORDER INFORMATION** can help you choose among the POD types available from Nohau.

Nohau has several categories of PODs: (1) External-mode PODs; (2) Bondout PODs; and (3) "Hooks"-mode PODs.

1. External-Mode PODs

External or microprocessor-mode PODs have several clear advantages. They are the cheapest type of POD. They have easily-replaceable standard microcontrollers. Your program runs on the real production part, with most of the lines directly connected to your target. You can use them where your microcontroller has external program or external read-write memory. With this type of POD, Port 2 is used only for the upper address bus and not as port input-output (I/O). Port 0 is used as a multiplexed address and data bus. In most of these PODs, P3.6 can only be used as the write line and P3.7 as only the read line. The **POD-31** is a typical external POD.

Your target system shares Port 0 and Port 2 with the emulator. The emulator uses Port 0 to run its monitor code when it is not executing your code in real-time. When you are not running real-time emulation, the POD holds the Program Store ENable line (PSEN/) high so that your external read-only memory (ROM) is not enabled. It also holds the ReaD/ line and WRite/ lines high so that none of your external read-write random access memory (RAM) or I/O is enabled.

But in the monitor (not emulating) mode, the address lines of both Port 2 and Port 0 are active and can output any address in the 64 kByte address range. It is up to your target system to prevent enabling any device unless the PSEN/ or the RD/ or the WR/ line goes low.

If you are emulating a ROM-less part, like the 8031, you can use the **POD-31** type of POD. You also might use this POD for some internal ROM applications. You could use it for designs that have all of Port 2 and Port 0 used as external buses. This is true even though the program can be executed from on-chip 8051 program memory in your final product. This is a case where your target schematic, rather than the device you are using lets you use this type of POD. So though the part may ultimately be an 8051 or 8751, you can use a **POD-31** because you are using the ports as buses.

If you are emulating the 80C31, 8032, 80C32 or 80C652, you can get external-mode PODs for these microcontrollers. With the **POD-31**, you can emulate all those parts in external mode at up to 12 MHz. You can get the POD with the correct micro installed, such as a **POD-32**. Or you can change among the types by changing the microcontroller component in the POD yourself.

16 MHz, 20 MHz, 24 MHz, 30 MHz and 33 MHz PODs are available for 40-pin parts. If you use a **POD-C31-1** with a 16 MHz rated emulator, you can support such parts as the 80C31-1, 80C32-1 and 80C652-1. Any higher-frequency POD and emulator set can support all the lower frequencies.

For emulating the 80C451, 80C552, or 80C562 other external mode PODs are available. PODs for dual inline package (DIP) parts have a DIP plug coming out of the bottom of the POD. PODs for plastic leaded chip carrier (PLCC) parts have a pin-grid array (PGA) plug coming out from the bottom of the POD. To plug a PGA POD into a PLCC socket, you can get an optional adapter. The PGA68-PLCC68 is a typical adapter. If your target board has feed-through holes and you solder a PGA socket into it, you don't need an adapter. If your target board already has a PGA socket, you don't need an adapter.

POD-31-S: All the above external PODs use P3.6 as WR/ and P3.7 as RD/. But there is a 40-pin external mode POD version that lets you use these two lines as I/O. The basic board is the **POD-31-S**. This special -S option lets you select whether the two lines are to be WR/ and RD/ or else I/O. They can be unidirectional port lines (input-input, input-output, output-input, or output-output). You also can get this POD in the variations of 16 MHz and 20 MHz and any of the 40-pin part variations listed above.

2. Bondout PODs

Bondout PODs use special microcontrollers that the semiconductor manufacturers designed to perform all the functions of the part. They also have additional lines "bonded out" from the chip that allow control for emulation. PODs with these special chips cost more than PODs with widely available commercial chips, but give you a greater flexibility in how you use the ports. These PODs allow you to use any combination of internal or external mode of the chip, and respond to the state of the External Access (EA) pin.

Nohau has bondout PODs for many different microcontrollers.

You can emulate the 80/80C51, 87/87C51, 80/80C31, 80C/83C/87C652, 83C/87C654, 80C/83C662 and 80C/83C851 with the **POD-C51B**. 16 MHz and 24 MHz versions are available.

To emulate the 83C451, 87C451 and 80C451 in the PLCC package, you can use the **POD-C451B-PGA**. See the previous discussion about PGAs.

For emulating the 80C/83C/87C552 and 80C/83C/87C562 you can use the **POD-C552B-PGA**. It also is available in a 16 MHz and 24MHz versions.

For emulating the 83C750, 87C750, 83C751 and 87C751, use the **POD-C751**. It has a 24 pin DIP plug on the bottom of the POD. You also can get an adapter to plug into 28 pin PLCC sockets. A 16 MHz version is available.

For emulating the 8XCL410, 8XCL31, and 8XCL51, use the **POD-CL410**.

For emulating the 83CL580, use the **POD-CL580** and a special adapter.

For emulating the 83CL781, 87CL781, 83CL782, 87CL782 and 80CL52, use the **POD-CL782**.

You should be aware of one small problem that most 8051 bondout parts have with serial communication. Specifically, if your program has written to SBUF, but the TI flag has not been set, and you then break emulation, the TI flag will never be set after you resume emulation.

Bondout PODs give you the best of both worlds. They let you mix how you use the ports. So you can emulate the single-chip mode using the ports as I/O. Or you can emulate external bus mode. Or you can emulate a mix of both modes.

3. "Hooks"-Mode PODs

The "hooks" type of POD Nohau offers uses a different technique for emulating. Certain chips can be put into a proprietary "hooks" mode that multiplexes the port information in and out of the part. The chip can still put out address and data. Most have 16 MHz versions available. This group includes these PODs:

The **POD-C752** is for emulating the 83C752 and 87C752. It has a 28 pin DIP plug on the bottom of the POD. You also can get an adapter to plug into 28 pin PLCC sockets.

The **POD-C52** can emulate the 80/80C/87/87C51, 80/80C/87/87C52, 80/80C31 and 80/80C32. You can use this POD when you need to emulate the 8052-type parts in single-chip mode when you are using Timer 2.

The **POD-L51P-16** can emulate the 8XL51FA/FB and can operate down to 3.0 volts.

The **POD-C528** can emulate the 80/80C/87/87C528 and 83/87C524.

The **POD-C592-PGA** can emulate the 8XC592. See the previous discussion about PGAs.

The **POD-C550-PGA** can emulate the 80/80C/87/87C550. It has a 44-pin PGA plug. You also can get an adapter to plug into 44-pin PLCC sockets.

The **POD-C558-16** can emulate the 83CE558 ad 89CE558.

The **POD-C575** can emulate the 8X575.

The **POD-C054** can emulate the 8XC053/054/055 "MTV" chip.

With the "Hooks"-Mode PODs, you use jumpers to select whether the code is to be fetched from the emulator RAM or from external ROM. You also select whether the read-write memory is in the emulator or on your target board. The ports on these PODs may have slightly different electrical characteristics than the microcontroller. Specifically, some output signals can appear up to three clock cycles later than on the actual part. Also, some ports have greater current sink or sink and source capacity or slightly higher impedance than the actual part. For almost all applications, these differences will have no detrimental effects.

You can use any size **EMUL51-PC** emulator and optional trace board with any POD you choose in Nohau's **EMUL51-PC** family. Choose an emulator and trace board with frequencies as fast or faster than the frequency of your fastest POD. If you need more information about which POD would best fit your application, please contact Nohau.

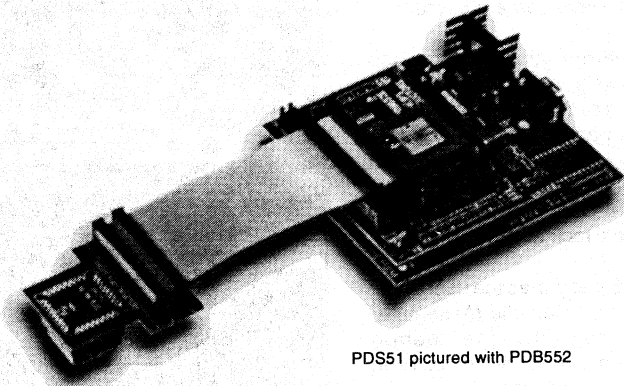
Description

The PDS51 is a board-level, full featured, 12 MHz In-Circuit Emulator for the Philips 8XC51 family of Microcontrollers. It allows the user complete access to the internal registers and full execution control with no chip resources being consumed. This means that the microcontroller in the target system can be replaced with the PDS51 enabling the target system to be easily run, monitored and debugged without any changes to code or hardware. The PDS51 is supplied with sufficient memory to enable emulation of the complete 64K of directly addressable code memory and a 28K line by 32 bit wide, real-time trace buffer.

The PDS51 system consists of two interconnected modules. One board (the motherboard), contains the systems that are common to any derivative emulation such as the control microcontroller, logic, communication interface and power supply. The second board (the daughterboard), contains the bondout microcontroller which determines the target devices able to be emulated. Most daughterboards are able to emulate more than one derivative, thereby enhancing system flexibility with minimal expense.

External breakpoint and trace control inputs, along with emulation running and fetch address controlled output trigger signals, are provided for interfacing and synchronising to external equipment.

The PDS51 is controlled via an RS232 serial interface by a PC running terminal emulation or the PDS51-IDE debugger supplied. The Integrated Development Environment is a windowed command and display environment which runs on PC XT, AT, 386, 486 or compatibles. The PDS51 requires an external 6.5-9.5V 750mA DC power source.



PDS51 pictured with PDB552

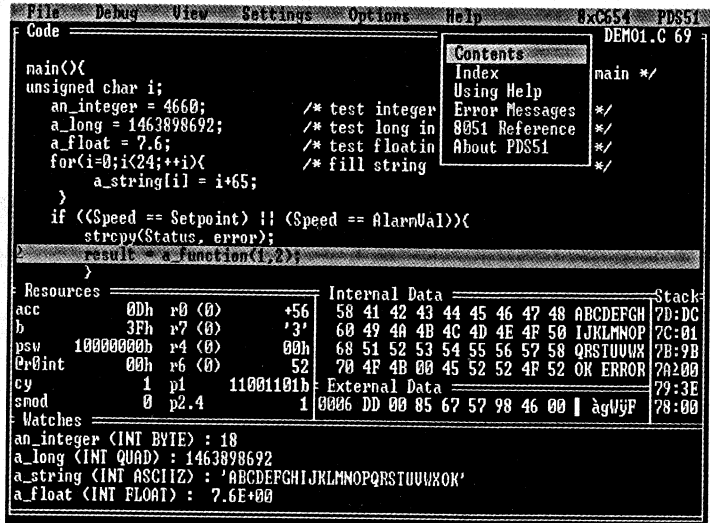
FEATURES

- Supports Philips 80C51 family
- Universal motherboard
- Bondout dependent daughterboard
- In-Circuit emulation
- No stolen resources
- 64K emulation code space
- Hardware breakpoints (64K) on:
 - Fetch address
 - Fetch address and external signal
- Real-time trace (28K frames)
- Traced information:
 - Fetch Address
 - 12 External trace probes
 - Control info (inta, c1)
- Conditional trace on:
 - Fetch address
 - External signal
- RS232 interface to PC
- Controlled via the Integrated Debugger or Terminal Emulation using the SDS command set
- Integrated Development Environment:
 - Windowed interface
 - Pull down menus
 - Borland style keystrokes
 - Symbolic or Source level debugging
- File formats supported:
 - BSO/Tasking
 - Kiel
 - Franklin
 - IAR
 - Archemedies
 - Metalink
 - Intel Hex
 - OMF51
- On-line help:
 - context sensitive
 - hypertext style

The PDS51 IDE (Integrated Development Environment) is a full screen, full featured debug and analysis interface to the PDS51 Development System.

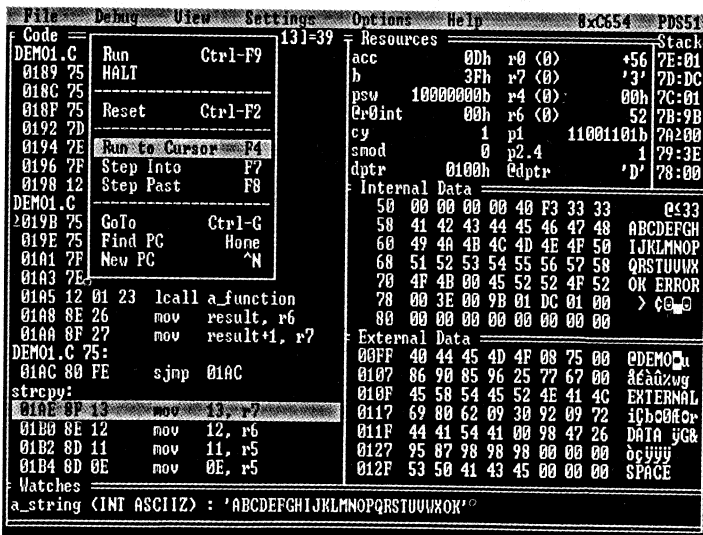
It provides an easy to use windowed environment that consists of 6 non-overlapping resizable panes that display Code, SFR Resources, Watch Variables, Internal Data RAM, External Data RAM (if enabled) and the Stack area.

The IDE integrates the user's text Editor and Compiler/Assembler programs into the debug environment by providing fast and easy hot-key access.



The enhanced PDS51 IDE now facilitates source-level debugging for a variety of high level language compilers for the 80C51 including BSO/Tasking, Franklin/Keil and IAR/Archemedes 'C' compilers, as well as assemblers like Metalink.

Borland compatible hot-keys provide fast and familiar access to frequently used functions, while pull down menus lead the new user to the desired operation. A comprehensive on-line help facility provides indexed, context sensitive, and hypertext style help on PDS51 IDE operation, as well as 80C51 architecture and instruction set.

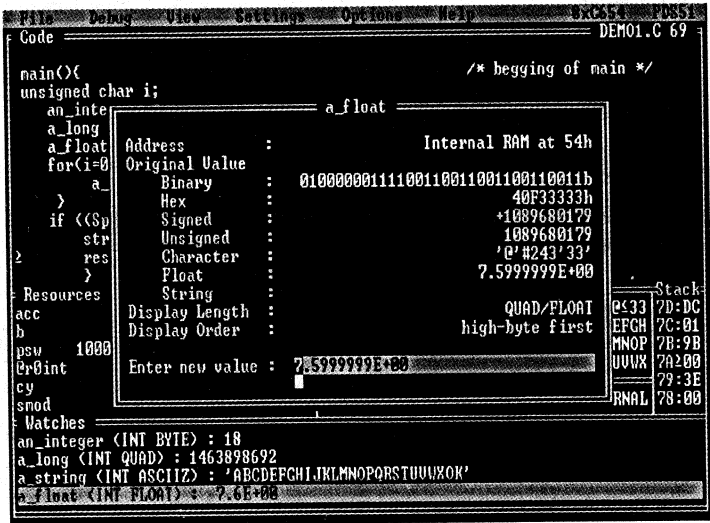


The code pane can be selected to display either a disassembly of the currently loaded program, or the source code itself. These modes are termed assembly-level and source-level respectively, and are readily switched between. A cursor is used to direct the debug operations with execution proceeding either at full speed or single stepped. Displayed variables are updated to accurately reflect the status of the emulator.

When programming using a compiler, your source code (C, Pascal etc.) will appear in the code pane in source-level mode, and debugging can be done

directly in terms of the high-level language statements. By switching to assembly-level mode, you will see the assembly-language code which the compiler has generated. When programming using an assembler, your source code (including all comments etc.) will appear in the code pane in source-level mode and by switching to assembly-level mode, you will see only the assembly-language code.

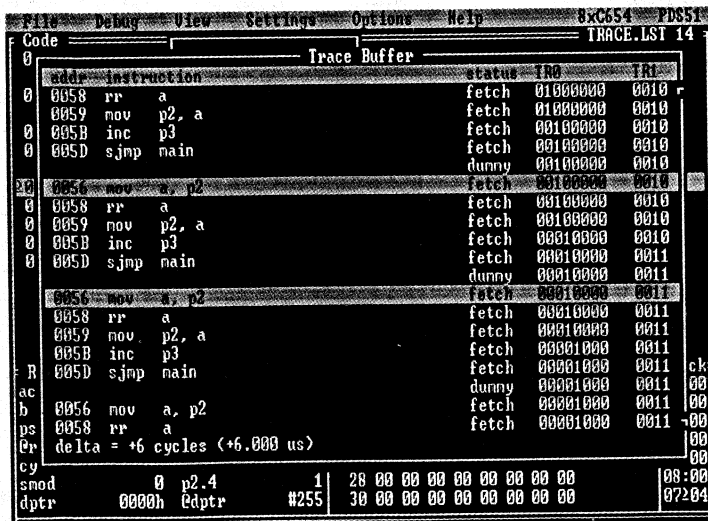
Breakpoints, triggers, trace inhibits and external breakpoints may be hot-key set and cleared at the cursor and are visible to the user as background shading of the corresponding lines of code. These settings may be specified in partitions and / or selected from a symbolic picklist.



The resource pane provides access to the special function register resources of the emulated derivative, while the watch pane provides access to user defined variables contained in the internal/external RAM. The layout, content and display format of these panes are user definable and may be edited at any time. Valid entry and display formats are hexadecimal, ascii, binary, signed and unsigned decimal, as well as IEEE floating point, Pascal string and 'C' string. Data may also be entered directly into internal and external memory and stack areas.

The huge 28K x 32 bit trace buffer is loaded in real time to provide subsequent viewing of past events. Executed code is presented in symbolic format on a cycle by cycle basis

along with 12 user configurable sampled inputs. Trace buffer viewing is aided by a cursor and a moveable reference bar between which code execution times are calculated.



An Execution Profiler is provided for identifying hot-spots or dead code for subsequent code optimization. This is in the form of a histogram of program fetches vs address bins. The address bin size and scale are adjustable.

Support for enhanced video display modes is provided for VGA and EGA displays to allow for more information on screen.

SPECIFICATIONS

- **Emulation:**
 - 64K bytes of code space
 - In-circuit emulation via bondout
 - No stolen resources
 - Real time
 - Universal motherboard
 - Bondout dependant daughterboard (each emulating several derivatives)
- **Trace Memory:**
 - 28K bytes deep, 32 bits wide
 - 16 bit program address
 - 12 bits of port lines or user signals
 - External probes sampled S5P1
 - Control info (C1, INTA)
 - Trace input filtering on the fetch address
- **Breakpoints:**
 - Hardware implemented
 - Up to 64K on program fetch addresses
 - Up to 64K on external signal input AND specified program fetch address
- **User Interfaces:**
 - 1) Windowed full screen debug environment
 - Requires:
 - PC/XT/AT/386/486 or compatible
 - MSDOS ver 3.10 or later
 - 512K bytes of available RAM
 - Monochrome/EGA/VGA
 - 2) Dumb Terminal Emulation using SDS command line instructions
- **Interface Signals:**
 - Force Trace I/P pin
 - Overrides trace input filter
 - External Breakpoint I/P pin
 - Breaks on an external event.
 - Fetch address sensitive over 64K range (sampled S2P1)
 - Emulation Active O/P pin
 - Signals when program running
 - External Trigger O/P pin
 - Trigger O/P for synchronising external test equipment.
 - Fetch address controlled over 64K range (updated S2P1)
- **Speed:**
 - 3.5 to 12 MHz
 - no wait states
 - no stolen cycles
- **Serial Interface:**
 - RS232C
 - Asynchronous, 8 data, no parity, 1 stop bit
 - 9 Pin 'D' connector
 - Rxd/Txd jumper swappable
 - 9.6K/19.2K/38.4K/115.2K Baud (jumper selectable)
 - Xon/Xoff handshaking
- **Size:**
 - Motherboard with daughterboard
 - 130mm x 110mm x 42mm (L.W.H)
- **Power Supply:**
 - 9V nominal (6.5-9.5V)
 - 800mA max, 500mA typical
 - 3.5mm Phono plug centre +ve

- **Integrated Development Environment:**
 - File Formats Supported with Symbolic or Source-Level Debugging:
 - BSO/Tasking, Franklin/Keil, IAR/Archemedes, Metalink, Intel Hex, OMF51, and more.
 - Windowed environment with 6 non-overlapping resizable panes:
 - HLL source or symbolic disassembly code
 - SFR resources
 - Internal/external RAM
 - Stack area
 - User watch variables
 - Pull-down menus
 - Enhanced video display modes available (28/50 line VGA, 43 line EGA)
 - Keyboard control, intuitive keystrokes and Borland compatible hotkeys
 - Fast hotkey access to user Editor/Assembler/Compiler applications
 - Data entry and display formats (examine/modify):
 - Hexadecimal, Ascii, Binary, Signed and unsigned decimal
 - IEEE floating point, Pascal string or 'C' string
 - Emulation Controls:
 - Reset bondout
 - Run from current program counter
 - Run from current program counter to cursor
 - Modify program counter
 - Single step (source-level line or assembly instruction)
 - Step over calls
 - Step into next call or procedure
 - Reposition cursor at address (or label selected from picklist)
 - Follow program without executing
 - Trace Display:
 - Window into 28K lines of trace buffer
 - Fetch addresses
 - Symbolic code format
 - Status (fetch, dummy, inta etc.)
 - 12 external probes (boolean format)
 - Adjustable references for execution time calculation
 - 64K of trace ON/OFF settings (trace filter)
 - Execution Profiler:
 - Bar graph: No. of Fetches vs Address bins
 - Adjustable bin sizes and scaling
 - Settings:
 - Breakpoints, external breakpoints, triggers, trace filters
 - all setable via specified address range, symbol picklist, or keybinding over 64K code space.
 - On-line Help:
 - Context sensitive
 - Indexed, Hypertext style
 - 8051 architecture/instruction set

PDS51 DAUGHTERBOARD SELECTION GUIDE			
DEVICE	DAUGHTERBOARDS ABLE TO EMULATE	PACKAGE EMULATION	
		SUPPLIED	OPTIONAL
8xC053	PDB054	SDIP42	-
8xC054	PDB054	SDIP42	-
8xC055	PDB054	SDIP42	-
8xC51	PDB51FB, PDB52, PDB654, PDB552	DIP40	PLCC44
8xC51FA	PDB51FB	DIP40	PLCC44
8xC51FB	PDB51FB	DIP40	PLCC44
8xC51FC	PDB51FC*	DIP40	PLCC44
8xC52	PDB52, PDB51FB	DIP40	PLCC44
8xC54	*	DIP40	PLCC44
8xC58		DIP40	PLCC44
8xC504	PDB504*	DIP40	PLCC44
8xC524	PDB528*	DIP40	PLCC44
8xC528	PDB528*	DIP40	PLCC44
8xC550	PDB550*	DIP40	PLCC44
8xC552	PDB552	PLCC68	-
8xC562	PDB552	PLCC68	-
8xC592	PDB592*	PLCC68	-
8xC652	PDB654, PDB552	DIP40	PLCC44
8xC654	PDB654, PDB552	DIP40	PLCC44
8xC748	PDB752	SDIP24	PLCC28
8xC749	PDB752	DIP28	PLCC28
8xC750	PDB752	SDIP24	PLCC28
8xC751	PDB752	SDIP24	PLCC28
8xC752	PDB752	DIP28	PLCC28

* DAUGHTER BOARD IN DEVELOPMENT, PLEASE ENQUIRE

Ordering Information:

Please quote the following codes when ordering:

PDS51	Motherboard	9351 719 40112
PDB752	Daughterboard	9351 720 00112
PDB654	Daughterboard	9351 719 80112
PDB552	Daughterboard	9351 719 70112
PDB52	Daughterboard	9351 719 60112
PDB51FB	Daughterboard	9351 719 50112

(not all codes available at time of printing, please enquire)

Description

The P8051LCP40 SD and P8051LCPX SD are low cost, board level products which together program most of Philips 87C51 microcontroller derivatives.

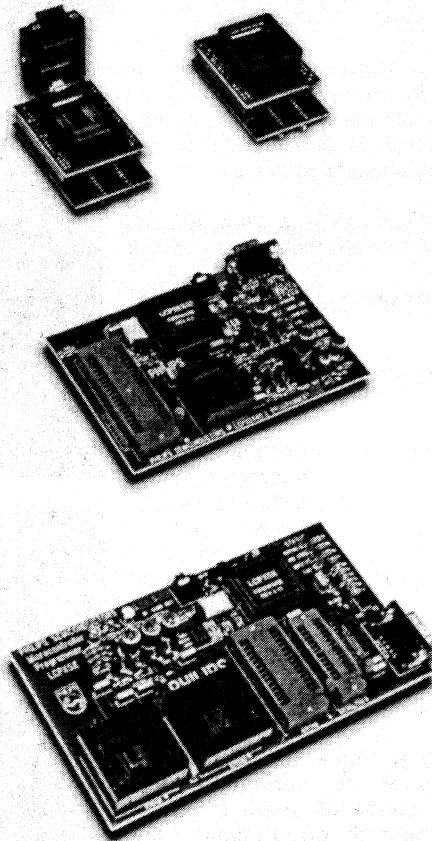
The programmers connect to the serial port of a PC or compatible and are controlled by the interface software supplied.

The programmers will read, blank check, program and verify code bytes, security bits and the encryption array. Both EPROM and EEPROM programming algorithms are supported.

The P8051LCP40 SD has ZIF sockets which accommodate 40-Pin DIP and 44-Pin PLCC 87C51 derivatives while the P8051LCPX SD has ZIF sockets which accommodate 24- and 28-Pin DIP, and 68-Pin PLCC 87C51 derivatives. Adapters are available which extend the range of devices and package types able to be programmed by each programmer.

Up to 8 programmers may be serially connected together ("ganged") for the simultaneous programming of multiple devices.

An external 9V DC power source is required (200mA per board).



Features

- Program code space
- Read code space
- Verify
- Blank Check
- Program security fuses
- Program encryption array
- PC software user interface
- Diskfile I/O (Intel Hex)
- Cascadable for "gang" programming
- Zero insertion force sockets
- Adapters extend the range of programmable devices
- Upgradable

The P8051LCPX and P8051LCP40 programmers are supplied with a full screen PC interface.

The intuitive display shows an outline of the programmer hardware indicating what sockets and devices to use and accurately reflecting the programmers current status.

The function keys allow fast and easy control of the basic operations:

- Blank Check Check ROM locations are all FFh
- Program Writes code from buffer into device
- Verify Ensures a match between buffer code and micro controller code
- Security Program and verify security bits

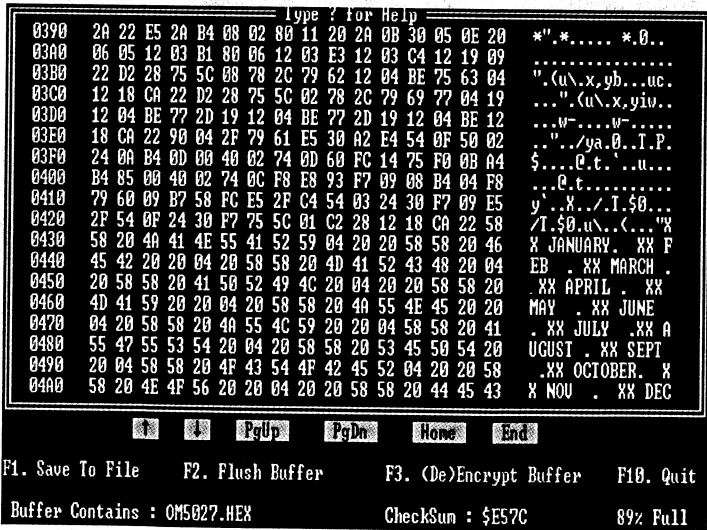
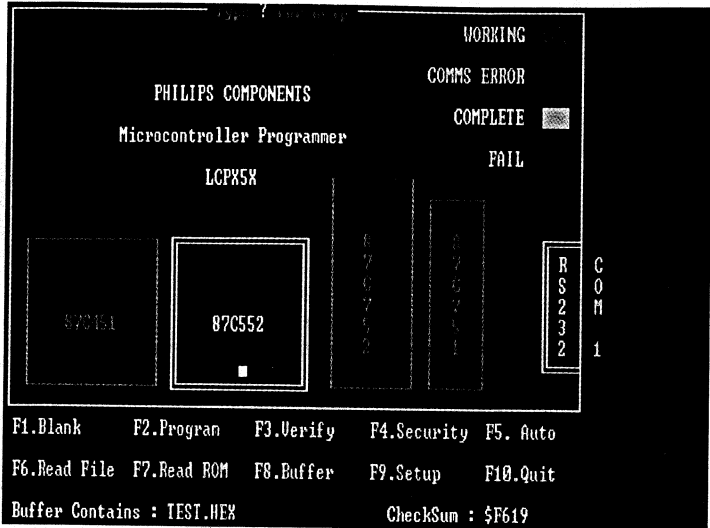
The Auto function performs all of the above operations sequentially to minimise operator intervention.

Only a single PC interface is required to coordinate the operations of up to 8 'gang-connected' programmers for programming multiple devices simultaneously.

The Read ROM function allows the master ROM contents to be read from the target microcontroller into the PC code buffer which may then be inspected (hex format), saved to disk (Intel hex file), or reprogrammed into other devices.

Hex files may also be read from disk to the buffer for subsequent programming, viewing or encryption.

The programmer interface is fully equipped for programming and



verifying encryption arrays and encrypted code. Data read or verified is automatically de-encrypted.

The new operator is assisted by the on-line context-sensitive help facility.

The interface is continuously upgraded in support of new devices as they become available. Upgrade kits are available from your nearest Philips dealer.

The software requires a host IBM PC/XT/AT/386/486 personal computer with an RS-232 serial port.

Programmers for 87C51 and Derivatives

LCP Family of Programmers

Each Programmer Kit Includes:

(both kits include the necessary hardware and software to program Philips microcontrollers using an IBM compatible PC/XT/AT/386/486)

- Programmer board with zero insertion force I.C sockets, 9 pin serial D connector (RS232), 9V power connector
- Control software on 3½" floppy disk
- User manual
- Phono power plug and cable for connection to external power supply

(plus in U.S.A only: RS-232 cable, 9-25 pin D connector adaptor, 110V AC to 9V DC adaptor)

Adaptors:

A range of adaptors is available which extends the list of devices and package types able to be programmed. Most adaptors are two-piece adaptors which consist of a Package Adaptor fitted to a specific Device Adaptor, which in turn is inserted into the 40 pin DIL ZIF socket on the P8051LCP40 SD. They are designed so one package adaptor may be shared between several device adaptors, thus minimising costs. See the selection table opposite.

Ordering Information:

Please quote the following codes when ordering from your nearest Philips distributor.

P8051LCP40 SD	9350 282 90112
P8051LCPX SD	9350 278 80112
LCP40-QFP44	9350 451 30112
LCP40-P80QFP	9350 750 40112
LCP40-P68LCC	9350 750 50112
LCP40-D552	9350 750 60112
LCP40-D558	9350 750 70112
LCP40-D592	9350 750 80112
LCP40-D598	9350 750 90112

Note: If the order code for the desired part is not listed, please enquire at your nearest Philips distributor.

LCP PROGRAMMER SELECTION GUIDE			
DEVICE	PACKAGE	PROGRAMMER	ADAPTOR
87C51	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C51FA	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C51FB	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
87L51FB	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
87C52	QFP44	P8051LCP40 SD	LCP40-QFP44
	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
87C52	QFP44	P8051LCP40 SD	LCP40-QFP44
	DIL40	P8051LCP40 SD	-
87C054	SDIL42	P8051LCPX SD	LCPX-054SDIL
87C055	SDIL42	P8051LCPX SD	LCPX-054SDIL
87C451	PLCC68	P8051LCPX SD	-
87C524	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C528	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C550	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
87C552	PLCC68	P8051LCPX SD	-
	QFP80	P8051LCP40 SD	LCP40-D552 & LCP40-P68LCC
	QFP80	P8051LCP40 SD	LCP40-D552 & LCP40-P80QFP
87C575	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C592	PLCC68	P8051LCP40 SD	LCP40-D592 & LCP40-P68LCC
87C598	QFP80	P8051LCP40 SD	LCP40-D598 & LCP40-P80QFP
	DIL40	P8051LCP40 SD	-
87C652	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C654	DIL40	P8051LCP40 SD	-
	PLCC44	P8051LCP40 SD	-
	QFP44	P8051LCP40 SD	LCP40-QFP44
87C748	SDIL24	P8051LCPX SD	-
	PLCC28	P8051LCPX SD	LCPX-751PLCC
87C749	DIL28	P8051LCPX SD	-
	PLCC28	P8051LCPX SD	LCPX-752PLCC
87C750	SDIL24	P8051LCPX SD	-
	PLCC28	P8051LCPX SD	LCPX-751PLCC
87C751	SDIL24	P8051LCPX SD	-
	PLCC28	P8051LCPX SD	LCPX-751PLCC
87C752	DIL28	P8051LCPX SD	-
	PLCC28	P8051LCPX SD	LCPX-752PLCC
89CE558	QFP80	P8051LCP40 SD	LCP40-D558 & LCP40-P80QFP
89CE559	QFP80	P8051LCP40 SD	LCP40-D558 & LCP40-P80QFP

*Support for the 87C504, 87C51FC, 87C54, 87C58, 87C576, 87C134 and 24/28 pin SSOP devices is intended but not available at the time of printing.

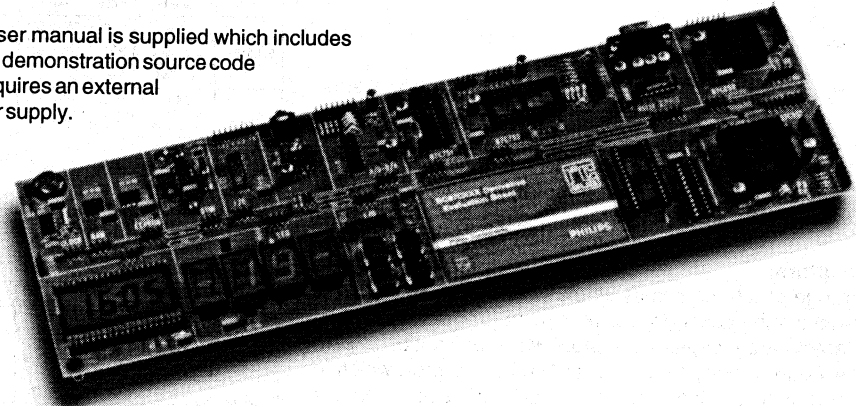
Description

The S87C00KSD Evaluation Board is a low cost, board-level product designed for evaluation of the I²C bus and Philips 80C51 microcontrollers. Its modular board layout and modular software make it useful for fast prototyping and as an educational tool.

This board is supplied with an 87C751 microcontroller pre-programmed with software which demonstrates I²C implementation, and application of a number of I²C peripherals.

Sockets for the 87C752, 87C654/528, and the 87C552 are included on the board to facilitate customer development of these microcontrollers.

A comprehensive user manual is supplied which includes circuit diagrams and demonstration source code listing. The board requires an external 9V 200mA DC power supply.



Board Modules:

- Microcontroller (87C751)
- LCD Display (PCF8577 / LTD226F-12)
- LED Display (SAA1064)
- 8-bit Parallel I/O (8574A)
- 8-bit Parallel I/O (8574)
- RAM (PCF8570)
- EEPROM (PCF8582)
- Clock/Calendar (PCF8583)
- DTMF Dialer (PCF3312 / TDA7050T)
- A/D and D/A (PCF8591)
- Socket for 87C752
- Socket for 87C654/528
- Socket for 87C552 with external RAM and Latch
- RS232 Interface
- Prototyping Area

Ordering Information:

Please quote the following code when ordering from any Philips distributor.

S87C00KSD

9350 107 50112

CAN evaluation board with the 8XC552 and 82C200 CAN controller

OM4130

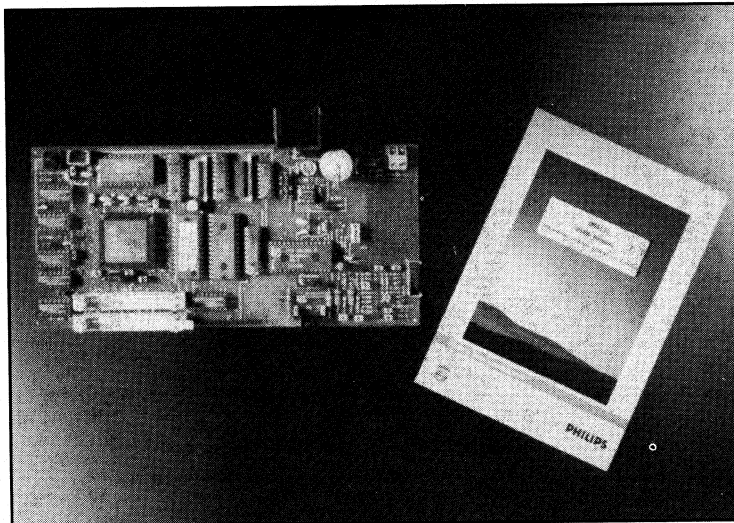
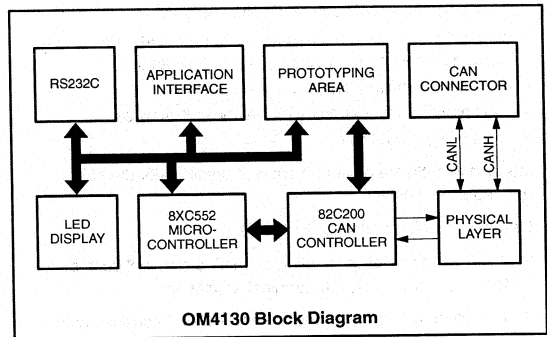
For evaluation of the 82C200 CAN-bus controller hooked to a 8XC552 microcontroller, Philips Semiconductors offers the OM4130 board. It is a low-cost prototyping board for interactive development of programs for full access to the CAN bus.

Major features of the OM4130 include:

- Evaluation of the 82C200 CAN controller
- Full support of the CAN communication utilities:
 - demonstration and monitor software for the CAN-bus system supports CAN communication of up to 1 Mbits/sec
 - interactive configuration of the CAN controller
 - CAN physical layer circuitry for balanced bus wires
 - LED display
- Interactive software development with up/down-loading of user-specific hex files
- On-board monitor software (EPROM)
- RS232C interface for connection to a PC, a terminal such as a VT100 or a workstation with terminal emulation
- Wire-wrap area for prototyping, an application interface and a two-wire CAN-bus connector

- Supply voltage: 7.5 to 18 V
- Dimensions: 100 × 200 mm.

The OM4130 comes with serial interface and supply voltage connectors.



OM4130 — CAN Evaluation Board with 8XC552 Microcontroller and 82C200 CAN Controller

Ordering Code: OM4130

CAN evaluation board with the 8XC592 microcontroller

OM4239

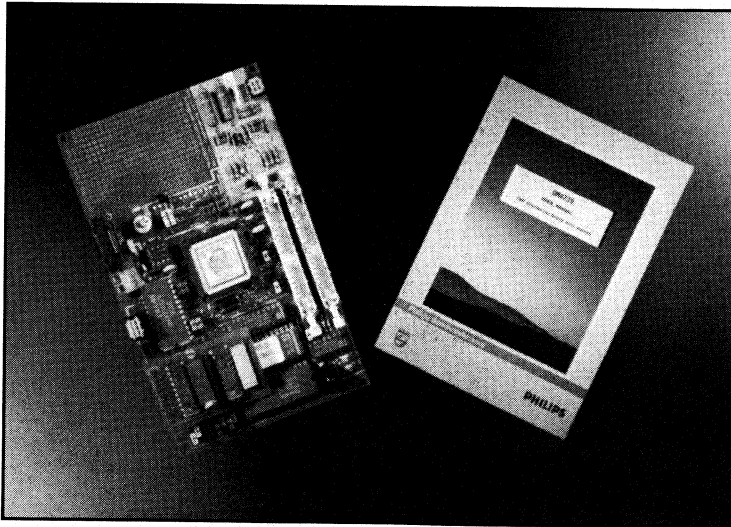
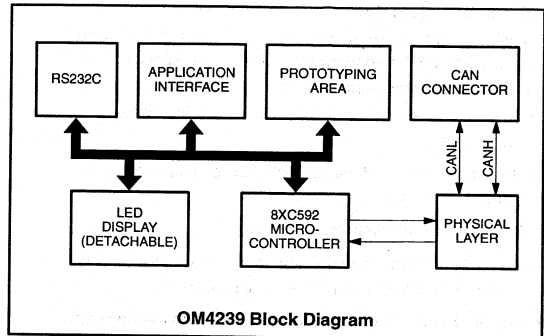
For evaluation of the 8XC592 microcontroller's CAN-bus interface, Philips Semiconductors offers the OM4239 board. It is a low-cost prototyping board providing powerful monitor functions for interactive development of programs for full access to the CAN bus.

Major features of the OM4239 include:

- Evaluation of the 8XC592 microcontroller's CAN interface
- Full support of the CAN communication utilities:
 - demonstration and monitor software for the CAN-bus system supports CAN communication of up to 1 Mbits/sec
 - interactive configuration of the CAN controller
 - CAN physical layer circuitry for balanced bus wires
 - detachable LED display
- Interactive software development with up/down-loading of user-specific hex files
- On-board monitor software (EPROM)
- RS232C interface for connection to a PC, a terminal such as a VT100 or a workstation with terminal emulation
- Wire-wrap area for prototyping, an application interface and a two-wire CAN-bus connector

- Supply voltage: 7.5 to 18 V
- Dimensions: 100 × 160 mm.

The OM4239 comes with serial interface and supply voltage connectors.



OM4239 — CAN Evaluation Board with the 8XC592 Microcontroller

Ordering Code: OM4239

Evaluation board for the 8XCE598 microcontroller

OM4240

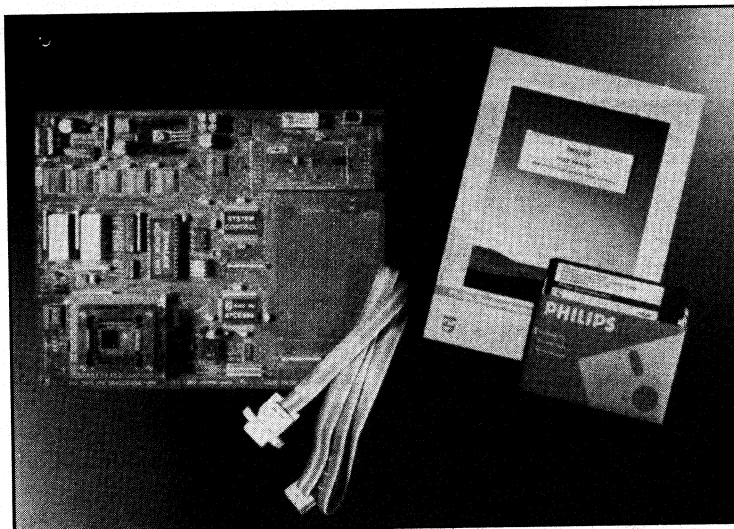
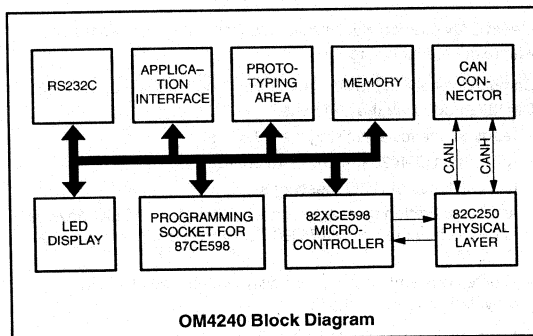
For evaluation of the 8XCE598 microcontroller and its CAN-bus interface, Philips Semiconductors offers the OM4240 board. It is a low-cost prototyping board providing powerful monitor functions for interactive development of programs for full access to the CAN bus.

Major features of the OM4240 include:

- Evaluation of the 8XCE598 microcontroller and its CAN interface
- Full support of the CAN communication utilities:
 - demonstration and monitor software for the CAN-bus system supports CAN communication of up to 1 Mbits/sec
 - interactive configuration of the CAN controller
 - a plug-in CAN physical layer baby board with the 82C250 CAN transceiver IC for balanced bus wires
 - LED display
- Interactive software development with up/down-loading of user-specific hex files, single/multi-step breakpoint setting and software trace functions
- On-board monitor software (EPROM) and a PC-DOS compatible cross-assembler on a diskette
- On-board programming of a 87CE598
- RS232C interface for connection to a PC

- Wire-wrap area for prototyping, an application interface and a two-wire CAN-bus connector
- Supply voltage: 7.5 to 18 V
- Dimensions: 200 × 160 mm.

The OM4240 comes with serial interface cable and a supply voltage connector.



OM4240 — Evaluation Board for the 8XCE598 Microcontroller

Ordering Code: OM4240

SLIO evaluation board with the 82C150 and 82C250 CAN ICs

OM4272

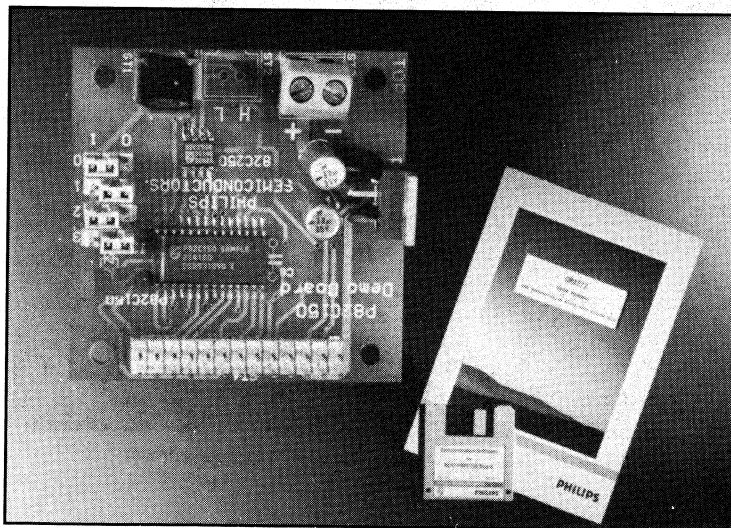
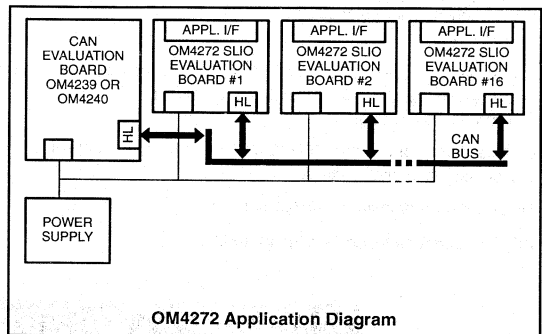
For a cost-effective implementation of CAN nodes, Philips Semiconductors offers the OM4272 SLIO (Serial Linked I/O) board. The board includes a P82C150 single-chip I/O device with a CAN protocol controller and a PCA82C250 CAN transceiver which interfaces to a physical bus complying with the ISO/DIS 11898 standard.

Major features of the OM4272 include:

- Low-cost interface between I/O and CAN bus
- Meets CAN protocol specification version 2.0A and B (passive) with restricted bit timing
- Sample software is provided for control/configuration of up to 16 OM4272 nodes via the CAN bus:
 - ready to use for exercising the P82C150
 - runs on an OM4239 or OM4240 board
 - uses all types of SLIO messages
 - sends calibration messages, checks presence of SLIO devices and switches the on-board LED connected to a port pin of the P82C150
 - source code written in C and down-loadable hex files are provided on a 3.5" diskette
- Full automatic bit rate detection and calibration
- 10-bit analog-to-digital conversion with up to six multiplexed analog input channels
- Up to 2 quasi-analog outputs with 10-bit accuracy

- On-board facilities for low-pass RC filters for analog-to-digital converter and quasi-analog outputs
- The application interface connector is hooked to all I/O ports, reset, test, and power pins
- Optional external clock mode via the application connector
- Supply voltage: 7.5 to 18 V
- Dimensions: 50 × 52 mm.

The OM4240 comes with serial interface cable and a supply voltage connector.



OM4272 — SLIO Evaluation Board with the 82C150 and 82C250 CAN ICs

Ordering Code: OM4272

P83C852 Smart Card crypto-controller demonstration kit

OM4280

For the fast growing market of smart cards, Philips Semiconductors offers the OM4280 demonstration kit for evaluating its P83C852 Crypto Controller. This demo kit enables the user to evaluate the smart card operating system implemented on the P83C852. The operating system provides several functions including PIN verification, various memory functions and debiting. It also supports more complex functions required for, e.g., point-of-sale applications and encryption/decryption functions for confidential data transfer including a personal signature in, e.g., open FAX systems.

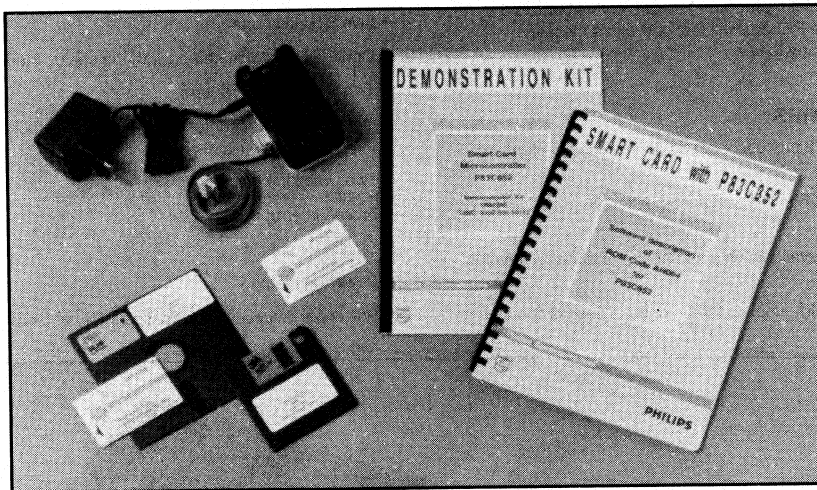
The wide spread use of smart cards requires thorough information security. The OM4280 demo kit particularly addresses this issue. The P83C852 Crypto Controller has an on-chip accelerator for fast asymmetric enciphering using the RSA algorithm.

A sample program on the OM4280 supports secret and public keys

for two persons named Alice and Bob. The user can assign, however, any other pair of secret/private keys on the card.

The demonstration kit is supplied with:

- An operating manual
- A full description of the card operating system
- Two Smart Cards each with a P83C852 CMOS crypto controller
- A card reader
- A PC interface cable
- A mains adaptor
- Diskettes with the Smart Card demo software and source code for an RS-232 interface to a PC's serial port



OM4280, P83C852 Smart Card Crypto Controller Demonstration Kit

Ordering Code: OM4280

Evaluation board

PEB552

The PEB552 is a low-cost, easily customized single extended euro-card for the Philips high performance PCB80C552 microcontroller. The PEB552 can be programmed in high-level PLM51, Forth, Basic, C languages or conventional assemblers.

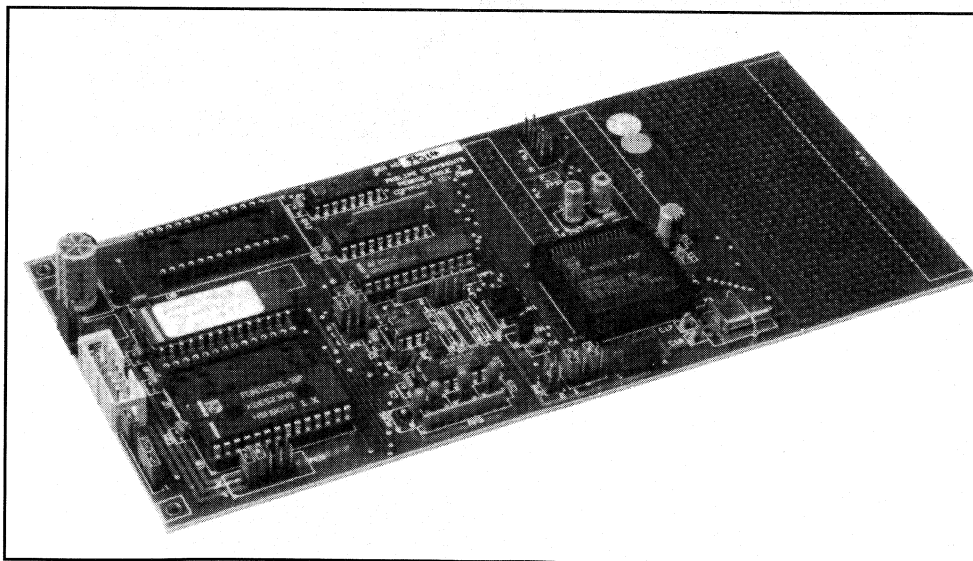
A major design feature of the PEB552 is its large, 11 × 38, wire-wrap prototyping area in a 0.1-inch grid. All the microprocessor signals, I/O decoding and an uncommitted 96-pin connector are conveniently routed to the edge of the prototyping area. The fully expanded memory offers 124 kbytes of RAM and EPROM and 4 kbytes of reconfigurable decoded I/O space.

The PEB552 has been designed for on-board EPROM programming. These programs can be executed at power-up or reset. The high programming voltages required for the EPROMs are provided by a piggy-back module which readily plugs into the PEB552.

To operate the PEB552, simply connect it to your PC and run a suitable terminal emulator program. The turn on the board's 5 V power supply and the PEB552 will respond with a message indicating that it is ready to use.

PEB552 FEATURES

- CMOS PCB80C552 processor based on the industry standard 80C51
- Wire-wrap prototyping area with uncommitted 96-pin DIN41612 connector
- All processor pins available on pin-headers near the prototyping area
- Full duplex RS232 interface to 25-pin or 9-pin D-shell connectors
- Two-wire I²C serial communication bus
- 32 kbytes of RAM expandable to 64 kbytes
- Optional battery back-up
- 256 bytes of RAM on the PCB80C552
- 32 kbytes of EPROM including a powerful monitor program. Expandable to 64 kbytes.
- Optional on-board EPROM programming capability for up to 32 kbytes of EPROM
- Optional on-board 87C552 microcontroller programming
- Reconfigurable memory-mapped I/O decoding
- Eight analog inputs multiplexed to a 10-bit ADC
- Two PWM outputs can be used as 8-bit DACs
- Five 8-bit I/O ports
- One 8-bit input port shared with the analog inputs
- Two 16-bit timer/counters
- One 16-bit timer counter coupled to four capture registers and three compare registers
- Two watchdog timers
- Fifteen interrupt vectors
- Single 5 V supply
- Power-on reset
- Power-fail detect and automatic battery back-up switch over.



PEB552 Evaluation Board

Evaluation board

PEB552

HARDWARE SPECIFICATIONS

Processor and Memory

The heart of the PEB552 is the Philips PCB80C552 microcontroller operating at 11.059 MHz. The board is supplied with 32 kbytes of static RAM and 32 kbytes of EPROM. The memory can be further expanded to 64 kbytes of RAM and 64 kbytes of EPROM. Two RAM sockets enable the use of either 8 or 32 kbyte devices and two EPROM sockets enable the use of 8, 16 or 32 kbyte devices. The second of the two EPROM sockets is available for EPROM programming. The memory and I/O maps are completely configurable by either jumper selections or, for customized requirements, via an industry standard programmable logic device.

Serial I/O

A full-duplex RS232 interface is implemented on the PEB552. The RS232 connector on the board enables a flat cable to be made up for the 25-pin or 9-pin D-shell connector conventions required by the IBM PC/XT and /AT computers and compatibles. Jumpers are provided to select the conventions for the transmit, receive and ground signals. The 11.059 MHz crystal frequency on the PEB552 can be used in conjunction with an on-chip timer to generate baud rates up to 19200 baud.

An I²C-bus controller for two-wire serial communication with I²C-bus compatible ICs is also provided. The I²C serial I/O has complete autonomy in byte handling and operates in 4 modes:

- Master transmitter
- Master receiver
- Slave transmitter
- Slave receiver

Parallel I/O

All of the processor signals are routed to pin headers at the edge of the prototyping area. The PCB80C552 processor offers five 8-bit I/O ports and one 8-bit input port. When using the processor in expanded mode, a dedicated 4 kbyte memory-mapped I/O space is enabled. The memory-mapped I/O is further decoded to produce 8 I/O select signals available on a pin header close to the prototyping area.

Watchdog Timers

Watchdog timers are an important feature of embedded microcontroller applications. The purpose of a watchdog timer is to reset the microcontroller if it enters an erroneous state within a reasonable period of time. The PEB552 has two watchdog timers: one is external to the PCB80C552 and has a time-out period of 1.4 s, and the other is part of the processor and has a programmable time-out period between 2 ms and 500 ms. Both watchdog timers are enabled with jumpers on the PEB552. If either watchdog times out, a reset pulse is generated.

External Connections

An uncommitted DIN41612 connector is available at the edge of the card and positioned in front of the prototyping area. Its 96 connections are routed to holes on the prototyping matrix.

All of the PCB80C552 processor pins with the exception of the crystal pins are available on the pin headers.

EPROM and 87C662 Processor Programming

For easy on-board EPROM programming, sufficient board area is available on the PEB552 for a 28-pin ZIF socket to hold an 8, 16, or 32 kbyte EPROM. Source code developed using high-level languages or assemblers can be burnt into the EPROM by using the optional PEB552VA voltage converter adapter and a pre-programmed 87C552 microcontroller. The PEB552VA plugs into a connector on the PEB552 and converts the 5 V supply voltage to 5, 6, and 12.5 V required for EPROM programming. The adapter is

controlled by a signal from the controller. A 87C552 can be programmed using an optional 87C552 programming adapter.

Battery Back-up

An external 2 to 4 V battery can be used to retain data in RAM. A precision voltage level detector automatically switches the 5 V supply over to the battery supply when the 5 V supply drops 50 mV below the battery level. Battery supply is switched to the main supply when the main voltage rises 50 mV above the battery level.

The precision voltage level detector also provides a power-fail signal that is available as an interrupt signal to the processor.

SOFTWARE SUPPORT

The PEB552 is supplied with a monitor program — MON552 — consisting of a diskette and a 27C256 EPROM. The diskette is IBM PC-compatible and contains the host module for the monitor program. The EPROM code executes on the PEB552 and uses approximately 2 kbytes of the supplied 32 kbytes EPROM. Existing systems can be upgraded with a pre-programmed 87C552 to replace the EPROM monitor and to provide both EPROM and 87C552 programming capability with the aid of the appropriate adapters.

The monitor module provides the minimum functionality required to:

- Read and write to internal and external RAM
- Read from program memory
- Read and write to the special function registers
- Load and save Intel hex files
- Program execution control such as jump to a start address
- Disassemble program memory
- Single step through a program in RAM or EPROM
- Set multiple break points

The purpose of the host module is to provide a user interface for the monitor module. The host module communicates with the PEB552 via the serial RS232 interface. Commands and parameters are transmitted to the PEB552. The monitor module then interprets these commands and transmits the required data back to the host. The host software provides a window environment, a lexical analyzer and command interpreter. Higher level languages including PLM51, Forth, Basic, C and assemblers are readily available from various sources to develop executable code for the PEB552.

USER SUPPORT

Users must register the serial number of their PEB552 with Micro Amps Ltd., the designer and manufacturer of the board, to receive support. This registration entitles the user to receive telephone support, information regarding new software releases, details regarding associated software tools, operating systems and customization facilities.

ORDERING INFORMATION

The ordering code for the complete PEB552 package: 9339 953 0112

This package includes:

- PEB552 evaluation board
- PEB552 hardware manual
- MON552 monitor manual
- PCB80C552 user manual
- REG552 registration form.

Optionally available:

- PEB552VA voltage adapter and 87C552 microcontroller for on-board EPROM programming
- Adapter for on-board 87C552 microcontroller programming

DATA HANDBOOK SYSTEM

Philips Semiconductors data handbooks contain all pertinent data available at the time of publication and each is revised and reissued regularly.

Loose data sheets are sent to subscribers to keep them up-to-date on additions or alterations made during the lifetime of a data handbook.

Catalogs are available for selected product ranges (some catalogs are also on floppy discs).

Our data handbook titles are listed here.

Integrated Circuits

<i>Book</i>	<i>Title</i>
IC01	Semiconductors for Radio and Audio Systems
IC02	Semiconductors for Television and Video Systems
IC03	Semiconductors for Telecom Systems
IC04	CMOS HE4000B Logic Family
IC06	High-speed CMOS Logic Family
IC11	General-purpose/Linear ICs
IC12	I ² C Peripherals
IC13	Programmable Logic Devices (PLD)
IC14	8048-based 8-bit Microcontrollers
IC15	FAST TTL Logic Series
IC16	CMOS Integrated Circuits for Clocks and Watches
IC17	RF/Wireless Communications
IC18	Semiconductors for In-car Electronics
IC19	ICs for Datacommunications
IC20	80C51-based 8-bit Microcontrollers
IC22	Desktop Video
IC23	QUBiC Advanced BiCMOS Bus Interface Logic ABT, MULTIBYTE™
IC24	Low Voltage CMOS & BiCMOS Logic

Discrete Semiconductors

<i>Book</i>	<i>Title</i>
SC01	Diodes
SC02	Power Diodes
SC03	Thyristors and Triacs
SC04	Small-signal Transistors
SC05	Low-frequency Power Transistors and Hybrid IC Power Modules
SC06	High-voltage and Switching NPN Power Transistors
SC07	Small-signal Field-effect Transistors
SC08a	RF Power Bipolar Transistors
SC08b	RF Power MOS Transistors
SC09	RF Power Modules
SC10	Surface Mounted Semiconductors
SC13	Power MOS Transistors including TOPFETs and IGBTs
SC14	RF Wideband Transistors, Video Transistors and Modules
SC15	Microwave Transistors
SC16	Wideband Hybrid IC Modules
SC17	Semiconductor Sensors

Professional Components

PC01	High-power Klystrons and Accessories
PC06	Circulators and Isolators

MORE INFORMATION FROM PHILIPS SEMICONDUCTORS?

For more information about Philips Semiconductors data handbooks, catalogs and subscriptions, contact your nearest Philips Semiconductors national organization, select from the **address list on the back cover of this handbook**. Product specialists are at your service and inquiries are answered promptly.

OVERVIEW OF PHILIPS COMPONENTS DATA HANDBOOKS

Our sister product division, Philips Components, also has a comprehensive data handbook system to support their products. Their data handbook titles are listed here.

Display Components

Book	Title
DC01	Colour TV Picture Tubes and Assemblies Colour Monitor Tubes
DC02	Monochrome Monitor Tubes and Deflection Units
DC03	Television Tuners, Coaxial Aerial Input Assemblies
DC05	Flyback Transformers, Mains Transformers and General-purpose FXC Assemblies

Magnetic Products

MA01	Soft Ferrites
MA03	Piezoelectric Ceramics Specialty Ferrites
MA04	Dry-reed Switches

Passive Components

PA01	Electrolytic Capacitors
PA02	Varistors, Thermistors and Sensors
PA03	Potentiometers
PA04	Variable Capacitors
PA05	Film Capacitors
PA06	Ceramic Capacitors
PA07	Quartz Crystals for Special and Industrial Applications
PA08	Fixed Resistors
PA10	Quartz Crystals for Automotive and Standard Applications
PA11	Quartz Oscillators

Professional Components

PC04	Photo Multipliers
PC05	Plumbicon Camera Tubes and Accessories
PC07	Vidicon and Newvicon Camera Tubes and Deflection Units
PC08	Image Intensifiers
PC12	Electron Multipliers

MORE INFORMATION FROM PHILIPS COMPONENTS?

For more information contact your nearest Philips Components national organization shown in the following list.

Argentina: BUENOS AIRES, Tel. (541)786 7635, Fax. (541)786 9367.
Australia: NORTH RYDE, Tel. (02)805 4455, Fax. (02)805 4466.
Austria: WIEN, Tel. (01)60101 1820, Fax. (01)60101 1210.
Belgium: EINDHOVEN, The Netherlands, Tel. (31)40 783749, Fax. (31)40 788399.
Brazil: SÃO PAULO, Tel. (011)821 2333, Fax. (011)829 1849.
Canada: SCARBOROUGH, Tel. (0416)292 5161, Fax. (0416)754 6248.
Chile: SANTIAGO, Tel. (02)77 38 16, Fax. (02)735 3594.
China (Peoples Republic of): SHANGHAI, Tel. (021)326 4140, Fax. (021)320 2160.
Columbia: BOGOTA, Tel. (571)249 7624/(571)217 4609, Fax. (571)217 4549.
Denmark: COPENHAGEN, Tel. (032)883 333, Fax. (031)571 949.
Finland: ESPOO, Tel. (9)0-52061, Fax. (9)0-520971.
France: SURESNES, Tel. (01)4099 6161, Fax. (01)4099 6431.
Germany: HAMBURG, Tel. (040)3296-0, Fax. (040)3296 213.
Greece: TAVROS, Tel. (01)489 4339/(01)489 4911, Fax. (01)481 5180.
Hong Kong: KWAI CHUNG, Tel. (852)424 5121, Fax. (852)428 6729.
India: BOMBAY, Tel. (022)4938 541, Fax. (022)4938 722.
Indonesia: JAKARTA, Tel. (021)5201122, Fax. (021)5205189.
Ireland: DUBLIN, Tel. (01)640 203, Fax. (01)640 210.
Israel: TEL AVIV, Tel. (9723)6450333, Fax. (9723)493272.
Italy: MILANO, Tel. (02)6752.3302, Fax. (02)6752.3300.
Japan: TOKIO, Tel. (03)3740 5143, Fax. (03)3740 5035.
Korea (Republic of): SEOUL, Tel. (02)709-1412, Fax. (02)709-1415.
Malaysia: KUALA LUMPUR, Tel. (03)757 5511, Fax. (03)757 4880.
Mexico: CHI HUA HUA, Tel. (016)18-67-01/(016)18-67-02, Fax. (016)778 0551.
Netherlands: EINDHOVEN, Tel. (040)783749, Fax. (040)788399.
New Zealand: AUKLAND, Tel. (09)849-4160, Fax. (09)849-7811.
Norway: OSLO, Tel. (22)74 8000, Fax. (22)74 8341.
Pakistan: KARACHI, Tel. (021)587 4641-49, Fax. (021)577035/5874546.
Philippines: MANILA, Tel. (02)810-0161, Fax. (02)817-3474.
Portugal: LINDA-A-VELHA, Tel. (01)14163160/4163333, Fax. (01)14163174/4163366.
Singapore: SINGAPORE, Tel. (65)350 2000, Fax. (65)355 1758.
South Africa: JOHANNESBURG, Tel. (011)470-5911, Fax. (011)470-5494.
Spain: BARCELONA, Tel. (03)301 6312, Fax. (03)301 4243.
Sweden: STOCKHOLM, Tel. (08)632 2000, Fax. (08)632 2745.
Switzerland: ZÜRICH, Tel. (01)488 2211, Fax. (01) 481 7730.
Taiwan: TAIPEI, Tel. (02)388 7666, Fax. (02)382 4382.
Thailand: BANGKOK, Tel. (662)398-0141, Fax. (662)398-3319.
Turkey: ISTANBUL, Tel. (0212)279 2770, Fax. (0212)269 3094.
United Kingdom: LONDON, Tel. (071)590 6633, Fax. (071)636 0394.
United States: RIVIERA BEACH, Tel. (407)881-3200, Fax. (407)881-3300.
Uruguay: MONTEVIDEO, Tel. (02)704 044, Fax. (02)920 601.
For all other countries apply to: Philips Components. Marketing Communications, P.O. Box 218, 5600 MD, EINDHOVEN, The Netherlands Telex 35000 phtnl, Fax. +31-40-724547.

North American Sales Offices, Representatives and Distributors

PHILIPS SEMICONDUCTORS

811 East Arques Avenue
P.O. Box 3409
Sunnyvale, CA 94088-3409

ALABAMA

Huntsville

Philips Semiconductors
Phone: (205) 464-0111
(205) 464-9101

Elcom, Inc.
Phone: (205) 830-4001

ARIZONA

Scottsdale

Thorn Luke Sales, Inc.
Phone: (602) 451-5400

Tempe

Philips Semiconductors
Phone: (602) 820-2225

CALIFORNIA

Calabasas

Philips Semiconductors
Phone: (818) 880-6304

Irvine

Philips Semiconductors
Phone: (714) 453-0770

Loomis

B.A.E. Sales, Inc.
Phone: (916) 652-6777

San Diego

Philips Semiconductors
Phone: (619) 560-0242

San Jose

B.A.E. Sales, Inc.
Phone: (408) 452-8133

Sunnyvale

Philips Semiconductors
Phone: (408) 991-3737

COLORADO

Englewood

Philips Semiconductors
Phone: (303) 792-9011

Thorn Luke Sales, Inc.
Phone: (303) 649-9717

CONNECTICUT

Wallingford

JEBCO
Phone: (203) 265-1318

FLORIDA

Clearwater

Conley and Assoc., Inc.
Phone: (813) 572-8895

Oviedo

Conley and Assoc., Inc.
Phone: (407) 365-3283

GEORGIA

Norcross

Elcom, Inc.
Phone: (404) 447-8200

ILLINOIS

Itasca

Philips Semiconductors
Phone: (708) 250-0050

Schaumburg

Micro-Tex, Inc.
Phone: (708) 885-8200

INDIANA

Indianapolis

Mohrfield Marketing, Inc.
Phone: (317) 546-6969

Kokomo

Philips Semiconductors
Phone: (317) 459-5355

MARYLAND

Columbia

Third Wave Solutions, Inc.
Phone: (410) 290-5990

MASSACHUSETTS

Chelmsford

JEBCO
Phone: (508) 256-5800

Westford

Philips Semiconductors
Phone: (508) 692-6211

MICHIGAN

Monroe

S-J Associates
Phone: (313) 242-0450

Novi

Philips Semiconductors
Phone: (810) 347-1700

MINNESOTA

Bloomington

High Technology Sales
Phone: (612) 844-9933

MISSOURI

Bridgeton

Centech, Inc.
Phone: (314) 291-4230

Raytown

Centech, Inc.
Phone: (816) 358-8100

NEW JERSEY

Toms River

Philips Semiconductors
Phone: (908) 505-1200
(908) 240-1479

NEW YORK

Ithaca

Bob Dean, Inc.
Phone: (607) 257-1111

Rockville Centre

S-J Associates
Phone: (516) 536-4242

Wappingers Falls

Philips Semiconductors
Phone: (914) 297-4074

Bob Dean, Inc.
Phone: (914) 297-6406

NORTH CAROLINA

Charlotte

Elcom, Inc.
Phone: (704) 543-1229

Greensboro

Elcom, Inc.
Phone: (919) 273-8887

Matthews

Elcom, Inc.
Phone: (704) 847-4323

OHIO

Columbus

S-J Associates, Inc.
Phone: (614) 885-6700

Kettering

S-J Associates, Inc.
Phone: (513) 298-7322

Solon

S-J Associates, Inc.
Phone: (216) 349-2700

Toledo

S-J Associates, Inc.
Phone: (419) 727-8051

OREGON

Beaverton

Philips Semiconductors
Phone: (503) 627-0110

Western Technical Sales
Phone: (503) 644-8860

PENNSYLVANIA

Erie

S-J Associates, Inc.
Phone: (216) 888-7004

Hatboro

Delta Technical Sales, Inc.
Phone: (215) 957-0600

Pittsburgh

S-J Associates, Inc.
Phone: (216) 349-2700

SOUTH CAROLINA

Greenville

Elcom, Inc.
Phone: (803) 370-9119

TENNESSEE

Dandridge

Philips Semiconductors
Phone: (615) 397-5053

TEXAS

Austin

Philips Semiconductors
Phone: (512) 339-9945

Synergistic Sales, Inc.
Phone: (512) 346-2122

Houston

Synergistic Sales, Inc.
Phone: (713) 937-1990

Richardson

Philips Semiconductors
Phone: (214) 644-1610
(214) 705-9555

Synergistic Sales, Inc.
Phone: (214) 644-3500

UTAH

Salt Lake City

Electrodyne
Phone: (801) 264-8050

WASHINGTON

Bellevue

Western Technical Sales
Phone: (206) 641-3900

Spokane

Western Technical Sales
Phone: (509) 922-7600

WISCONSIN

Waukesha

Micro-Tex, Inc.
Phone: (414) 542-5352

CANADA

PHILIPS

SEMICONDUCTORS CANADA, LTD.

Calgary, Alberta

Philips Semiconductors/
Components, Inc.
Phone: (403) 293-5969

Tech-Trek, Ltd.
Phone: (403) 241-1719

Kanata, Ontario

Philips Semiconductors
Phone: (613) 599-8720

Tech-Trek, Ltd.
Phone: (613) 599-8787

Montreal, Quebec

Philips Semiconductors/
Components, Inc.
Phone: (514) 424-7320

Mississauga, Ontario

Tech-Trek, Ltd.
Phone: (416) 238-0366

Richmond, B.C.

Tech-Trek, Ltd.
Phone: (604) 276-8735

Scarborough, Ontario

Philips Semiconductors/
Components, Ltd.
(416) 292-5161

Ville St. Laurent, Quebec

Tech-Trek, Ltd.
Phone: (514) 337-7540

MEXICO

Anzures Section

Philips Components
Phone: (800) 234-7381

El Paso, TX

Philips Components
Phone: (915) 775-4020

PUERTO RICO

Caguas

Mectron Group
Phone: (809) 746-3522

DISTRIBUTORS

Contact one of our

local distributors:

Allied Electronics
Anthem Electronics
Arrow/Schweber Electronics
Future Electronics (Canada only)
Gerber Electronics
Hamilton Hallmark
Marshall Industries
Newark Electronics
Richardson Electronics
Wyle Electronics
Zeus Electronics

Philips Semiconductors - a worldwide company

Argentina: IEROD, Av. Juramento 1992 - 14.b (1428) BUENOS AIRES,
Tel. (541) 786 7633, Fax. (541) 786 9367

Australia: 34 Waterloo Road, NORTH RYDE, NSW 2113,
Tel. (02) 805 4455, Fax. (02) 805 4466

Austria: Triester Str. 64, A-1101 WIEN, P.O. Box 213,
Tel. (01) 60 101-1236, Fax. (01) 60 101-1211

Belgium: Postbus 90050, 5600 PB EINDHOVEN, The Netherlands,
Tel. (31)40 783 749, Fax. (31)40 788 399

Brazil: Rua do Rocio 220 - 5th Floor, Suite 51
CEP: 04552-903 SÃO PAULO-SP, Brazil
P.O. Box 7383 (01064-970),
Tel. (011) 821-2333, Fax (011) 829-1849

Canada: PHILIPS SEMICONDUCTORS/COMPONENTS:
Tel. (800) 234-7381, Fax. (708) 296-8556

Chile: Av. Santa Maria 0760, SANTIAGO,
Tel. (02) 773 816, Fax. (02) 777 6730

Colombia: IPRELENSO LTDA, Carrera 21 No. 56-17, 77621 BOGOTA,
Tel. (571)249 7624/(571)217 4609, Fax. (571)217 4549

Denmark: Prags Boulevard 80, PB 1919, DK-2300 COPENHAGEN S,
Tel. (032) 88 2636, Fax. (031) 57 1949

Finland: Sinikalliontie 3, FIN-02630 ESPOO,
Tel. (9)0-50261, Fax. (9)0-520971

France: 4 Rue du Port-aux-Vins, BP317, 92156 SURESNES Cedex,
Tel. (01)4099 6161, Fax. (01)4099 6427

Germany: P.O. Box 10 63 23, 20043 HAMBURG,
Tel. (040) 3296-0, Fax. (040) 3296 213

Greece: No. 15, 25th March Street, GR 17778 TAVROS,
Tel. (01) 4894 339/4894 911, Fax. (01) 4814 240

Hong Kong: PHILIPS HONG KONG Ltd., 6/F Philips Ind. Bldg.,
24-28 Kung Yip St., KWAI CHUNG, N.T.,
Tel. (852)424 5121, Fax. (852)428 6729

India: Philips INDIA Ltd., Shivsagar Estate, A Block,
Dr. Annie Besant Rd., Worli, BOMBAY 400 018,
Tel. (022)4938 541, Fax. (022)4938 722

Indonesia: Philips House, Jalan H.R. Rasuna Said Kav. 3-4,
P.O. Box 4252, JAKARTA 12950
Tel. (021)5201 122, Fax. (021)5205 189

Ireland: Newstead, Clonskeagh, DUBLIN 14,
Tel. (01)640 000, Fax. (01)640 200

Italy: PHILIPS SEMICONDUCTORS S.r.l.,
Piazza IV Novembre 3, 20124 MILANO,
Tel. (0039)2 6752 2531, Fax. (0039)2 6752 2557

Japan: Philips Bldg. 13-37, Kohnan 2-chome, Minato-ku, TOKYO 108,
Tel. (03)3740 5028, Fax. (03)3740 0580

Korea (Republic of): Philips House, 260-199 Itaewon-dong,
Yongsan-ku, SEOUL, Tel. (02)794 5011, Fax. (02)798 8022

Malaysia: No. 76 Jalan Universiti, 46200 PETALING JAYA,
SELANGOR, Tel. (03)750 5214, Fax. (03)757 4880

Mexico: 5900 Gateway East, Suite 200, EL PASO, TX 79905,
Tel. 9-5 (800)234-7381, Fax. (708)296-8556

Netherlands: Postbus 90050, 5600 PB EINDHOVEN, Bldg. VB
Tel. (040)783749, Fax. (040)788399

New Zealand: 2 Wagener Place, C.P.O. Box 1041, AUCKLAND,
Tel. (09)849-4160, Fax. (09)849-7811

Norway: Box 1, Manglerud 0612, OSLO,
Tel. (022)74 8000, Fax (022)74 8341

Pakistan: Philips Electrical Industries of Pakistan Ltd.,
Exchange Bldg. ST-2/A, Block 9, KDA Scheme 5, Clifton,
KARACHI 75600, Tel. (021)587 4641-49, Fax. (021)577035/5874546

Philippines: PHILIPS SEMICONDUCTORS PHILIPPINES Inc.,
106 Valero St. Salcedo Village, P.O. Box 2108 MCC, MAKATI,
Metro MANILA, Tel. (02)810 0161, Fax. (02)817 3474

Portugal: PHILIPS PORTUGUESA, S.A.,
Rua dr. AntOnio Loureiro Borges 5, Arquiparque - Miraflores,
Apartado 300, 2795 LINDA-A-VELHA,
Tel. (01)14163160/4163333, Fax. (01)14163174/4163366

Singapore: Lorong 1, Toa Payoh, SINGAPORE 1231,
Tel. (65)350 2000, Fax. (65)251 6500

South Africa: S.A. PHILIPS Pty Ltd.,
195-215 Main Road, Martindale, 2092 JOHANNESBURG,
P.O. Box 7430, Johannesburg 2000,
Tel. (011)470-5911, Fax. (011)470-5494

Spain: Balmes 22, 08007 BARCELONA,
Tel. (03)301 6312, Fax. (03)301 42 43

Sweden: Kottbygatan 7, Akalla. S-164 85 STOCKHOLM,
Tel. (0)8-632 2000, Fax. (0)8-632 2745

Switzerland: Allmendstrasse 140, CH-8027 ZÜRICH,
Tel. (01)488 2211, Fax. (01)481 7730

Taiwan: PHILIPS TAIWAN Ltd., 23-30F, 66, Chung Hsiao West Road,
Sec. 1. Taipei, Taiwan ROC, P.O. Box 22978, TAIPEI 100,
Tel. (02)388 7666, Fax. (02)382 4382

Thailand: PHILIPS ELECTRONICS (THAILAND) Ltd.,
209/2 Sanpavuth-Bangna Road Prakanong,
BANGKOK 10260, Thailand
Tel. (662)398-0141, Fax. (662)398-3319

Turkey: Talatpasa Cad. No. 5, 80640 GÜLTEPE/ISTANBUL,
Tel. (0212)279 2770, Fax. (0212)269 3094

United Kingdom: Philips Semiconductors Ltd.,
276 Bath Road, Hayes, MIDDLESEX UB3 5BX
Tel. (081)730-5000, Fax. (081)754-8421

United States: 811 East Arques Avenue, SUNNYVALE, CA 94088-3409,
Tel. (800)234-7381, Fax. (708)296-8556

Uruguay: Coronel Mora 433, MONTEVIDEO,
Tel. (02)70-4044, Fax (02)92 0601

For all other countries apply to: Philips Semiconductors,
International Marketing and Sales, Building BE-p,
P.O. Box 218, 5600 MD, EINDHOVEN, The Netherlands,
Telex 35000 phntnl, Fax +31-40-724825
SCD36 ©Philips Electronics N.V. 1995

All rights are reserved. Reproduction in whole or in part is prohibited
without the prior written consent of the copyright owner.

The information presented in this document does not form part of any
quotation or contract, is believed to be accurate and reliable and may be
changed without notice. No liability will be accepted by the publisher for
any consequence of its use. Publication thereof does not convey nor imply
any license under patent- or industrial or intellectual property rights.

Printed in the USA

5120M/61M/CR1/pp848

Date of release: 02-95

Document order number:

9397 750 00013



This book was printed on recycled paper.