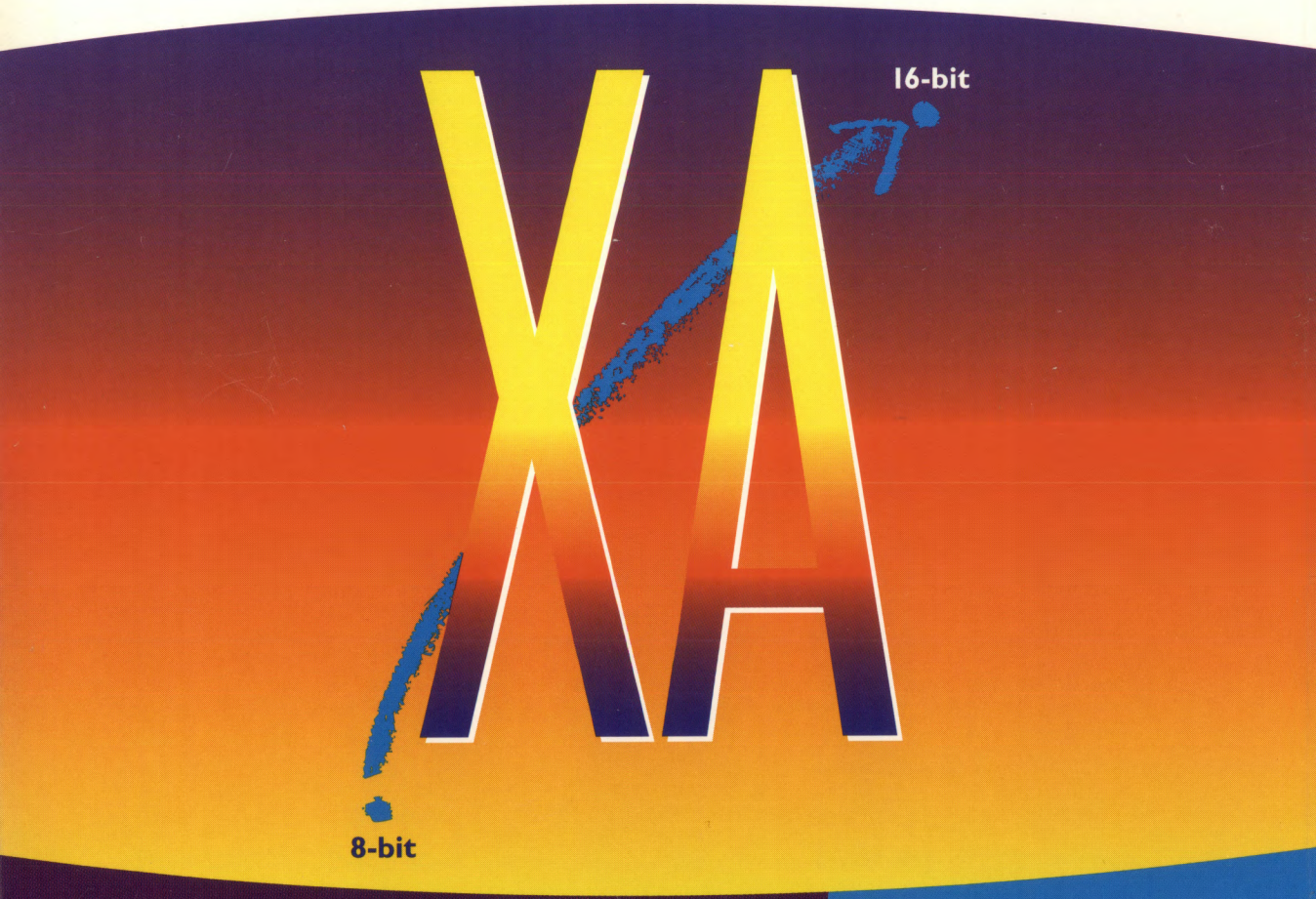
INTEGRATED CIRCUITS

16-Bit 80C51 XA Microcontrollers (eXtended Architecture)

Data Handbook IC25
CD-ROM included
1998



PHILIPS

Let's make things better.

http://www.semiconductors.philips.com

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.

16-bit 80C51XA (eXtended Architecture) Microcontrollers Data Handbook

CONTENTS

DEFINITIONS

Data Sheet Identification	Product Status	Definition (Note)
Objective Specification	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.
Preliminary Specification	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.
Product Specification	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.
Short-form specification	—	The data in this specification is extracted from a full data sheet with the same type number and title. For detailed information see the relevant data sheet or data handbook.

Limiting values

Limiting values given are in accordance with the Absolute Maximum Rating System (IEC 134). Stress above one or more of the limiting values may cause permanent damage to the device. These are stress ratings only and operation of the device at these or at any other conditions above those given in the Characteristics sections of the specification is not implied. Exposure to limiting values for extended periods may affect device reliability.

Application information

Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification

LIFE SUPPORT APPLICATIONS

These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such improper use or sale.

PURCHASE OF PHILIPS I²C COMPONENTS

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. This specification can be ordered using the code 9398 393 40011.

DISCLAIMER

Philips Semiconductors reserves the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

NOTE: Always check with your local Philips Semiconductors Sales Office to be certain that you have the latest data sheet(s) before completing a design.

Preface

XA Microcontrollers from Philips Semiconductors

Philips Semiconductors offers a wide range of microcontrollers based on the 8048, 80C51, and now the XA architectures. The XA is an architecture that was developed by Philips Semiconductors in response to the market need for higher performance than can be obtained from the 8-bit 80C51, while retaining compatibility with the 80C51 designed-in architecture. The XA successfully addresses both of these needs. It is compatible with the 80C51 at the source code level. All of the internal registers and operating modes of the 80C51 are fully supported within the XA, as are all of the 80C51 instructions. Yet compatibility with the 80C51 has in no way hindered the performance of the XA. It is a very high performance 16-bit architecture. The XA's performance is 3 to 4 times faster than that of the most popular 16 bit architectures and typically 10 times faster than the 80C51.

If you use or are familiar with the 80C51 and need higher performance, the XA is the architecture for you. You will find it very easy to understand. Rather than having to learn its programmer's model, you will find that you already know it, and, better, are very familiar with it. You will be able to focus on the enhanced features of the XA and quickly move your design to much higher performance. You will also notice that the features on the XA, in many cases, exceed what you need today. We have designed the XA so that it will meet your needs not only today but well into the future; you will not need to look for another architecture for many years to come.

Philips Semiconductors goal is to develop a family of XA derivatives as broad and robust as our 80C51/8048 family. The XA-G3 and XA-S3 derivatives have been received very well by customers with many design-ins. A FLASH derivative (XA-G49) and a serial communications controller (XA-SCC) will be introduced this year. In addition, we offer several application-specific XA derivatives for smartcards, global positioning (GPS), and text-processing for televisions. Many more XA derivatives are under development to be released in the next few years.

Philips Semiconductors offers you one of the industry's widest selections of microcontrollers. The XA architecture is an extension of this strategy that gives you the ability to easily upgrade your designs to very high performance with the only 16-bit, 80C51-compatible microcontroller available on the market.

This data handbook includes a CD-ROM which contains the entire contents of the book in machine readable form. The CD-ROM also contains a copy of the *80C51-Based 8-Bit Microcontrollers* Data Handbook, and support software and files for both the XA and 80C51 microcontroller families.

CONTENTS

IC25: 16-bit 80C51XA (eXtended Architecture) Microcontrollers

CONTENTS

CONTENTS

CONTENTS

Section 8 – Third Party Development Tools and Demo Software

Section 9 – Package Information

CONTENTS

Section 1
Selection Guides

CONTENTS

XA microcontroller development tools

COMPANY	WORLD WIDE WEB	TELEPHONE	FAX
Software: Compilers, Assemblers, Simulators			
Avocet Systems, Inc.	www.midcoast.com/~avocet/	1.207.236.9055	1.207.236.6713
Ceibo/Pantasoft	www.ceibo.com	1.314.830.4084 +972.9.9555387	1.314.830.4083 +972.9.9553297
CMX	www.cmx.com	1.508.872.7675	1.508.620.6828
Philips Semiconductors*	www.semiconductors.philips.com	1.408.991.51XA	1.408.991.3773
Tasking	www.tasking.nl	1.781.320.9400	1.781.320.9212
Emulators (including Debuggers)			
Ashling Microsystems	www.ashling.com	1.408.747.0440 +353.61.334466	1.408.747.0688 +353.61.334477
Ceibo	www.ceibo.com	1.314.830.4084 +972.9.9555387	1.314.830.4083 +972.9.9553297
Nohau Corp.	www.nohau.com/nohau/	1.408.866.1820 +46.40.592200	1.408.378.7869 +46.40.592229
Programmers			
Advin Systems	www.wco.com/~advin/	1.408.243.7000	1.408.736.2503
BP Microsystems	www.bpmicro.com	1.713.688.2675	1.713.688.0920
Ceibo	www.ceibo.com	1.314.830.4084 +972.9.9555387	1.314.830.4083 +972.9.9553297
Data I/O Corp.	sirius.data-io.com	1.425.881.6444	1.425.882.1043
Philips NZ	www.he.net/~pds/	1.408.991.51XA	1.408.991.3773
Programming Adapters			
EDI Corp.	–	1.702.735.4997	1.702.735.8339
Logical Systems	www.logicalsys.com	1.315.478.0722	1.315.479.6753
Translators			
Philips Semiconductors*	www.semiconductors.philips.com	1.408.991.51XA	1.408.991.3773
Real-Time Operating Systems			
CMX	www.cmx.com	1.508.872.7675	1.508.620.6828
Embedded Systems Products	www.esphou.com	1.281.561.9990	1.281.561.9980
R&D Books	www.rdbooks.com	1.785.841.1631	1.408.848.5784
Development Boards			
Future Designs, Inc.	members.aol.com/ teamfdi/teamfdi.htm	1.205.830.4116	1.205.830.9421
CMX	www.cmx.com	1.508.872.7675	1.508.620.6828
Philips NZ	www.he.net/~pds/	1.408.991.51XA	1.408.991.3773

* The cross assembler, simulator, and translator are available on the Philips Microcontroller Support File site at www.philipsmcu.com/. The file name is XA–TOOLS.ZIP

Microcontroller internet and bulletin board access

INTERNET ACCESS

Philips Semiconductors World Wide Web:

http://www.semiconductors.philips.com

Microcontroller Support Files:

Using a web browser: www.philipsmcu.com

Using FTP: ftp.philipsmcu.com

Philips Microcontroller Discussion Forum:

Send forum messages to: forum@philipsmcu.com

Forum messages on the web: webforum.philipsmcu.com

Email forum Subscriptions*: forum–request@philipsmcu.com

Philips Microcontroller Newsletter:

Newsletter Subscriptions*: news–request@philipsmcu.com

80C51 Applications Support Email Address:

80C51_help@sv.sc.philips.com

XA Applications Support Email Address:

XA_help@sv.sc.philips.com

* These are email-oriented internet services. To subscribe, send an email to the internet address listed above and include 'subscribe' in the subject category.

BULLETIN BOARD

To better serve our customers, Philips maintains a microcontroller bulletin board. This computer bulletin board system features microcontroller newsletters, application and demonstration programs for download, and the ability to send messages to microcontroller application engineers.

The telephone number is:

+31 40 2721102
MAX 14.400 baud
Standards V32/V42/V42.bis/HST
(The Netherlands)

Files from the former North American Bulletin Board are available on the world wide web (see previous page).

Sunnyvale ROMcode Bulletin Board

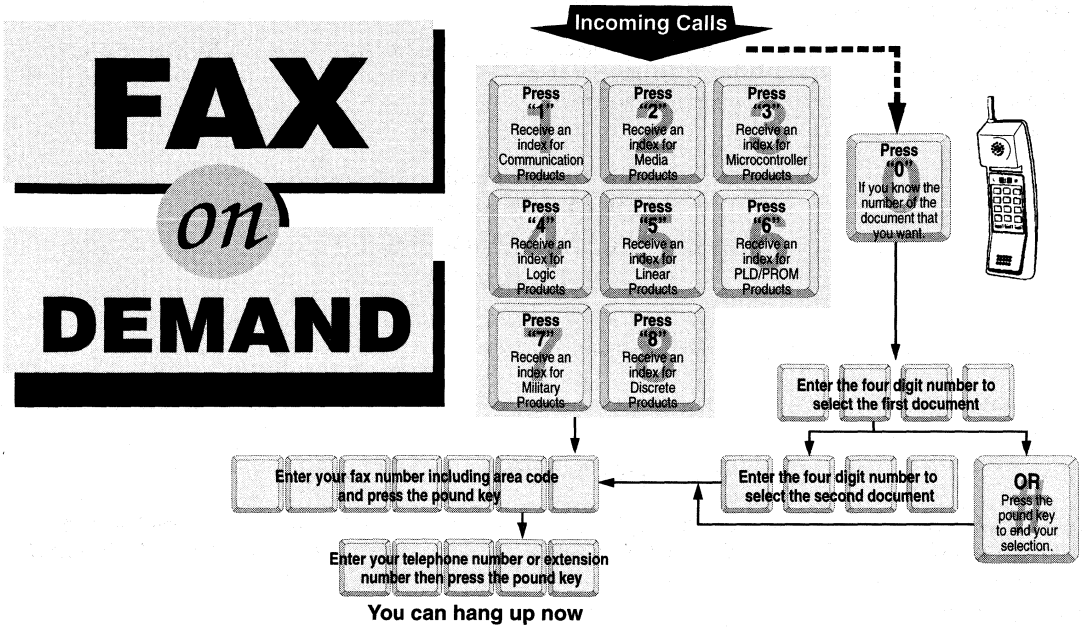
We also have a ROM code bulletin board through which you can submit ROM codes. This is a closed bulletin board for security reasons. To get an ID, contact your local sales office. The system can be accessed with a 2400, 1200, or 300 baud modem, and is available 24 hours a day.

The telephone number is:

(408) 991-3459

All code for application notes in this databook are available on the Philips web site.

FAX-on-DEMAND System



What is it?

The FAX-on-DEMAND system is a computer facsimile system that allows customers to receive selected documents by fax automatically.

How does it work?

To order a document, you simply enter the document number. This number can be obtained by asking for an index of available documents to be faxed to you the first time you call the system.

Our system has a selection of the latest product data sheets from Philips with varying page counts. As you know, it takes approximately one minute to FAX one page. This isn't bad if the number of pages is less than 10. But if the document is 37 pages long, be ready for a long transmission!

Philips Semiconductors also maintains product information on the World-Wide Web. Our home page can be located at:
http://www.semiconductors.philips.com

Who do I contact if I have a question about FAX-on-DEMAND?

Contact your local Philips sales office.

FAX-on-DEMAND phone numbers:

United Kingdom, Ireland, Benelux & Scandinavia	+44-181-730-5020
North America	1-800-282-2000
Asia/Pacific (Australia, China/HK, India, Indonesia, Japan, Korea, Malaysia, New Zealand, Philippines, Singapore, Taiwan, and Thailand)	+852 2811 9990

CMOS and NMOS 8-bit microcontroller family

8400 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	EMULATOR
84C81A	8k	256	16	DIL28/SO28	20 I/O lines 8-bit timer Byte I^2C			OM5501
84C12A	1k	64	16	DIL20/SO20	13 I/O lines 8-bit timer			OM5501
84C00B	0	256	10	piggyback	20 I/O lines 8-bit timer Byte I^2C	Piggyback		
84C00T	0	256	10	VSO-56		ROMless		
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	
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^2C, RC I^2C, LC RC LC I^2C, RC I^2C, LC RC LC I^2C, RC I^2C, LC RC LC	OM1074	For emulation of LC versions, use OM1074 + adapter_3 + 2 adapter_5 Baud for LCDS OM4831
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^2C, RC I^2C, RC	OM4829 + OM4832	OM4833 for LCD584

CMOS and NMOS 8-bit microcontroller family

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^2C			OM1025 (LCDS) + OM1026
8422 8442	2k 4k	64 128	6 6	DIL20 DIL20	13 I/O lines 8-bit timer Bit I^2C			
8401B	0	128	6	28-pin		Piggyback for 84X1		

3300 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	emulator
3349A	4k	224	1–16	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM5501/2
3350A	8k	128	1–16	QFP44 LQFP32	34 I/O lines 8-bit timer DTMF generator 256 bytes EEPROM		OM5501/2
3351A/C	2k	64	1–16	DIL28/SO28 LQFP32	20 I/O lines 8-bit timer DTMF generator 128 bytes EEPROM		OM5501/2
3352A/C	4k	128	1–16	DIL28/SO28 LQFP32	20 I/O lines 8-bit timer DTMF generator 128 byte EEPROM		OM5501/2
3353A/C	6k	128	1–16	DIL28/SO28 LQFP32	20 I/O lines 8-bit timer DTMF generator Ringer out 128 bytes EEPROM		OM5501/2
3354A	8k	256	1–16	QFP44	36 I/O lines 2x 8-bit timer DTMF generator Ringer out 256 bytes EEPROM		OM5501/2
3355A	8	128	1–16	DIL28 SO28 LQFP32	20 I/O lines 128 bytes EEPROM DTMF, 2x 8-bit counters		OM5501/2
3356A	8	128	1–16	DIL28 SO28 LQFP32	20 I/O lines 128 bytes EEPROM DTMF, 2x 8-bit counters		OM5501/2
3357A	6	128	1–16	DIL28 SO28 LQFP32	20 I/O lines 128 bytes EEPROM DTMF, 2x 8-bit counters		OM5501/2

3300 FAMILY CMOS (continued)

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	emulator
3359A	2	64	1–16	DIL28 SO28 LQFP32	20 I/O lines 128 bytes EEPROM DTMF, 2x 8-bit counters		OM5501/2
3745A	4.5k (OTP)	256	1–16	SO28 DIL28 LQFP32	16 I/O lines, 8-bit timer, RTC, V_{DD} 1.8V–6V 2 Programmable counters with 2 inputs		OM5501/2
3755A/3756A	8k(OTP)	128	1–16	DIL28/SO28	20 I/O lines 2x 8-bit timer DTMF generator Melody output 128 bytes EEPROM		OM5501/2
3354B					Piggyback for 3354A		
All 33xx +37xx							OM1025 + OM5024

NOTE: Further information on these products can be found in Databook IC03 or on our internet page http://www.semiconductors.philips.com

CMOS 16-bit microcontroller family

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 Ashling Future Designs
XA-G2	16k	512	30	3 timers, watchdog, 2 UARTs	–40 to +125°C	Nohau Ashling Future Designs
XA-G3	32k	512	30	3 timers, watchdog, 2 UARTs	–40 to +125°C	Nohau Ashling Future Designs
XA–S3	32K	1K	30	A/D converter, 3 Timers, PCA, watchdog, 2 UARTS, I^2C		Nohau Future Designs
XA-SCC	0K	256	30	DRAM controller, glueless support for most memory types, dynamic bus sizing, 4 high speed serial communication channels with hardware autobaud, v.54, and 2047, SCp, IDL, 2 timers, watchdog, 100-pin LQFP		Nohau Future Designs Tasking

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^2C, 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^2C, 3 timers (16 bit), Watchdog timer. 68000 software compatible, EMC, QFP64	–25 to +85°C	OM4162 Microcore 4	TRACE32 – (Lauterbach)
P90CL301	16MB external ROM 256 internal	512	27	2xUART, 12C, 2xtimer (16–bit), watchdog timer (21–bit), low power modes, 2xPWM (8–bit), 4xinput ADC (8–bit), LQPF80		OM5040 Microcore 5	TRACE32 (Lauterbach)

80C51 microcontroller family features guide

Memory from 1K to 8K

Prefix	Part Number ROM/ROMless/ OTP/Flash	Memory			New and Improved (Note 6)	Counter				I/O Pins	Serial Interfaces	Comments/ Special Features
		ROM	EPROM	RAM		#	PWM	PCA	WD			
P	83C750	1K		64		1	N	N	N	19	–	Lowest cost, 1 (16-bit) Timer, SSOP
P	87C750		1K	64		1	N	N	N	19	–	Lowest cost, 1 (16-bit) Timer, SSOP
P	83C748	2K		64		2	N	N	N	19	–	'751 w/o I^2C, 1 (16-bit) Timer, SSOP
P	87C748		2K	64		2	N	N	N	19	–	'751 w/o I^2C, 1 (16-bit) Timer, SSOP
S	83C751	2K		64		1	N	N	N	19	I^2C (bit)	1 (16-bit) Timer, SSOP
S	87C751		2K	64		1	N	N	N	19	I^2C (bit)	1 (16-bit) Timer, SSOP
P	83C749	2K		64		2	Y	N	N	21	–	'752 w/o I^2C, 1 (16-bit) Timer, SSOP
P	87C749		2K	64		2	Y	N	N	21	–	'752 w/o I^2C, 1 (16-bit) Timer, SSOP
S	83C752	2K		64		1	Y	N	N	21	I^2C (bit)	1 (16-bit) Timer, SSOP
S	87C752		2K	64		1	Y	N	N	21	I^2C (bit)	1 (16-bit) Timer, SSOP
P	***80C51/80C31***	***4K***		***128***	***Y***	***2***	***N***	***N***	***N***	***32***	***UART***	***CMOS***
P	***87C51***		***4K***	***128***	***Y***	***2***	***N***	***N***	***N***	***32***	***UART***	***CMOS***
P	80CL51/80CL31	4K		128		2	N	N	N	32	UART	Low voltage (1.8V–6V), Low power
P	83C434	4K		128								LCD driver
P	83CL410/80CL410	4K		128		2	N	N	N	32	I^2C	Low voltage (1.8V–6V), Low power
SC	83C451/80C451	4K		128		2	N	N	N	56	UART	Extended I/O, Processor bus interface
SC	87C451		4K	128		2	N	N	N	56	UART	Extended I/O, Processor bus interface
P	83C550/80C550	4K		128		2	N	N	Y	32	UART	8 channel 8-bit A/D w/Hdw WD
P	87C550		4K	128		2	N	N	Y	32	UART	8 channel 8-bit A/D w/Hdw WD
P	83C851/80C851	4K		128		2	N	N	N	32	UART	256B EEPROM, 80C51 pin-compatible
P	83C754	4K		256		3	Y	Y	N	11	UART	8-bit DAC, 3-input mux comparator, Ref V Out
P	87C754		4K	256		3	Y	Y	N	11	UART	(see above)
P	83C852	6K		256		2	N	N	N	16	–	Smartcard controller with 2K EEPROM (Data, Code) Cryptographic Calc Unit
P	83CL580/80CL580	6K		256		3	Y	Y	Y	40	UART, I^2C	4 channel 8-bit A/D, w/Hdw WD, low voltage (2.5V–6V), low power
P	***80C52/80C32***	***8K***		***256***	***Y***	***3***	***N***	***N***	***N***	***32***	***UART***	***80C51 pin-compatible***
P	***87C52***		***8K***	***256***	***Y***	***3***	***N***	***N***	***N***	***32***	***UART***	***80C51 pin-compatible***
P	***83C51RA+/80C51RA+***	***8K***		***512***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***w/Hdw WD, 2.7–5.5V versions***
P	***89C51RA+/87C51RA+***		***8K***	***512***	***Y***	***4***	***Y***	***Y***	***Y***	***32***		***(see above) (FLASH–5V only)***
P	83C652/80C652	8K		256		2	N	N	N	32	UART, I^2C	80C51 pin-compatible
S	87C652		8K	256		2	N	N	N	32	UART, I^2C	80C51 pin-compatible
P	83C453/80C453	8K		256		2	N	N	N	56	UART	Extended I/O, processor bus interface
P	87C453		8K	256		2	N	N	N	56	UART	Extended I/O, processor bus interface
P	***83C51FA/80C51FA***	***8K***		***256***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Enhanced UART, 3 timers + PCA***
P	***87C51FA***		***8K***	***256***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Enhanced UART, 3 timers + PCA***
P	83C575/80C575	8K		256		4	Y	Y	Y	32	UART	w/Hdw WD, low voltage detect, osc fail detect, analog comparators, PCA
P	87C575		8K	256		4	Y	Y	Y	32	UART	(see above)
P	83C576	8K		256		4	Y	Y	Y	32	UART	Same as 8xC575 plus UPI and 10-bit A/D
P	87C576	8K		256		4	Y	Y	Y	32	UART	(see above)
P	83C845	8K		256		2	Y	N	N	28	–	On-screen display, 9 PWM outputs, 3 software A/D inputs
P	83C880	8K		512								DDC interface for monitors, auto sync detection and sync processor
PCx	83C562/80C562	8K		256		3	Y	N	Y	48	UART	8 channel 8-bit A/D, 2 PWM outputs, Capture/Compare timer, w/Hdw WD
PCx	83C552/80C552	8K		256		3	Y	N	Y	48	UART, I^2C	8 channel 10-bit A/D, 2 PWM outputs, Capture/Compare timer, w/Hdw WD
S	87C552		8K	256		3	Y	N	Y	48	UART, I^2C	(see above)
P	83C834	8K		256								LCD driver
P	83CL883	8K	8K	256		3	N	N	Y	19	UART	1.8–3.6V operation, low voltage detection
P	83CL884	8K	8K	256		3	N	N	Y	18	UART	1.8–3.6V operation, low voltage detection

NOTES:
Part number prefixes are noted in the first column.
All combinations of part type, speed, temperature and package may not be available.
Parts in Italics bold are 51plus microcontrollers.

Memory from 1K to 8K (continued)

Part Number ROM/ROMless/ OTP/Flash		A/D		External Interrupt	Program Security ?	Clock Freq. (MHz)	Temperature Range (°C)			Package		
		Bits	Channels				0 to +70	–40 to +85	–55 to +125	PDIP	PLCC	PQFP/SSOP
83C750	S			2	N	3.5 to 40	X	X		N24	A28	DB24 (0–70F)
87C750	S			2	Y	3.5 to 40	X	X		N24	A28	DB24 (0–70F)
83C748	S			2	N	3.5 to 16	X	X		N24	A28	DB24 (0–70F)
87C748	S			2	Y	3.5 to 16	X	X		N24	A28	DB24 (0–70F)
83C751	S			2	N	3.5 to 16	X	X		N24	A28	DB24 (0–70F)
87C751	S			2	Y	3.5 to 16	X	X		N24	A28	DB24 (0–70F)
83C749	S	8	5	2	N	3.5 to 16	X	X		N28	A28	DB28 (0–70F)
87C749	S	8	5	2	Y	3.5 to 16	X	X		N28	A28	DB28 (0–70F)
83C752	S	8	5	2	N	3.5 to 16	X	X	X	N28	A28	DB28 (0–70F)
87C752	S	8	5	2	Y	3.5 to 16	X	X	X	N28	A28	DB28 (0–70F)
80C51/80C31	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44 (5)***
87C51	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44 (5)***
80CL51/80CL31	Z			10	N	0 to 16 (1)		X		N40 (2)		B44
83C434	T					12MHz		X		NB42		B44
83CL410/80CL410	Z			10	N	0 to 12 (1)		X		N40 (2)		B44
83C451/80C451	S			2	N	3.5 to 16	X	X		N64 (4)	A68	
87C451	S			2	Y	3.5 to 16	X	X		N64 (4)	A68	
83C550/80C550	S	8	8	2	Y	3.5 to 16	X	X		N40	A44	
87C550	S	8	8	2	Y	3.5 to 16	X	X		N40	A44	
83C851/80C851	H			2	Y	1.2 to 16	X	X		N40	A44	B44
83C754	S			2	Y	3.5 to 16	X					
87C754	S			2	Y	3.5 to 16	X					DB28
83C852	H			1	Y	1 to 12	X			SO28 or die		
83CL580/ 80CL580	Z	8	4	9	N	0 to 12 (1)		X		(3)		B64
80C52/80C32	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44 (5)***
87C52	***S***			***2***	***Y***	***0 to 33***	***X***	***X***	***X***	***N40***	***A44***	***B44 (5)***
83C51RA+/80C51RA+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
89C51RA+/87C51RA+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
83C652/80C652	H			2	Y	3.5 to 24	X	X	–40 to +125	N40	A44	B44
87C652	S			2	Y	3.5 to 16	X	X	X	N40	A44	
83C453/80C453	S			2	N	3.5 to 16	X	X			A68	
87C453	S			2	Y	3.5 to 16	X	X			A68	
83C51FA/80C51FA	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
87C51FA	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
83C575/80C575	S			2	Y	4 to 16	X	X	X	N40	A44	B44
87C575	S			2	Y	4 to 16	X	X	X	N40	A44	B44
83C576	S	10	6	2	Y	6 to 16	X	X	X	N40	A44	B44
87C576	S	10	6	2	Y	6 to 16	X	X	X	N40	A44	B44
83C845	T			2	N	3.5 to 20	X			NB42		
83C880												
83C562/80C562	H	8	8	2	N	3.5 to 16	X	X	–40 to +125		A68	B80
83C552/80C552	H	10	8	2	N	3.5 to 30	X	X	–40 to +125		A68	B80
87C552	S	10	8	2	Y	3.5 to 16	X				A68	
83C834	T					16		X		NB42		B44
83CL883	Z			8		3.58	–25 to +70					SO28
83CL884	Z			8		3.58	–25 to +70					SO28

NOTES:
Production Centers are indicated in the second column:
H = Hamburg; S = Sunnyvale, T = Taiwan, Z = Zurich
All combinatiuons of part type, speed, temperature and package may not be available.
Parts in Italics bold are 51plus microcontrollers.

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 16MHz only.
6. New and Improved devices operate from 2.7V–5.5V @ 16MHz. Static Core, 33MHz operation, Dual Data Pointers, and more.

80C51 microcontroller family features guide

Memory from 12K to 64K

Prefix	Part Number ROM/ROMless/ OTP/Flash	Memory			New and Improved (Note 6)	Counter				I/O Pins	Serial Interfaces	Comments/ Special Features
		ROM	EPROM/ FLASH	RAM		#	PWM	PCA	WD			
P	83C145	12K		256		2	Y	N	N	28	–	On-Screen Display, 9 PWM outputs, 3 software A/D inputs
P	83CL887/87CL887	12K	12K	256		3	N	N	Y	18	UART	1.8V–3.6V operation, low voltage detection
P	83C055	16K		256		2	Y	N	N	28	–	On-Screen Display, 9 PWM outputs 3 software A/D inputs
P	87C055		16K	256		2	Y	N	N	28	–	(see above)
P	83C180	16K	16K	512								DDC interface for monitors, auto sync detection and sync processor
P	***80C54***	***16K***		***256***	***Y***	***3***	***N***	***N***	***N***	***32***	***UART***	***Standard; 80C51 compatible***
P	***87C54***		***16K***	***256***	***Y***	***3***	***N***	***N***	***N***	***32***	***UART***	***Standard; 87C51 compatible***
P	89C138		16K	256		3	N	N	N	32	UART	Reduced EMI, Hdw. Watchdog timer
P	83C654/80C654	16K		256		2	N	N	N	32	UART, I²C	80C51 pin-compatible
S	87C654		16K	256		2	N	N	N	32	UART, I²C	80C51 pin-compatible
P	83CL781	16K		256		3	N	N	N	32	UART, I²C	Low voltage (1.8V–6V), low power
P	83CL782	16K		256		3	N	N	N	32	UART, I²C	83CL781 optimized 12MHz@3.1V
P	***83C51FB***	***16K***		***256***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Enhanced UART, 3 timers + PCA***
P	***87C51FB***		***16K***	***256***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Enhanced UART, 3 timers + PCA***
P	83C524	16K		512		3	N	N	Y	32	UART, I²C-bit	512 RAM
P	87C524		16K	512		3	N	N	Y	32	UART, I²C-bit	512 RAM
P	***83C51RB+***	***16K***		***512***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Extended RAM (512 bytes), 2.7V–5.5V versions***
P	***89C51RB+/87C51RB+***		***16K***	***512***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***(see above) (FLASH–5V only)***
P	83CL886/87CL886	16K	16K	512		3		N	Y	18	UART	1.8V–3.6V operation, low voltage detection
P	83C592/80C592/ 87C592	16K	16K	512		3	N	N	Y	48	UART, CAN	CAN bus controller, 8×10-bit A/D, 2 PWM outputs, Capture/Compare timer
P	83C280	24K		512								DDC interface for monitors, auto sync detection and sync processor
P	83C380	32K		512			Y					DDC interface for monitors, auto sync detection and sync processor
P	***80C58***	***32K***		***256***	***Y***	***3***	***N***	***N***	***N***	***32***	***UART***	***Standard; 80C51 compatible***
P	***87C58***		***32K***	***256***	***Y***	***3***	***N***	***N***	***N***	***32***	***UART***	***Standard; 87C51 compatible***
P	***83C51FC***	***32K***		***256***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Enhanced UART, 3 timers + PCA***
P	***87C51FC***		***32K***	***256***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Enhanced UART, 3 timers + PCA***
P	83C528/80C528	32K		512		3	N	N	Y	32	UART, I²C-bit	Large memory for high level languages
P	87C528		32K	512		3	N	N	Y	32	UART, I²C-bit	Large memory for high level languages
P	83CE528	32K		512		3	N	N	Y	32	UART, I²C-bit	8XC528 with Reduced EMI
P	***83C51RC+***	***32K***		***512***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Extended RAM (512 bytes), 2.7V–5.5V versions***
P	***89C51RC+/87C51RC+***		***32K***	***512***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***(see above) (FLASH–5V only)***
P	89C238		32K	256		3	N	N	Y	32	UART	Reduced EMI, Hdw. Watchdog timer
P	83CE598/80CE598/ 89CE598	32K	32K	512		3	Y	N	Y	48	UART, CAN	CAN bus controller, 8×10-bit A/D, 2 PWM outputs, WD, T2, Reduced EMI
P	83CE558/80CE558/ 89CE558	32K	32K	1024		3	Y	N	Y	48	UART, I²C	Low EMI, 8 Channel 10-bit A/D, 2 PWM outputs, Capture/Compare Timer
P	89C738	64K		512		3	N	N	N	32	UART	Open Collector outputs
P	***83C51RD+***	***64K***		***1024***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***Extended RAM (1K), 2.7V–5.5V versions***
P	***89C51RD+/87C51RD+***		***64K***	***1024***	***Y***	***4***	***Y***	***Y***	***Y***	***32***	***UART***	***(see above) (FLASH–5V only)***
P	87CL881		64K	2048		3	Y	N	Y	32	UART	1.8V–3.6V operation, low voltage detection
XA Family												
P	51XAG30/G37/G33	32K	32K	512		3	N	N	Y	32	2 UARTs	2.7V–5.5V operation, 16-bit XA core, compatible with 80C51
P	51XAS3		32K	1024		4	Y	Y	Y	50	2 UARTs, I²C	16M byte address range, 2.7V–5.5V operation, 8-bit DAC

NOTES:
Part number prefixes are noted in the first column.
All combinations of part type, speed, temperature and package may not be available.
Parts in Italics bold are 51plus microcontrollers.

Memory from 12K to 64K (continued)

Part Number ROM/ROMless/ OTP/Flash		A/D		External Interrupt	Program Security?	Clock Freq. (MHz)	Temperature Range (°C)			Package		
		Bits	Channels				0 to +70	–40 to +85	–55 to +125	PDIP	PLCC	PQFP/ Other
83C145	T			2	N	3.5 to 20	X			NB42		
83CL887/ 87CL887	T			8	N	3.58	–25 to +70					SO28
83C055	T			2	N	3.5 to 20	X			NB42		
87C055	T			2	N	3.5 to 20	X			NB42		
83C180	T					16		–25 to +85		NB42		
80C54	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
87C54	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
89C138	T			3	Y	3.5 to 40	X			NB42	A44	B44
83C654/80C654	H			2	Y	3.5 to 24	X	X	–40 to +125	R42, N40	A44	B44
87C654	S			2	Y	3.5 to 20	X	X		N40	A44	B44
83CL781	Z			10	N	0 to 12 (1)		X		N40		B44
83CL782	Z			10	N	0 to 12 (1)			–25 to +55	N40		B44
83C51FB	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
87C51FB	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
83C524	H			2	Y	3.5 to 24	X	X		N40	A44	B44
87C524	S			2	Y	3.5 to 16	X	X		N40	A44	B44
83C51RB+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
89C51RB+/87C51RB+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
83CL886/87CL886	Z			8								
83C592/80C592/ 87C592	H	10	8	6	Y	1.2 to 16		X	–40 to +125		A68	
83C280	T					16		–25 to +85		NB42		
83C380	T					16		–25 to +85		NB42		
80C58	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
87C58	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
83C51FC	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
87C51FC	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
83C528/80C528	H			2	Y	3.5 to 16	X	X	–40 to +125	N40, R42	A44	B44
87C528	S			2	Y	3.5 to 16	X	X		N40	A44	B44
83CE528	H			2	Y	3.5 to 16		X			A44	B44
83C51RC+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
89C51RC+/87C51RC+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
89C238	T			3	Y	3.5 to 40	X			NB42	A44	B44
83CE598/80CE598 89CE598	H	10	8	6	Y	1.2 to 16		X	–40 to +125			B80
83CE558/80CE558 89CE558	T	10	8	2	Y	1.2 to 16	X	X	–40 to +125 Except 89CE558			B80
89C738	T			3	N	3.5 to 16	X			N40	A44	B44
83C51RD+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
89C51RD+/87C51RD+	***S***			***2***	***Y***	***0 to 33***	***X***	***X***		***N40***	***A44***	***B44***
87CL881	T			8		1 to 10	–25 to +70					BD44
XA Family												
51XAG30/G37/G33	S			2	Y	30	X	X			A44	BD44
51XAS3	S	8	8	2	Y	30	X	X			A68	B80

NOTES:
Production Centers are indicated in the second column:
H = Hamburg; S = Sunnyvale, T = Taiwan, Z = Zurich
All combinatiuons of part type, speed, temperature and package may not be available.
Parts in Italics bold are 51plus microcontrollers.

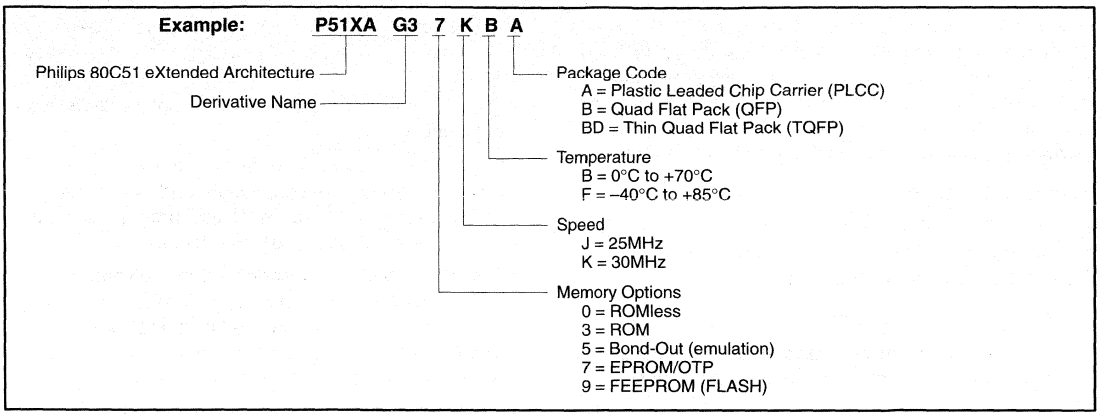
1. Oscillator options start from 32kHz.
2. Also available in VSO40 package.
3. Also available in VSO56 package.
4. Not recommended for new design.
5. Package available up to 16MHz only.
6. New and Improved devices operate from 2.7V–5.5V @ 16MHz. Static Core, 33MHz operation, Dual Data Pointers, and more.

Section 2
General Information

CONTENTS

Ordering Information

MICROCONTROLLER PRODUCTS – XA



TOTAL QUALITY MANAGEMENT

Philips Semiconductors is a Quality Company, renowned for the high quality of our products and service. We keep alive this tradition by constantly aiming towards one ultimate standard, that of zero defects. This aim is guided by our Total Quality Management (TQM) system, the basis of which is described in the following paragraphs.

Quality assurance

Based on ISO 9000 standards, customer standards such as Ford TQE and IBM MDQ. Our factories are certified to ISO 9000 by external inspectorates.

Partnerships with customers

PPM co-operations, design-in agreements, ship-to-stock, just-in-time and self-qualification programmes, and application support.

Partnerships with suppliers

Ship-to-stock, statistical process control and ISO 9000 audits.

Quality improvement programme

Continuous process and system improvement, design improvement, complete use of statistical process control, realization of our final objective of zero defects, and logistics improvement by ship-to-stock and just-in-time agreements.

ADVANCED QUALITY PLANNING

During the design and development of new products and processes, quality is built-in by advanced quality planning. Through Failure Mode Effects Analysis (FMEA) the critical parameters are detected and measures taken to ensure good performance on these parameters. This method is also used to evaluate the capability of process steps.

PRODUCT CONFORMANCE

The assurance of product conformance is an integral part of our quality assurance (QA) practice. This is achieved by:

- Incoming material management through partnerships with suppliers.
- In-line quality assurance to monitor process reproducibility during manufacture and initiate any necessary corrective action. Critical process steps are 100% under statistical process control.
- Acceptance tests on finished products to verify conformance with the device specification. The test results are used for quality feedback and corrective actions. The inspection and test requirements are detailed in the general quality specifications.
- Periodic inspections to monitor and measure the conformance of products.

PRODUCT RELIABILITY

With the increasing complexity of Original Equipment Manufacturer (OEM) equipment, components reliability must be extremely high. Our research laboratories and development departments study the failure mechanisms of semiconductors. Their studies result in design rules and process optimization for the highest built-in product reliability. Highly accelerated tests are applied to the product reliability evaluation. Rejects from reliability tests and from customer complaints are submitted to failure analysis, to result in corrective action and continuous improvement.

CUSTOMER RESPONSES

Our quality improvement depends on joint action with our customer. We need our customer's inputs and we invite constructive comments on all aspects of our performance. Please contact our local sales representative.

RECOGNITION

The high quality of our products and services is demonstrated by many Quality Awards granted by major customers and international organizations.

RATING SYSTEMS

The rating systems described are those recommended by the IEC in its publication number 134.

Definitions of terms used

ELECTRONIC DEVICE

An electronic tube or valve, transistor or other semiconductor device. This definition excludes inductors, capacitors, resistors and similar components.

CHARACTERISTIC

A characteristic is an inherent and measurable property of a device. Such a property may be electrical, mechanical, thermal, hydraulic, electro-magnetic or nuclear, and can be expressed as a value for stated or recognized conditions. A characteristic may also be a set of related values, usually shown in graphical form.

BOGEY ELECTRONIC DEVICE

An electronic device whose characteristics have the published nominal values for the type. A bogey electronic device for any particular application can be obtained by considering only those characteristics that are directly related to the application.

RATING

A value that establishes either a limiting capability or a limiting condition for an electronic device. It is determined for specified values of environment and operation, and may be stated in any suitable terms. Limiting conditions may be either maxima or minima.

RATING SYSTEM

The set of principles upon which ratings are established and which determine their interpretation. The rating system indicates the division of responsibility between the device manufacturer and the circuit designer, with the object of ensuring that the working conditions do not exceed the ratings.

Absolute maximum rating system

Absolute maximum ratings are limiting values of operating and environmental conditions applicable to any electronic device of a specified type, as defined by its published data, which should not be exceeded under the worst probable conditions.

These values are chosen by the device manufacturer to provide acceptable serviceability of the device, taking no responsibility for equipment variations, environmental variations, and the effects of changes in operating conditions due to variations in the characteristics of the device under consideration and of all other electronic devices in the equipment.

The equipment manufacturer should design so that, initially and throughout the life of the device, no absolute maximum value for the intended service is exceeded with any device, under the worst probable operating conditions with respect to supply voltage variation, equipment component variation, equipment control adjustment, load variations, signal variation, environmental conditions, and variations in characteristics of the device under consideration and of all other electronic devices in the equipment.

Design maximum rating system

Design maximum ratings are limiting values of operating and environmental conditions applicable to a bogey electronic device of a specified type as defined by its published data, and should not be exceeded under the worst probable conditions.

These values are chosen by the device manufacturer to provide acceptable serviceability of the device, taking responsibility for the effects of changes in operating conditions due to variations in the characteristics of the electronic device under consideration.

The equipment manufacturer should design to that, initially and throughout the life of the device, no design maximum value for the intended service is exceeded with a bogey electronic device, under the worst probable conditions with respect to supply voltage variation, equipment component variation, variation in characteristics of all other devices in the equipment, equipment control adjustment, load variation, signal variation and environmental conditions.

Design centre rating system

Design centre ratings are limiting values of operating and environmental conditions applicable to a bogey electronic device of a specified type as defined by its published data, and should not be exceeded under normal conditions.

These values are chosen by the device manufacturer to provide acceptable serviceability of the device in average applications, taking responsibility for normal changes in operating conditions due to rated supply voltage variation, equipment component variation, equipment control adjustment, load variation, signal variation, environmental conditions, and variations in the characteristics of all electronic devices.

The equipment manufacturer should design so that, initially, no design centre value for the intended service is exceeded with a bogey electronic device in equipment operating at the stated normal supply voltage.

ELECTROSTATIC CHARGES

Electrostatic charges can exist in many things; for example, man-made-fibre clothing, moving machinery, objects with air blowing across them, plastic storage bins, sheets of paper stored in plastic envelopes, paper from electrostatic copying machines, and people. The charges are caused by friction between two surfaces, at least one of which is non-conductive. The magnitude and polarity of the charges depend on the different affinities for electrons of the two materials rubbing together, the friction force and the humidity of the surrounding air.

Electrostatic discharge is the transfer of an electrostatic charge between bodies at different potentials and occurs with direct contact or when induced by an electrostatic field. All of our MOS devices are internally protected against electrostatic discharge but they **can** be damaged if the following precautions are not taken.

WORK STATION

Figure 1 shows a working area suitable for safely handling electrostatic sensitive devices. It has a work bench, the surface of which is conductive or covered by an antistatic sheet. Typical resistivity for the bench surface is between 1 and 500 kΩ per cm^2. The floor should also be covered with antistatic material. The following precautions should be observed:

- Persons at a work bench should be earthed via a wrist strap and a resistor
- All mains-powered electrical equipment should be connected via an earth leakage switch
- Equipment cases should be earthed
- Relative humidity should be maintained between 50 and 65%
- An ionizer should be used to neutralize objects with immobile static charges

RECEIPT AND STORAGE

MOS devices are packed for dispatch in antistatic/conductive containers, usually boxes, tubes or blister tape. The fact that the contents are sensitive to electrostatic discharge is shown by warning labels on both primary and secondary packing.

The devices should be kept in their original packing whilst in storage. If a bulk container is partially unpacked, the unpacking should be performed at a protected work station. Any MOS devices that are stored temporarily should be packed in conductive or antistatic packing or carriers.

ASSEMBLY

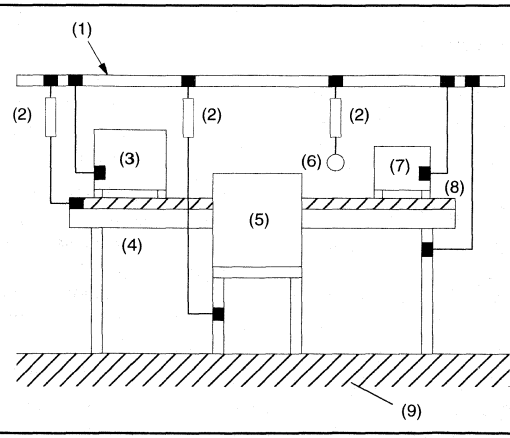
MOS devices must be removed from their protective packing with earthed component pincers or short-circuit clips. Short-circuit clips must remain in place during mounting, soldering and cleansing/drying processes. Do not remove more devices from the storage packing than are needed at any one time. Production/assembly documents should state that the product contains electrostatic sensitive devices and that special precautions need to be taken.

During assembly, ensure that the MOS devices are the last of the components to be mounted and that this is done at a protected work station.

All tools used during assembly, including soldering tools and solder baths, must be earthed. All hand tools should be of conductive or antistatic material and, where possible, should not be insulated.

Measuring and testing of completed circuit boards must be done at a protected work station. Place the soldered side of the circuit board on conductive or antistatic foam and remove the short-circuit clips. Remove the circuit board from the foam, holding the board only at the edges. Make sure the circuit board does not touch the conductive surface of the work bench. After testing, replace the circuit board on the conductive foam to await packing.

Assembled circuit boards containing MOS devices should be handled in the same way as unmounted MOS devices. They should also carry warning labels and be packed in conductive or antistatic packing.



(1) Earthing rail.
(2) Resistor (500 kΩ ± 10%, 0.5 W).
(3) Ionizer.
(4) Work bench.
(5) Chair
(6) Wrist strap.
(7) Electrical equipment.
(8), (9) Conductive surface/antistatic sheet.

Figure 1. Protected work station

Section 3
XA User Guide

CONTENTS

1 The XA Family - High Performance, Enhanced Architecture 80C51-Compatible 16-Bit CMOS Microcontrollers

1.1 Introduction

The role of the microcontroller is becoming increasingly important in the world of electronics as systems which in the past relied on mechanical or simple analog electrical control systems have microcontrollers embedded in them that dramatically improve functionality and reliability, while reducing size and cost. Microcontrollers also provide the general purpose solutions needed so that common software and hardware can be shared among multiple designs to reduce overall design-in time and costs.

The requirements of systems using microcontrollers are also much more demanding now than a few years ago. Whether called by the name "microcontrollers", "embedded controllers" or "single-chip microcomputers", the systems that use these devices require a much higher level of performance and on-chip integration.

As microcontrollers begin to enter into more complex control environments, the demand for increased throughput, increased addressing capability, and higher level of on-chip integration has led to the development of 16-bit microcontrollers that are capable of processing much more information than 8-bit microcontrollers. However, simply integrating more bits or more peripheral functions does not solve the demand of the control systems being developed today. New microcontrollers must provide high-level-language support, powerful debugging environments, and advanced methods of real time control in order to meet the more stringent functionality and cost requirements of these systems.

To meet the above goals The XA or "eXtended Architecture" family of general-purpose microcontrollers from Philips is being introduced to provide the highest performance/cost ratio for a variety of high performance embedded-systems-control applications including real-time, multi-tasking environments. The XA family members add to the CPU core a specific complement of on-chip memory, I/Os, and peripherals aimed at meeting the requirements of different application areas. The core-based architecture allows easy expansion of the family according to a wide variety of customer requirements. The powerful instruction set supports faster computing power, faster data transfer, multi-tasking, improved response to external events and efficient high-level language programming.

Upward (assembly-level) code compatibility with the Philips 80C51 family of controllers provides a smooth design transition for system upgrades by providing tremendously enhanced performance.

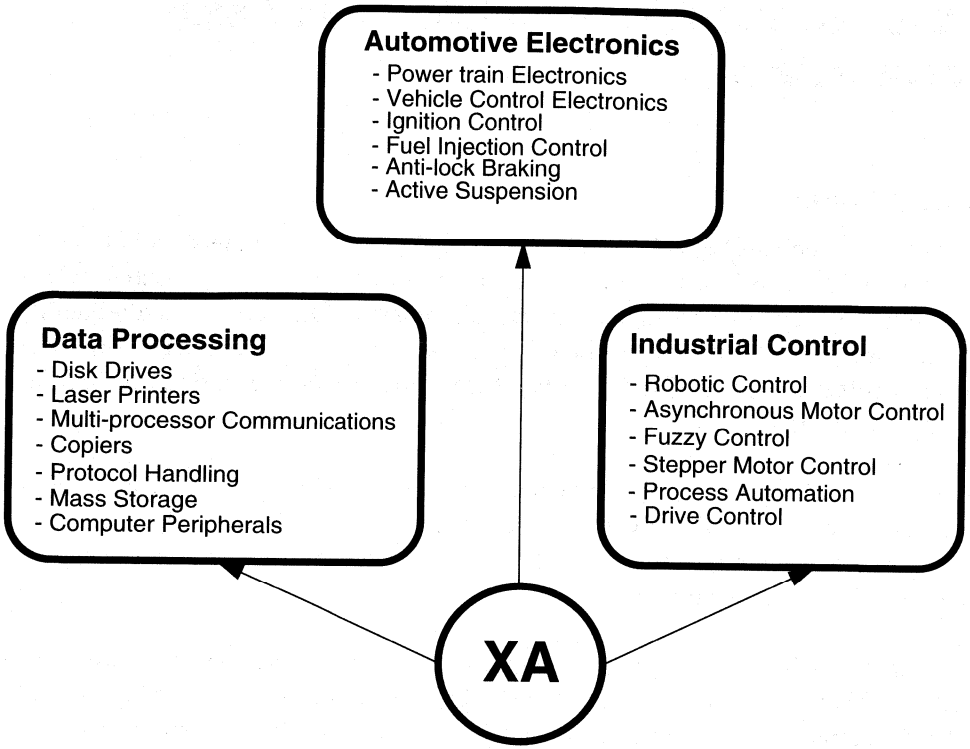


Figure 1. Applications of Philips XA microcontrollers

1.2 Architectural Features of XA

- Upward compatibility with the standard 8XC51 core (assembly source level)
- 24-bit address range (16 Megabytes code and data space)
- 16-bit static CPU
- Enhanced architecture using both 16-bit words and 8-bit bytes
- Enhanced instruction set
- High code efficiency; most of the instructions are 2-4 bytes in length
- Fast 16X16 Multiply and 32x16 Divide Instructions
- 16-bit Stack Pointers and general pointer registers
- Capability to support 32 vectored interrupts - 31 maskable and 1 NMI
- Supports 16 hardware and 16 software traps
- Power Down and Idle power reduction modes
- Hardware support for multi-tasking software

2 Architectural Overview

2.1 Introduction

The Philips XA (eXtended Architecture) has a general purpose register-register architecture to provide the best cost-to-performance trade-off available for a high speed microcontroller using today's technology. Intended as both an upward compatibility path for 80C51 users who need greater performance or more memory, and as a powerful, general-purpose 16-bit controller, the XA also incorporates support for multi-tasking operating systems and high-level languages such as C, while retaining the comprehensive bit-oriented operations that are the hallmark of the 80C51.

This overview introduces the concepts and terminology of the XA architecture in preparation for the detailed descriptions in the following sections of this manual.

2.2 Memory Organization

The XA architecture has several distinct memory spaces. The architecture and the instruction encoding are optimized for register based operations; in addition, arithmetic and logical operations may be done directly on data memory as well. Thus, the XA architecture avoids the bottleneck of having a single accumulator register.

2.2.1 Register File

The register file (Figure 2.1) allows access to 8 words of data at any one time; the eight words are also addressable as 16 bytes. The bottom 4 word registers are "banked". That is, there are four groups of registers, any one of which may occupy the bottom 4 words of the register file at any one time. This feature may be used to minimize the time required for context switching during interrupt service, and to provide more register space for complicated algorithms.

For some instructions –32-bit shifts, multiplies, and divides– adjacent pairs of word registers are referenced as double words.

The upper four words of the register file are not banked. The topmost word register is the stack pointer, while any other word register may be used as a general purpose pointer to data memory.

The entire register file is bit addressable. That is, any bit in the register file (except the 3 unselected banks of the bottom 4 words) may be operated on by bit manipulation instructions.

The XA instruction encoding allows for future expansion of the register file by the addition of 8 word registers. If implemented, these additional registers will be word data registers only and cannot be used as pointers or addressed as bytes.

The overall XA register file structure provides a superset of the 80C51 register structure. For details, refer to the section on 80C51 compatibility.

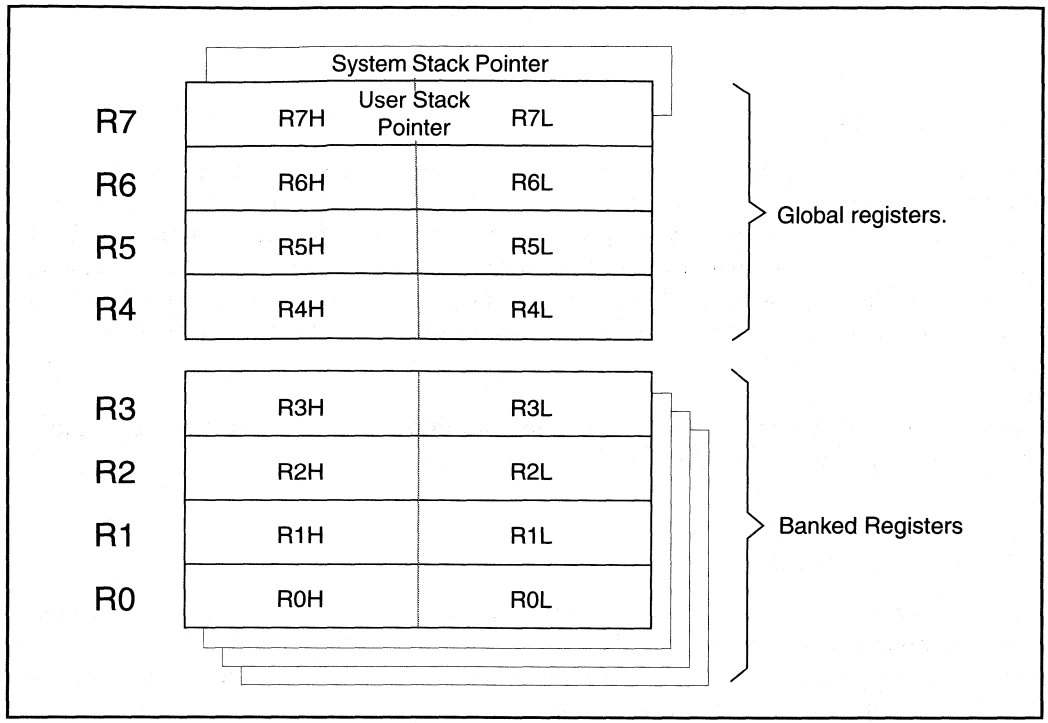


Figure 2.1 XA register file diagram

2.2.2 Data Memory

The XA architecture supports a 16 megabyte data memory space with a full 24-bit address. Some derivative parts may implement fewer address lines for a smaller range. The data space beginning at address 0 is normally on-chip and extends to the limit of the RAM size of a particular XA derivative. For addresses above that on a derivative, the XA will automatically roll over to external data memory.

Data memory in the XA is divided into 64K byte segments (Figure 2.2) to provide an intrinsic protection mechanism for multi-tasking systems and to improve performance. Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require large data memories (Figure 2.3).

The XA provides 2 segment registers used to access data memory, the Data Segment register (DS) and the Extra Segment register (ES). Each pointer register is associated with one of the segment registers via the Segment Select (SSEL) register. Pointer registers retain this association until it is changed under program control.

The XA provides flexible data addressing modes. Most arithmetic, logic, and data movement instructions support the following modes of addressing data memory:

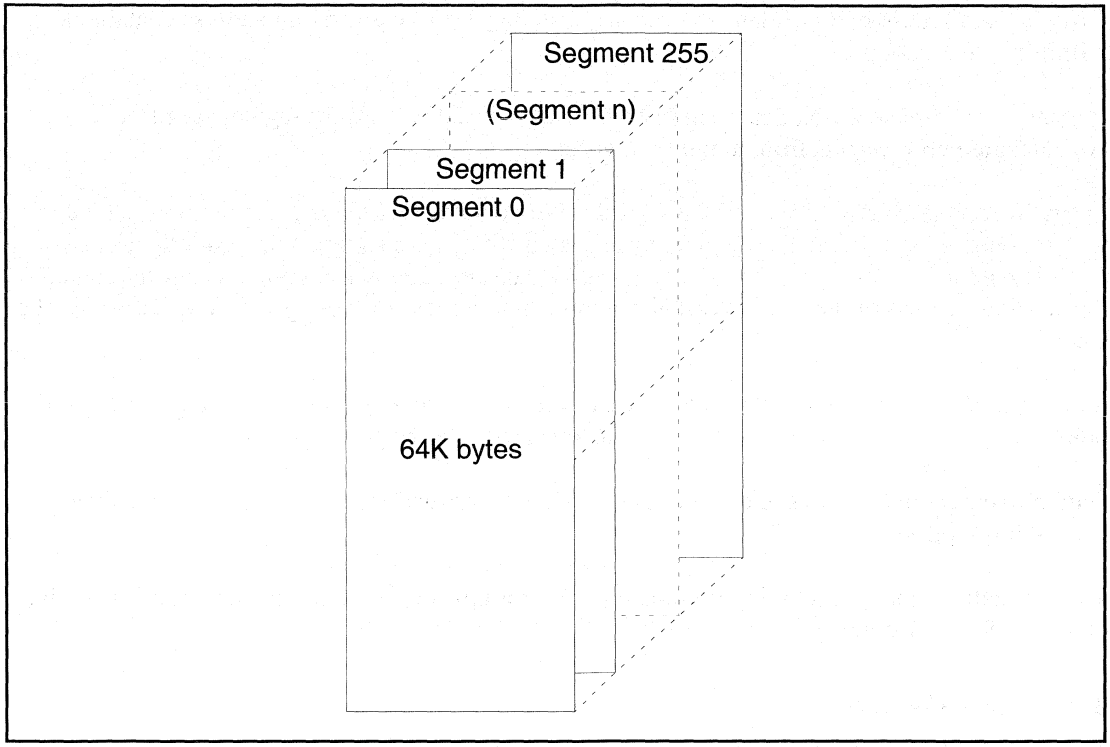


Figure 2.2 XA data memory segments

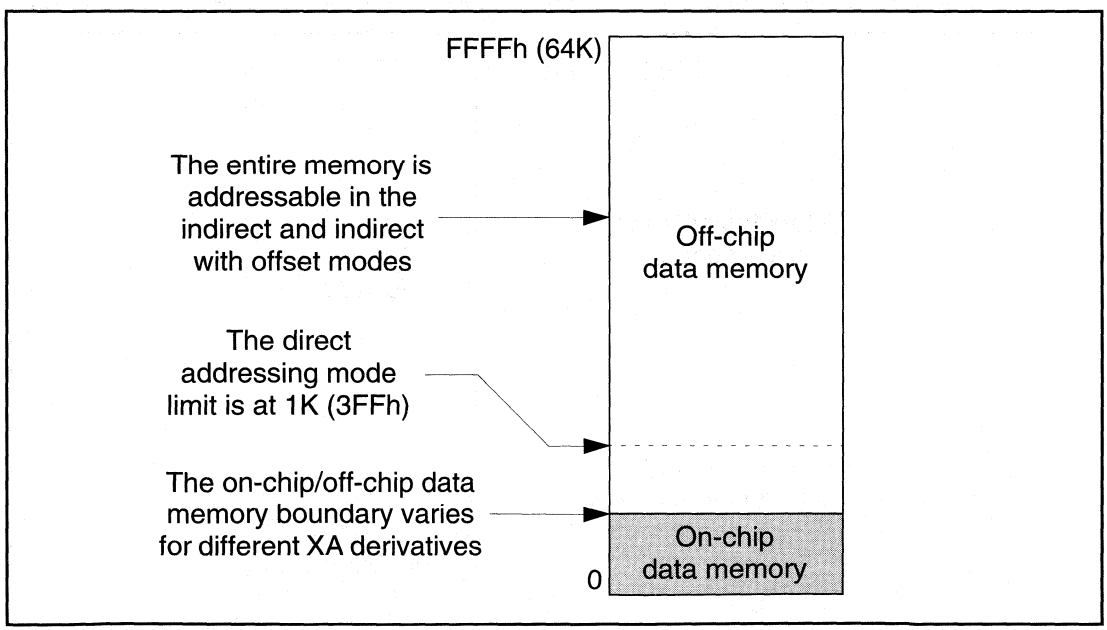


Figure 2.3 Simplified XA data memory diagram

Direct. The first 1K bytes of data on each segment may be accessed by an address contained within the instruction.

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with 16-bits from a pointer register.

Indirect with offset. An 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode allows access into a data structure when a pointer register contains the starting address of the structure. It also allows subroutines to access parameters passed on the stack.

Indirect with auto-increment. The address is formed in the same manner as plain indirect, but the pointer register contents are automatically incremented following the operation.

Data movement instructions and some special purpose instructions also have additional data addressing modes.

The XA data memory addressing scheme provides for upward compatibility with the 80C51. For details, refer to Chapter 9.

2.2.3 Code Memory

The XA is a Harvard architecture device, meaning that the code and data spaces are separate. The XA provides a continuous, unsegmented linear code space that may be as large as 16 megabytes (Figure 2.4). In XA derivatives with on-chip ROM or EPROM code memory, the on-

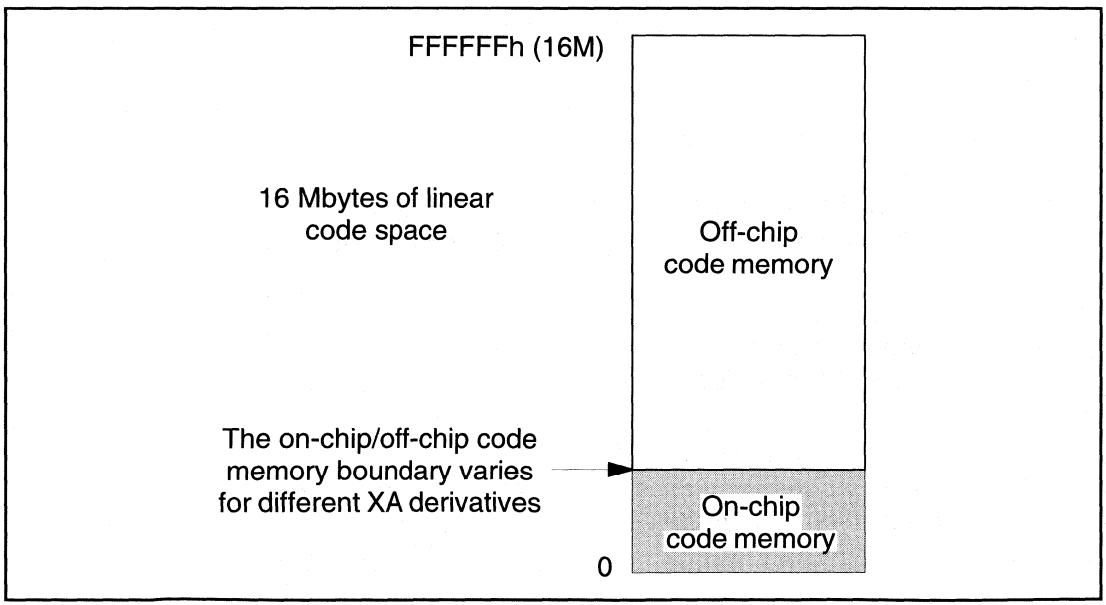


Figure 2.4 XA code memory map

chip space always begins at code address 0 and extends to the limit of the on-chip code memory. Above that, code will be fetched from off-chip. Most XA derivatives will support an external bus for off-chip data and code memory, and may also be used in a ROM-less mode, with no code memory used on-chip.

In some cases, code memory may be addressed as data. Special instructions provide access to the entire code space via pointers. Either a special segment register (CS or Code Segment) or the upper 8-bits of the Program Counter (PC) may be used to identify the portion of code memory referenced by the pointer.

2.2.4 Special Function Registers

Special Function Registers (SFRs) provide a means for the XA to access Core registers, internal control registers, peripheral devices, and I/O ports. Any SFR may be accessed by a program at any time without regard to any pointer or segment. An SFR address is always contained entirely within an instruction. See Figure 2.5.

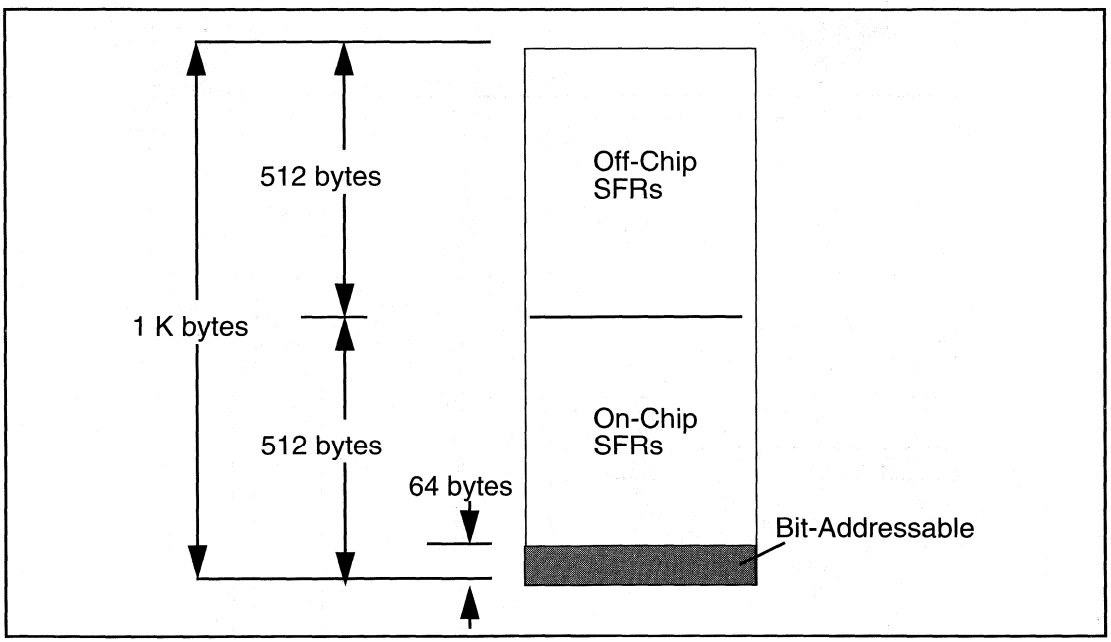


Figure 2.5 SFR Address Space

The total SFR space is 1K bytes in size. This is further divided into two 512 byte regions. The lower half is assigned to on-chip SFRs, while the second half is reserved for off-chip SFRs. This allows provides a means to add off-chip I/O devices mapped into the XA as SFRs. Off-chip SFR access is not implemented on all XA derivatives.

On-chip SFRs are implemented as needed to provide control for peripherals or access to CPU features and functions. Each XA derivative may have a different number of SFRs implemented

because each has a different set of peripheral functions. Many SFR addresses will be unused on any particular XA derivative.

The first 64 bytes of on-chip SFR space are bit-addressable. Any CPU or peripheral register that allows bit access will be allocated an address within that range.

2.3 CPU

Figure 2.6 shows the XA architecture as a whole. Each of the blocks shown are described in this section.

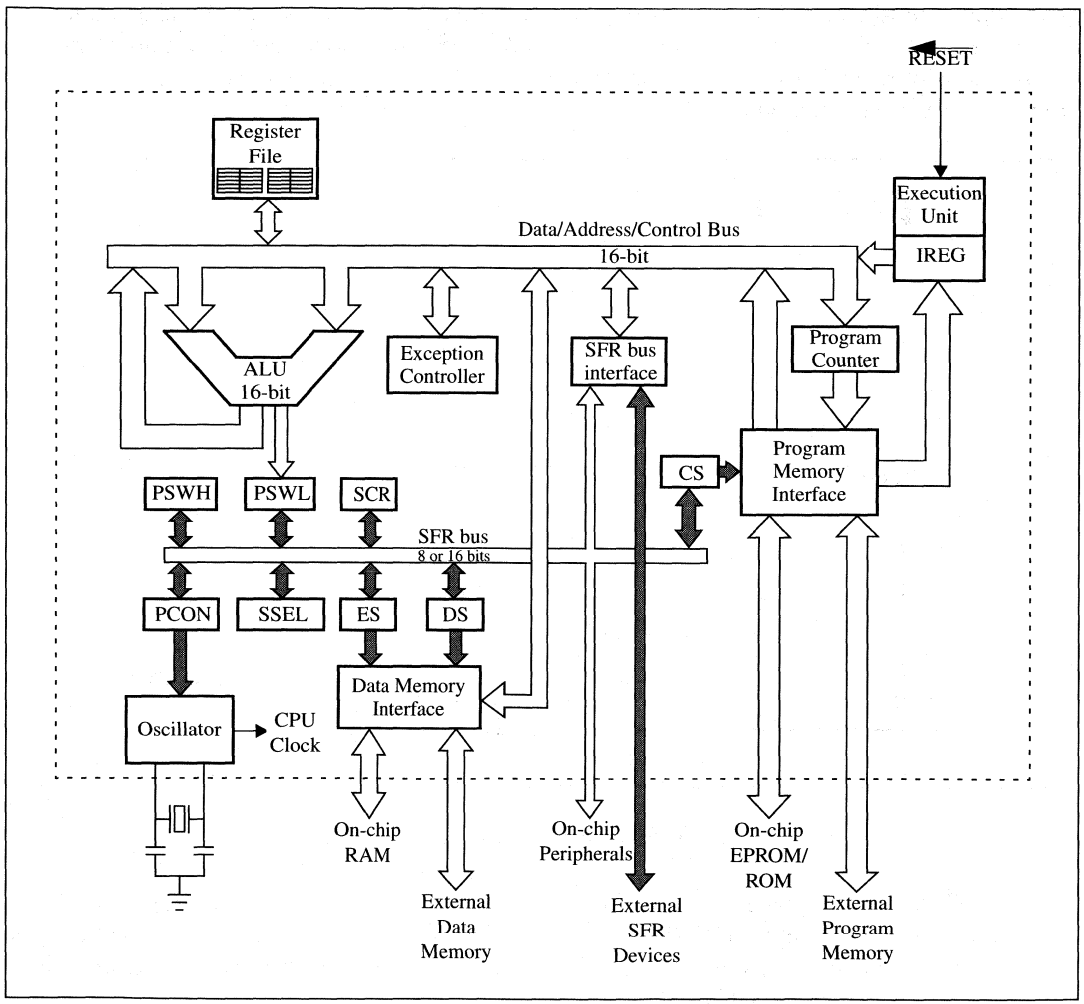


Figure 2.6 The XA Architecture

2.3.1 CPU Blocks

The XA processor is composed of several functional blocks: Instruction fetch and decode; Execution unit; ALU; Exception controller; Interrupt controller; Register File and core registers; Program memory (ROM or EPROM), Data memory (RAM); SFR and external bus interface; Oscillator; and on-chip peripherals and I/O ports.

Certain functional blocks that exist on most XA derivatives are not part of the CPU core and may vary in each derivative. These are: the external bus interface, the Special Function Register bus (SFR bus) interface, specific peripherals, I/O ports, code and data memories, and the interrupt controller.

CPU Performance Features

The XA core is partially pipelined and performs some CPU functions in parallel. For instance, instruction fetch and decode, and in some cases data write-back, are done in parallel with instruction execution. This partial pipelining gives very fast instruction execution at a very low cost. For instance, the instruction execution time for most register-to-register operations on the XA is 3 CPU clocks, or 100 nanoseconds with a 30 MHz oscillator.

ALU

Data operations in the XA core are accomplished with a 16-bit ALU, providing both 8-bit and 16-bit functions. Special circuitry has been included to allow some 32-bit functions, such as shifts, multiply, and divide.

Core Registers

The XA core includes several key Special Function Registers which are accessed by programs.

The System Configuration Register (SCR) sets up the basic operating modes of the XA. The Program Status Word (PSW) contains status flags that show the result of ALU operations, the register select bits for the four register file banks, the interrupt mask bit, and other system flags. The Data Segment (DS), Extra Segment (ES), and Code Segment (CS) registers contain the segment numbers of active data memory segments. The Segment Select register (SSEL), contains bits that determine which segment register is used by each pointer register in the register file. Bits in the Power Control register (PCON) control the reduced power modes of the processor.

Execution and Control

The Execution and Control block fetches instructions from the code memory and decodes the instructions prior to execution. The XA normally attempts to fetch instructions from the code memory ahead of what is immediately needed by the execution unit. These pre-fetched instructions are stored in a 7 byte queue contained in the fetch and decode unit.

If the fetch unit has instructions in the queue, the execution unit will not have to wait for a fetch to occur when it is ready to begin execution of a new instruction. If a program branch is taken, the queue is flushed and instructions are fetched from the new location. This block also decides whether to attempt instruction fetches from on or off-chip code memory.

The instruction at the head of the queue is decoded into separate functional fields that tell the other CPU blocks what to do when the instruction is executed. These fields are stored in staging registers that hold the information until the next instruction begins executing.

Execution Unit

The execution unit controls many of the other CPU blocks during instruction execution. It routes addressing information, sends read and write commands to the register file and memory control blocks, tells the fetch and decode unit when to branch, controls the stack, and ensures that all of these operations are performed in the proper sequence. The execution unit obtains control information for each instruction from a microcode ROM.

Interrupt Controller

The interrupt controller can receive an interrupt request from any of the sources on a particular XA derivative. It prioritizes these based on user programmable registers containing a priority for each interrupt source. It then compares the priority of the highest pending interrupt (if any) to the interrupt mask bits from the PSW. If the interrupt has a higher priority than the currently running code, the interrupt controller issues a request to the execution unit.

The interrupt controller also contains extra registers for processing software interrupts. These are used to run non-critical portions of interrupt service routines at a decreased priority without risking "priority inversion."

While the interrupt controller is not part of the XA core, it is present in some form on all XA derivatives.

Exception Controller

The exception controller is similar to the interrupt controller except that it processes CPU exceptions rather than hardware and software interrupt requests. Sources of exceptions are: stack overflow; divide by zero; user execution of an RETI instruction; hardware breakpoint; trace mode; and non-maskable interrupt (NMI).

Exceptions are serviced according to a fixed priority ranking. Generally, exceptions must be serviced immediately since each represents some important event or problem that must be dealt with before normal operation can resume.

The Exception Controller is part of the XA core and is always present.

Interrupt and Exception Processing

Interrupt and exception processing both make use of a vector table that resides in the low addresses of the code memory. Each interrupt and exception has an entry in the vector table that includes the starting address of the service routine and a new PSW value to be used at the beginning of the service routine. The starting address of a service routine must be within the first 64K of code memory.

When the XA services an exception or interrupt, it first saves the return address on the stack, followed by the PSW contents. Next, the PC and the PSW are loaded with the starting address of the appropriate service routine and the new PSW contents, respectively, from the vector table.

When the service routine completes, it returns to the interrupted code by executing the RETI (return from interrupt) instruction. This instruction loads first the PSW and then the Program Counter from the stack, resuming operation at the point of interruption. If more than the PC and PSW are used by the service routine, it is up to that routine to save and restore those registers or other portions of the machine state, normally by using the stack, and often by switching register banks.

Reset

Power up reset and any other external reset of the XA is accomplished via an active low reset pin. A simple resistor and capacitor reset circuit is typically used to provide the power-on reset pulse. the reset pin is a Schmitt trigger input, in order to prevent noise on the reset pin from causing spurious or incomplete resets.

The XA may be reset under program control by executing the RESET instruction. This instruction has the effect of resetting the processor as if an external reset occurred, except that some hardware features that are latched following a hardware reset (such as the state of the EA pin and bus width programming) are not re-latched by a software reset. This distinction is necessary because external circuitry driving those inputs cannot determine that a reset is in progress.

Some XA derivatives also have a hardware watchdog timer peripheral that will trigger an equivalent chip reset if it is allowed to time out.

Oscillator and Power Saving Modes

XA derivatives have an on-chip oscillator that may be used with crystals or ceramic resonators to provide a clock source for the processor.

The XA supports two power saving modes of operation: Idle mode and Power Down mode. Either mode is activated by setting a bit in the Power Control (PCON) register. The Idle mode shuts down all processor functions, but leaves most of the on-chip peripherals and the external interrupts functioning. The oscillator continues to run. An interrupt from any operating source will cause the XA to resume operation where it left off.

The Power Down mode goes one step further and shuts down everything, including the on-chip oscillator. This reduces power consumption to a tiny amount of CMOS leakage plus whatever loads are placed on chip pins. Resuming operation from the power down mode requires the oscillator to be restarted, which takes about 10 milliseconds. Power down mode can be terminated either by resetting the XA or by asserting one of the external interrupts, if one was left enabled when power down mode was entered. In Power Down mode, data in on-board RAM is retained. Further power savings may be made by reducing Vdd in Power Down mode; see the device data sheet for details.

Stack

The processor stack provides a means to store interrupt and subroutine return addresses, as well as temporary data. The XA includes 2 stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP), which correspond to 2 different stacks: the system stack and the user stack. See Figure 2.7. The system stack always resides in the first data memory segment,

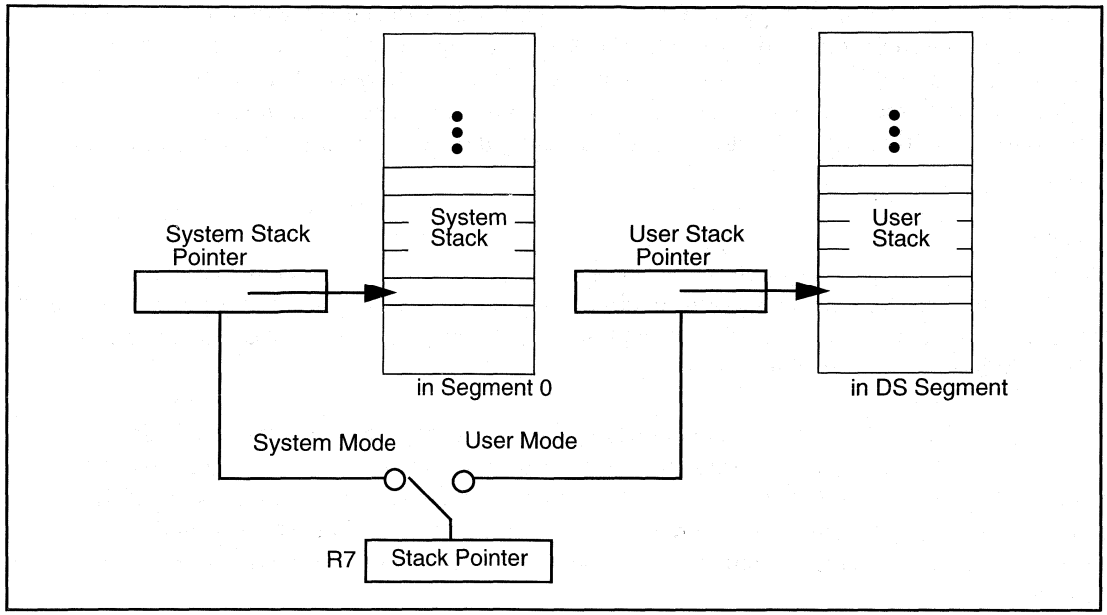


Figure 2.7 XA Stacks

segment 0. The user stack resides in the data memory segment identified by the current value of the data segment (DS) register. Executing code has access to only one of these stacks at a time, via the Stack Pointer, R7. Since each stack resides in a single data memory segment, its maximum size is 64K bytes. The purpose of having two stack pointers will be discussed in more detail in the section on Task Management below.

The XA stack grows downwards, from higher addresses to lower addresses within data memory. The current stack pointer always points to the last item pushed on the stack, unless the stack is empty. Prior to a push operation, the stack pointer is decremented by 2, then data is written to memory. When the stack is popped, the reverse procedure is used. First, data is read from memory, then the stack pointer is incremented by 2. Data on the stack always occupies an even number of bytes and is word aligned in data memory.

Debugging Features

The XA incorporates some special features designed to aid in program and system debugging. There is a software breakpoint instruction that may be inserted in a user's program by a debugger program, causing the user program to break at that point and go to the breakpoint service routine, which can transmit the CPU state so that it can be viewed by the user.

The trace mode is similar to a breakpoint, but is forced by hardware in the XA after the execution of every instruction. The trace service routine can then keep track of every instruction executed by a user program and transmit information about the CPU state to a serial port or other peripheral for display or storage. Trace mode is controlled by a bit in the PSW. The XA is able to alter the trace mode bit whenever an interrupt or exception vector is taken. This gives very flexible use of trace mode, for instance by allowing all interrupts to run at full speed to comply with system hardware requirements, while single stepping through mainline code.

With these two features, a simple monitor debugger routine can allow a user to single step through a program, or to run a program at full speed, stopping only when execution reaches a breakpoint, in either case viewing the CPU state before continuing.

2.4 Task Management

Several features of the XA have been included to facilitate multi-tasking. Multi-tasking can be thought of as running several programs at once on the same processor, with a supervisory program determining when each program, or task, runs, and for how long. Since each task shares the same CPU, the system resources required by each must be kept separate and the CPU state restored when switching execution from one task to another. The problem is much simpler for a microcontroller than it is for a microprocessor, because the code executed by a microcontroller always comes from the same source: the designers of the system it runs on. Thus, this code can be considered to be basically trustworthy and extreme measures to prevent misbehavior are not necessary. The emphasis in the XA design is to protect against simple accidents.

The first step in supporting multi-tasking is to provide two execution contexts, one for the basic tasks –on the XA termed "user mode"– and one for the supervisory program –"system mode.". A program running in system mode has access to all of the processor's resources and can set up and launch tasks.

Code running in system and user mode use different stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP) respectively. The system stack is always located in the first 64K data memory segment, where it can take advantage of the fast on-chip RAM. The user stack is located within each task's local data segment, identified by the DS register. The fact that user mode code uses a different stack than system mode code prevents tasks from accidentally destroying data on the system stack and in other task spaces.

Additional protection mechanisms are provided in the form of control bits and registers that are only writable by system mode code. For instance the DS register, that identifies the local data segment for user mode code, is only writable in the system mode. While tasks can still write to the other segment register, the ES register, they cannot write to memory via the ES register unless specifically allowed to do so by the system. The data memory segmentation scheme thus prevents tasks from accessing data memory in unpredictable ways.

Other protected features include enabling of the Trace Mode and alteration of the Interrupt Mask.

The 4 register banks are a feature that can be useful in small multi-tasking systems by using each bank for a different task, including one for system code. This means less CPU state that must be saved during task switching.

2.5 Instruction Set

The XA instruction set is designed to support common control applications. The instruction encoding is optimized for the most commonly used instructions: register to register or register with indirect arithmetic and logic operations; and short conditional and unconditional branches. These instructions are all encoded as 2 bytes. The bulk of XA instructions are encoded as either 2 or 3 bytes, although there are a few 1 byte instructions as well as 4, 5, and 6 byte instructions.

The execution of instructions normally overlaps instruction fetch, and sometimes write-back operations, in order to further speed processing.

2.5.1 Instruction Syntax

The instruction syntax chosen for the XA is similar in many ways to that of the 80C51. A typical XA instruction has a basic mnemonic, such as "ADD", followed by the operands that the operation is to be performed on. The basic syntax is illustrated in Figure 2.8. The direction of operation flow is determined by the order in which operands occur in the source line. For instance, the instruction: "ADD R1, R2" would cause the contents of R1 and R2 to be added together and the result stored in R1. Since R1 and R2 are word registers in the XA, this is a 16-bit operation.

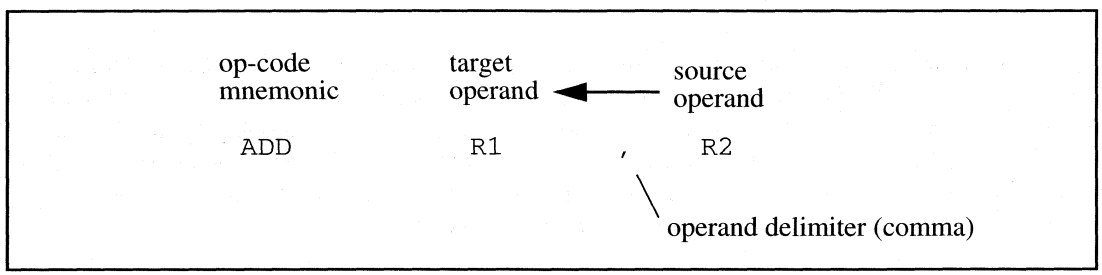


Figure 2.8 Basic Instruction Syntax

An indirect reference (a reference to data memory using the contents of a register as an address) is specified by enclosing the operand in square brackets, as in: "ADD R1, [R2]". See Figure 2.9. This instruction causes the contents of R1 and the data memory location pointed to by R2 (appended to its associated segment register) to be added together and the result stored in R1. Reversing the operand order ("ADD [R2], R1") causes the result to be stored in data memory, as shown in Figure 2.10.

Most instructions support an additional feature called auto-increment that causes the register used to supply the indirect memory address to be automatically incremented after the memory access takes place. The source line for such an operation is written as follows: "ADD R1, [R2+]". As illustrated in Figure 2.11, the auto-increment amount always matches the data size used in the instruction. In the previous example, R2 will have 2 added to it because this was a word operation.

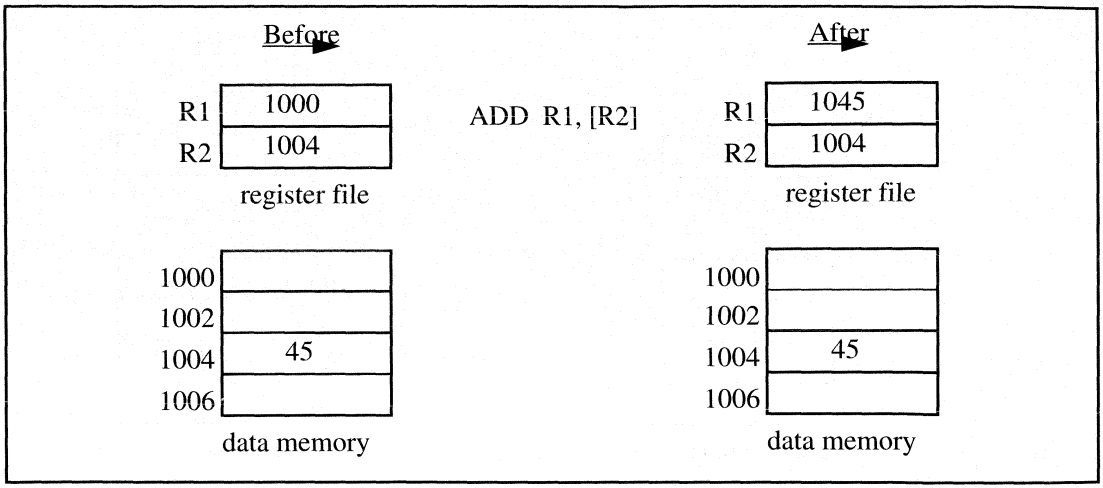


Figure 2.9 Basic Indirect Addressing Syntax, to register

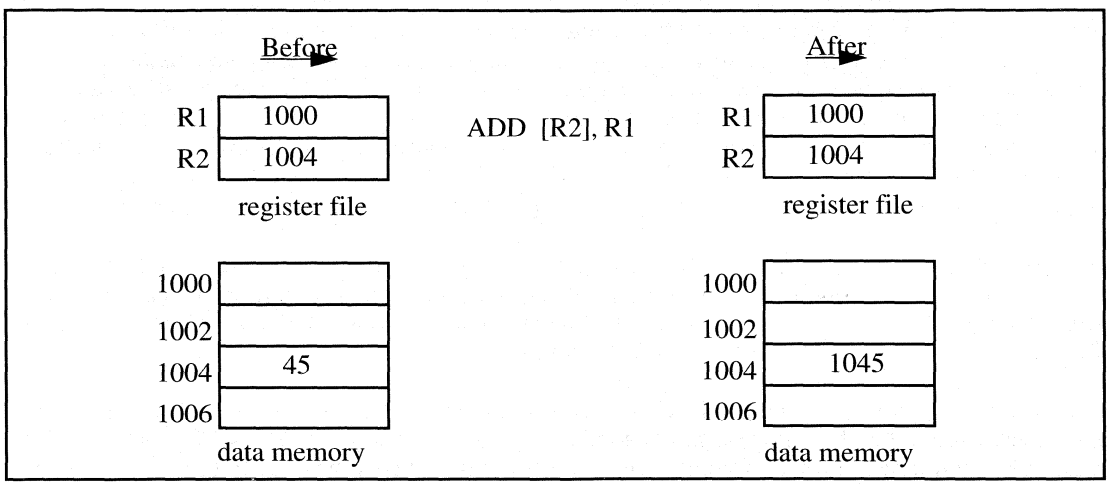


Figure 2.10 Basic Indirect Addressing Syntax, from Register

Another version of indirect addressing is called indirect with offset mode. In this version, an immediate value from the instruction word is added to the contents of the indirect register in order to form the actual address. This result of the add is 16 bits in size, which is then appended to the segment register for that pointer register. If the offset calculation overflows 16 bits, the overflow is ignored, so the indirect reference always remains on the same segment. The immediate data from the instruction is a signed 8-bit or 16-bit offset. Thus, the range is +127 bytes to -128 bytes for an 8-bit offset, and +32,767 to -32,768 bytes for a 16-bit offset. Note that since the address calculation is limited to 16-bits, the 16-bit offset mode allows access to an entire data segment.

When an instruction requires an immediate data value (a value stored within the instruction itself), it is written using the "#" symbol. For example: "ADD R1, #12" says to add the value 12 to register R1.

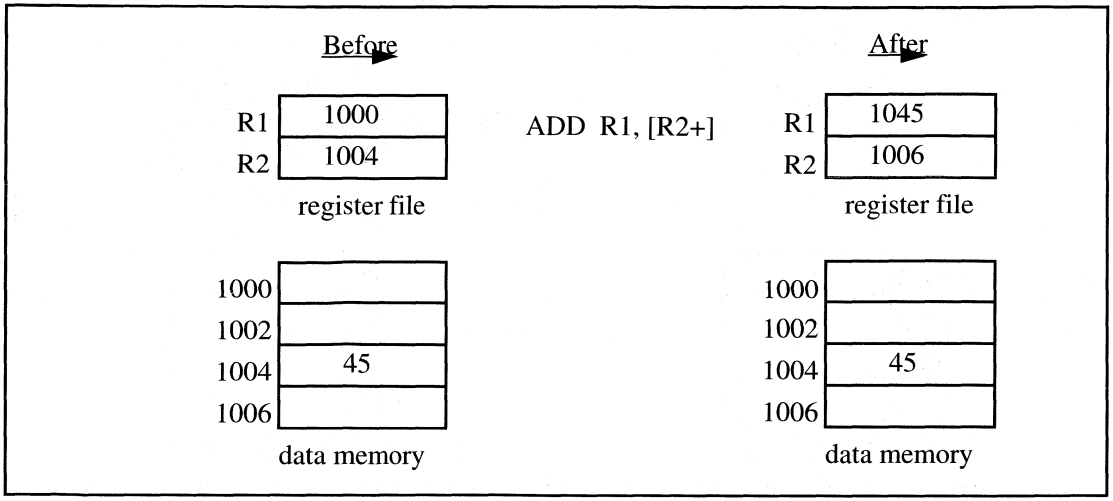


Figure 2.11 Indirect Addressing with Auto-Increment

Since indirect memory references and immediate data values do not implicitly identify the size of the operation to be performed, a few XA instructions must have an operation size explicitly called out. An example would be the instruction: "MOV [R1], #1". The immediate data value does not specify the operation size, and the value stored in memory at the location pointed to by R1 could be either a byte or a word. To clarify the intent of such an instruction, a size identifier is added to the mnemonic as follows: "MOV.b [R1], #1". This tells us that the operation should be performed on a byte. If the line read "MOV.w [R1], #1", it would be a word operation.

If a direct data address is used in an instruction, the address is simply written into the instruction: "ADD 123, R1", meaning to add the contents of register R1 to the data memory value stored at direct address 123. In an actual program, the direct data address could be given a name to make the program more readable, such as "ADD Count, R1".

Operations using Special Function Registers (SFRs) are written in a way similar to direct addresses, except that they are normally called out by their names only: "MOV PSW,#12". Using actual SFR addresses rather than their names in instructions makes the code both harder to read and less transportable between XA derivatives.

Bit addresses within instructions may be specified in one of several ways. A bit may be given a unique name, or it may be referred to by its position within some larger register or entity. An example of a bit name would be one of the status flags in the PSW, for instance the carry ("C") flag. To clear the carry flag, the following instruction could be used: "CLR C". The same bit could be addressed by its position within the PSW as follows: "CLR PSWL.7", where the period (".") character indicates that this is a bit reference. A program may use its own names to identify bits that are defined as part of the application program.

Finally, code addresses are written within instructions either by name or by value. Again, a program is more readable and easier to modify if addresses are called out by name. Examples are: "JMP Loop" and "JMP 124".

2.5.2 Instruction Set Summary

The following pages give a summary of the XA instruction set. For full details, consult Chapter 6.

Basic Arithmetic, Logic, and Data Movement Instructions

The most used operations in most programs are likely to be the basic arithmetic and logic instructions, plus the MOV (move data) instruction. The XA supports the following basic operations:

ADD	Simple addition.
ADDC	Add with carry.
SUB	Subtract.
SUBB	Subtract with borrow.
CMP	Compare.
AND	Logical AND.
OR	Logical OR.
XOR	Exclusive-OR.

These instructions support all of the following standard XA data addressing mode combinations::

Operands	Description
R, R	The source and destination operands are both registers.
R, [R]	The source operand is indirect, the destination operand is a register.
[R], R	The source operand is a register, the destination operand is indirect.
R, [R+]	The source operand is indirect with auto-increment, the destination operand is a register.
[R+], R	The source operand is a register, the destination operand is indirect with auto-increment.
R, [R+offset]	The source operand is indirect with an 8 or 16-bit offset, the destination operand is a register.
[R+offset], R	The source operand is a register, the destination operand is indirect with an 8 or 16-bit offset.
direct, R	The source operand is a register, the destination operand is a direct address.
R, direct	The source operand is a direct address, the destination operand is a register.
R, #data	The source operand is an 8 or 16-bit immediate value, the destination operand is a register.
[R], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect.

Operands	Description
[R+], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with auto-increment.
[R+offset], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with an 8 or 16-bit offset.
direct, #data	The source operand is an 8 or 16 bit immediate value, the destination operand is a direct address.

Other instructions on the XA use different operand combinations. All XA instructions are covered in detail in the Instruction Set section. Following is a summary of other instruction types:Additional arithmetic instructions

Additional arithmetic instructions

ADDS	Add short immediate (4-bit signed value).
NEG	Negate (twos complement).
SEXT	Sign extend.
MUL	Multiply.
DIV	Divide.
DA	Decimal adjust.
ASL	Arithmetic shift left.
ASR	Arithmetic shift right.
LEA	Load effective address.

Additional logic instructions

CPL	Complement (ones complement or logical inverse).
LSR	Logical shift right.
NORM	Normalize.
RL	Rotate left.
RLC	Rotate left through carry.
RR	Rotate right.
RRC	Rotate right through carry.

Other data movement instructions

MOVS	Move short immediate (4-bit signed value).
MOVC	Move to or from code memory.
MOVX	Move to or from external data memory.
PUSH	Push data onto the stack.
POP	Pop data from the stack.
XCH	Exchange data in two locations.

Bit manipulation instructions

SETB	Set (write a 1 to) a bit.
CLR	Clear (write a 0 to) a bit.
MOV	Move a bit to or from the carry flag.
ANL	Logical AND a bit (or its inverse) to the carry flag.
ORL	Logical OR a bit (or its inverse) to the carry flag.

Jump, branch, and call instructions

BR	Branch to code address (plus or minus 256 byte range).
JMP	Jump to code address (range depends on specific JMP variation).
CALL	Call subroutine (range depends on specific CALL variation).
RET	Return from subroutine or interrupt.
Bcc	Conditional branches with 15 possible condition variations.
JB, JNB	Jump if a bit set or not set.
CJNE	Compare two operands and jump if they not equal.
DJNZ	Decrement and jump if the result is not zero.
JZ, JNZ	Jump on zero or not zero (included for 80C51 compatibility).

Other instructions

NOP	No operation (used mainly to align branch targets).
BKPT	Breakpoint (used for debugging).
TRAP	Software trap (used to call system services in a multitasking system).
RESET	Reset the entire chip.

2.6 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices, and initiates code read, data read, or data write strobes. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

As described in section 4.4.4, the initial external bus width is hardware settable, and the XA determines its value (8 or 16 bits) during the reset sequence.

2.6.1 External Bus Signals

The standard XA external bus supports 8 or 16-bit data transfers and up to 24 address lines. The precise number of address lines varies by derivative. The standard control signals and their functions for the external bus are as follows:

Signal name	Function
ALE	Address Latch Enable. This signal directs an external address latch to store a portion of the address for the next bus operation. This may be a data address or a code address.
$\overline{\text{PSEN}}$	Program Store Enable. Indicates that the XA is reading code information over the bus. Typically connected to the Output Enable pin of external EPROMs.
$\overline{\text{RD}}$	Read. The external data read strobe. Typically connected to the $\overline{\text{RD}}$ pin of external peripheral devices.
$\overline{\text{WRL}}$	Write. The low byte write strobe for external data. Typically connected to the $\overline{\text{WR}}$ pin of external peripheral devices. For an 8-bit data bus, this is the only write strobe. For a 16-bit data bus, this strobe applies only to the lower data byte.
$\overline{\text{WRH}}$	Write High. This is the upper byte write strobe for external data when using a 16-bit data bus.
WAIT	Wait. Allows slowing down any type external bus cycle. When asserted during a bus operation, that operation waits for this signal to be de-asserted before it is completed.

2.6.2 Bus Configuration

The standard XA bus is user configurable in several ways. First, the bus size may be configured to either 8 bits or 16 bits. This may be configured by the logic level on a pin at reset, or under firmware control (if code is initially executed from on-chip code memory) prior to any actual external bus operations. As on the 80C51, the $\overline{\text{EA}}$ pin determines whether or not on-chip code memory is used for initial code fetches.

Second, the number of address lines may be configured in order to make optimal use of I/O ports. Since external bus functions are typically shared with I/O ports and/or peripheral I/O functions, it is advantageous to set the number of address lines to only what is needed for a particular application, freeing I/O pins for other uses.

2.6.3 Bus Timing

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, $\overline{\text{PSEN}}$ width, $\overline{\text{RD}}$ and $\overline{\text{WRL}}/\overline{\text{WRH}}$ width, and data hold time from $\overline{\text{WRL}}/\overline{\text{WRH}}$. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

The following figures show the basic sequence of events and timing of typical XA bus accesses. For more detailed information, consult Section 7 and the device data sheet.

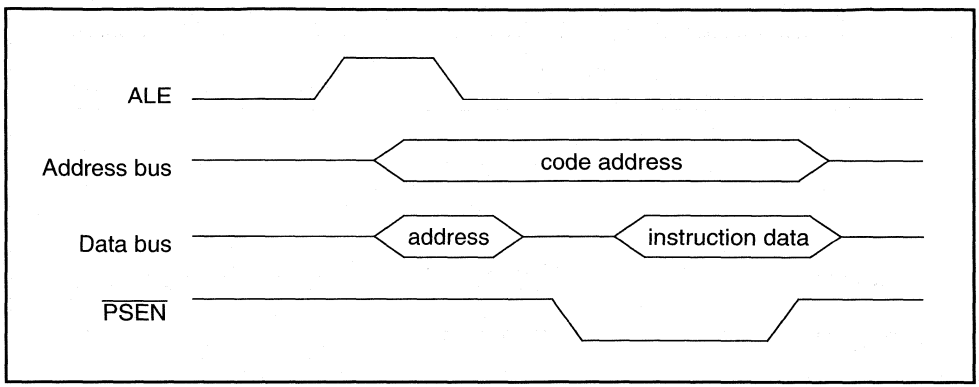


Figure 2.12 Typical External Code Read.

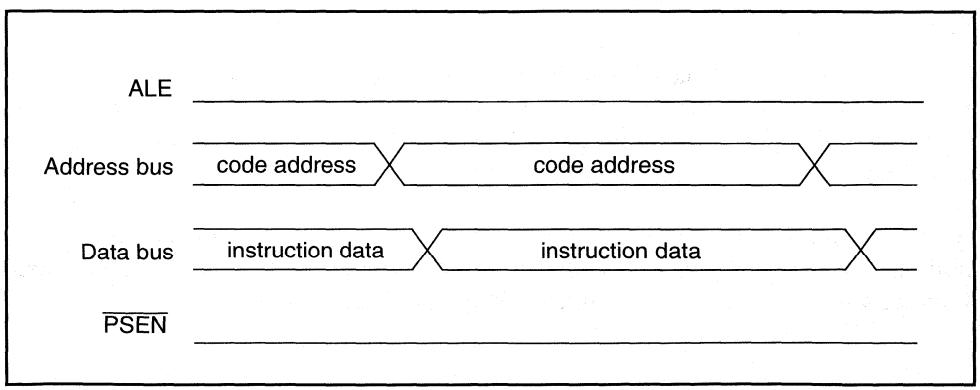


Figure 2.13 Optimized (Sequential Burst) External Code Read.

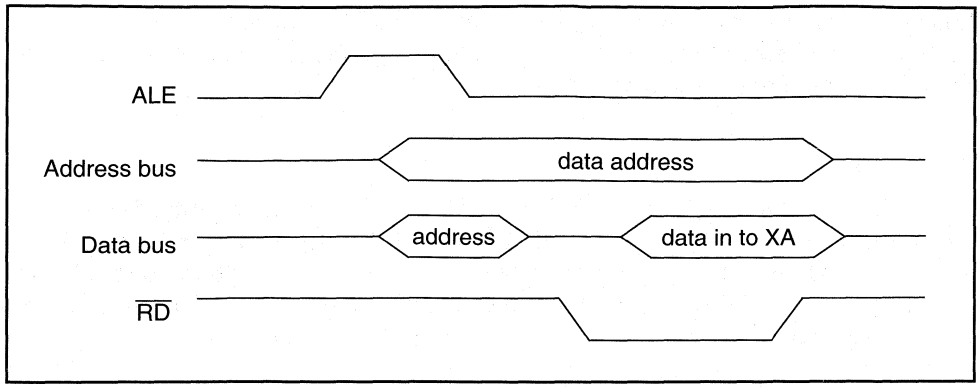


Figure 2.14 Typical External Data Read.

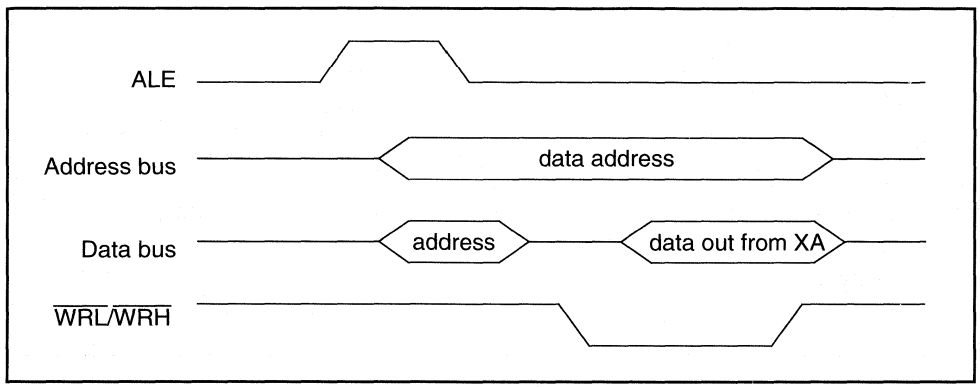


Figure 2.15 Typical External Data Write.

2.7 Ports

Standard I/O ports on the XA have been enhanced to provide better versatility and programmability than was previously available in the 80C51 and most of its derivatives. Access to the I/O ports from a program is through SFR addresses assigned to those ports. Ports may be read and written is the same manner as any other SFR.

The XA provides more flexibility in the use of I/O ports by allowing different output configurations. See Figure 2.16. Port outputs may be programmed to be quasi-bidirectional (80C51 style ports), open drain, push-pull, and high impedance (input only).

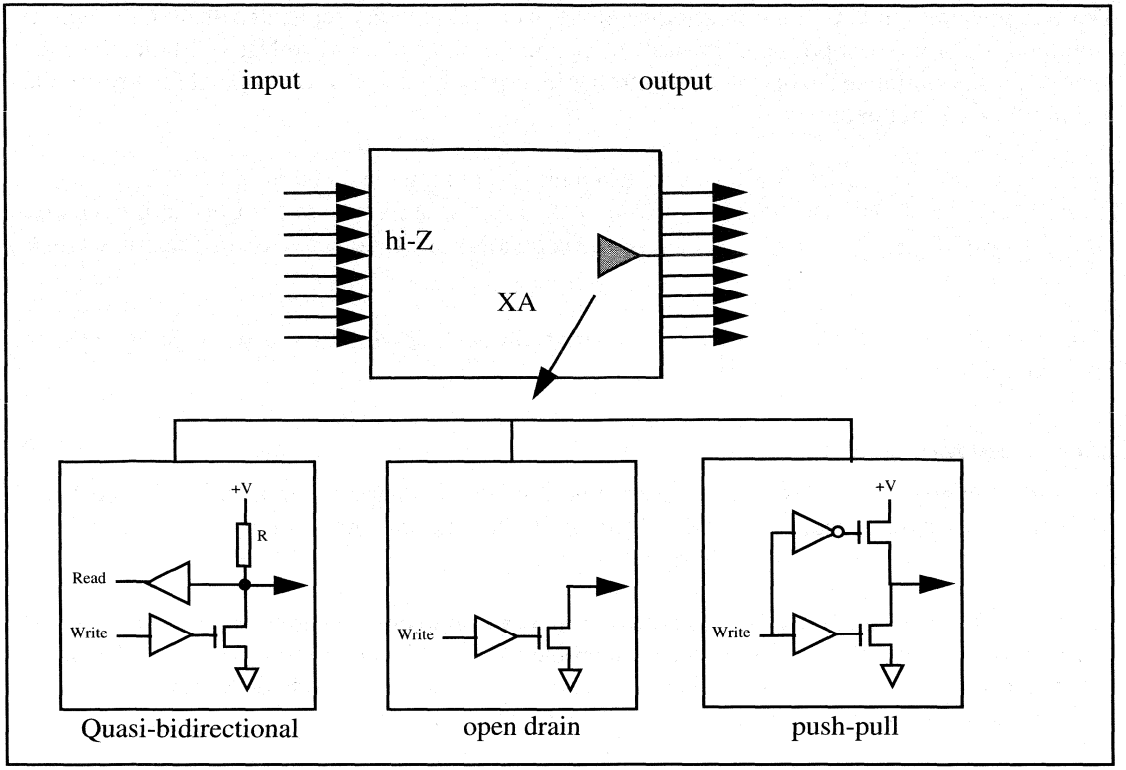


Figure 2.16 XA Port Pins with Driver Option Detail

2.8 Peripherals

The XA CPU core is designed to make derivative design fast and easy. Peripheral devices are not part of the core, but are attached by means of a Special Function Register bus, called the SFR bus, which is distinct from the CPU internal buses. So, a new XA derivative may be made by designing a new SFR bus compatible peripheral function block, if one does not already exist, then attaching it to the XA core.

2.9 80C51 Compatibility

The 80C51 is the most extensively designed-in 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this device family. For customers who use the 80C51 or one of its derivatives, preservation of their investment in code development is an important consideration. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with this higher-performance 16-bit controller. At the same time, the XA hardware was designed with the clear goal of upward compatibility. 80C51 designs may be migrated to the XA with very few changes necessary to software source or hardware.

The XA provides an 80C51 Compatibility Mode, which essentially replicates the 80C51 register architecture for the best possible upward compatibility. In the alternative Native Mode, the XA operates as an optimized 16-bit microcontroller incorporating the best conceptual features of the original 80C51 architecture.

Many trade-offs and considerations were taken into account in the creation of the XA architecture. The most important goal was to make it possible for a software translator to convert 80C51 assembler source code to XA source code on a 1:1 basis, i.e., one XA instruction for one 80C51 instruction.

Some specific compatibility issues are summarized in the following two sections. See Chapter 9 for a complete description of compatibility.

2.9.1 Software Compatibility

Several basic goals were observed in order to design 80C51 software compatibility for the XA, while avoiding over-complicating the XA design. Following are some key issues for XA software:

• Instruction mapping. Each 80C51 instruction translates into one XA instruction. Multi-instruction combinations that could result in problems if split by an interrupt were avoided as much as possible. Only one 80C51 instruction does not have a one-to-one direct replacement with an XA instruction (this instruction, XCHD, is extremely rarely used).

• "As-is" instructions. Most XA instructions are more powerful supersets of 80C51 instructions. Where this was not possible, the original 80C51 instruction is included "as-is" in the XA instruction set.

• Timing. Instruction timing must necessarily change in order to improve performance. The XA does not attempt to retain timing compatibility with the 80C51; rather, the design simply maximizes instruction execution speed. When 80C51 code that is timing critical is translated to the XA, the user must re-analyze the timing and make adjustments.

• SFR Access. Translation of SFR accesses is usually simple, since SFRs are normally referenced by name. Such references are simply retained in the translated XA code. If program source code from a specific 80C51 derivative references an SFR by its address, the translator can directly substitute the appropriate XA SFR, provided both the 80C51 and the XA derivative are correctly identified to the translator.

2.9.2 Hardware Compatibility

The key goal for hardware was to produce a familiar architecture with a good deal of upward compatibility.

• Memory Map. A major consideration in hardware compatibility of the XA with the 80C51 is the memory map. The XA approaches this issue by having each memory area (registers, data memory, code memory, stack, SFRs) be a superset of the corresponding 80C51 area.

• Stack. One area where a functional change could not be avoided is in the use of the processor stack. Due to the fact that the XA supports 16-bit operations in memory, it was necessary to change the direction of stack growth to downward –the standard for 16-bit processors– in order to match stack usage with efficient access of 16-bit variables in memory. This is an important consideration for support of high-level language compilers such as C.

• Pin-for-pin compatibility. XA derivatives are not intended to be exactly pin-compatible with other 80C51 derivatives that have similar features. Many on-chip XA peripherals, for example, have improved capabilities, and maintaining pin-for-pin compatibility would limit access to these capabilities. In general, peripherals have been made upward compatible with the original 80C51 devices, and most enhancements are added transparently. In these cases, 80C51 code will operate correctly on the 80C51 functional subset.

• Bus Interface. The external bus on the XA is an example of a trade-off between 80C51 compatibility and performance. In order to provide more flexibility and maximum performance, the 80C51 bus had to be changed somewhat. The differences are described in detail in the section on the external bus.

3 XA Memory Organization

3.1 Introduction

The memory space of XA is configured in a Harvard architecture which means that code and data memory (including sfrs) are organized in separate address spaces. The XA architecture supports 16 Megabytes (24-bit address) of both code and data space. The size and type of memory are specific to an XA derivative.

The XA supports different types of both code and data memory e.g.,code memory could be Eprom, EEProm, OTP ROM, Flash, and Masked ROM whereas data memory could be RAM, EEProm or Flash.

This chapter describes the XA Memory Organization of register, code, and data spaces; how each of these spaces are accessed, and how the spaces are related.

3.2 The XA Register File

The XA architecture is optimized for arithmetic, logical, and address-computation operations on the contents of one or more registers in the XA Register File.

3.2.1 Register File Overview

The XA architecture defines a total of 16 word registers in the Register File:
In the baseline XA core, only R0 through R7 are implemented. These registers are available for unrestricted use except R7– which is the XA stack pointer, as illustrated in Figure 3.1. In effect, the XA registers provide users with at least 7 distinct "accumulators" which may be used for all operations. As will be seen below, the XA registers are accessible at the bit, byte, word, and doubleword level.

Additional global registers, R8 through R15, are reserved and may be implemented in specific XA derivatives. These registers, when available, are equivalent to R0 through R7 except byte access and use as pointers will not be possible (only word, double-word, and bit-addressable). The Register File is independent of all other XA memory spaces (except in Compatibility Mode; see chapter 9).

Register File Detail

Figure 3.2 describes R0 through R7 in greater detail.

Byte, Word, and Doubleword Registers

All registers are accessible as bits, bytes, words, and –in a few cases– doublewords. Bit access to registers is described in the next section. As for byte and word accesses, R1 –for example– is a word register that can be word referenced simply as "R1". The more significant byte is labeled as "R1H" and the less significant byte of R1 is referenced as "R1L". Double-word registers are always formed by adjacent pairs of registers and are used for 32 bit shifts, multiplies, and divides. The pair is referenced by the name of the lower-numbered register (which contains the

less significant word), and this must have an even number. Thus valid double-register pairs are (R0,R1), (R2,R3), (R4,R5) and (R6, R7).

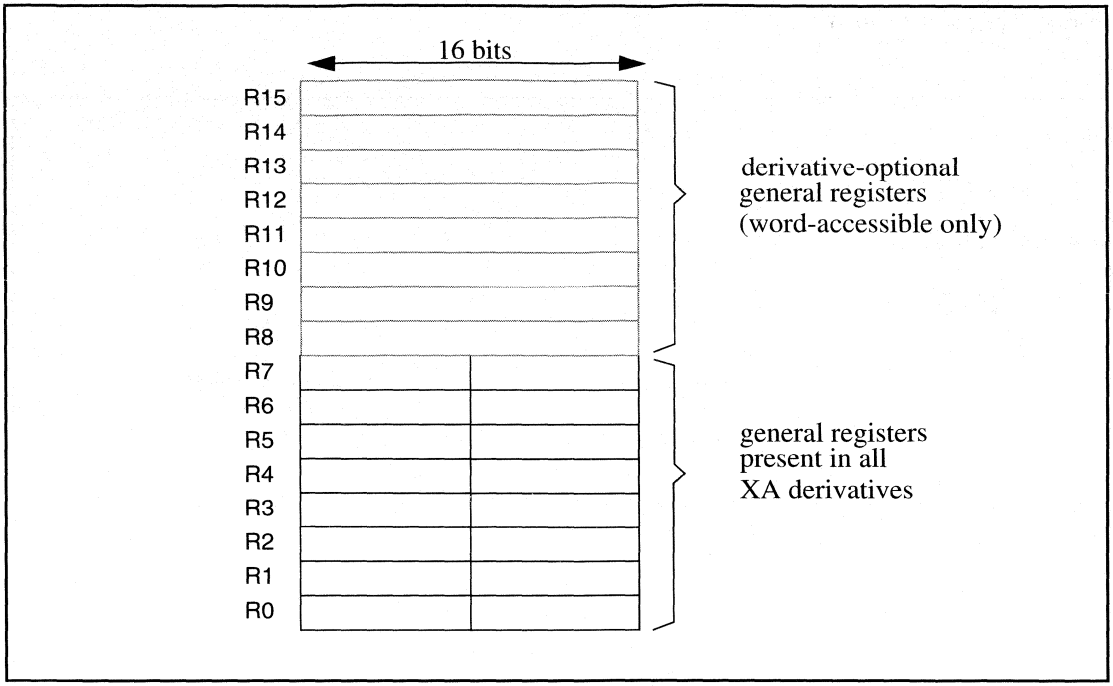


Figure 3.1 XA Register File Overview

As described in section 4.7, there are two stack pointers, one for user mode and another for system mode. At any given instant only one stack pointer is accessible and its value is in R7. When PSW.SM is 0, user mode is active and the USP is accessible via R7. When PSW.SM is 1, the XA is operating in system mode, and SSP is in SP (R7). (Note however, as described in Chapter 4, all interrupts save stack frames on the system stack, using the SSP, regardless of the current operating mode.)

There are four distinct instances of registers R0 through R3. At any given time, only 1 set of the 4 banks is active, referenced as R0 through R3, and the contents of the other banks are inaccessible. This allows high-speed context-switching, for example, for interrupt service routines. **PSW** bits **RS1** and **RS0** select the active register bank:

RS1	RS0	visible register bank
0	0	bank 0
0	1	bank 1
1	0	bank 2
1	1	bank 3

PSW.RSn are writable when the XA is operating in system or user mode, and programs running in either mode may explicitly change these bits to make selected banks visible one at a time. More commonly, the interrupt mechanism, as described in Chapter 4, provides automatic implicit register bank switching so interrupt handlers may immediately begin operating in a reserved register context.

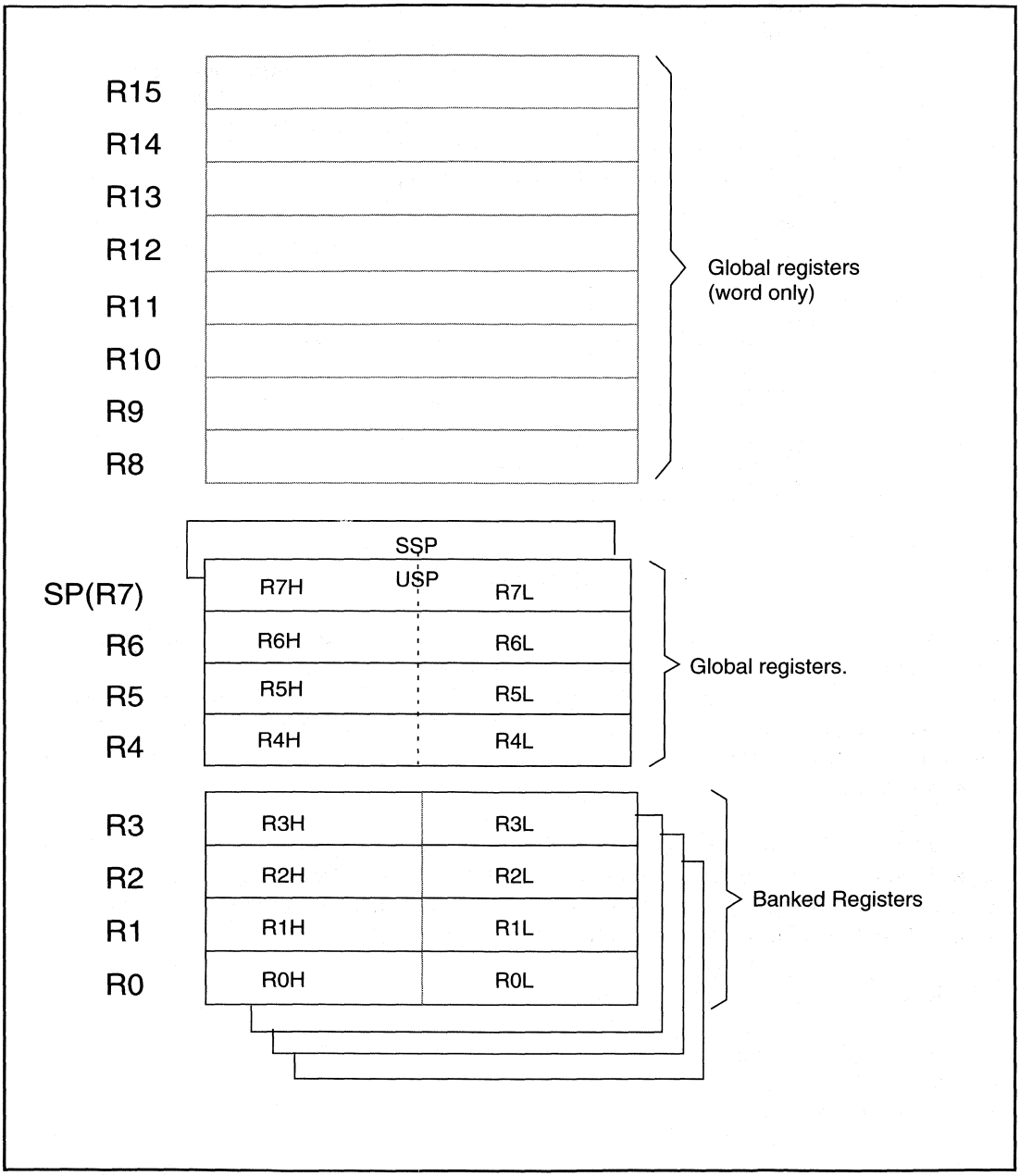


Figure 3.2 XA Register File

Bit Access to Registers

The XA Registers are all bit addressable. Figure 3.3 shows how bit addresses overlie the basic register file map. In general, absolute bit references as given in this map are unnecessary. XA software development tools provide symbolic access to bits in registers. For example, bit 7 may be designated as "R0.7" with no ambiguity

Bit references to banked registers R0 through R3 access the currently accessible register bank, as set by **PSW** bits **RS1**, **RS0** and the currently selected stack pointer USP or SSP. The unselected registers are inaccessible..

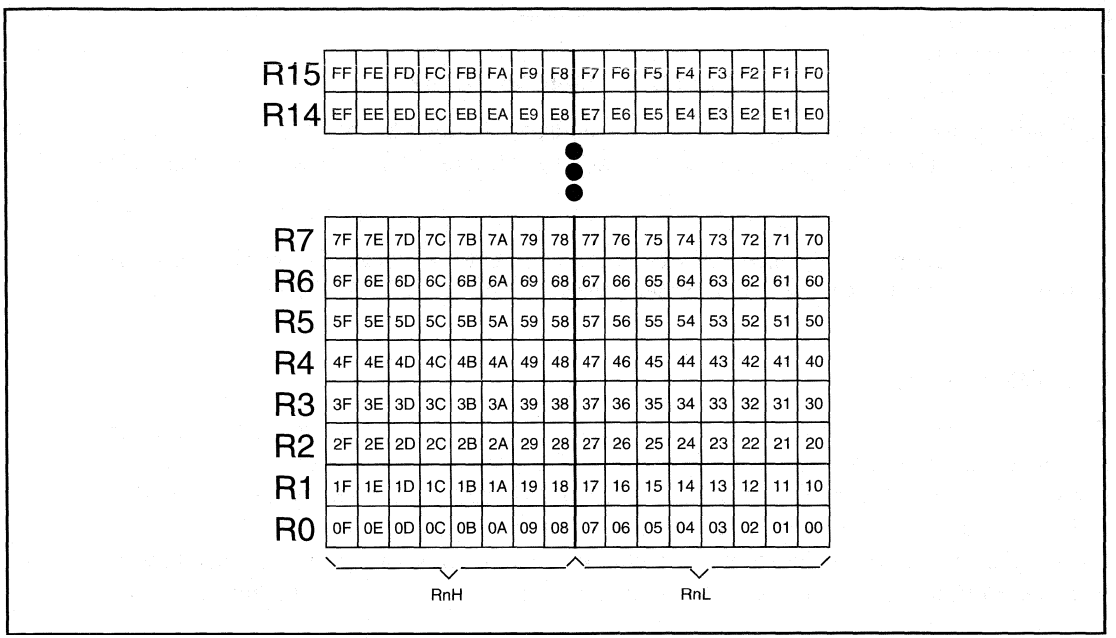


Figure 3.3 Bit Address to Registers

3.3 The XA Memory Spaces

The XA divides physical memory into program and data memory spaces. Twenty-four address bits, corresponding to a 16MB address space, are defined in the XA architecture. In any given XA implementation, fewer than all twenty-four address bits may actually be used, and there is provision for a small-memory mode which uses only 16-bit addresses; see Chapter 4.

Code and data memory may be on-chip or external, depending on the XA variant and the user implementation. Whether a specific region is on-chip or external does not, in general, affect access to the memory.

3.3.1 Bytes, Words, and Alignment

XA memory is addressed in units of *bytes*, where each byte consists of 8 bits. A *word* consists of two bytes, and the word storage order is "Little-Endian", that is, the less significant byte of word data is located at a lower memory address. See Figure 3.4.

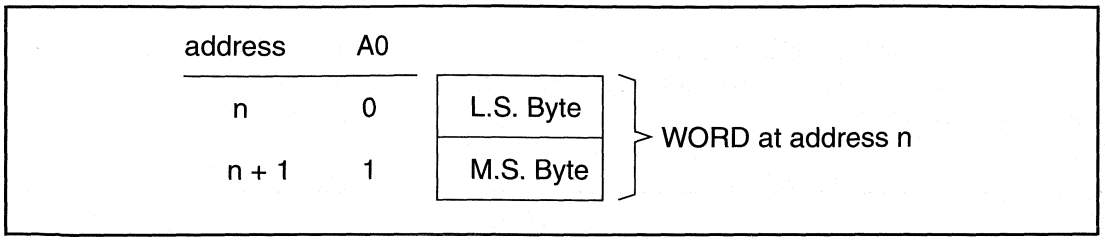


Figure 3.4 Memory byte order

Any word access must be aligned at an even address (Address bit A0=0). If an odd-aligned word access is attempted the word at the next-smallest even address will be accessed, that is, A0 will be set to 0.

The external XA memory spaces may be accessed in byte or word units but the hardware access method does not affect the even alignment restriction on word accesses.

3.4 Data Memory

The data memory space starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. As will be described below, the data memory space is segmented into 256 segments of 64K bytes each. *External Data Memory* starts at the first address following the highest *Internal Data Memory* location. In general, at least 512 bytes of Internal Data Memory, starting at location 0, will be provided in all XA implementations; however, there is no inherent minimum or maximum architectural limitation on Internal Data Memory.

The upper 16 segments of data memory (addresses F0:0000 through FF:FFFF hexadecimal) are reserved for special functions in XA derivatives. A similar range is reserved in the code memory space, see section 3.5.

3.4.1 Alignment in Data Memory

There are no data memory alignment restrictions except that placed on word accesses to all memory: Words must be fetched from even addresses. An attempt to fetch a word at an odd address will fetch a word from the preceding even address.

3.4.2 External and Internal Overlap

If External Data Memory is placed by external logic at addresses that overlaps Internal Data Memory, the Internal Data Memory generally takes precedence. The overlapped portion of the External memory may be accessed only by using a form of the MOVX instruction; see Chapter 6. The use of MOVX always forces external data memory fetch in XA. For non-overlapped portion of external data memory, no MOVX is required.

3.4.3 Use and Read/Write Access

Data memory is defined as read-write, and is intended to contain read/write data. It is logically impossible to execute instructions from XA Data Memory. It is possible, and a common practice, to add logic to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such a modified Harvard architecture, implemented with external logic, it is possible –but not recommended– to write self-modifying XA code. No such overlap is possible for internal data memory.

3.4.4 Data Memory Addressing

XA data memory addressing is optimized for the needs of embedded processing. Data memory in the XA is divided into 64K byte segments. This provides an intrinsic protection mechanism for multitasking applications and improves performance by requiring fewer address bits for localized accesses.

Addressing through Segment Registers

Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require full use of the XA 16 Mbyte address space. Two segment registers are defined in the XA architecture for use in accessing data memory, the Data Segment Register (**DS**), and the Extra Segment Register (**ES**). As user stacks are located in the segment specified by **DS**, it is probably most convenient to address user data structures through **ES**. Each pointer register, namely R0 through R6, is associated with one of the segment registers via the Segment Select (**SSEL**) register as illustrated in Figure 3.5.

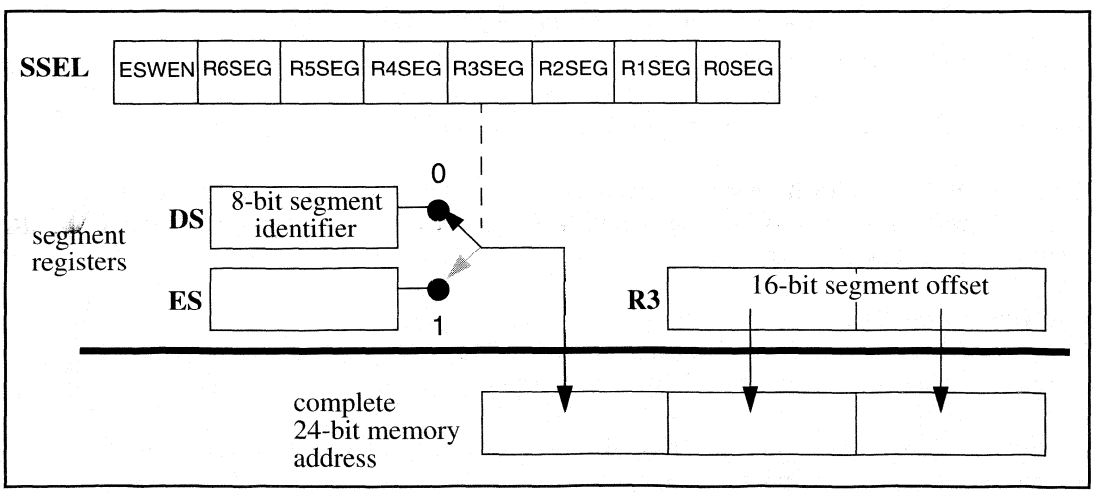


Figure 3.5 Address generation

A 0 in the SSEL bit corresponding to the pointer register selects DS (default on RESET) and 1 selects the ES. For example, when R3 contains a pointer value, the full 24 bit address is formed by concatenating DS or ES, as determined by the state of SSEL bit 3, as the most significant 8 bits. As a consequence of segmented addressing, the XA data memory space may be viewed as 256 segments of 64K bytes each (Figure 3.6).

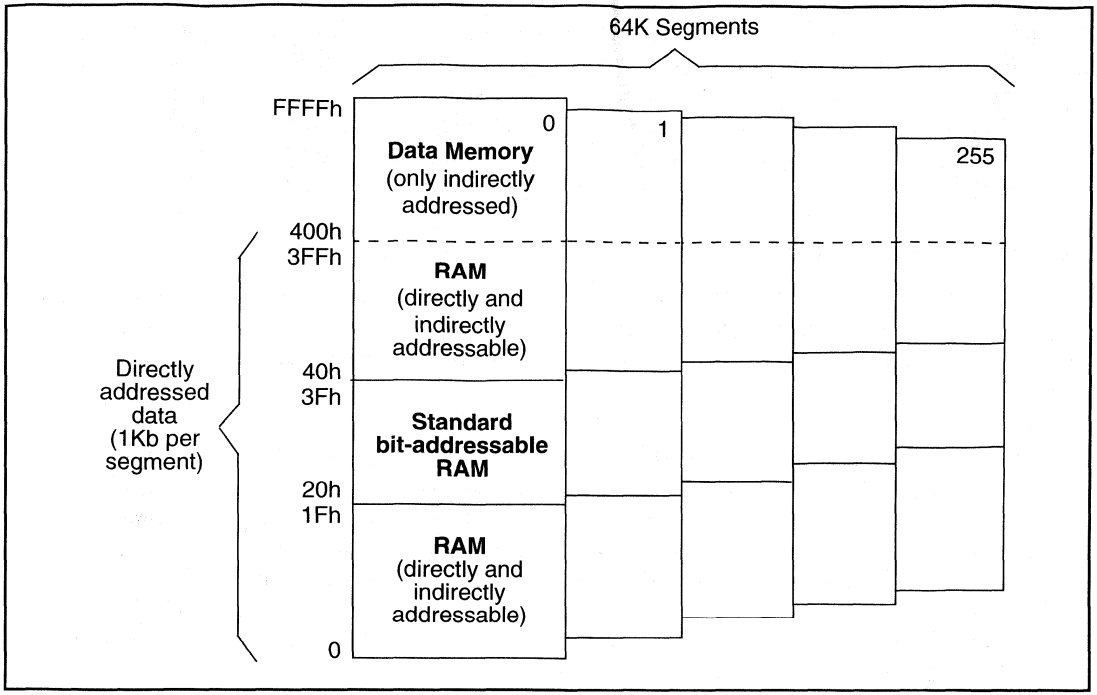


Figure 3.6 Data memory segmentation

If R7 (the stack pointer) is used as a normal indirect pointer, the data segment addressed will always be segment 0 in System Mode and the DS segment in User Mode. More information about the System and User modes may be found in sections 4 and 5.

The ESWEN (bit 7 of SSEL) can be programmed only in the System Mode to enable (1) or disable (0) write privileges to data segment via ES register in the User Mode. This bit defaults to the disabled (0) state after reset.

Addressing Modes

The XA provides flexible data addressing modes. Arithmetic, logic, and data movement instructions generally support the following data memory access:

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with a 16-bit pointer in a register.

Direct. The first 1K bytes of data in each segment may be accessed by an address contained within the instruction. *Indirect with offset.* A signed byte/word offset contained within the instruction is added to the contents of a pointer register, and the result is concatenated with the 8-bit segment register DS to produce a complete 24-bit address.

Indirect with auto-increment. Indirect addresses are formed as above and the pointer register contents are automatically incremented.

Bit-level. Bit-level addresses are absolute references to specific bits.

Data move instructions and some special purpose instructions also have additional data addressing modes as described in Chapter 6.

Indirect Addressing

The entire 16 MByte address space is accessible via register-indirect addressing with a segment register, as illustrated by Figure 3.7 (Note that for simplicity, this figure omits showing how the Extra Segment or Data Segment Register is chosen using **SSEL**.).

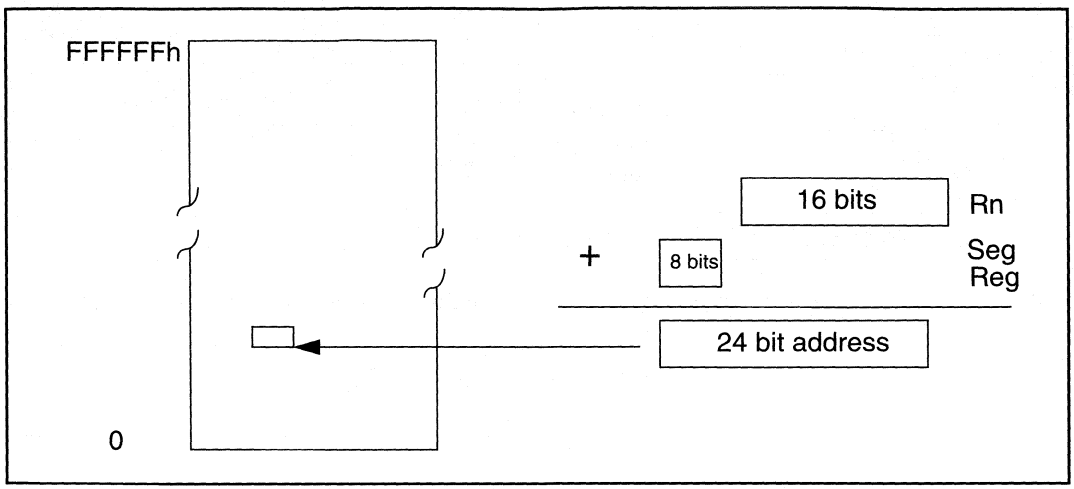


Figure 3.7 Indirect Access to 24 Bit Address Space

Indirect addressing with an offset is a variant of general indirect addressing in which an 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode gives access to data structures when a pointer register contains the starting address of the structure. It also supports stack-based parameter passing.

Indirect addressing with autoincrement is another variant of indirect addressing in which the pointer register contents are automatically incremented following the operation. When the operand is a byte, the increment is one; when the operand is a word, the increment is 2. Using indirect addressing with auto-increment provides a convenient method of traversing data structures smaller than 64K bytes. For data structures exceeding 64K bytes in length, the program code must explicitly adjust the segment register at page boundaries.

Address generation in these two modes of indirect addressing is illustrated inFigures 3.8 and 3.9. When using indirect addressing care is necessary to avoid accessing a word quantity at an odd address. This will result in an access using the next-lower even address, which is generally not desirable. Note that the indirect addressing with an offset will be successful in this case as long as the final, effective address is even. That is, both the base address and the offset may be odd.

Direct Addressing

The first 1K of each segment is directly addressable. Address generation for the direct address mode is summarized in Figure 3.10. Segment register DS is always used.

Direct data-reference instructions encode a maximum of 10 address bits, which are zero extended to sixteen bits and concatenated with DS to form an absolute 24 bit address. In all segments, direct addressing can be used to access any byte in the first 1K bytes of the segment.

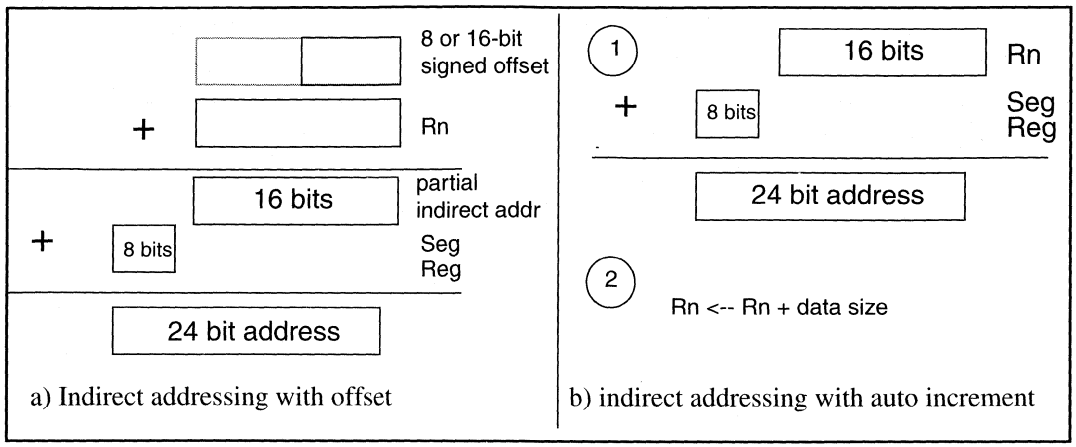


Figure 3.8 Indirect Addressing

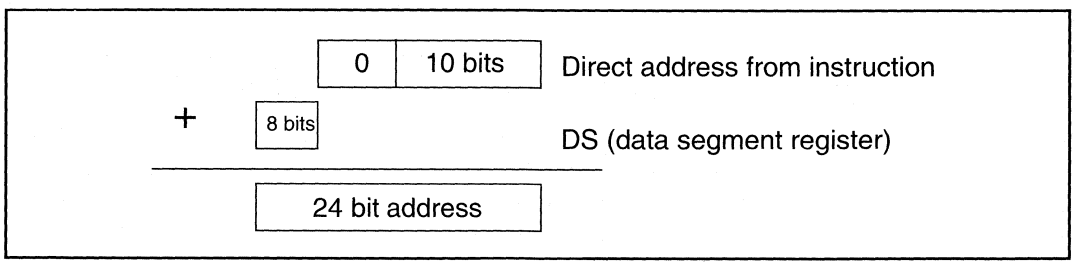


Figure 3.9 Direct address generation

SFR Addressing

A 1K portion of the direct address space, addresses 400h through 7FFh, is reserved for SFR addresses. The SFR address space uses a portion of the direct address space, but represents a completely distinct logical area that is not related to the data memory segmentation scheme. See section 3.6 for a complete description of SFR access.

Bit Addressing

Thirty-two bytes of each segment of data memory are also bit-addressable, starting at offset 20h in the segment addressed by the DS register. Address generation for bit addressing in the data memory space is shown in Figure 3.10. As described in chapter 6, bits are encoded in instructions as 10 bits. Figure 3.11 shows the bit addresses as they appear in memory .

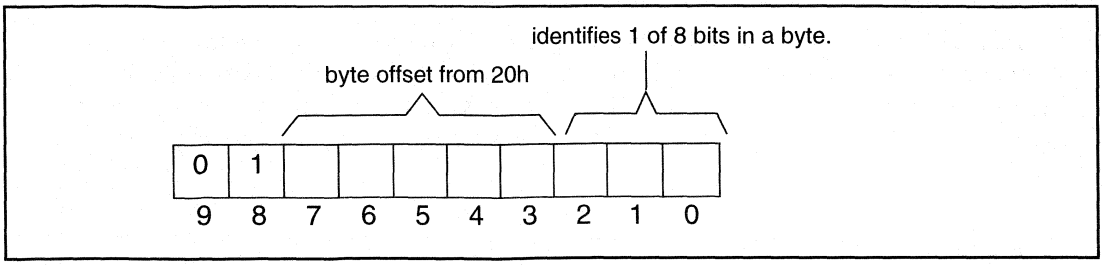


Figure 3.10 Bit address generation in direct memory space

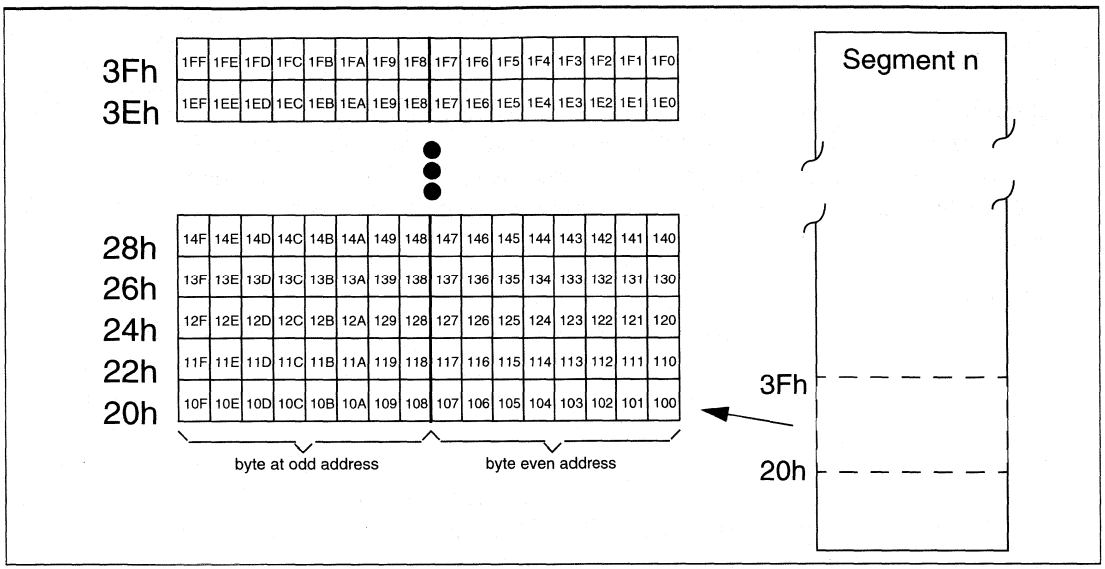


Figure 3.11 Direct memory bit addressing

3.5 Code Memory

Code memory starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. *External Code Memory* (off-chip) starts at the first address following the highest *Internal Code Memory* (on-chip) location, if any. If code memory is present on-chip, it always starts at location 0.

The upper sixteen 64K byte code pages (addresses F00000 through FFFFFF hexadecimal) are reserved for special functions in XA derivatives. The same address range is reserved in the data memory space, see section 3.4.

3.5.1 Alignment in Code Memory

As instructions are variable in length, from 1 to 6 bytes (see Chapter 6), instructions in code memory can be located at odd addresses. As described in Chapter 6, instruction branch targets, i.e., targets of jumps, calls, branches, traps, and interrupts must be aligned on an even address.

3.5.2 External and Internal Overlap

If External Code Memory is placed by external logic at locations that overlap Internal Code Memory, the Internal Code Memory takes precedence, and the overlapped portion of the External memory will in not be accessed. However, on XA implementations that provide an External Address ($\overline{EA}$) hardware input, setting EA low will cause external program memory to be used.

3.5.3 Access

Code memory is intended to contain executable XA instructions. The XA architecture supports storing constant data in Code Memory and provides special access modes for retrieving this information. Constant data is implicitly stored within the instruction of many data manipulation instructions when immediate operands are specified.

It is possible, and a common practice, to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such an architecture, implemented with external logic, code memory is logically read-only memory that is writable when accessed as external data memory. No such overlap is possible for internal code memory.

MOVC addressing in Code Memory

A special instruction, MOVC, is defined in the XA for accessing constant data (e.g lookup tables, string constants etc.) stored in code memory. There is a standard form of MOVC that reflects the native XA architecture, and there are two variations that reflect 80C51 compatibility; see Chapter 9 for details of 80C51 compatibility. The standard form of MOVC uses a 16-bit register value as a pointer, appended to either the top 8 bits of the Program Counter (PC) or the Code Segment register (CS) to form a 24-bit address, as shown in Figure 3.12. The source for the upper 8 address bits is determined by the setting of the segment selection bit (0 = PC and 1= CS) in the SSEL register that corresponds to the operand register.

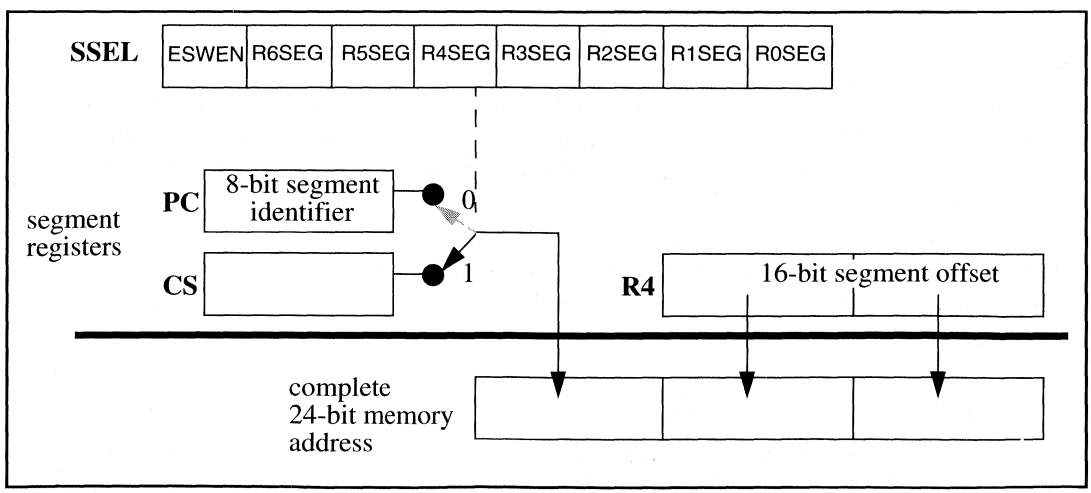


Figure 3.12 MOVC addressing in code memory

3.6 Special Function Registers (SFRs)

Special Function Registers (SFRs) provide a means for programs to access CPU control and status registers, peripheral devices, and I/O ports. The SFR mechanism provides a consistent mechanism for accessing standard portions of the XA core, peripheral functions added to the core within each XA derivative, and external devices as implemented in future derivatives.

Figure 3.13 highlights the core registers that are accessed as SFRs: **PCON**, **SCR**, **SSEL**, **PSWH**, **PSWL**, **CS**, **ES**, **DS**. Communication with these registers as well as on-chip peripheral devices is performed via the dedicated Special Function Register Bus (see section 8).

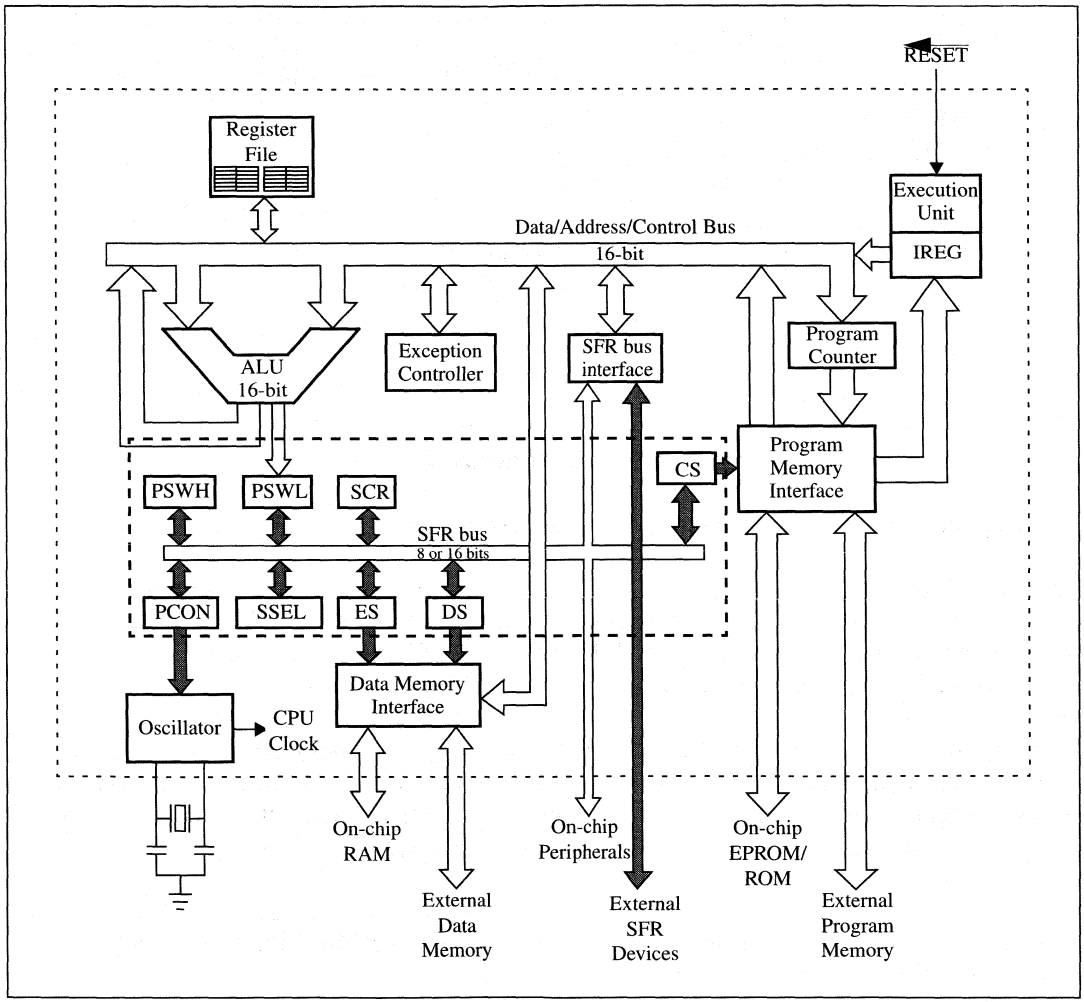


Figure 3.13 XA Core with SFRs highlighted

The SFR address space is 1K bytes (Figure 3.14). The first half of this space (400h through 5FFh) is dedicated to accessing core registers and on-chip peripherals outside the XA core. SFRs

assigned addresses in the range 400h through 43Fh are both byte and bit-addressable. The second half (600h through 7FFh) of the SFR space is reserved for providing access to off-chip SFRs. The off-chip sfr space is provided to allow faster access of off-chip memory mapped I/O devices without having to create a pointer for each access.

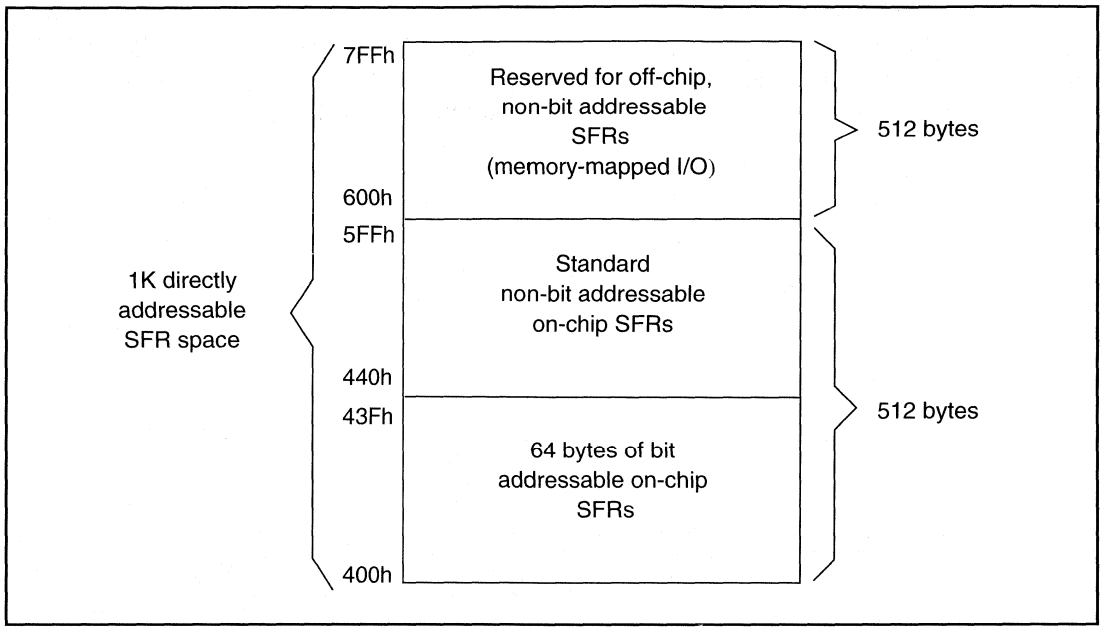


Figure 3.14 SFR address space

Following are some key points to remember when using SFRs:

SFRs should be symbolically addressed. Because SFR assignments may vary from derivative to derivative, it is important to always use symbolic references to SFRs. XA software development tools provide symbolic constants for all SFRs in the form of header/include files and the tools will be updated as new SFRs are added with each added XA derivative.

Verify that your application uses the right header/include files. Although baseline SFRs are likely to retain their addresses in future XA derivatives, this is not guaranteed. SFRs used for optional peripherals may well have different addresses on different derivatives, and the same address on one derivative may access a different peripheral SFR.

Any SFR may be accessed at any time without reference to a pointer or segment. SFR access is independent of any segment register, so SFRs are always accessible with the 10 bit address encoded in instructions accessing SFRs.

SFRs may not be accessed via indirect address. Any time indirection is used, data memory is accessed. If an SFR address is referenced as an indirect address, physical RAM at that address – if it exists– is accessed.

An SFR address is always contained entirely within an instruction. The SFR address is always encoded in the instruction providing the access, and there is no other way of addressing an SFR.

Details of access to external SFRs is determined by derivative implementation. Access to off-chip SFRs is a reserved feature not implemented in the baseline XA. Consult derivative product datasheets for details of external SFR access, e.g., timing.

3.7 Summary of Bit Addressing

Several sections of this chapter have described portions of the XA that are bit-addressable. There are a total of 1024 addressable bits distributed in the XA architecture, chosen to make important data structures immediately accessible via XA bit-processing instructions, specifically, all registers in the register file, R0 through R7 (and R8 through R15 if implemented); directly addressable RAM addresses 20h through 3Fh in the page currently specified by DS, and a portion of the on-chip SFRs. Figure 3.15 summarizes all the bit-addressable portions of the XA.5

bit space			overlaps bytes...		
start		end	type	start	end
0	◄►	0FFh	registers	R0	◄► R15
100h	◄►	1FFh	direct RAM	20h	◄► 3Fh
200h	◄►	3FFh	on-chip SFRs	400h	◄►43Fh

Figure 3.15 Bit addressing summary

4 CPU Organization

This chapter describes the Central Processing Unit (CPU) of the XA Core. The CPU contains all status and control logic for the XA architecture. The XA reset sequence and the system oscillator interface with the CPU, and power control is handled here. The CPU performs interrupt and exception handling. The XA CPU is equipped with special functions to support debugging.

4.1 Introduction

Figure 4.1 is a block diagram of the XA Core.

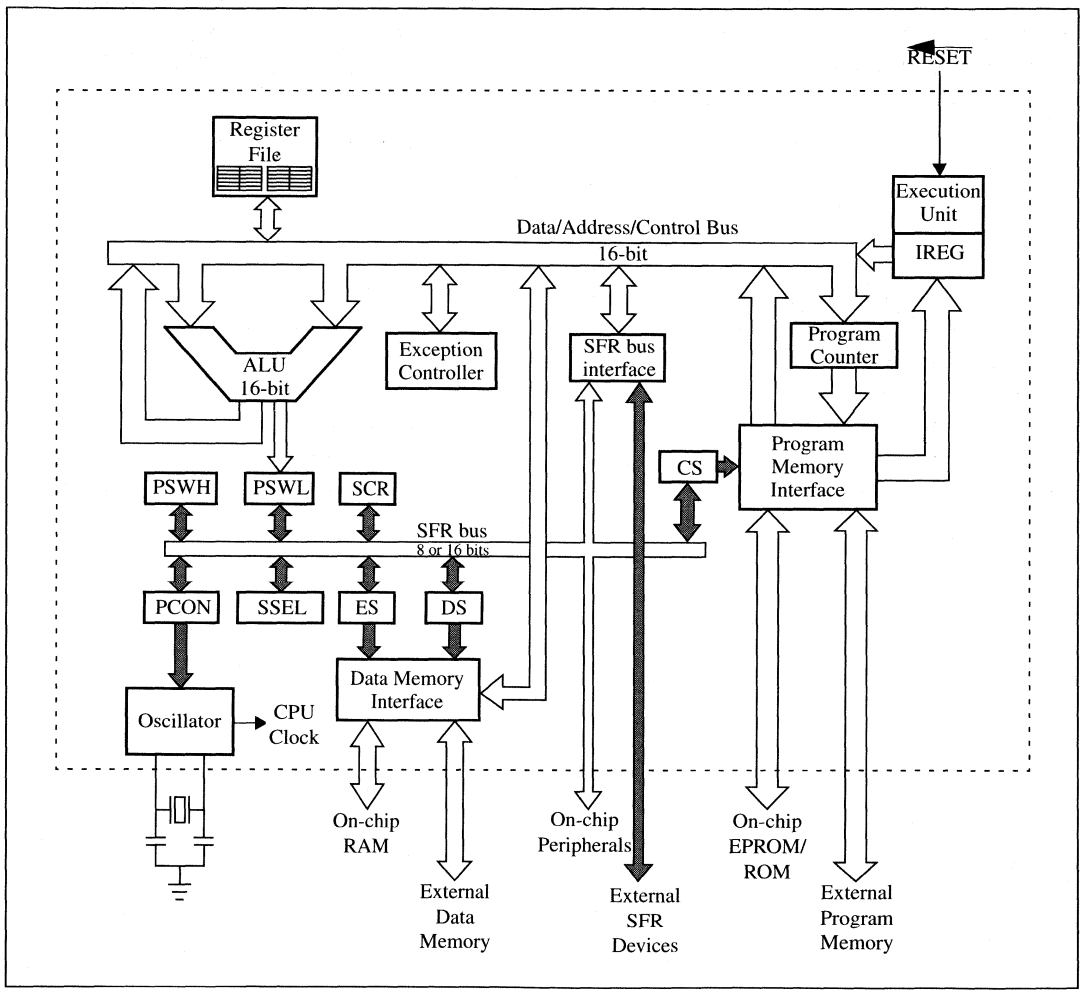


Figure 4.1 The XA Core

Here is an overview of core elements: The XA Core oscillator provides a basic system clock. Timing and control logic are initialized by an external reset signal; once initialized, this logic

provides internal and external timing for program and data memory access. This logic supervises loading the Program Counter and storing instructions fetched by the Program Memory Interface into the Instruction Register. The timing and control logic sequences data transfers to and from the Data Memory Interface. Under the same control, the ALU performs Arithmetic and Logical operations. The ALU stores status information in the low byte of the Program Status Word (**PSWL**). The on-board register file is used for intermediate storage and contains the current value of the Stack Pointer (**SP**). The high byte of the Program Status Word (**PSWH**) chooses between a privileged System Mode and a restricted User Mode; controls a Trace Mode used for single-step debugging, chooses the active register bank, and records the priority of the currently executing process. The System Configuration Register (**SCR**) is initialized to choose native XA mode execution or an 80C51 family compatibility mode. The Segment Selection Register (**SSL**) controls the use of the Code Segment (**CS**), Data Segment (**DS**), and the Extra Segment (**ES**) registers. The XA Core architecture supports interfaces to on- and off-chip RAM, ROM/ EPROM, and Special Function Registers (SFRs).

This chapter describes all these core elements in detail.

4.2 Program Status Word

The Program Status Word (**PSW**) is a two-byte SFR register that is a focal point of XA operations. The least significant byte contains the CPU status flags, which generally reflect the result of each XA instruction execution. This byte is readable and writable by programs running in both User and System modes.

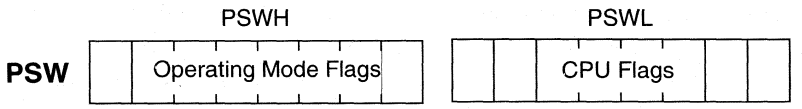


Figure 4.2 XA PSW

The most significant byte of **PSW** is written by programs to set important XA operating modes and parameters: system/user mode, trace mode, register bank select bits, and task execution priority. **PSWH** is readable by any process but only the register select bits may be modified by User mode code. All of the flags may be modified by code running in System Mode.

It should be noted that the XA includes a special SFR that mimics the original 80C51 PSW register. This register, called PSW51, allows complete compatibility with 80C51 code that manipulates bits in the PSW. See Chapter 9 for details of 80C51 compatibility.

4.2.1 CPU Status Flags

The PSW CPU flags (Figure 4.3) signify Carry, Auxiliary Carry, Overflow, Negative, and Zero. Some instructions affect all these flags, others only some of them, and a few XA instructions have no effect on the PSW status flags. In general, these flags are read by programs in order to make logical decisions about program flow. Chapter 6 describes comprehensively how CPU

Status Flags are affected by each instruction type. Consult reference pages in Chapter 6 for details about how individual instructions affect the PSW Status Flags.

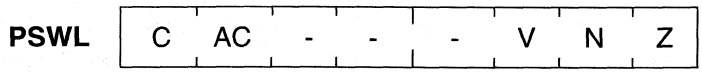


Figure 4.3 PSW CPU status flags

C, the Carry Flag, generally reflects the results of arithmetic and logical operations. It contains the carry out of the most significant bit of an arithmetic operation, if any, for the instructions ADD, ADDC, CMP, CJNE, DA, SUB, and SUBB.The carry flag is also used as an intermediate bit for shift and rotate instructions ASL, ASR, LSR, RLC, and RRC.

The multiply and divide instructions (MUL16, MULU8, MULU16, DIV16, DIV32, DIVU8, DIVU16, and DIVU32) unconditionally clear the carry flag.

AC, the auxiliary carry flag, is updated to reflect the result of arithmetic instructions ADD, ADDC, CMP, SUB, and SUBB with the carry out of the least significant nibble of the ALU. This flag is used primarily to support BCD arithmetic using the decimal adjust instruction (DA).

V is the overflow flag. It is set by an arithmetic overflow condition during signed arithmetic using instructions ADD, ADDC, CMP, NEG, SUB, and SUBB.

V is also set when the result of a divide instruction (DIV16, DIV32, DIVU8, DIVU16, DIVU32) exceeds the size of the specified destination register and when a divide-by-zero has occurred. For multiply instructions (MUL16, MULU8, MULU16) this flag is set when the result of a multiply instruction exceeds the source operand size. In this case "overflow" provides an indication to the program that the result is a larger data type than the source, such as a long integer product resulting from the multiply of two integers).

N reflects the twos complement sign (the high-order or "negative" bit) of the result of arithmetic operations and the value transferred by data moves. This flag is unaffected by PUSH, POP, SEXT, LEA, and XCH instructions.

Z ("zero") reflects the value of the result of arithmetic operations and the value transferred by data moves. This flag is set if the result or value is zero, otherwise it is cleared. The flag is unaffected by PUSH, POP, SEXT, LEA, and XCH instructions.

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.2.2 Operating Mode Flags

The PSW operating mode flags (Figure 4.4) set several aspects of the XA operating mode. All of the flags in the upper byte of the PSW (PSWH) except the bits RS1 and RS0 may be modified only by code running in system mode.

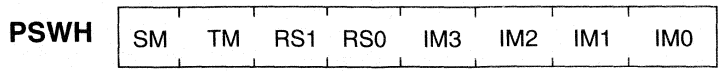


Figure 4.4 PSW operating mode flags

The System Mode bit, **SM**, when asserted, allows the currently running program full System Mode access to all XA registers, instructions, and memories. (For example, most of PSWH can only be modified when **SM** is asserted.) When this bit is cleared, the XA is running in User Mode and some privileges are denied to the currently running program.

The Trace Mode bit, **TM**, when set to 1, enables the built-in XA debugging facilities described in section 4.9. When **TM** is cleared, the XA debugging features are disabled.

The bits **RS1** and **RS0** identify one of the four banks of word registers R0 through R3 as the active register set. The other three banks are not accessible as registers (but also see the Compatibility Mode description in the System Configuration Register section).

The 4 bits **IM3** through **IM0** (Interrupt Mask bits) identify the execution priority of the current executing program. The event interrupt controller compares the setting of the IM bits to the priority of any pending interrupts to decide whether to initiate an interrupt sequence. The value 0 in the IM bits indicates the lowest priority, or fully interruptible code. The value 15 (or F hexadecimal) indicates the highest priority, not interruptible by event interrupts. Note that priority 15 does not inhibit servicing of exception interrupts or NMI.

The value of the IM bits may be written only by code operating in the system mode. Their value may be read by interrupt handler code to implement software-based interrupt priorities. Note that simply writing a new value to the interrupt mask bits can sometimes cause what is called a priority inversion, that is, the currently executing code may have a lower priority than previously interrupted code. The Software Interrupt mechanism is included on some XA derivatives specifically to avoid priority inversion in complex systems. Refer to the section on Software Interrupts for details.

4.2.3 Program Writes to PSW

The bytes comprising the PSW, namely PSWH and PSWL, are accessible as SFRs, and there is a potential ambiguity when a write to the PSW is performed by an instruction whose execution also modifies one or more PSW bits. The XA resolves this by giving full precedence to explicit writes to the PSW.

For example, executing

```
MOV.b R0L,#81h
```

sets PSW bit **N** to 1, since the byte value transferred is a twos complement negative number. However, executing

```
MOV.b PSWL, #81h
```

will set PSW bits **C** and **Z** and leave bit **N** cleared, since the value explicitly written to PSW takes precedence.

This precedence rule suppresses *all* PSW flag updates. When a value is written to the PSW, for example when executing

```
OR.b PSWH, #30
```

the contents of PSWL are unaffected.

4.2.4 PSW Initialization

As described below, at XA reset, the initial PSW value is loaded from the reset vector located at program memory address 0. Philips recommends that the PSW initialization value in the reset vector sets **IM3** through **IM0** to all 1's so that XA initialization is marked as the highest priority process (and therefore cannot be interrupted except by an exception or NMI). At the conclusion of the initialization code, the execution priority is typically reduced, often to 0, to allow all other tasks to run. It is also recommended that the reset vector set the **SM** bit to 1, so that execution begins in System Mode.

4.3 System Configuration Register

The System Configuration Register (**SCR**), described in Figure 4.5, sets XA global operating mode. **SCR** is intended to be written once during system start-up and left alone thereafter. Four bits are currently defined:

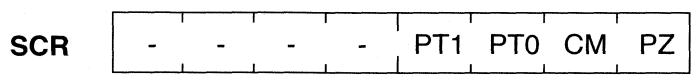


Figure 4.5 System Configuration Register (SCR)

PZ set to 0 (the default) puts the XA in the Large-Memory mode that uses full 24-bit XA addressing. When **PZ** = 1 the XA uses a small-memory "Page 0" mode that uses 16 bit addresses. The intent of Page 0 mode is to save stack space and improve interrupt latency in systems with less than 64K bytes of code and data memory. See the following sections for details.

CM chooses between standard "native" mode XA operation and 80C51 compatibility mode. When 80C51 compatibility mode is enabled, two things happen. First, the bottom 32 bytes of data memory in each data segment are replaced by the four banks of R0 through R3 from the register file. R0L of bank 0 will appear at data address 0, R0H of bank 0 will appear at data address 1, etc. Second, the use of R0 and R1 as indirect pointers is altered. To mimic 80C51 indirect addressing, indirect references to R0 use the byte R0L (zero extended to 16-bits) as the actual pointer value. References to R1 similarly use the byte R0H (zero extended to 16-bits) as the actual pointer value. Note that R0L and R0H on the XA are the same registers as R0 and R1 on the 80C51. No other XA features are altered or affected by compatibility mode. Operation of the XA with compatibility mode off (CM = 0) is reflected in descriptions found in the first 8 chapters of this User Guide. Operation with compatibility mode on (CM = 1) is discussed in Chapter 9.

PT1 and **PT0** select a submultiple of the oscillator clock as a Peripheral Timing clock source, in particular for timers but possibly for other peripherals in XA derivatives. Here are the values for these bits and the resulting clock frequency:

PT1	PT0	Peripheral Clock
0	0	oscillator/4
0	1	oscillator/16
1	0	oscillator/64
1	1	reserved

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.3.1 XA Large-Memory Model Description

When the default XA operation is chosen via the **SCR** (**CM** = 0 and **PZ** = 0), all addresses are maintained by the core as 24 bit values, providing a full 16 MByte address space. On a specific XA derivative, fewer than 24 bits may be available at the external bus interface. All 24 address bits are pushed on the stack during calls and interrupts and 24 bits are popped by RETs and RETIs.

4.3.2 XA Page 0 Memory Model Description

When XA Page 0 mode is chosen, only 16 address bits are maintained by the XA core. This operating mode supports XA applications for which a 64K byte address space is sufficient. The external memory interface port used for the upper 8 address bits, if present, is available for other uses. A single 16-bit word is pushed on the stack during calls and interrupts and 16 bits are, in turn popped by RETs and RETIs. Using Page 0 mode when only a small memory model is needed saves stack space and speeds up address PUSH and POP operations on the stack.

Switching into or out of Page 0 mode after the original initialization is not recommended. First, switching into Page 0 mode can only be done by code running on Page 0, since the code address will be truncated to 16-bits as soon as Page 0 mode takes effect. Instructions already in the XA pre-fetch queue would have been fetched prior to Page 0 mode taking effect. Any addresses that may have been pushed onto the stack previously also become invalid when Page 0 mode is changed. Thus Page 0 mode could not be changed while in an interrupt service routine, or any subroutine.

4.4 Reset

The term "reset" refers specifically to the hardware input required when power is first applied to the XA device, and generally to the sequence of initialization that follows a hardware reset, which may occur at any time. The term also refers to the effect of the RESET instruction (see Chapter 6); in addition, an overflowing Watchdog timer (if this peripheral is present) has an identical effect.

This section describes the XA reset sequence and its implications for user hardware and software.

4.4.1 Reset Sequence Overview

A specific hardware reset sequence must be initiated by external hardware when the XA device is powered-up, before execution of a program may begin. If a proper reset at power up is not done, the XA may fail wholly or in part. The XA reset sequence includes the following sequential components:

- Reset signal generated by external hardware
- Internal Reset Sequence occurs
- $\overline{\text{RST}}$ line goes high
- External bus width and memory configuration determined
- Reset exception interrupt generated
- Startup Code executed

Figure 4.6 illustrates this process.

4.4.2 Power-up Reset

This section describes the reset sequence for powering up an XA device.

The XA $\overline{\text{RST}}$ input must be held low for a minimum reset period after Vdd has been applied to the XA device and has stabilized within specifications. The minimum reset period for a typical system with a reasonably fast power supply ramp-up time is 10 milliseconds. This reset period provides sufficient time for the XA oscillator to start and stabilize and for the CPU to detect the reset condition. At this point, the CPU initiates an internal reset sequence. $\overline{\text{RST}}$ must continue to be low for a sufficient time for the internal reset sequence to complete.

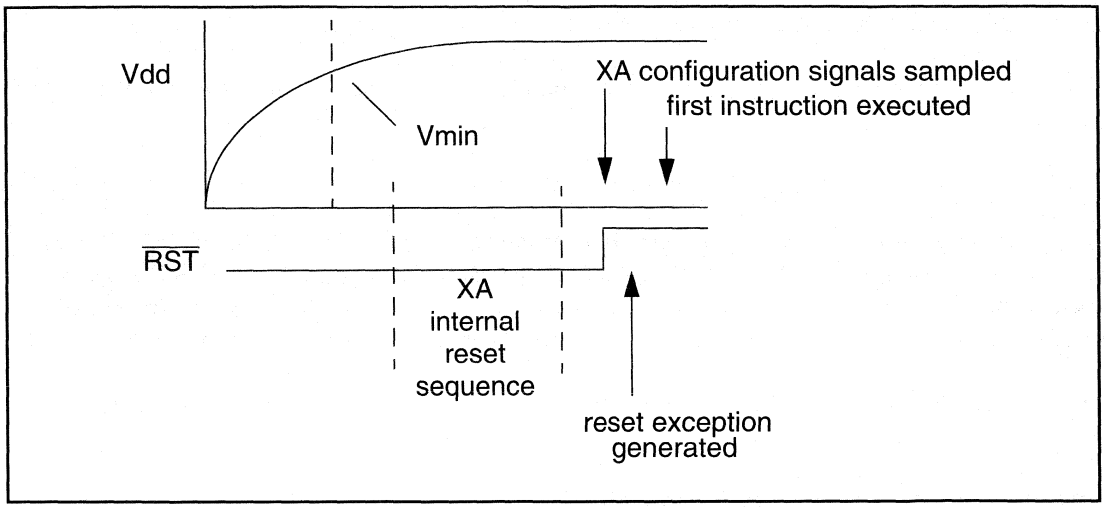


Figure 4.6 XA power-up sequence

4.4.3 Internal Reset Sequence

The XA internal reset sequence occurs after power-up or any time a sufficiently long reset pulse is applied to the $\overline{\text{RST}}$ input while the XA is operating. This sequence requires a minimum of a 10 microseconds (or 10 clocks, whichever is greater) to complete, and $\overline{\text{RST}}$ must remain low for at least this long.

The internal reset sequence does the following:

- Writes a 00 to most core and many peripheral SFRs. Other values are written to some peripheral SFRs. Consult the data sheet of a specific device for details.
- Sets **CS**, **DS**, and **ES** to 0.
- Sets **SSEL** = 0, i.e., sets all accesses through DS.
- Sets all registers in the Register File to 0.
- Sets the user and the system stack pointers (**USP** and **SSP**) to 0100h.
- Clears SCR bit **PZ**, i.e., 24-bit memory addresses will be used by default.
- Clears SCR bit **CM**, i.e., starts execution in XA Native Mode.
- Clears IE bit **EA**, disabling all maskable interrupts.

Note that the internal reset sequence does not initialize internal or external RAM. Note also that the contents of **PSW** at this point is not important, as it will immediately be replaced as described further below.

The effect of the internal reset sequence on components outside the XA core depends on the peripheral complement and configuration of the specific XA derivative. In general, the internal reset sequence has the following effects:

- Sets all port pins to inputs (quasi-bidirectional output configuration with port value = FF hex)
- Clears most SFRs to 0
- Initializes most other SFRs to appropriate non-zero values

Note that serial port buffers, PCA capture registers, and WatchDog feed registers (if present) are unaffected. Consult the XA derivative data sheet for more information.

After the XA internal reset sequence has been completed, the device is quiescent until the $\overline{RST}$ line goes high.

4.4.4 XA Configuration at Reset

As the $\overline{RST}$ line goes high, the value on two input pins is sampled to determine the XA memory and bus configuration. The $\overline{EA}$ and BUSW pins (if present on a specific XA derivative) have special function during the reset sequence, to allow external hardware to determine the use of internal or external program memory, and to select the default external bus width.

Immediately after the $\overline{RST}$ line goes high, the CPU triggers a reset exception interrupt, as described in the next section.

Selecting Internal or External Program Memory

The XA is capable of reading instructions from internal or external memory, both of which may be present. The XA $\overline{EA}$ input pin determines whether internal or external program memory will be used. The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse. If $\overline{EA}$ = 0, the XA will operate out of external program memory, otherwise it will use internal code memory. The selection of external or internal code memory is fixed until the next time $\overline{RST}$ is asserted and released; until then all code fetches will access the selected code memory.

The XA cannot detect inconsistencies between the setting detected on the $\overline{EA}$ input and the hardware memory configuration. For example, setting $\overline{EA}$ = 1 on a ROMless XA variant will cause the XA to attempt to execute internal code memory, which is undefined on a ROMless device, typically resulting in a system failure.

Selecting External Bus Width

The XA is capable of accessing an 8 or 16 bit external data bus. The BUSW pin tells the XA the external data bus configuration. BUSW=0 selects an 8-bit bus and BUSW=1 selects an 16-bit bus. On power-up, the XA defaults to the 16-bit bus (due to an on-chip weak pull-up on BUSW). The BUSW pin is sampled on the rising edge of the $\overline{RST}$ pulse. If BUSW is low, the XA operates its external bus interface in 8 bit mode, otherwise, the XA uses 16 bit bus operation. The bus width may also be set under software control on derivatives equipped with the **BCR** ("Bus Configuration Register") SFR.

After $\overline{RST}$ is released, the BUSW pin may be used an alternate function on some XA derivatives. Consult derivative data sheets for exact pinouts and details of how pins such as these may be shared to keep package size small.

4.4.5 The Reset Exception Interrupt

Immediately after the $\overline{RST}$ line goes high, the CPU generates a Reset Exception Interrupt. As a result, the initial PSW and address of the first instruction (the "start-up code") is fetched from the reset vector in code memory at location 0. Here's an example in generalized assembler format of the setup for the Reset Exception:

```
            code_seg                ; establish code segment
            org  0h                 ; start at address 0

; reset_vector
            dw   initial_PSW        ; define a word constant
            dw   startup_code       ; define a word constant

            org  120h               ; move to address 120h
                                    ; (above vector table)

startup_code:
            ...                     ; put startup code here
```

The initial value of **PSWL** set in the Reset Vector is generally of no special system-wide importance and may be set to zero or some other value to meet special needs of the XA application. The initial **PSWH** value sets the stage for system software initialization and its value requires more attention. Here's an example set of declarations that create the recommended initial value of **PSWH**:

```
system_mode     equ  8000h
max_priority    equ  0F00h
initial_PSW     equ  system_mode + max_priority
```

It is generally appropriate to initialize the XA in System Mode so that the start-up code has unrestricted access to the entire architecture. This is done by using a initial value that sets the PSWH bit **SM**.

Philips recommends initializing the execution priority of the start-up code to the highest possible value of 15 (that is, IM0 through IM3 to all ones) so that the start-up code is recognizable as the highest priority process. As described above, the hardware initialization sequence turns off all possible interrupts, so the only potential interrupting process would arise from a non-maskable interrupt (NMI). It is generally a good idea to prevent NMI generation with a hardware lock-out until XA start-up procedures are completed.

The **PSWH** initialization value given in this example sets System Mode (**SM**), selects register bank 0 (any register bank could be used) and clears **TM** so that Trace Mode is inactive.

4.4.6 Startup Code

Philips recommends that the first instruction of start-up code set the value of the System Configuration Register (**SCR**), described in section 4.3, to reflect the system architecture.

The next recommended step is explicitly initializing the stack pointers. The default values (section 4.7) are usually insufficient for application needs.

The start-up code sequence may be concluded by a simple branch or jump to application code. A RETI may not be used at the conclusion of a Reset Exception Interrupt handler (which causes the start-up code to run) because a reset initializes the SP and does not leave an interrupt stack frame.

4.4.7 Reset Interactions with XA Subsystems

The following describes how the reset process interacts with some key subsystems:

- Trace Exception. The trace exception is aborted by an external reset; see section 4.9.
- WatchDog. In XA derivatives equipped with a WatchDog timer feature, an internal reset will be asserted for a derivative-defined number of clocks.
- Resets while in Idle Mode or during normal code execution. Since the XA oscillator is running in Idle Mode, the $\overline{\text{RST}}$ input must be kept low for only 10 microseconds (or 10 clocks, whichever is greater) to achieve a complete reset.
- Resets while in Power-Down Mode. The XA oscillator is stopped in Power-Down mode, so the $\overline{\text{RST}}$ input must be low for at least 10 milliseconds. An exception to this is when an external oscillator is used and the XA is in Power-Down mode. In this case, if the external oscillator is running, a reset during Power-Down mode may be the same as a reset in Idle Mode.

4.4.8 An External Reset Circuit

The $\overline{\text{RST}}$ pin is a high-impedance Schmitt trigger input pin. For applications that have no special start-up requirements, it is practical to generate a reset period known to be much longer than that required by the power supply rise time and by the XA under all foreseeable conditions. One simple way to build a reset circuit is illustrated in Figure 4.7.

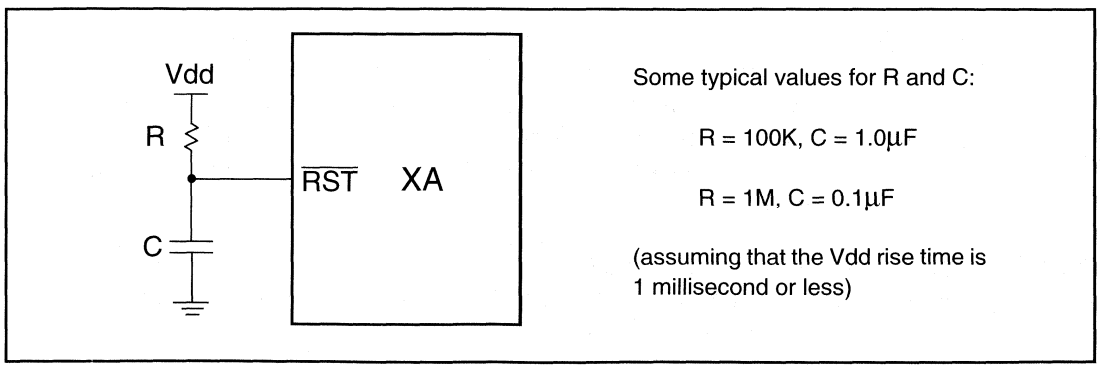


Figure 4.7 An external reset circuit

4.5 Oscillator

The XA contains an on-chip oscillator which may be used as the clock source for the XA CPU, or an external clock source may be used. A quartz crystal or ceramic resonator may be connected as shown in Figure 4.8a to use the internal oscillator. To use an external clock, connect the source to pin XTAL1 and leave pin XTAL2 open, as shown in Figure 4.8b.

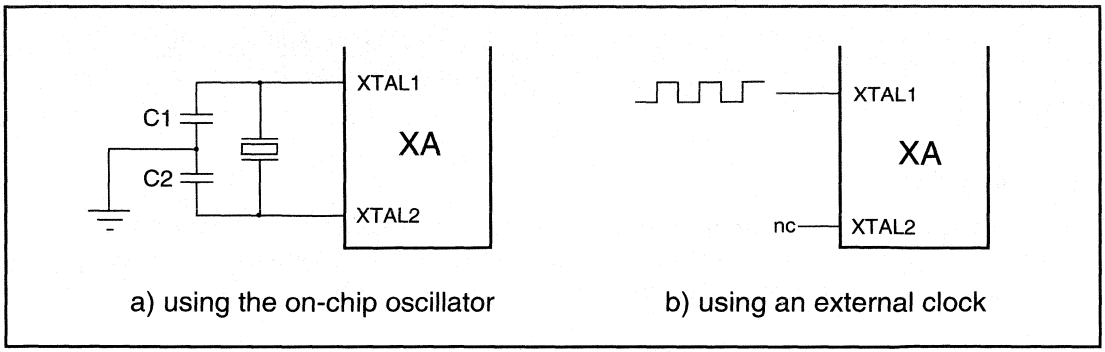


Figure 4.8 XA clock sources

The on-chip oscillator of the XA consists of a single stage linear inverter intended for use as a positive reactance oscillator. In this application, the crystal is operated in its fundamental response mode as an inductive reactance in parallel resonance with capacitance external to the crystal.

A quartz crystal or ceramic resonator is connected between the XTAL1 and XTAL2 pins, capacitors ar connected from both pins to ground. In the case of a quartz crystal, a parallel resonant crystal must be used in order to obtain reliable operation. The capacitor values used in the oscillator circuit should normally be those recommended by the crystal or resonator manufacturer. For crystals, the values may generally be from 18 to 24 pF for frequencies above 25 MHz and 28 to 34 pF for lower frequencies. Too large or too small capacitor values may prevent oscillator start-up or adversely affect oscillator start-up time.

4.6 Power Control

The XA CPU implements two modes of reduced power consumption: Idle mode, for moderate power savings, and Power-Down mode. Power-Down reduces XA consumption to a bare minimum. These modes are initiated by writing SFR **PCON**, as illustrated in Figure 4.9.

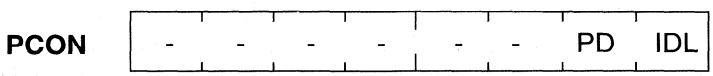


Figure 4.9 PCON

Idle Mode is activated by setting the PCON bit **IDL**. This stops CPU execution while leaving the oscillator and some peripherals running.

Power-Down mode is activated set by setting the PCON bit **PD**. This shuts down the XA entirely, stopping the oscillator.

The reset values of **IDL** and **PD** are 0. If a 1 is written to both bits simultaneously, **PD** takes precedence and the XA goes into Power-Down mode.

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.6.1 Idle Mode

Idle mode stops program execution while leaving the oscillator and selected peripherals active. This greatly reduces XA power consumption. Those peripheral functions may cause interrupts (if the interrupt is enabled) that will cause the processor to resume execution where it was stopped.

In the Idle mode, the port pins retains their logical states from their pre-idle mode. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). ALE and $\overline{\text{PSEN}}$ are held in their respective non-asserted states. When Idle is exited normally (via an active interrupt), port values and configurations will remain in their original state.

4.6.2 Power-Down Mode

Power-Down mode stops program execution and shuts down the on-chip oscillator. This stops all XA activity. The contents of internal registers, SFRs and internal RAM are preserved. Further power savings may be gained by reducing XA Vdd to the RAM retention voltage in Power Down mode; see the device data sheet for the applicable Vdd value. The processor may be re-activated by the assertion of $\overline{\text{RST}}$ or by assertion of one of an external interrupt, if enabled. When the processor is re-activated, the oscillator will be restarted and program execution will resume where it left off.

In Power-Down mode, the ALE and $\overline{\text{PSEN}}$ outputs are held in their respective non-asserted states. The port pins output the values held by their respective SFRs. Thus, port pins that are not configured to be part of an external bus retain their state. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). If Power-Down mode is exited via Reset, all port values and configurations will be set to the default Reset state.

In order to use an external interrupt to re-activate the XA while in Power-Down mode, the external interrupt must be enabled and be configured to level sensitive mode. When Power-Down mode is exited via an external interrupt, port values and configurations will remain in their original state. Since the XA oscillator is stopped when the XA leaves Power-Down mode via an interrupt, time must be allowed for the oscillator to re-start. Rather than force the external logic asserting the interrupt to remain active during the oscillator start-up time, the XA implements its own timer to insure proper wake-up. This timer counts 9,892 oscillator clocks before allowing the XA to resume program execution, thus insuring that the oscillator is running and stable at

that time. Once the oscillator counter times out, the XA will execute the interrupt that woke it up, if that interrupt is of a higher priority than the currently executing code.

Note that if an external oscillator is used, power supply current reduction in the Power-Down mode is reduced from what would be obtained when using the XA on-chip oscillator. In this case, full power savings may be gained by turning off the external clock source or stopping it from reaching the XTAL1 pin of the XA. If the clock source may be turned off, it may be advantageous to use Idle mode rather than Power-Down mode, to allow more ways of terminating the power reduction mode and to avoid the 9,892 clock waiting period for exiting Power-Down mode.

4.7 XA Stacks

The XA stacks are word-aligned LIFO data structures that grow downward in data memory, from high to low address. This and some other details of the XA stack implementation differ from 80C51 stack operation. Refer to the chapter on 8051 compatibility for a detailed discussion of this topic.

The XA implements two distinct stacks, one for User Mode and one for System Mode. The User Stack may be placed anywhere in data memory, while the System Stack must be located in the first 64K bytes, i.e., segment 0.

4.7.1 The Stack Pointers

The XA has two stacks, the system stack and the user stack. Each stack has an associated stack pointer, the System Stack Pointer (SSP) and the User Stack Pointer (USP), respectively. Only one of these stacks is active at a given time. The current stack pointer at any instant (which may be the SSP or the USP) appears as word register SP (R7) in the register file; the other stack pointer will not be visible. The value of the PSW bit **SM** determines which stack is active (and whose stack pointer therefore appears as R7). In User Mode (**SM** = 0), SP (R7) contains the User Stack Pointer. In System Mode (**SM** =1), SP (R7) contains the System Stack Pointer. The XA automatically switches SSP and USP values when the operating mode is changed. Note that the terms "USP" and "SSP" are logical terms, denoting the value of SP (R7) in each mode.

Segments and Protection

The User stack is always addressed relative to the current data segment (DS) value. This is consistent with each user task being associated with a specific data segment. Moreover, code running in User Mode cannot modify **DS**, so there is no possibility of changing the segment in which the stack resides within the User context. The System Stack must always be located in segment 0, that is, the first 64K of data memory.

4.7.2 PUSH and POP

The PUSH operation is illustrated by Figure 4.10. The stack pointer always points to an existing data item at the top of the stack, and is decremented by 2 prior to writing data.

The POP operation copies the data at the top of the stack and then adds two to the stack pointer, as follows shown in Figure 4.11.

All stack pushes and pops occur in word multiples. If a byte quantity is pushed on the stack it is stored as the least significant byte of a word and the high byte is left unwritten; see Figure 4.12. A POP to a byte register removes a word from the stack and the byte register receives the least significant 8 bits of the word, as shown in Figure 4.13.

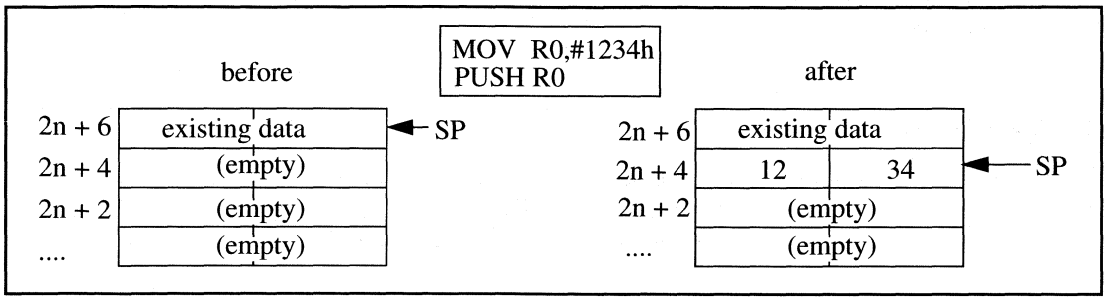


Figure 4.10 PUSH operation

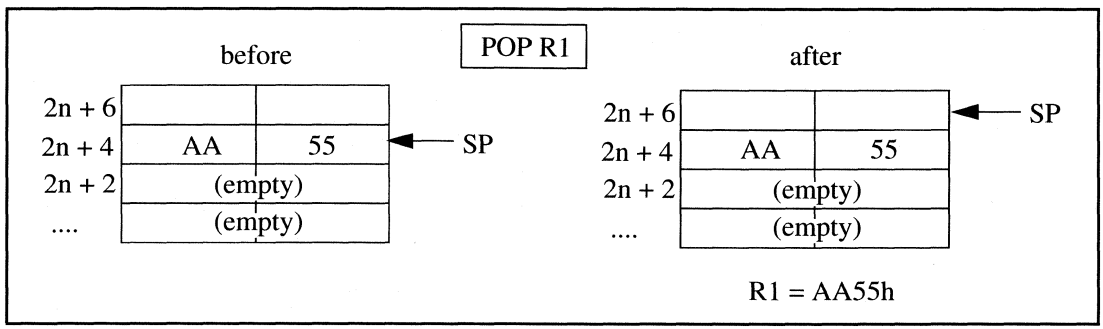


Figure 4.11 POP operation

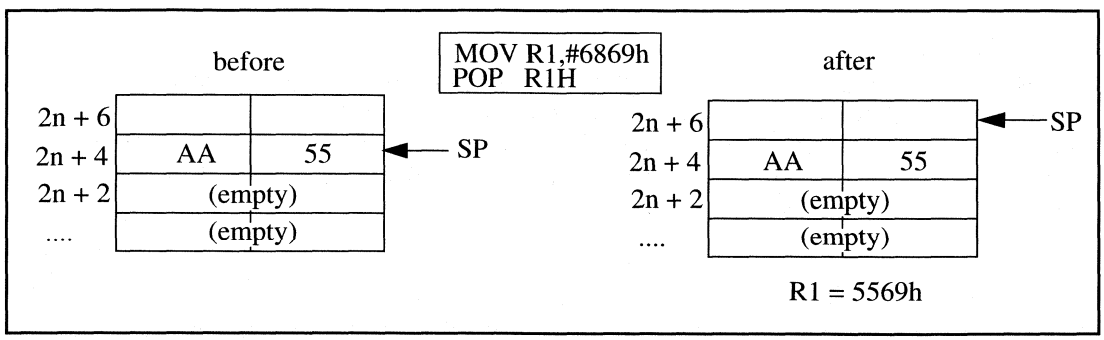


Figure 4.12 POP a byte

The stack should always be word-aligned. If the SP (R7) is modified to an odd value, the offending LSB of the stack pointer is ignored and the word at the next-lower even address is accessed.

Note that neither PUSH or POP operations have any effect on the PSW flags.

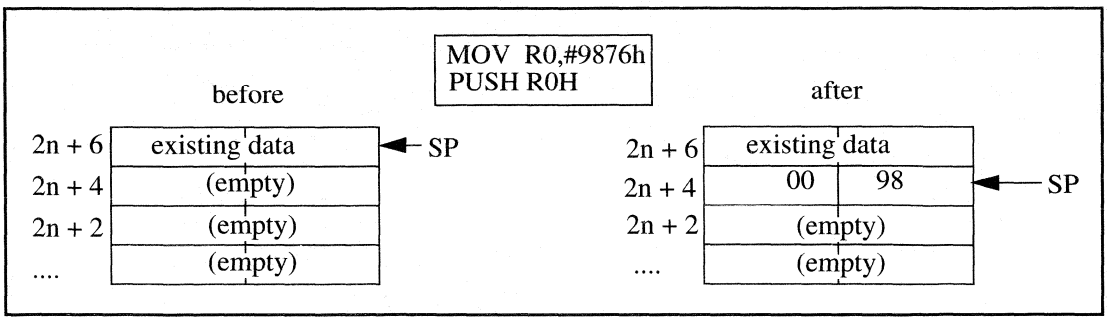


Figure 4.13 PUSH a byte

4.7.3 Stack-Based Addressing

Stack-based data addressing is fully supported by the XA. R0 through R7 may be used in all indexed address modes; the stack pointer in R7 is equally valid as an index.
Figure 4.14 illustrates an example of stack-based addressing. The segment used for stack relative addressing is always the same as for other stack operations (Segment 0 for System mode code and DS for User mode code).

Note that the precautions mentioned in section 3.3.4 apply here: when referencing a word quantity, the final (effective) address must be even, otherwise incorrect data will be accessed. This topic is discussed further in the section Stack Pointer Misalignment.

4.7.4 Stack Errors

Special attention is required to avoid problems due to stack overflow, stack underflow, and stack pointer misalignment

Stack Overflow

Stack overflow occurs when too many items are pushed, either explicitly or as the result of interrupts. As items are pushed on to the stack, it may grow downward past the memory allocated to it. It is not always possible for programs to detect stack overflow, so the XA triggers a Stack Overflow Exception Interrupt whenever the value of the *current* stack pointer (SSP or USP) decrements from 80h to 7Eh (simply setting SP to a value lower than 80h would NOT cause a stack overflow). This value was chosen so that stack space sufficient to handle a stack overflow exception interrupt is always guaranteed, as follows:

The 80h limit leaves 64 bytes available for stack overflow processing. A worst case might be occurs when the Stack Pointer is at 80h and a program executes an 8 word push; this generates a stack overflow. If an NMI occurs at the same time, 3 additional words are pushed. The balance

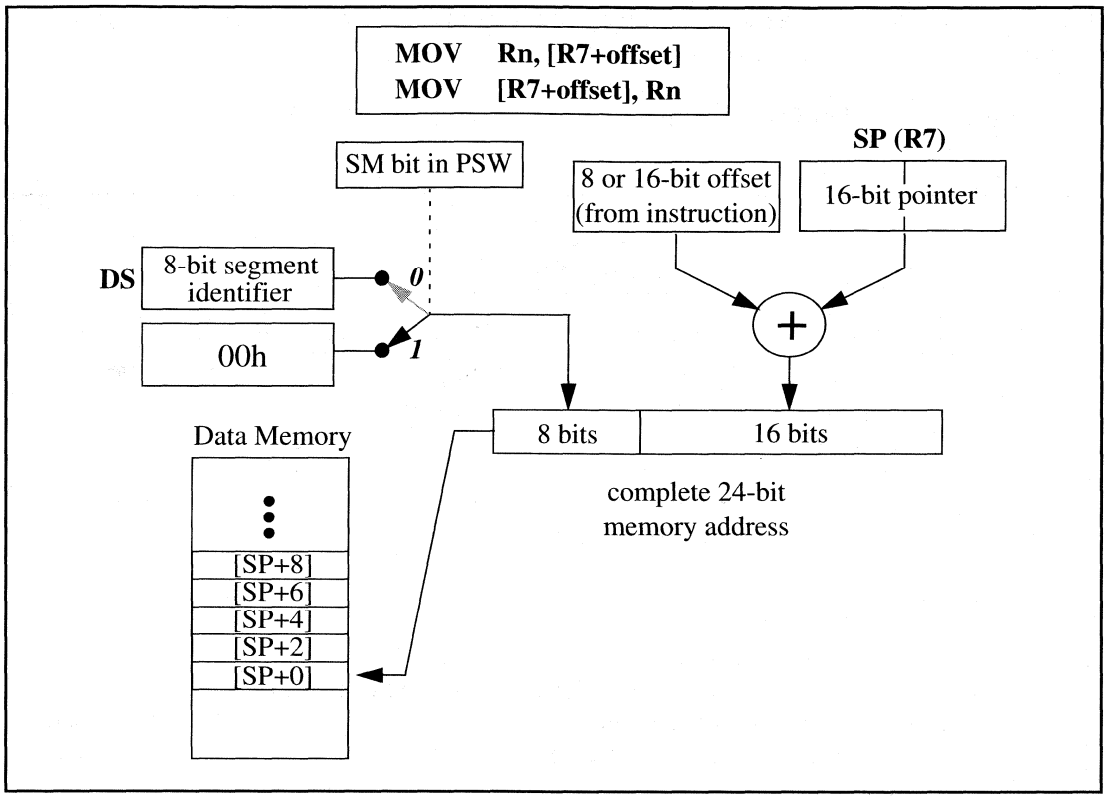


Figure 4.14 Stack-based addressing

of the 64 bytes on the stack is available for handler processing, which should carefully limit further use of the stack.

Stack Underflow

Stack underflow occurs when too many items are popped and the stack pointer value becomes greater than its initial value, i.e., the stack top. The XA does not support stack underflow detection.

Stack Pointer Misalignment

Pointer misalignment occurs when a pointer contains an odd value and is used by an instruction to access a word value in memory. The same situation could occur if some program action forced the stack pointer to an odd value. In these cases, the XA ignores the bottom bit of the pointer and continues with a word memory access.

4.7.5 Stack Initialization

At power-on reset, *both* USP and SSP in all XA derivatives are initialized to 100h. Since SP is pre-decremented, the first PUSH operation will store a word at location FEh and the stack will grow downwards from there.

These default stack pointer start-up values overlap the System and User stacks and are applicable only when one of these stacks will never be used.

Since the System stack is used for all exception and interrupt processing, this may not be appropriate in all XA applications. The startup code should normally set new and different values of both USP and SSP.

4.8 XA Interrupts

The XA architecture defines four kinds of interrupts. These are listed below in order of intrinsic priority:

- Exception Interrupts
- Event Interrupts
- Software Interrupts
- Trap Interrupts

Exception interrupts reflect system events of overriding importance. Examples are stack overflow, divide-by-zero, and Non-Maskable Interrupt. Exceptions are always processed immediately as they occur, regardless of the priority of currently executing code.

Event interrupts reflect less critical hardware events, such as a UART needing service or a timer overflow. Event interrupts may be associated with some on-chip device or an external interrupt input. Event interrupts are processed only when their priority is higher than that of currently executing code. Event interrupt priorities are settable by software.

Software interrupts are an extension of event interrupts, but are caused by software setting a request bit in an SFR. Software interrupts are also processed only when their priority is higher than that of currently executing code. Software interrupt priorities are fixed at levels from 1 through 7.

Trap interrupts are processed as part of the execution of a TRAP instruction. So, the interrupt vector is always taken when the instruction is executed.

All forms of interrupts trigger the same sequence: First, a *stack frame* containing the address of the next instruction and then the current value of the PSW is pushed on the System Stack. A vector containing a new PSW value and a new execution address is fetched from code memory. The new PSW value entirely replaces the old, and execution continues at the new address, i.e., at the specific interrupt handler.

The new PSW value may include a new setting of PSW bit **SM**, allowing handler routines to be executed in System or User mode, and a new value of PSW bits **IM3** through **IM0**, reflecting the execution *priority* of the new task. These capabilities are basic to multi-tasking support on the XA. See Chapter 5 for more details.

Returns from all interrupts should in most cases be accomplished by the RETI instruction, which pops the System Stack and continues execution with the restored PSW context. Since RETI executed while in User Mode will result in an exception trap, as described further below, interrupt service routines will normally be executed in System Mode.

The XA architecture contains sophisticated mechanisms for deciding when and if an interrupt sequence actually occurs. As described below, Exception Interrupts are always serviced as soon as they are triggered. Event Interrupts are deferred until their execution priority is higher than that of the currently executing code. For both exception and event interrupts, there is a systematic way of handling multiple simultaneous interrupts. Software and trap interrupts occur only when program instructions generating them are executed so there is no need for conflict resolution.

The Non-Maskable Interrupt requires special consideration. It is generated outside the XA core, and in that respect is an event interrupt. However, it shares many characteristics of exception interrupts, since it is not maskable. Note that NMI, while part of the XA CPU core, may not always be connected to a pin or other event source on all XA derivatives.

4.8.1 Interrupt Type Detailed Descriptions

This section describes the four kinds of interrupts in detail.

Exception Interrupts

Exception interrupts reflect events of overriding importance and are always serviced when they occur. Exceptions currently defined in the XA core include: Reset, Breakpoint, Divide-by-0, Stack overflow, Return from Interrupt (RETI) executed in User Mode, and Trace. Nine additional exception interrupts are reserved. NMI is listed in the table of exception interrupts (Table 4.1) below because NMI is handled by the XA core in same manner as exceptions, and factors into the precedence order of exception processing.

Since exception interrupts are by definition not maskable, they must always be serviced immediately regardless of the priority level of currently executing code, as defined by the IM bits in the PSW. In the unusual case that more than one exception is triggered at the same time, there is a hard-wired *service precedence* ranking. This determines which exception vector is taken first if multiple exceptions occur. In these cases, the exception vector taken *last* may be considered the highest priority, since its code will execute first. Of course, being non-maskable, any exception occurring during execution of the ISR for another exception will still be serviced immediately.

Programmers should be aware of the following when writing exception handlers:

1. Since another exception could interrupt a stack overflow exception handler routine, care should be taken in all exception handler code to minimize the possibility of a destructive stack overflow. Remember that stack overflow exceptions only occur once as the stack crosses the bottom address limit, 80h.

2. The breakpoint (caused by execution of the BKPT instruction, or a hardware breakpoint in an emulation system) and Trace exceptions are intended to be mutually exclusive. In both cases, the handler code will want to know the address in user code where the exception occurred. If a breakpoint occurs during trace mode, or if trace mode is activated during execution of the breakpoint handler code, one of the handlers will see a return address on the stack that points within the other handler code.

Table 4.1: Exception interrupts, vectors, and precedence

Exception Interrupt	Vector Address	Service Precedence
Breakpoint	0004h:0007h	0
Trace	0008h:000Bh	1
Stack Overflow	000Ch:000Fh	2
Divide-by-zero	0010h:0013h	3
User RETI	0014h:0017h	4
<reserved>	0018h - 003Fh	5
NMI	009Ch:009Fh	6
Reset	0000h:0003h	7 (always serviced immediately, aborts other exceptions)

Event Interrupts

Event Interrupts are typically related to on-chip or off-chip peripheral devices and so occur asynchronously with respect to XA core activities. The XA core contains no inherent event interrupt sources, so event interrupts are handled by an interrupt control unit that resides on-chip but outside of the processor core.

On typical XA derivatives, event interrupts will arise from on-chip peripherals and from events detected on interrupt input pins. Event interrupts may be globally disabled via the **EA** bit in the Interrupt Enable register (IE) and individually masked by specific bits the IE register or its extension. When an event interrupt for a peripheral device is disabled but the peripheral is not turned off, the peripheral interrupt flag can still be set by the peripheral and an interrupt will occur if the peripheral is re-enabled. An event interrupt that is enabled is serviced when its priority is higher than that of the currently executing code. Each event interrupt is assigned a priority level in the Interrupt Priority register(s). If more than one event interrupt occurs at the same time, the priority setting will determine which one is serviced first. If more than one interrupt is pending at the same level priority, a hardwares precedence scheme is used to choose the first to service. The XA architecture defines 15 interrupt occurrence priorities that may be programmed into the Interrupt Priority registers for Event Interrupts. Note that some XA implementations may not support all 15 levels of occurrence priority. Consult the data sheet for a specific XA derivative for details.

Note that, like all other forms of interrupts, the PSW (including the Interrupt Mask bits) is loaded from the interrupt vector table when an event interrupt is serviced. Thus, the priority at which the interrupt service routine executes could be different than the priority at which the interrupt occurred (since that was determined not by the PSW image in the vector table, but by the Interrupt Priority register setting for that interrupt). Normally, it is advisable to set the execution priority in the interrupt vector to be the same as the Interrupt Priority register setting that will be used in the program.

Furthermore, the occurrence priority of an interrupt should never be set higher than the execution priority. This could lead to infinite interrupt nesting where the interrupt service routine is re-interrupted immediately upon entry by the same interrupt source.

Software Interrupts

Software Interrupts act just like event interrupts, except that they are caused by software writing to an interrupt request bit in an SFR. The standard implementation of the software interrupt mechanism provides 7 interrupts which are associated with 2 Special Function Registers. One SFR, the software interrupt request register (SWR), contains 7 request bits: one for each software interrupt. The second SFR is an enable register (SWE), containing one enable bit matching each software interrupt request bit.

Software interrupts are initiated by setting one of the request bits in the SWR register. If the corresponding enable bit in the SWE register is also set, the software interrupt will occur when it becomes the highest priority pending interrupt and its priority is higher than the current execution level. The software interrupt request bit in SWR must be cleared by software prior to returning from the software interrupt service routine.

Software interrupts have fixed interrupt priorities, one each at priorities 1 through 7. These are shown in Table 4.2 below. Software Interrupts are defined outside the XA core and may not be present on all XA derivatives; consult the specific XA derivative data sheet for details.

Table 4.2: Software interrupts, vectors, and fixed priorities

Software Interrupt	Vector Address	Fixed Priority
SWI1	0100h:0103h	1
SWI2	0104h:0107h	2
SWI3	0108h:010Bh	3
SWI4	010Ch:010Fh	4
SWI5	0110h:0113h	5
SWI6	0114h:0117h	6
SWI7	0118h:011Bh	7

The primary purpose of the software interrupt mechanism is to provide an organized way in which portions of event interrupt routines may be executed at a lower priority level than the one

at which the service routine began. An example of this would be an event Interrupt Service Routine that has been given a very high priority in order to respond quickly to some critical external event. This ISR has a relatively small portion of code that must be executed immediately, and a larger portion of follow-up or "clean-up" code which does not need to be completed right away. Overall system performance may be improved if the lower priority portion of the ISR is actually executed at a lower priority level, allowing other more important interrupts to be serviced.

If the high priority ISR simply lowers its execution priority at the point where it enters the follow-up code, by writing a lower value to the IM bits in the PSW, a situation called "priority inversion" could occur. Priority inversion describes a case where code at a lower priority is executing while a higher priority routine is kept waiting. An example of how this could occur by writing to the IM bits follows, and is illustrated in Figure 4.15.

Suppose code is executing at level 0 and is interrupted by an event interrupt that runs at level 10. This is again interrupted by a level 12 interrupt. The level 12 ISR completes a time-critical portion of its code and wants to lower the priority of the remainder of its code (the non-time critical portion) in order to allow more important interrupts to occur. So, it writes to the IM bits, setting the execution priority to 5. The ISR continues executing at level 5 until a level 8 event interrupt occurs. The level 8 ISR runs to completion and returns to the level 5 ISR, which also runs to completion. When the level 5 ISR returns, the previously interrupted level 10 ISR is re-activated and eventually competes.

It can be seen in this example that lower priority ISR code executed and completed while higher priority code was kept waiting on the stack. This is priority inversion.

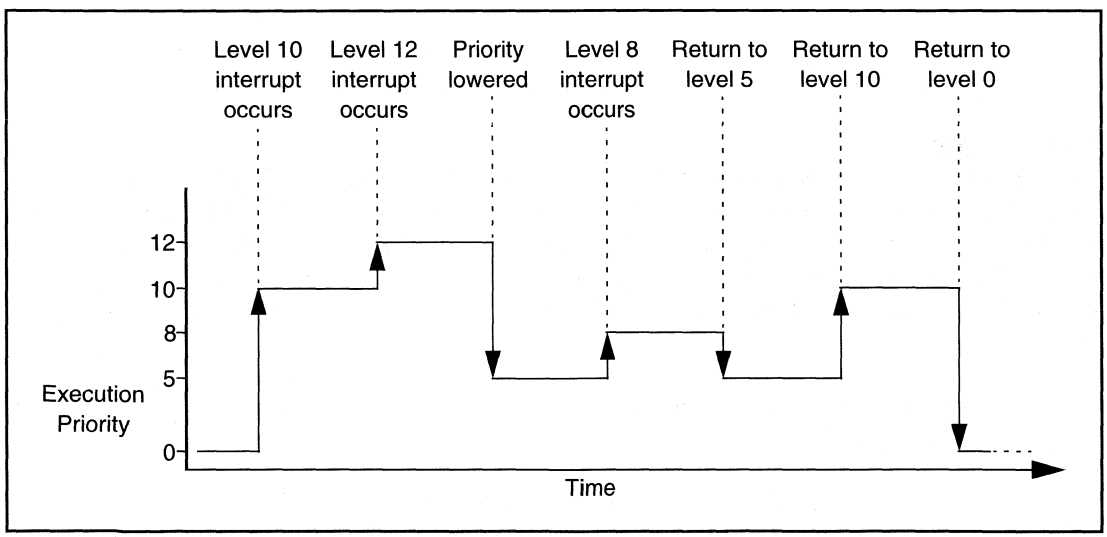


Figure 4.15 Example of priority inversion (see text)

In those cases where it is desirable to alter the priority level of part of an ISR, a software interrupt may be used to accomplish this without risk of priority inversion. The ISR must first be

split into 2 pieces: the high priority portion, and the lower priority portion. The high priority portion remains associated with the original interrupt vector. The lower priority portion is associated with the interrupt vector for software interrupt 5. At the completion of the high priority portion of the ISR, the code sets the request bit for software interrupt 5, then returns. the remainder of the ISR, now actually the ISR for software interrupt 5, executes when it becomes the highest priority pending interrupt.

The diagram in Figure 4.16 shows the same sequence of events as in the example of priority inversion, except using software interrupt 5 as just described. Note that the code now executes in the correct order (higher priority first).

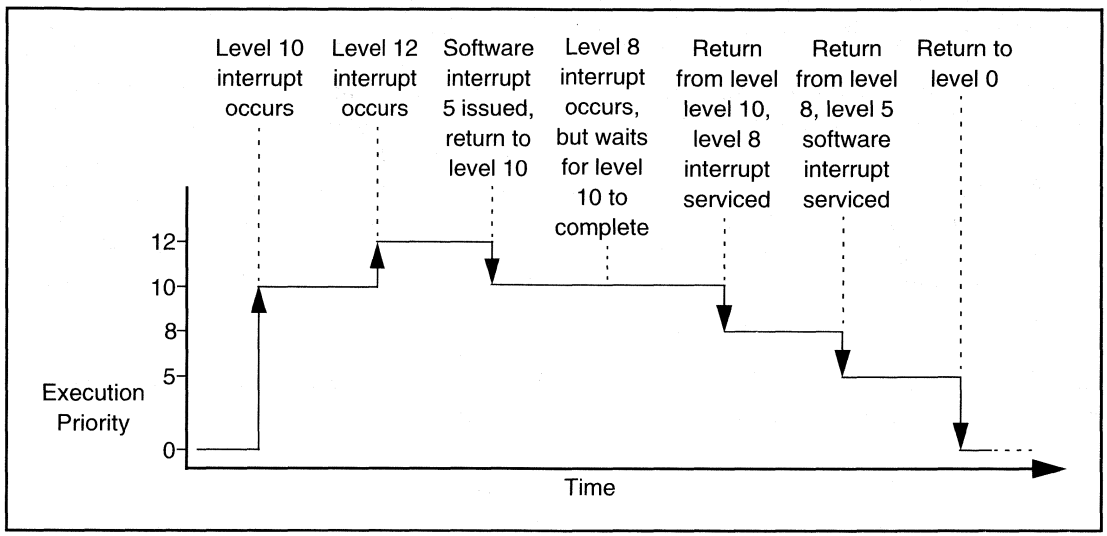


Figure 4.16 Example use of software interrupt (see text)

Trap Interrupts

Trap Interrupts are generated by the TRAP instruction. TRAP 0 through TRAP 15 are defined and may be used as required by applications. Trap Interrupts are intended to support application-specific requirements, as a convenient mechanism to enter globally used routines, and to allow transitions between user mode and system mode. A trap interrupt will occur if and only if the instruction is executed, so there is no need for a precedence scheme with respect to simultaneous traps.

The effect of a TRAP is immediate, the corresponding TRAP service routine is entered upon completion of the TRAP instruction.

See Chapter 6 for a detailed description of the TRAP instruction.

4.8.2 Interrupt Service Data Elements

There are two data elements associated with XA interrupts. The first is the stack frame created when each interrupt is serviced. The second is the interrupt vector table located at the beginning

of code memory. Understanding the structure and contents of each is essential to the understanding of how XA interrupts are processed.

Interrupt Stack Frame

A stack frame is generated, always on the System Stack, for each XA interrupt. With one exception, the stack frame is stored for the duration of interrupt service and used to return to and restore the CPU state of the interrupted code. (The exception is an Exception Interrupt triggered by a Reset event. Since Reset re-initializes the stack pointers, no stack frame is preserved. See section 4.4 for details.) The stack frame in the native 24-bit XA operating mode is illustrated in Figure 4.17. Three words are stored on the stack in this case. The first word pushed is the low-order 16 bits of the current PC, i.e., the address of the next instruction to be executed. The next word contains the high-order byte of the current PC. A zero byte is used as a pad. In sum, a complete 24-bit address is stored in the stack frame. The third word contains a copy of the PSW at the instant the interrupt was serviced.

When the XA is operating in Page 0 Mode (SCR bit **PZ** = 1) the stack frame is smaller because, in this mode, only 16 address bits are used throughout the XA. The stack frame in Page 0 Mode is illustrated in Figure 4.18. Obviously it is very important that stack frames of both sizes not be mixed; this is one reason for the admonition in section 4.3 to set the System Configuration Register once during XA initialization and leave it unchanged thereafter.

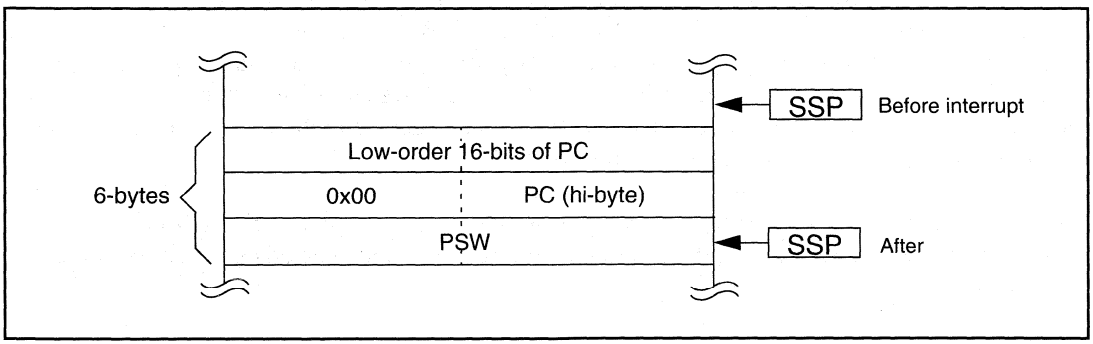


Figure 4.17 Interrupt stack frame (non- page zero mode)

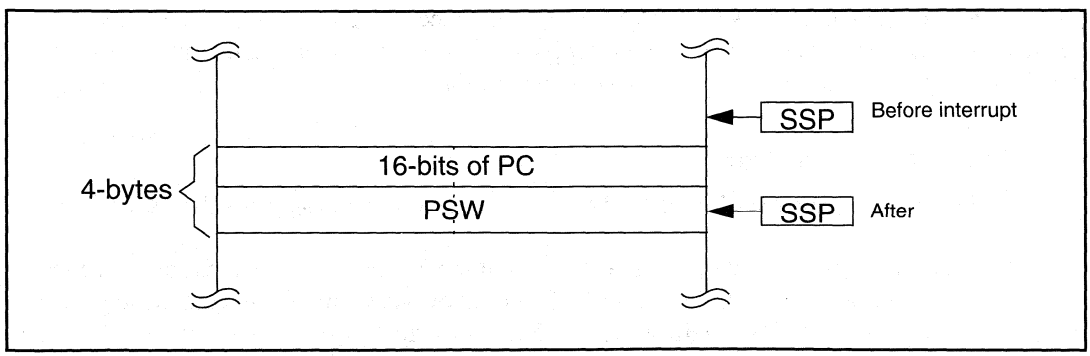


Figure 4.18 Interrupt stack frame (page 0 mode)

Interrupt Vector Table

The XA uses the first 284 bytes of code memory (addresses 0 through 11B hex) for an interrupt vector table. The table may contain up to 71 double-word entries, each corresponding to a particular interrupt event.

The double-word entries each consist of a 16 bit address of an interrupt service routine address and a 16 bit PSW replacement value. Because vector addresses are 16-bit, the first instruction of service routines must be located in the first 64K bytes of XA memory. The first instruction of all service routines must be word-aligned. Key elements of the replacement PSW value are the choice of System or User mode for the service routine, the Register Bank selection, and an Execution Priority setting. For more details on PSW elements, see section 4.2.2.

The first 16 vectors, starting at code memory address 0 are reserved for Exception Interrupt vectors. The second 16 vectors are reserved for Trap Interrupts. The following 32 vectors in the table are reserved for Event Interrupts. The final 7 vectors are used for Software Interrupts. Figure 4.19 illustrates the XA vector table and the structure of each component vector. Of the vectors assigned to Exceptions, 6 are assigned to events specific to the XA CPU and 10 are reserved. All 16 Trap Interrupts may be used freely. Assignments of Event Interrupt vectors are derivative-independent and vary with the peripheral device complement and pinout of each XA derivative.

Unused interrupt vectors should normally be set to point to a dummy service routine. The dummy service routine should clear the interrupt flag (if it is not self-clearing) and execute an RETI to return to the user program. This is especially true of the exception interrupts and NMI, since these could conceivably occur in a system where the designer did not expect them. If these vectors are routed to a dummy service routine, the system can essentially ignore the unexpected exception or interrupt condition and continue operation.

Note that when using some hardware development tools, it may be preferable not to initialize unused vector locations, allowing the development tool to flag unexpected occurrences of these conditions.

4.9 Trace Mode Debugging

The XA has an optional Trace Mode in which a special trace exception is generated at the conclusion of each instruction. Trace Mode supports user-supplied debugger/monitor programs which can single-step through any code, even code in ROM.

4.9.1 Trace Mode Operation

Trace Mode is initiated by asserting **PSW.TM** in the context of the program to be traced.

Using Trace Mode requires a detailed understanding of the XA instruction execution sequence because when and if a trace exception occurs depends on events within the execution sequence of a single instruction. Figure 4.20 illustrates the XA instruction sequence in overview.

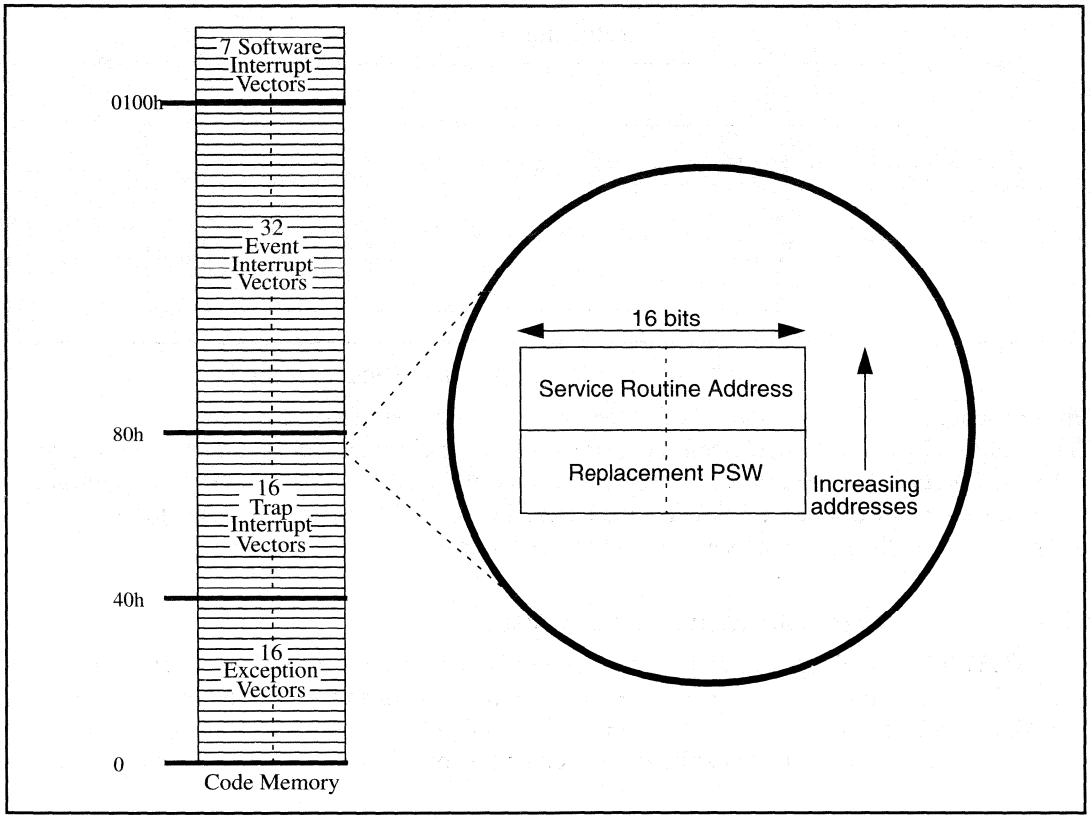


Figure 4.19 Interrupt vectors

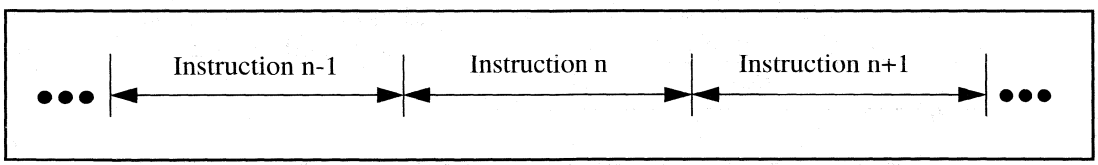


Figure 4.20 XA Instruction Sequence Overview

A detailed model of this sequence is shown in Figure 4.21: First, at the beginning of the instruction cycle, the state of the TM flag is latched. Next, the instruction is checked to see if it is valid; undefined instructions or disallowed operations (like a write through ES in User Mode) are simply not executed, and there is no chance for a trace to occur. The sequence then checks for instructions illegal in the current context (currently only an IRET while in User Mode is detected here) and services an exception if one is found. If, and only if, none of these special conditions occur, the instruction is actually executed. Just after execution, if the Trace Mode bit had been latched TRUE at the beginning of the instruction cycle, the Trace is serviced. Finally, the cycle checks for a pending interrupt and performs interrupt service if one is found
Note that an external reset may occur at any point during the cycle illustrated in Figure 4.21. This will abort processing when it occurs.

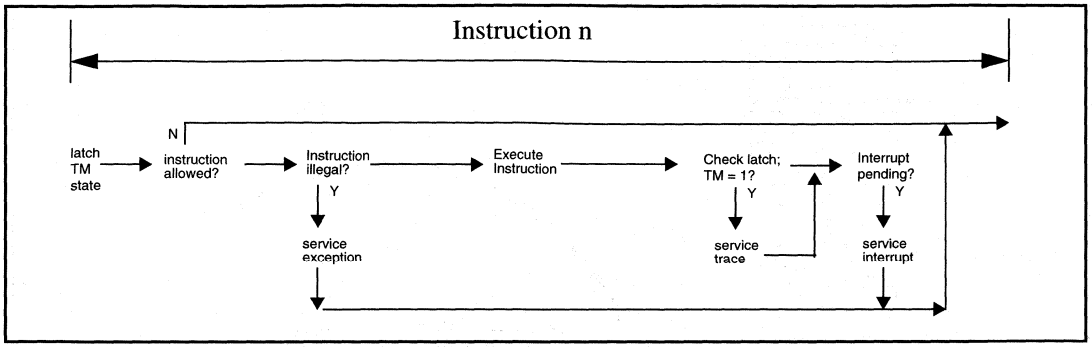
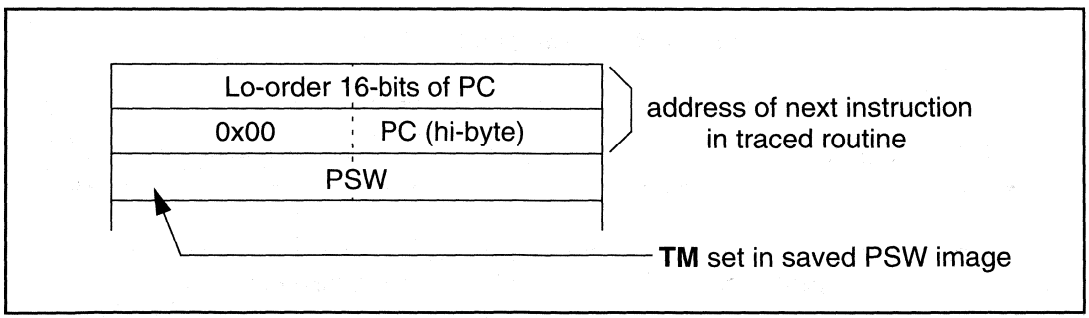


Figure 4.21 Instruction Execution Clock Detail

One consequence of this sequence is that the instruction that sets TM = 1 cannot generate a Trace, since TM is not latched when the instruction is actually executed. Another consequence is that an instruction that generates an exception will never be traced. Finally if an event interrupt occurs during an instruction clock when the instruction being executed is a TRAP, the TRAP will be executed, then the trace service, and finally the interrupt will be serviced.

4.9.2 Trace Mode Initialization and Deactivation

Since **PSW.TM** is in the protected portion of the PSW (i.e., in PSWH), only code executing in System Mode can initiate or turn off Trace Mode. In practice, this may be done by invoking a trap whose replacement PSW clears this bit, or by executing a RETI instruction with a synthetic Exception/Interrupt stack frame explicitly pushed on the top of the System Stack, as follows:



Tracing will continue until the PSW bit **TM** is cleared. This may be done by the trace service routine by examining the stack frame at the top of the system stack and clearing the TM bit prior to returning to the currently traced process. A similar method may be used to initiate trace mode. Note that stack frames generated by exception interrupts are always placed on the System stack. It is probably a good idea for the trace service routine to verify that the item in the stack frame is consistent with the traced process before modifying the TM bit.

5 Real-time Multi-tasking

Multi-tasking as the name suggests, allows tasks, which are pieces of code that do specific duties, to run in an apparently concurrent manner. This means that tasks will seem to all run at the same time, doing many specific jobs simultaneously.

High end applications (like automotive) require instantaneous responses when dealing with high speed events, such as engine management, traction control and adaptive braking system (ABS) and hence there is a trend towards multi-tasking in a wide variety of high performance embedded control applications.

Real-time application programs are often comprised of multiple tasks. Each task manages a specific facet of application program. Building a real-time application from individual tasks allows subdividing a complicated application program into independent and manageable modules. Each task shares the processor with other tasks in the application program according to an assigned priority level.

In real-time multi-tasking, the main concern is the *system overhead.* Switching tasks involve moving lots of data of the terminated and initiated tasks, and extensive book-keeping to be able to restore dormant tasks when required. Thus it is extremely crucial to minimize the system overhead as much as possible. In some cases, some of the tasks may be associated with real-time response, which further complicates the requirements from the system.

The following section analyzes the requirements and the XA suitability to these applications.

5.1 Multi-tasking Support in XA

The XA has numerous provisions to support multi-tasking systems. The architecture provides direct support for the concept of a multi-tasking OS by providing two (System/User) privilege levels for isolation between tasks. High performance, interrupt driven, multi-tasking applications systems requiring protection are feasible with the XA.

The XA architecture offers the following features which will appeal to multi-tasking implementations.

5.1.1 Dual stack approach

The architecture defines a System Stack Pointer (SSP) as well as an User Stack Pointer (USP). The dual stack feature supports fast task switching, and ease the creation of a multi-tasking monitor kernel. The separation of the two offers a reduction is storing and retrieving stack pointers or using a single stack, when switching to the kernel and back to an application. It also serves to speed up interrupt processing in large systems with external data memory. User stacks can be allocated in the slower external memory, while system memory is in internal SRAM, allowing for fast interrupt latency in this environment. The dual stack approach also adds the benefit of a better potential to recover from an ill-behaved task, since the system stack is still intact when an error is sensed.

5.1.2 Register Banks

The XA also supports 4 banks of 8 byte/4 word registers, in addition to 12 shared registers. In some applications, the register banks can be designated statically to tasks, cutting significantly on the overhead for saving and restoring registers on context switching.

5.1.3 Interrupt Latency and Overhead

Interrupt latency is extremely critical in a multitasking environment. For a real-time multitasking environment, a fast interrupt response is crucial for switching between tasks. The XA is designed to provide such fast task switching environment through improved interrupt latency time.

The interrupt service mechanism saves the PC (1 or 2 words, depending on the Page0 mode flag PZ) and the PSW (1 word) on the stack. The interrupt stack normally resides in the internal data memory, and interrupt call including saving of three words takes 23 clocks. Prefetching the service routine takes 3 additional clocks.

When interrupt or an exception/trap occurs, the current instruction in progress always gets executed prior servicing the interrupt. This present an overhead, while increasing the effective interrupt latency, since the event that interrupted the machine cannot be dealt with before the book-keeping is completed. In XA, the longest uninterrupted instruction is the signed 32x16 Divide, which takes 24 clocks.

This puts the worst case interrupt latency at [24 + 23 + 3] = 50 clocks (3.125 microseconds at 16.0 MHz, 2.5 microseconds at 20.0 MHz, and 1.67 microseconds at 30.0 MHz). Saving the state of the lower registers can be done by simply switching the register bank.

In the general case, up to 16 registers would be saved on the stack, which takes 32 clocks. The total latency+overhead at start of an interrupt is a maximum of 68 clocks (4.25 microsecond at 16 MHz, 3.4 at 20 MHz and 2.27 at 30 MHz). This allows for extremely fast context switching for multitasking environments.

5.1.4 Protection

The issue is mentioned here simply to clarify what is and what is not supported by the XA architecture. Dual stack pointer and minor privileges to what looks like a supervisor mode do not mean full protection. It is assumed that code in a microcontroller does not require guarding from intentional system break-in by a lower privilege task. A table of the protected features in XA is given below. Note that features marked "disallowed" are simply not completed if attempted in the User mode. There are no exceptions or flags associated with these occurrences.

Table 5.1: Segment and Stack Register Protection

Mode	Write to DS	Write through DS	Write to ES	Write through ES	Read through DS	Read through ES	Read through SSP	Write to SSP	Write to SSEL bit 7
System	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
User	Dis-allowed	Allowed	Allowed	Select-able [1]	Allowed	Allowed	Not possible	Not possible	Dis-allowed

Note 1: The MSB of SSEL (bit 7) selects whether write through ES is allowed in User mode. However, this bit is accessible only in System mode.

Table 5.2: PSW bit protection

Mode	Write to SM bit	Write to RS0:1 bits	Write to TM bit	Write to IM0:3 bits
System	Allowed	Allowed	Allowed	Allowed
User	Disallowed	Allowed	Disallowed	Disallowed

In addition to the above, the System Stack is protected from corruption by User Mode execution of the RETI instruction. If User Mode code attempts to execute that instruction, it causes an exception interrupt. If it is necessary to run TRAP routines, for instance, in User Mode, the User RETI exception handler can perform the return for the User Mode code. To accomplish this, the User RETI exception handler may pop the topmost return address from the stack (2 or 3 words, depending on whether the XA is in Page Zero mode) and then execute the RETI.

Protection Via Data Memory Segmentation

In User/Application mode, each task is protected from all others via the separation of data spaces (unless explicit sharing is planned in advance). If the address spaces of two tasks include no shared data, one task cannot affect the data of another, but it can read any data in the full address space. Code sharing is always safe since code memory may never be written[1]. An application mode program is prohibited from writing the segment registers, thus confining the writable area per an ill-behaved task to its dedicated segment. Most applications, which are not expected to utilize multi-tasking or use external memory, do not require any protection. They will remain after reset in system mode, and could access all system resources.

At any given instant, two segments of memory are immediately accessible to an executing XA program. These are the data segment DS, where the stack and local variables reside, and the extra segment ES, which may be used to read remote data structures. Restricting the addressability of task modules helps gain complete control of system resources for efficient, reliable operation in a multi-tasking environment.

1. True for non-writable code memory only like EPROM, ROM, OTP. This might change for FLASH parts.

Protection Via Dual Stack Pointers

The XA provides a two-level user/supervisor protection mechanism. These are the *user* or *application* mode and the *system* or *supervisor* mode. In a multitasking environment, tasks in a supervisor level are protected from tasks in the application level.

The XA has two stack pointers (in the register file) called the System Stack Pointer (SSP) and the User Stack Pointer (USP). In multitasking systems one stack pointer is used for the supervisory system and another for the currently active task. This helps in the protection mechanism by providing isolation of system software from user applications. The two stack pointers also help to improve the performance of interrupts. If the stack for a particular application would exceed the space available in the on-chip RAM, or on-chip RAM is needed for other time critical purposes (since on-chip RAM is accessed more quickly than off-chip memory), the main stack can be put off-chip and the interrupt stack (using the System SP) may be put in on-chip RAM.

These features of the XA place it well above the competition in suitability to multi-tasking applications.

6 Instruction Set and Addressing

This section contains information about the addressing modes and data types used in the XA. The intent is to help the user become familiar with the programming capabilities of the processor.

6.1 Addressing Modes

Addressing modes are ways to form effective addresses of the operands. The XA provides seven *basic* powerful addressing modes for access on word, byte, and bit data, or to specify the target address of a branch instruction. These *basic* addressing modes are uniformly available on a large number of instructions. Table 6.1 includes the basic addressing modes in the XA. An instruction could use a combination of these basic addressing modes, e.g., ADD R0, #020 is a combination of Register and Immediate addressing modes.

All modes (non-register) generate ADDR[15:0]. This address is combined with DS/ES[23:16] for data and PC/CS[23:16] for code to form a 24-bit address[1].

An XA instruction can have zero, one, two, or three operands, whose locations are defined by the addressing mode. A *destination* operand is one that is replaced by a result, or is in some way affected by the instruction. The destination operand is listed first in an addressing mode expression. A *source* operand is a value that is moved or manipulated by the instruction, but is not altered. The source is listed second in an addressing mode expression.

Table 6.1 Basic Addressing Modes

MODE	MNEMONIC	OPERANDS
Register	R	operand(s) in register (in Register file)
Indirect	[R]	Byte/Word whose 16-bit address is in R
Indirect-Offset	[R+off 8/16]	Byte or Word data whose address (16-bit) contained in R, is offset by 8/16-bit signed integer "off 8/16'
Direct	mem_addr	Byte/Word at given memory "mem_addr'
SFR [1]	sfr_addr	Byte/Word at "sfr_addr' address
Immediate	#data 4/5 #data 8/16	Immediate 4/5 and 8/16-bit integer constants "data8/16"
Bit	bit	10-bit address field specifying Register File, Data Memory or SFR bit address space

1. This is a special case of direct addressing mode but separately identified, as SFR space is separate from data memory.

1. Exception is Page 0 mode, where all addresses are 16-bit.

6.2 Description of the Modes

6.2.1 Register Addressing

Instructions using this addressing mode contain a field that addresses the Register File that contains an operand. The Register file is byte[2], word, double-word or bit addressable.

Example: ADD R6, R4

Before: R4 contains 005Ah
R6 contains A5A5h

After: R4 contains 005Ah
R6 contains A5FFh

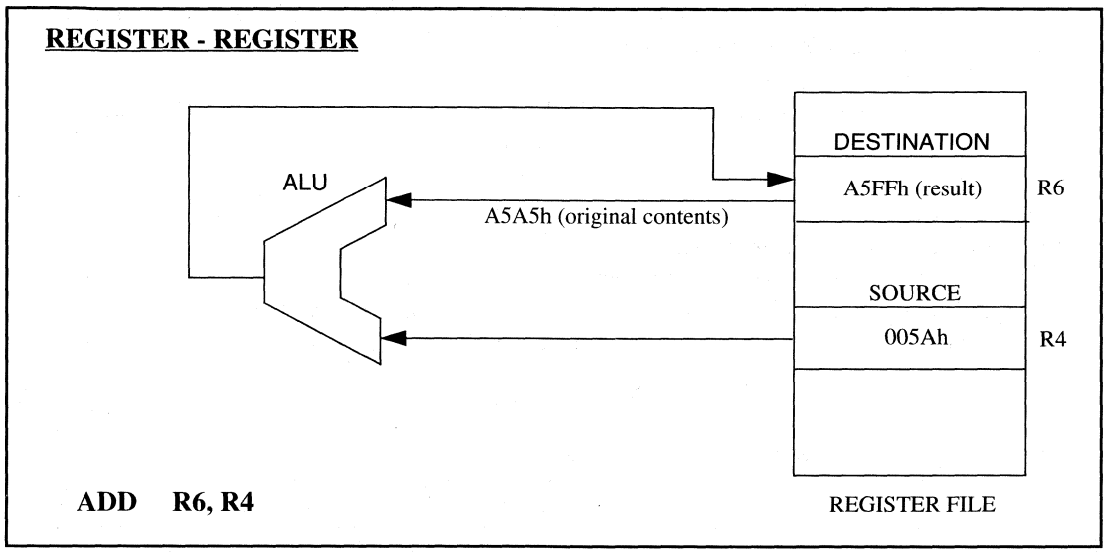


Figure 6.1

2. The unimplemented 8 word registers are not Byte addressable

6.2.2 Indirect Addressing

Instructions using this addressing mode contain a 16-bit address field. This field is contained in 1 out of 8 pointer registers in the Register File (that contain the 16-bit address of the operand in any 64K data segment). For data, the segment is identified by the 8-bit contents of DS or the ES and for code by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL corresponding to the indirect register number. The address of the pointer word for word operands should be even

Example:	ADD R6, [R4]	Before:	R6 contains 1005h
	SSEL.4 = 1		R4 contains A000h
	i.e., the operand is in		Word at A000h contains A5A5h
	segment determined		
	by the contents of ES	After:	R4 contains A000h
	So, if ES = 08, the		R6 contains B5AAh
	operand is in		Word at A000h in segment 8
	segment 8 of data memory.		of data memory contains A5A5h

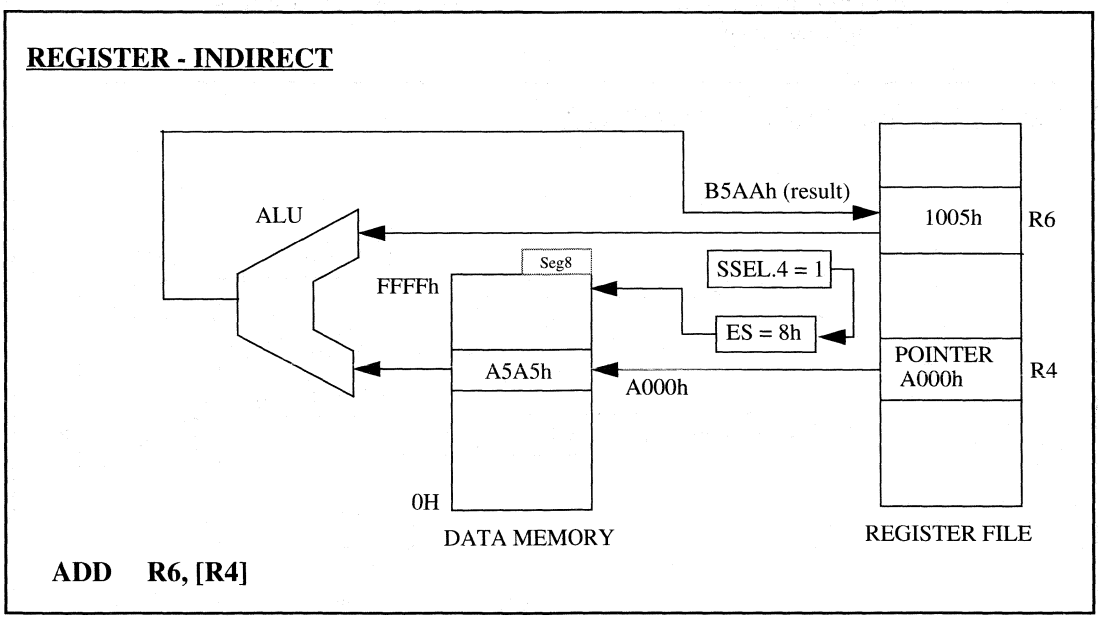


Figure 6.2

6.2.3 Indirect-Offset Addressing

This addressing mode is just like the Register-Indirect addressing mode above except that an additional displacement value is added to obtain the final effective address. Instructions using this addressing mode contain a 16-bit address field and an 8 or 16-bit signed displacement field. This field addresses 1 out of 8 pointer registers in the Register File that contains the 16-bit address of the operand in any 64K data segment. The contents of the pointer register are added to the signed displacement to obtain the effective address[3] (which *must* be even) of the operand. For data the segment is identified by the 8-bit contents of DS or the ES and for code, by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL.

Example:	ADD R5, [R3 +30h] SSEL.3 = 1 i.e., the operand is in segment determined by the contents of ES So, if ES = 04, the operand is in segment 4 of data memory.	Before:	R3 contains C000h R5 contains 0065h Word at C030h = A540h
		After:	R3 contains C000h R5 contains A5A5h Word at C030h = A540h

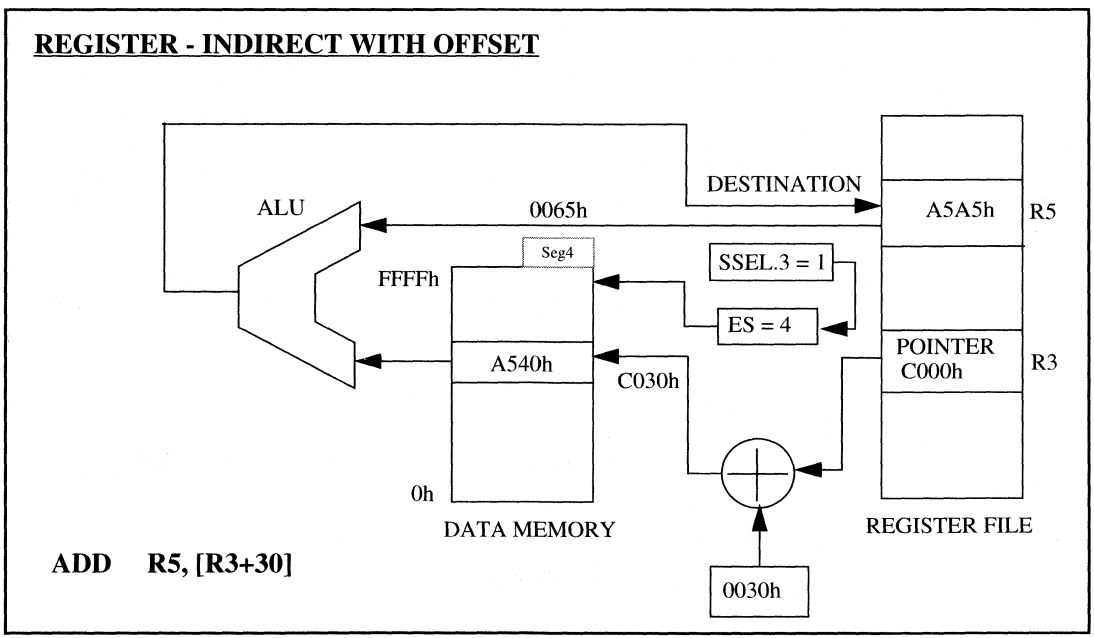


Figure 6.3

3. In case of an odd address, the XA forces the operand fetch from the next lower even boundary (address.bit0 = 0)

6.2.4 Direct Addressing

Instructions using this addressing mode contain an 10-bit address field, which contains the actual address of the operand in any 64K data memory segment or sfr space.The direct address data memory space is always the bottom 1K byte (0:3FFh) of any segment. The associated data segment is always identified by the 8-bit contents of DS.

Example: SUB R0, 200h
If DS = 02, the operand is in segment 2 of data memory.

Before: R0 contains A5FFh
200H contains 5555h

After: R0 contains 50AAh
200h contains 5555h

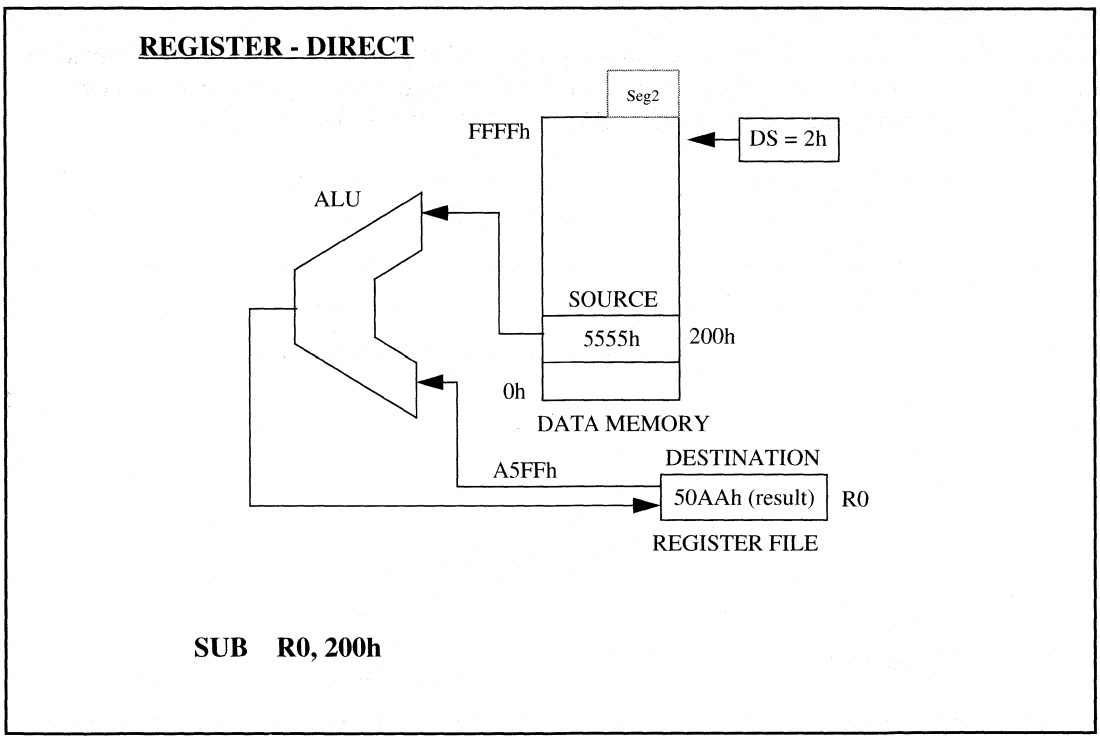


Figure 6.4

6.2.5 SFR Addressing

This is identical to the direct addressing mode described before, except it addresses the 1K SFR space. Although encoded into the same instruction field as the direct addressing described above, this is actually a separate space. Instructions using this addressing mode contain an 10-bit SFR address. The 1K SFR space is always directly addressed (400:7FFh) and is mapped directly above the 1K direct-addressed RAM space.

Example: MOV R0H, 406h[4]

Before: R0H contains 05h
406h contains A5h

After: R0H contains A5h
406h contains A5h

6.2.6 Immediate Addressing

In immediate addressing, the actual operand is given explicitly in the instruction.The immediate operand is either an 4/5, 8 or 16-bit integer which constitutes the source operand. 4-bit short immediate operands used with instructions ADDS and MOVS are sign extended.

Example: ADD R0L,#0B9h

Before: R0 contains 13h
After: R0L contains CCh

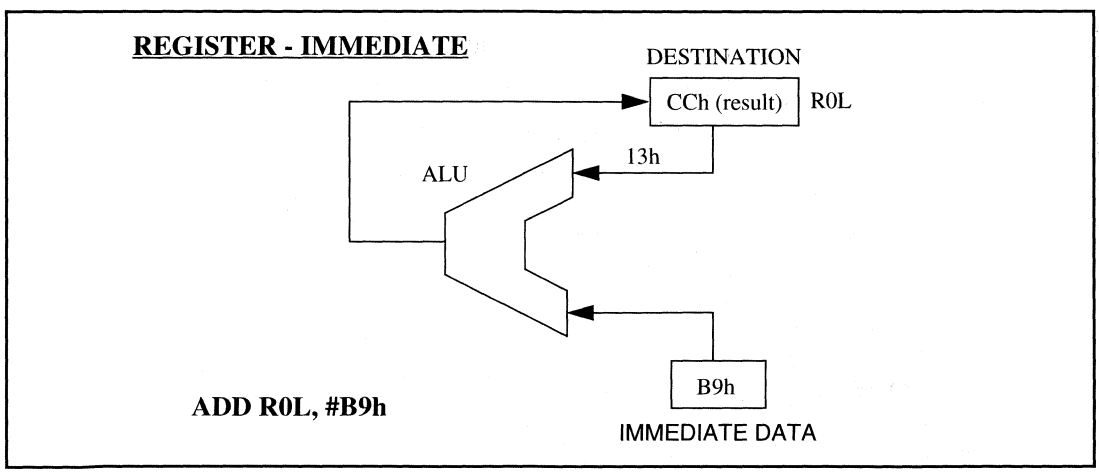


Figure 6.5

4. The syntax always refers to the SFR address starting from the base address of 400H.

6.2.7 Bit Addressing

Instructions using the bit addressing mode contain a 10-bit field containing the address of the bit operand. The XA supports three bit address spaces, which are encoded into the same format. The spaces are: 256 bits in the register file (the entire active register file); 256 bits in the data memory (byte addresses 20 through 3F hex on the current data segment); and 512 bits in the SFR space (byte addresses 400 through 43F hex).

Bit addresses 0 to FF hex map to the register file, bit addresses 100 to 1FF hex map to data memory, and bit addresses 200 to 3FF map to the SFR space.

A separate bit-addressable space (20-3F hex) in the direct-address data memory, exists for each segment. The current working segment for the direct-address space being always identified by the DS register.

The encoding of the 10-bit field for bit addresses is as follows:

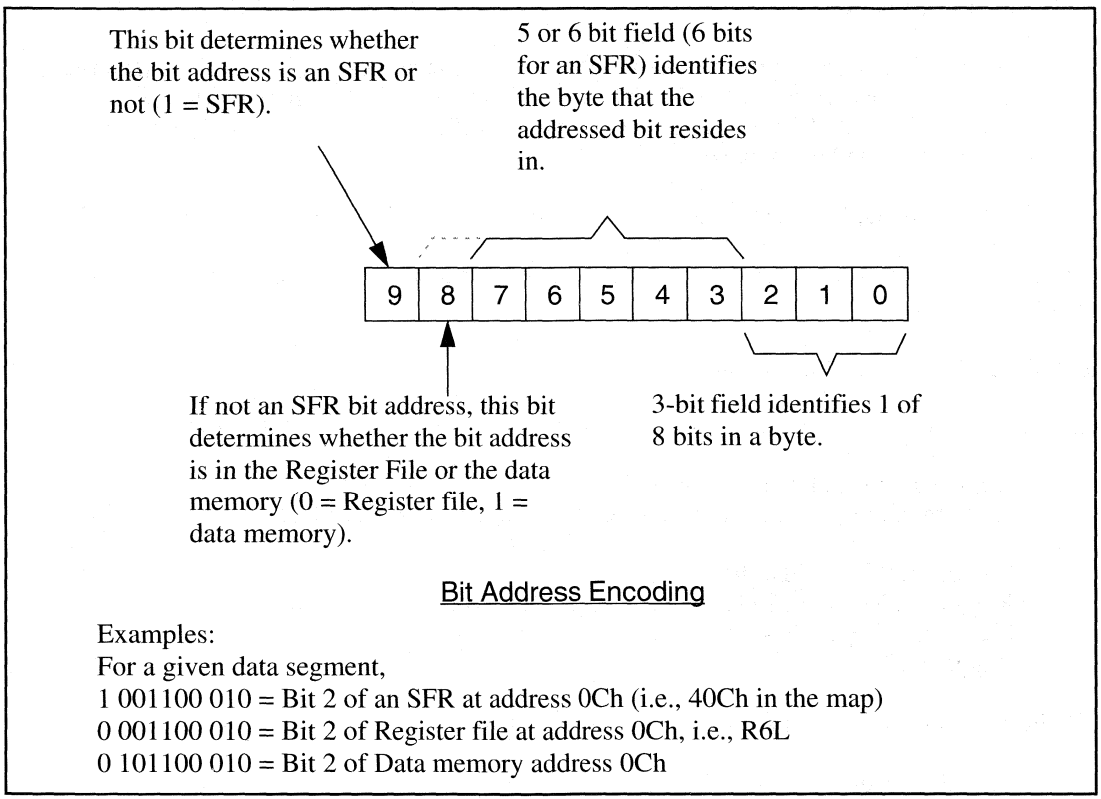


Figure 6.6

6.3 Relative Branching and Jumps

Program memory addresses as referenced by Jumps, Calls, and Branch instructions must be word aligned in XA. For instance, a branch instruction may occur at any code address, but it may only branch to an even address. This forced alignment to even address provides three benefits:

- Branch ranges are doubled without providing an extra bit in the instruction and
- Faster execution as XA always fetches first two byte of an instruction simultaneously.
- Allows translated 8051 code to have branches extended over intervening code that will tend to grow when translated and generally increase the chances of a branch target being in that range.

The *rel8* displacement is a 9-bit two's complement integer which is encoded as 8-bits that represents the relative distance in words from the current PC to the destination PC. Similarly, the *rel16* displacement is a 17-bit twos complement integer which is encoded as 16-bits. The value of the PC used in the target address calculation is the address of the instruction following the Branch, Jump or Call instruction.

The 8-bit signed displacement is between -128 to +127. The branch range for rel8 is (sample calculation shown below) is really +254 bytes to -256 bytes for instructions located at an *even* address, and +253 to -257 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

The 16-bit signed displacement is -32,768 to +32,767. The branch range is therefore +65,534 bytes to -65,536 bytes for instructions located at an *even* address, and +65,533 to -65,537 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

Sample calculation for rel8 range:

Assuming word aligned branch target, forward range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = +127; So, rel8 << 1 is +254

If Current PC = ODD, then
Range = 254 - 1 = +253 as PC is forced to an even location, else
If current PC = EVEN, then
Range = +254

Similarly, reverse range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = -128; So, rel8 << 1 is -256

If Current PC = ODD, then
Range = -257
Else if current PC = EVEN, then
Range = -256

6.4 Data Types in XA

The XA uses the following types of data:

- Bits
- 4/5-bit signed integers
- 8-bit (byte) signed and unsigned integers
- 8-bit, two digit BCD numbers
- 16-bit (word) signed and unsigned integers
- 10-bit address for bit-addressing in data memory and SFR space
- 24-bit effective address comprising of 16-bit address and 8-bit segment select. See addressing modes for more information.

A byte consists of 8-bits. A word is a 16-bit value consisting of two contiguous bytes. A double word consists of two 16-bit words packed in two contiguous words in memory.

Negative integers are represented in twos complement form. 4-bit signed integers (sign extended to byte/word) are used as immediate operands in MOVS and ADDS instructions.

Binary coded decimal numbers are packed, 2 digits per byte. BCD operations use byte operands.

6.5 Instruction Set Overview

The XA uses a powerful and efficient instruction set, offering several different types of addressing modes. A versatile set of "branch" and "jump" instructions are available for controlling program flow based on register or memory contents. Special emphasis has been placed on the instruction support of structured high-level languages and real-time multi-tasking operating systems.

This section discusses the set of instructions provided in the XA microcontroller, and also shows how to use them. It includes descriptions of the instruction format and the operands used by the instructions. After a summary of the instructions by category, the section provides a detailed description of the operation of each instruction, in alphabetical order.

Five summary tables are provided that describes the available instructions. The first table is a summary of instructions available in the XA along with their common usage. The second and third table are tables of addressing modes and operands, and the instruction type they pertain to. A fourth table that lists the summary of status flags update by different instructions. A fifth table lists the available instructions with their different addressing modes and briefly describes what each instruction does along with the number of bytes, and number of clocks required for each instruction.

The formats have been chosen to optimize the length and execution speed of those instructions that would be used the most often in critical code. Only the first and sometimes the second byte of an instruction are used for operation encoding. The length of the instruction and the first execution cycle activity are determined from the first byte. Instruction bytes following the first two bytes (if any) are always immediate operands, such as addresses, relative displacements, offsets, bit addresses, and immediate data.

General:

offset8	An 8-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
offset16	A 16-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
direct	An 11-bit immediate address contained in the instruction.
#data4	4 bits of immediate data contained in the instruction. (range +7 to -8 for signed immediate data and 0-15 for shifts)
#data5	5 bits of immediate data contained in the instruction. (0-31 for shifts)
#data8	8 bits of immediate data contained in the instruction. (+127 to -128)
#data16	16 bits of immediate data contained in the instruction. (+32,767 to -32,768)
bit	The 10-bit address of an addressable bit.
rel8	An 8-bit relative displacement for branches. (+254 to -256)
rel16	An 16-bit relative displacement for branches.(+65,534 to -65,536)
addr16	A 16-bit absolute branch address within a 64K code page.
addr24	A 24-bit absolute branch address, able to access the entire XA address space.
SP	The current Stack Pointer (User or System) depending on the operation mode.
USP	The User Stack Pointer.
SSP	The System Stack Pointer
C	Carry flag from the PSW.
AC	Auxiliary Carry flag from the PSW.
V	Overflow flag from the PSW.
N	Negative flag from the PSW.
Z	Zero flag from the PSW.
DS	Data segment register. Holds the upper byte of the 24-bit data address space of the XA. Used mainly for local data segments.
ES	Extra segment register. Holds the upper byte of the 24-bit data address space of the XA. Used mainly for addressing remote data structures.

direct	Uses the current DS for data memory for the upper byte of the 24-bit address or none (uses only the low 16-bit address) for accessing the special functions register (SFR) space. The interpretation should be as below:
	if (data range)
	then (direct = (DS:direct)
	if (sfr range)
	then (direct) = (sfr)

Operation encoding fields:

SZ	Data Size. This field encodes whether the operation is byte, word or double-word.
IND	This field flags indirect operation in some instructions.
H/L	This field selects whether PUSH and POP Rlist use the upper or lower half of the available registers.
dddd	Destination register field, specifies one of 16 registers in the register file.
ddd	Destination register field for indirect references, specifies one of 8 pointer registers in the register file.
ssss	Source register field, specifies one of 16 registers in the register file.
sss	Source register field for indirect references, specifies one of 8 pointer registers in the register file.

Mnemonic text:

Rs	Source register.
Rd	Destination register.
[ ]	In the instruction mnemonic, indicates an indirect reference (e.g.: [R4] refers to the memory address pointed to by the contents of register 4).
[R+]	Used to indicate an automatic increment of the pointer register in some indirect addressing modes.
[WS:R]	Indicates that the pointer register (R) is extended to a 24-bit pointer by the selected segment register (either DS or ES for all instructions except MOVC, which uses either PC_{23-16} or CS).
Rlist	A bitmap that represents each register in the register file during a PUSH or POP operation. These registers are R0-R7 for word or R0L-R7H for byte.

Pseudocode:

( )	Used to indicate "contents of" in the instruction operation pseudocode (e.g.: (R4) refers to the contents of register 4).
<---	Pseudocode assignment operator. Occasionally used as <--> to indicate assignment in both directions (interchange of data).
((SP))	Data memory contents at the location pointed to by the current stack pointer. In system mode, the current SP is the SSP, and the segment used is always segment 0. In user mode, the current SP is the USP, and the segment used is the Data Segment (DS). This segment apply to the uses of the SP, not just PUSH and POP. In a few cases, "((SSP))" or "((USP))" indicate that a specific SP is used, regardless of the operating mode.
Rn.x	Indicates bit x of register n.
Rn.x-y	Indicates a range of bits from bit x to bit y of register n.

Note: all indirect addressing is accomplished using the contents of the data segment register as the upper 8 address bits unless otherwise specified. Example: [ES:Rs] indicates that the extra segment register generates the upper 8 bits of the address in this case.

Execution time:

PZ	- In Page 0
nt	- Not Taken
t	- Taken

Syntax For Operand size:
.w = For word operands
.b = byte operands
.d = double-word operands

Default operand size is dependant on the operands used e.g MOV R0,R1 is always word-size whereas MOV R0L, R0H is always byte etc. For INDIRECT_IMMEDIATE, DIRECT_IMMEDIATE, DIRECT_DIRECT, etc., user must specify operand size.

Others

0x = prefix for Hex values
[] = For indirect addressing
[[]] = For Double-indirect addressing
dest = destination
src = source

Table 6.2 Instruction Set in XA

Mnemonic	Usage
MOV, MOVC, MOVS, MOVX, LEA, XCH, PUSH, POP, PUSHU, POPU	Data Movement
ADD, ADDS, ADDC, SUB, SUBB	Add and Subtract
MULU.b, MULU.w, MUL.w DIVU.b, DIVU.w, DIVU.d, DIV.w, DIV.d	Multiply and Divide
RR, RRC, RL, RLC, LSR, ASR, ASL, NORM	Shifts and Rotates
CLR, SETB, MOV, ANL, ORL	Bit Operations
JB, JBC, JNB, JNZ, JZ, DJNZ, CJNE,	Conditional Jumps/Calls
BOV, BNV, BPL, BCC, BCS, BEQ, BNE, BG, BGE, BGT, BL, BLE, BLT, BMI	Conditional Branches
AND, OR, XOR	Boolean Functions
JMP, FJMP, CALL, FCALL, BR	Unconditional Jumps/Calls/Branches
RET, RETI	Return from subroutines, interrupts
SEXT, NEG, CPL, DA	Sign Extend, Negate, Complement, Decimal Adjust
BKPT, TRAP#, RESET	Exceptions
NOP	No Operation

Table 6.3 shows a summary of the basic addressing modes available for data transfer and calculation related instructions.

Table 6.3 Instruction Addressing Modes

Modes/ Operands	MOVX	MOV	CMP	ADD ADDC	SUB SUBB	AND OR XOR	ADDS MOVS	MUL DIV	Shift	XCH	bytes
R, R		•	•	•	•	•		•	•	•	2
R, [R]	•	•	•	•	•	•				•	2
[R], R	•	•	•	•	•	•					2
R, [R+off8]		•	•	•	•	•					3
[R+off8], R		•	•	•	•	•					3
R, [R+off16]		•	•	•	•	•					4
[R+off16], R		•	•	•	•	•					4
R, [R+]		•	•	•	•	•					2
[R+], R		•	•	•	•	•					2
[R+], [R+]		•									2
dir, R		•	•	•	•	•					3
R, dir		•	•	•	•	•				•	3
dir, [R]		•									3
[R], dir		•									3
R, #data		•	•	•	•	•	•	•	•		2*/3/4
[R], #data		•	•	•	•	•	•				2*/3/4
[R+], #data		•	•	•	•	•	•				2*/3/4
[R+off8], #data		•	•	•	•	•	•				3*/4/5
[R+off16], #data		•	•	•	•	•	•				4*/5/6
dir, #data		•	•	•	•	•	•				3*/4/5
dir, dir		•									4
R, USP		•									2

Notes:
- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.
* : ADDS and MOVS uses a short immediate field (4 bits).
** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Modes/ Operands	**MOVC**	PUSH POP	DA, SEXT CPL, NEG	JUMP CALL	DJNZ	CJNE	BIT OPS	MISC	bytes
R, [R+]	•								2
[R+], R	•								2
A, [A+DPTR]	•								2
A, [A+PC]	•								2
direct		•							3
Rlist		•							2
R			•						2
addr24				•					4
[R]				•					2
[A+DPTR]				JMP					2
R, rel					•				3
direct, rel					•				4
R, direct, rel						•			4
R, #data, rel						•			4/5
[R], #data, rel						•			4/5
bit							•		3
bit, C; C, bit							•		3
C, /bit							•		3
rel				•				Cond. Branch	2
bit, rel								Cond. Branch	4
#data4								TRAP	2
R, R+off8								LEA	3
r, R+off16								LEA	4
<none> **								•	1/2

Notes:
- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.
* : ADDS and MOVS uses a short immediate field (4 bits).
** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Table 6.4 summarizes the status flag updates for the various XA instruction types.

Table 6.4 Status Flag Updates

Instruction Type	Flags Updated				
	C	AC	V	N	Z
ADD, ADDC, CMP, SUB, SUBB	X	X	X	X	X
ADDS, MOVS	-	-	-	X	X
AND, OR, XOR	-	-	-	X	X
ASR, LSR	*	-	-	X	X
branches, all bit operations, NOP	-	-	-	-	-
Calls, Jumps, and Returns	-	-	-	-	-
CJNE	X	-	-	X	X
CPL	-	-	-	X	X
DA	*	-	-	X	X
DIV, MUL	*	-	*	X	X
DJNZ	-	-	-	X	X
LEA	-	-	-	-	-
MOV, MOVC, MOVX	-	-	-	X	X
NEG	-	-	X	X	X
NORM	-	-	-	X	X
PUSH, POP	-	-	-	-	-
RESET	*	*	*	*	*
RL, RR	-	-	-	X	X
RLC, RRC	*	-	-	X	X
SEXT	-	-	-	-	-
TRAP, BKPT	-	-	-	-	-
XCH	-	-	-	-	-
ASL	*	-	X	X	X

Notes:
-: flag not updated.
X: flag updated according to the standard definition.
*: flag update is non-standard, refer to the individual instruction description.
Note: Explicit writes to PSW flags takes precedence over flag updates.

Instruction Set Summary

Table 6.5 lists the entire XA instruction set by instruction type. This can be used as a quick reference to find specific instructions that may be looked up in the detailed alphabetical description section.

Instruction timing data given in this table and in the following detailed instruction description section are based on code execution from internal code memory and data accesses to internal RAM and registers only. Due to the highly programmable timing of accesses to external code and data memory on the XA and the interaction of pipelined functions, detailed timing for all conditions cannot be documented in a concise fashion. The instruction timing data given here also assumes that the CPU does not need to stall while the instruction is read into the pre-fetch queue.

In the case of branches, one on-chip code fetch (16 bits) is built into the timing numbers. The time given will be valid if the instruction that is branched to is not longer than two bytes. For longer instructions, the CPU will wait until the entire instruction is contained in the pre-fetch queue before resuming execution. This may take one or two additional fetches since the XA has instructions up to six bytes in length.

Following is a summary of events or conditions that may cause timing differences from the given data. These are generally stalls that occur when the CPU must wait for some information to become available.

— Instruction fetch. Execution stalls if the pre-fetch queue does not contains a complete instruction when it is needed. Except following branches, the state of the queue depends upon the history of instructions that have previously executed.

— Instruction sequence dependencies. This typically occurs when an instruction that reads data from a resource such as the SFR bus or the external bus follows an instruction that caused a write to the same resource. The CPU must stall while the write completes (which otherwise requires no CPU time) before the read can begin. Execution cannot resume until the read is complete.

— Internal data memory versus SFR accesses. SFR reads require an additional 2 clocks to complete. Because XA peripherals run from the CPU clock divided by 2, there may be one clock used to synchronize the CPU and the SFR bus.

— Program flow changes. When any change occurs in the program flow, the XA must flush the pre-fetch queue and begin loading it from the new execution address. The published timing values include one internal code fetch for all branches, jumps, calls, etc. If the instruction at the new address is longer than two bytes, additional fetch cycles must occur to obtain a complete instruction in the queue. In the case of a return from subroutine or interrupt, the first code fetch may only obtain one byte of the next instruction since returns may resume execution at odd code addresses.

— Internal versus external code execution. Programmable bus timing and other bus considerations result in a different timing for internal and external code accesses. Use of the 8-bit bus width for external code access has a substantial effect on overall performance. Possible use of the WAIT signal adds an additional variable to this effect. The external bus requirement for an ALE cycle at 16-byte address boundaries, during program flow changes, and after external bus data accesses also adds to the variability.

— Internal versus external data access. Programmable bus timing again causes different timing for internal and external data accesses. The 8-bit data bus setting contributes to the differences. Use of the WAIT signal may vary the timing still further.

— Collision of external code fetch and external data access. When an externally executing program accesses data on the external bus, the pre-fetch queue tends to starve more often that for internal execution.

Table 6.5

Mnemonic		Description	Bytes	Clocks
Arithmetic Operations				
ADD	Rd, Rs	Add registers direct	2	3
ADD	Rd, [Rs]	Add register-indirect to register	2	4
ADD	[Rd], Rs	Add register to register-indirect	2	4
ADD	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register	3	6
ADD	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset	3	6
ADD	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register	4	6
ADD	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset	4	6
ADD	Rd, [Rs+]	Add register-indirect with auto increment to register	2	5
ADD	[Rd+], Rs	Add register-indirect with auto increment to register	2	5
ADD	direct, Rs	Add register to memory	3	4
ADD	Rd, direct	Add memory to register	3	4
ADD	Rd, #data8	Add 8-bit immediate data to register	3	3
ADD	Rd, #data16	Add 16-bit immediate data to register	4	3
ADD	[Rd], #data8	Add 8-bit immediate data to register-indirect	3	4
ADD	[Rd], #data16	Add 16-bit immediate data to register-indirect	4	4
ADD	[Rd+], #data8	Add 8-bit immediate data to register-indirect with auto-increment	3	5
ADD	[Rd+], #data16	Add 16-bit immediate data to register-indirect with auto-increment	4	5
ADD	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset	4	6
ADD	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset	5	6
ADD	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADD	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset	6	6
ADD	direct, #data8	Add 8-bit immediate data to memory	4	4
ADD	direct, #data16	Add 16-bit immediate data to memory	5	4
ADDC	Rd, Rs	Add registers direct with carry	2	3
ADDC	Rd, [Rs]	Add register-indirect to register with carry	2	4
ADDC	[Rd], Rs	Add register to register-indirect with carry	2	4
ADDC	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register with carry	3	6
ADDC	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset with carry	3	6
ADDC	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register with carry	4	6
ADDC	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset with carry	4	6
ADDC	Rd, [Rs+]	Add register-indirect with auto increment to register with carry	2	5
ADDC	[Rd+], Rs	Add register-indirect with auto increment to register with carry	2	5
ADDC	direct, Rs	Add register to memory with carry	3	4
ADDC	Rd, direct	Add memory to register with carry	3	4
ADDC	Rd, #data8	Add 8-bit immediate data to register with carry	3	3
ADDC	Rd, #data16	Add 16-bit immediate data to register with carry	4	3
ADDC	[Rd], #data8	Add 16-bit immediate data to register-indirect with carry	3	4
ADDC	[Rd], #data16	Add 16-bit immediate data to register-indirect with carry	4	4
ADDC	[Rd+], #data8	Add 8-bit immediate data to register-indirect and auto-increment with carry	3	5
ADDC	[Rd+], #data16	Add 16-bit immediate data to register-indirect and auto-increment with carry	4	5
ADDC	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset and carry	4	6
ADDC	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset and carry	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADDC	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset and carry	5	6
ADDC	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset and carry	6	6
ADDC	direct, #data8	Add 8-bit immediate data to memory with carry	4	4
ADDC	direct, #data16	Add 16-bit immediate data to memory with carry	5	4
ADDS	Rd, #data4	Add 4-bit signed immediate data to register	2	3
ADDS	[Rd], #data4	Add 4-bit signed immediate data to register-indirect	2	4
ADDS	[Rd+], #data4	Add 4-bit signed immediate data to register-indirect with auto-increment	2	5
ADDS	[Rd+offset8], #data4	Add register-indirect with 8-bit offset to 4-bit signed immediate data	3	6
ADDS	[Rd+offset16], #data4	Add register-indirect with 16-bit offset to 4-bit signed immediate data	4	6
ADDS	direct, #data4	Add 4-bit signed immediate data to memory	3	4
ASL	Rd, Rs	Logical left shift destination register by the value in the source register	2	See Note1
ASL	Rd, #data4	Logical left shift register by the 4-bit immediate value	2	See Note1
ASR	Rd, Rs	Arithmetic shift right destination register by the count in the source	2	See Note1
ASR	Rd, #data4	Arithmetic shift right register by the 4-bit immediate count	2	See Note1
CMP	Rd, Rs	Compare destination and source registers	2	3
CMP	[Rd], Rs	Compare register with register-indirect	2	4
CMP	Rd, [Rs]	Compare register-indirect with register	2	4
CMP	[Rd+offset8], Rs	Compare register with register-indirect with 8-bit offset	3	6
CMP	[Rd+offset16], Rs	Compare register with register-indirect with 16-bit offset	4	6
CMP	Rd, [Rs+offset8]	Compare register-indirect with 8-bit offset with register	3	6
CMP	Rd,[Rs+offset16]	Compare register-indirect with 16-bit offset with register	4	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
CMP	Rd, [Rs+]	Compare auto-increment register-indirect with register	2	5
CMP	[Rd+], Rs	Compare register with auto-increment register-indirect	2	5
CMP	direct, Rs	Compare register with memory	3	4
CMP	Rd, direct	Compare memory with register	3	4
CMP	Rd, #data8	Compare 8-bit immediate data to register	3	3
CMP	Rd, #data16	Compare 16-bit immediate data to register	4	3
CMP	[Rd], #data8	Compare 8 -bit immediate data to register-indirect	3	4
CMP	[Rd], #data16	Compare 16-bit immediate data to register-indirect	4	4
CMP	[Rd+], #data8	Compare 8-bit immediate data to register-indirect with auto-increment	3	5
CMP	[Rd+], #data16	Compare 16-bit immediate data to register-indirect with auto-increment	4	5
CMP	[Rd+offset8], #data8	Compare 8-bit immediate data to register-indirect with 8-bit offset	4	6
CMP	[Rd+offset8], #data16	Compare 16-bit immediate data to register-indirect with 8-bit offset	5	6
CMP	[Rd+offset16], #data8	Compare 8-bit immediate data to register-indirect with 16-bit offset	5	6
CMP	[Rd+offset16], #data16	Compare 16-bit immediate data to register-indirect with 16-bit offset	6	6
CMP	direct, #data8	Compare 8-bit immediate data to memory	4	4
CMP	direct, #data16	Compare 16-bit immediate data to memory	5	4
DA	Rd	Decimal Adjust byte register	2	4
DIV.w	Rd, Rs	16x8 signed register divide	2	14
DIV.w	Rd, #data8	16x8 signed divide register with immediate word	3	14
DIV.d	Rd, Rs	32x16 signed double register divide	2	24
DIV.d	Rd, #data16	32x16 signed double register divide with immediate word	4	24
DIVU.b	Rd, Rs	8x8 unsigned register divide	2	12
DIVU.b	Rd, #data8	8X8 unsigned register divide with immediate byte	3	12

Table 6.5

Mnemonic		Description	Bytes	Clocks
DIVU.w	Rd, Rs	16X8 unsigned register divide	2	12
DIVU.w	Rd, #data8	16X8 unsigned register divide with immediate byte	3	12
DIVU.d	Rd, Rs	32X16 unsigned double register divide	2	22
DIVU.d	Rd, #data16	32X16 unsigned double register divide with immediate word	4	22
LEA	Rd, Rs+offset8	Load 16-bit effective address with 8-bit offset to register	3	3
LEA	Rd, Rs+offset16	Load 16-bit effective address with 16-bit offset to register	4	3
MUL.w	Rd, Rs	16X16 signed multiply of register contents	2	12
MUL.w	Rd, #data16	16X16 signed multiply 16-bit immediate data with register	4	12
MULU.b	Rd, Rs	8X8 unsigned multiply of register contents	2	12
MULU.b	Rd, #data8	8X8 unsigned multiply of 8-bit immediate data with register	3	12
MULU.w	Rd, Rs	16X16 unsigned register multiply	2	12
MULU.w	Rd, #data16	16X16 unsigned multiply 16-bit immediate data with register	4	12
NEG	Rd	Negate (twos complement) register	2	3
SEXT	Rd	Sign extend last operation to register	2	3
SUB	Rd, Rs	Subtract registers direct	2	3
SUB	Rd, [Rs]	Subtract register-indirect to register	2	4
SUB	[Rd], Rs	Subtract register to register-indirect	2	4
SUB	Rd, [Rs+offset8]	Subtract register-indirect with 8-bit offset to register	3	6
SUB	[Rd+offset8], Rs	Subtract register to register-indirect with 8-bit offset	3	6
SUB	Rd, [Rs+offset16]	Subtract register-indirect with 16-bit offset to register	4	6
SUB	[Rd+offset16], Rs	Subtract register to register-indirect with 16-bit offset	4	6
SUB	Rd, [Rs+]	Subtract register-indirect with auto increment to register	2	5
SUB	[Rd+], Rs	Subtract register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUB	direct, Rs	Subtract register to memory	3	4
SUB	Rd, direct	Subtract memory to register	3	4
SUB	Rd, #data8	Subtract 8-bit immediate data to register	3	3
SUB	Rd, #data16	Subtract 16-bit immediate data to register	4	3
SUB	[Rd], #data8	Subtract 8-bit immediate data to register-indirect	3	4
SUB	[Rd], #data16	Subtract 16-bit immediate data to register-indirect	4	4
SUB	[Rd+], #data8	Subtract 8-bit immediate data to register-indirect with auto-increment	3	5
SUB	[Rd+], #data16	Subtract 16-bit immediate data to register-indirect with auto-increment	4	5
SUB	[Rd+offset8], #data8	Subtract 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUB	[Rd+offset8], #data16	Subtract 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUB	[Rd+offset16], #data8	Subtract 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUB	[Rd+offset16], #data16	Subtract 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUB	direct, #data8	Subtract 8-bit immediate data to memory	4	4
SUB	direct, #data16	Subtract 16-bit immediate data to memory	5	4
SUBB	Rd, Rs	Subtract with borrow registers direct	2	3
SUBB	Rd, [Rs]	Subtract with borrow register-indirect to register	2	4
SUBB	[Rd], Rs	Subtract with borrow register to register-indirect	2	4
SUBB	Rd, [Rs+offset8]	Subtract with borrow register-indirect with 8-bit offset to register	3	6
SUBB	[Rd+offset8], Rs	Subtract with borrow register to register-indirect with 8-bit offset	3	6
SUBB	Rd, [Rs+offset16]	Subtract with borrow register-indirect with 16-bit offset to register	4	6
SUBB	[Rd+offset16], Rs	Subtract with borrow register to register-indirect with 16-bit offset	4	6
SUBB	Rd, [Rs+]	Subtract with borrow register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUBB	[Rd+], Rs	Subtract with borrow register-indirect with auto increment to register	2	5
SUBB	direct, Rs	Subtract with borrow register to memory	3	4
SUBB	Rd, direct	Subtract with borrow memory to register	3	4
SUBB	Rd, #data8	Subtract with borrow 8-bit immediate data to register	3	3
SUBB	Rd, #data16	Subtract with borrow 16-bit immediate data to register	4	3
SUBB	[Rd], #data8	Subtract with borrow 8-bit immediate data to register-indirect	3	4
SUBB	[Rd], #data16	Subtract with borrow 16-bit immediate data to register-indirect	4	4
SUBB	[Rd+], #data8	Subtract with borrow 8-bit immediate data to register-indirect with auto-increment	3	5
SUBB	[Rd+], #data16	Subtract with borrow 16-bit immediate data to register-indirect with auto-increment	4	5
SUBB	[Rd+offset8], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUBB	[Rd+offset8], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUBB	[Rd+offset16], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUBB	[Rd+offset16], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUBB	direct, #data8	Subtract with borrow 8-bit immediate data to memory	4	4
SUBB	direct, #data16	Subtract with borrow 16-bit immediate data to memory	5	4
Logical Operations				
AND	Rd, Rs	Logical AND registers direct	2	3
AND	Rd, [Rs]	Logical AND register-indirect to register	2	4
AND	[Rd], Rs	Logical AND register to register-indirect	2	4
AND	Rd, [Rs+offset8]	Logical AND register-indirect with 8-bit offset to register	3	6
AND	[Rd+offset8], Rs	Logical AND register to register-indirect with 8-bit offset	3	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
AND	Rd, [Rs+offset16]	Logical AND register-indirect with 16-bit offset to register	4	6
AND	[Rd+offset16], Rs	Logical AND register to register-indirect with 16-bit offset	4	6
AND	Rd, [Rs+]	Logical AND register-indirect with auto increment to register	2	5
AND	[Rd+], Rs	Logical AND register-indirect with auto increment to register	2	5
AND	direct, Rs	Logical AND register to memory	3	4
AND	Rd, direct	Logical AND memory to register	3	4
AND	Rd, #data8	Logical AND 8-bit immediate data to register	3	3
AND	Rd, #data16	Logical AND 16-bit immediate data to register	4	3
AND	[Rd], #data8	Logical AND 8-bit immediate data to register-indirect	3	4
AND	[Rd], #data16	Logical AND16-bit immediate data to register-indirect	4	4
AND	[Rd+], #data8	Logical AND 8-bit immediate data to register-indirect and auto-increment	3	5
AND	[Rd+], #data16	Logical AND16-bit immediate data to register-indirect and auto-increment	4	5
AND	[Rd+offset8], #data8	Logical AND8-bit immediate data to register-indirect with 8-bit offset	4	6
AND	[Rd+offset8], #data16	Logical AND16-bit immediate data to register-indirect with 8-bit offset	5	6
AND	[Rd+offset16], #data8	Logical AND8-bit immediate data to register-indirect with 16-bit offset	5	6
AND	[Rd+offset16], #data16	Logical AND16-bit immediate data to register-indirect with 16-bit offset	6	6
AND	direct, #data8	Logical AND 8-bit immediate data to memory	4	4
AND	direct, #data16	Logical AND16-bit immediate data to memory	5	4
CPL	Rd	Complement (ones complement) register	2	3
LSR	Rd, Rs	Logical right shift destination register by the value in the source register	2	See Note 1
LSR	Rd, #data4	Logical right shift register by the 4-bit immediate value	2	See Note 1
NORM	Rd, Rs	Logical shift left destination register by the value in the source register until MSB set	2	See Note 1

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	Rd, Rs	Logical OR registers	2	3
OR	Rd, [Rs]	Logical OR register-indirect to register	2	4
OR	[Rd], Rs	Logical OR register to register-indirect	2	4
OR	Rd, [Rs+offset8]	Logical OR register-indirect with 8-bit offset to register	3	6
OR	[Rd+offset8], Rs	Logical OR register to register-indirect with 8-bit offset	3	6
OR	Rd, [Rs+offset16]	Logical OR register-indirect with 16-bit offset to register	4	6
OR	[Rd+offset16], Rs	Logical OR register to register-indirect with 16-bit offset	4	6
OR	Rd, [Rs+]	Logical OR register-indirect with auto increment to register	2	5
OR	[Rd+], Rs	Logical OR register-indirect with auto increment to register	2	5
OR	direct, Rs	Logical OR register to memory	3	4
OR	Rd, direct	Logical OR memory to register	3	4
OR	Rd, #data8	Logical OR 8-bit immediate data to register	3	3
OR	Rd, #data16	Logical OR 16-bit immediate data to register	4	3
OR	[Rd], #data8	Logical OR 8-bit immediate data to register-indirect	3	4
OR	[Rd], #data16	Logical OR 16-bit immediate data to register-indirect	4	4
OR	[Rd+], #data8	Logical OR 8-bit immediate data to register-indirect with auto-increment	3	5
OR	[Rd+], #data16	Logical OR 16-bit immediate data to register-indirect with auto-increment	4	5
OR	[Rd+offset8], #data8	Logical OR 8-bit immediate data to register-indirect with 8-bit offset	4	6
OR	[Rd+offset8], #data16	Logical OR 16-bit immediate data to register-indirect with 8-bit offset	5	6
OR	[Rd+offset16], #data8	Logical OR 8-bit immediate data to register-indirect with 16-bit offset	5	6
OR	[Rd+offset16], #data16	Logical OR 16-bit immediate data to register-indirect with 16-bit offset	6	6
OR	direct, #data8	Logical OR 8-bit immediate data to memory	4	4

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	direct, #data16	Logical OR16-bit immediate data to memory	5	4
RL	Rd, #data4	Rotate left register by the 4-bit immediate value	2	See Note 1
RLC	Rd, #data4	Rotate left register though carry by the 4-bit immediate value	2	See Note 1
RR	Rd, #data4	Rotate right register by the 4-bit immediate value	2	See Note 1
RRC	Rd, #data4	Rotate right register though carry by the 4-bit immediate value	2	See Note 1
XOR	Rd, Rs	Logical XOR registers	2	3
XOR	Rd, [Rs]	Logical XOR register-indirect to register	2	4
XOR	[Rd], Rs	Logical XOR register to register-indirect	2	4
XOR	Rd, [Rs+offset8]	Logical XOR register-indirect with 8-bit offset to register	3	6
XOR	[Rd+offset8], Rs	Logical XOR register to register-indirect with 8-bit offset	3	6
XOR	Rd, [Rs+offset16]	Logical XOR register-indirect with 16-bit offset to register	4	6
XOR	[Rd+offset16], Rs	Logical XOR register to register-indirect with 16-bit offset	4	6
XOR	Rd, [Rs+]	Logical XOR register-indirect with auto increment to register	2	5
XOR	[Rd+], Rs	Logical XOR register-indirect with auto increment to register	2	5
XOR	direct, Rs	Logical XOR register to memory	3	4
XOR	Rd, direct	Logical XOR memory to register	3	4
XOR	Rd, #data8	Logical XOR 8-bit immediate data to register	3	3
XOR	Rd, #data16	Logical XOR 16-bit immediate data to register	4	3
XOR	[Rd], #data8	Logical XOR 8-bit immediate data to register-indirect	3	4
XOR	[Rd], #data16	Logical XOR 16-bit immediate data to register-indirect	4	4
XOR	[Rd+], #data8	Logical XOR 8-bit immediate data to register-indirect with auto-increment	3	5
XOR	[Rd+], #data16	Logical XOR 16-bit immediate data to register-indirect with auto-increment	4	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
XOR	[Rd+offset8], #data8	Logical XOR 8-bit immediate data to register-indirect with 8-bit offset	4	6
XOR	[Rd+offset8], #data16	Logical XOR 16-bit immediate data to register-indirect with 8-bit offset	5	6
XOR	[Rd+offset16], #data8	Logical XOR 8-bit immediate data to register-indirect with 16-bit offset	5	6
XOR	[Rd+offset16], #data16	Logical XOR 16-bit immediate data to register-indirect with 16-bit offset	6	6
XOR	direct, #data8	Logical XOR 8-bit immediate data to memory	4	4
XOR	direct, #data16	Logical XOR16-bit immediate data to memory	5	4
Data transfer				
MOV	Rd, Rs	Move register to register	2	3
MOV	Rd, [Rs]	Move register-indirect to register	2	3
MOV	[Rd], Rs	Move register to register-indirect	2	3
MOV	Rd, [Rs+offset8]	Move register-indirect with 8-bit offset to register	3	5
MOV	[Rd+offset8], Rs	Move register to register-indirect with 8-bit offset	3	5
MOV	Rd, [Rs+offset16]	Move register-indirect with 16-bit offset to register	4	5
MOV	[Rd+offset16], Rs	Move register to register-indirect with 16-bit offset	4	5
MOV	Rd, [Rs+]	Move register-indirect with auto increment to register	2	4
MOV	[Rd+], Rs	Move register-indirect with auto increment to register	2	4
MOV	direct, Rs	Move register to memory	3	4
MOV	Rd, direct	Move memory to register	3	4
MOV	[Rd+], [Rs+]	Move register-indirect to register-indirect, both pointers auto-incremented	2	6
MOV	direct, [Rs]	Move register-indirect to memory	3	4
MOV	[Rd], direct	Move memory to register-indirect	3	4
MOV	Rd, #data8	Move 8-bit immediate data to register	3	3
MOV	Rd, #data16	Move 16-bit immediate data to register	4	3

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOV	[Rd], #data8	Move 16-bit immediate data to register-indirect	3	3
MOV	[Rd], #data16	Move 16-bit immediate data to register-indirect	4	3
MOV	[Rd+], #data8	Move 8-bit immediate data to register-indirect with auto-increment	3	4
MOV	[Rd+], #data16	Move 16-bit immediate data to register-indirect with auto-increment	4	4
MOV	[Rd+offset8], #data8	Move 8-bit immediate data to register-indirect with 8-bit offset	4	5
MOV	[Rd+offset8], #data16	Move 16-bit immediate data to register-indirect with 8-bit offset	5	5
MOV	[Rd+offset16], #data8	Move 8-bit immediate data to register-indirect with 16-bit offset	5	5
MOV	[Rd+offset16], #data16	Move 16-bit immediate data to register-indirect with 16-bit offset	6	5
MOV	direct, #data8	Move 8-bit immediate data to memory	4	3
MOV	direct, #data16	Move 16-bit immediate data to memory	5	3
MOV	direct, direct	Move memory to memory	4	4
MOV	Rd, USP	Move User Stack Pointer to register (system mode only)	2	3
MOV	USP, Rs	Move register to User Stack Pointer (system mode only)	2	3
MOVC	Rd, [Rs+]	Move data from WS:Rs address of code memory to register with auto-increment	2	4
MOVC	A, [A+DPTR]	Move data from code memory to the accumulator indirect with DPTR	2	6
MOVC	A, [A+PC]	Move data from code memory to the accumulator indirect with PC	2	6
MOVS	Rd, #data4	Move 4-bit sign-extended immediate data to register	2	3
MOVS	[Rd], #data4	Move 4-bit sign-extended immediate data to register-indirect	2	3
MOVS	[Rd+], #data4	Move 4-bit sign-extended immediate data to register-indirect with auto-increment	2	4
MOVS	[Rd+offset8], #data4	Move register-indirect with 8-bit offset to 4-bit sign-extended immediate data	3	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOVS	[Rd+offset16], #data4	Move register-indirect with 16-bit offset to 4-bit sign-extended immediate data	4	5
MOVS	direct, #data4	Move 4-bit sign-extended immediate data to memory	3	3
MOVX	Rd, [Rs]	Move external data from memory to register	2	6
MOVX	[Rd], Rs	Move external data from register to memory	2	6
PUSH	direct	Push the memory content (byte/word) onto the current stack	3	5
PUSHU	direct	Push the memory content (byte/word) onto the user stack	3	5
PUSH	Rlist	Push multiple registers (byte/word) onto the current stack	2	See Note 2
PUSHU	Rlist	Push multiple registers (byte/word)from the user stack	2	See Note 2
POP	direct	Pop the memory content (byte/word) from the current stack	3	5
POPU	direct	Pop the memory content (byte/word) from the user stack	3	5
POP	Rlist	Pop multiple registers (byte/word) from the current stack	2	See Note 3
POPU	Rlist	Pop multiple registers (byte/word) from the user stack	2	See Note 3
XCH	Rd, Rs	Exchange contents of two registers	2	5
XCH	Rd, [Rs]	Exchange contents of a register-indirect address with a register	2	6
XCH	Rd, direct	Exchange contents of memory with a register	3	6
Program Branching				
BCC	rel8	Branch if the carry flag is clear	2	6t/3nt
BCS	rel8	Branch if the carry flag is set	2	6t/3nt
BEQ	rel8	Branch if the zero flag is set	2	6t/3nt
BNE	rel8	Branch if the zero flag is not set	2	6t/3nt
BG	rel8	Branch if greater than (unsigned)	2	6t/3nt
BGE	rel8	Branch if greater than or equal to (signed)	2	6t/3nt
BGT	rel8	Branch if greater than (signed)	2	6t/3nt

Table 6.5

Mnemonic		Description	Bytes	Clocks
BL	rel8	Branch if less than or equal to (unsigned)	2	6t/3nt
BLE	rel8	Branch if less than or equal to (signed)	2	6t/3nt
BLT	rel8	Branch if less than (signed)	2	6t/3nt
BMI	rel8	Branch if the negative flag is set	2	6t/3nt
BPL	rel8	Branch if the negative flag is clear	2	6t/3nt
BNV	rel8	Branch if overflow flag is clear	2	6t/3nt
BOV	rel8	Branch if overflow flag is set	2	6t/3nt
BR	rel8	Short unconditional branch	2	6
CALL	[Rs]	Subroutine call indirect with a register	2	8/5(PZ)
CALL	rel16	Relative call (+/- 64K)	3	7/4(PZ)
CJNE	Rd,direct,rel8	Compare direct byte to register and jump if not equal	4	10t/7nt
CJNE	Rd,#data8,rel8	Compare immediate byte to register and jump if not equal	4	9t/6nt
CJNE	Rd,#data16,rel8	Compare immediate word to register and jump if not equal	5	9t/6nt
CJNE	[Rd],#data8,rel8	Compare immediate word to register-indirect and jump if not equal	4	10t/7nt
CJNE	[Rd],#data16,rel8	Compare immediate word to register-indirect and jump if not equal	5	10t/7nt
DJNZ	Rd,rel8	Decrement register and jump if not zero	3	8t/5nt
DJNZ	direct,rel8	Decrement memory and jump if not zero	4	9t/5nt
FCALL	addr24	Far call (anywhere in the 24-bit address space)	4	12/8 (PZ)
FJMP	addr24	Far jump (anywhere in the 24-bit address space)	4	6
JB	bit,rel8	Jump if bit set	4	10t/6nt
JBC	bit,rel8	Jump if bit set and then clear the bit	4	11t/7nt
JMP	rel16	Long unconditional branch	3	6
JMP	[Rs]	Jump indirect to the address in the register (64K)	2	7
JMP	[A+DPTR]	Jump indirect relative to the DPTR	2	5
JMP	[[Rs+]]	Jump double-indirect to the address (pointer to a pointer)	2	8

Table 6.5

Mnemonic		Description	Bytes	Clocks
JNB	bit,rel8	Jump if bit not set	4	10t/6nt
JNZ	rel8	Jump if accumulator not equal zero	2	6t/3nt
JZ	rel8	Jump if accumulator equals zero	2	6t/3nt
NOP		No operation	1	3
RET		Return from subroutine	2	8/6(PZ)
RETI		Return from interrupt	2	10/ 8(PZ)
Bit Manipulation				
ANL	C, bit	Logical AND bit to carry	3	4
ANL	C, /bit	Logical AND complement of a bit to carry	3	4
CLR	bit	Clear bit	3	4
MOV	C, bit	Move bit to the carry flag	3	4
MOV	bit, C	Move carry to bit	3	4
ORL	C, bit	Logical OR a bit to carry	3	4
ORL	C, /bit	Logical OR complement of a bit to carry	3	4
SETB	bit	Sets the bit specified	3	4
Exception / Trap				
BKPT		Cause the breakpoint trap to be executed.	1	23/ 19(PZ)
RESET		Causes a hardware Reset, identical to an external Reset	2	18
TRAP	#data4	Causes 1 of 16 hardware traps to be executed	2	23/ 19(PZ)

Note 1: For 8 and 16 bit shifts, it is 4+1 per additional two bits. For 32-bit shifts, it is 6+1 per additional two bits.
Note 2: 3 clocks per register pushed.
Note 3: 4 clocks for the first register and two clocks for each additional register.

Syntax: ADD dest, source

Operation: dest <- src + dest

Description: Performs a twos complement binary addition of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

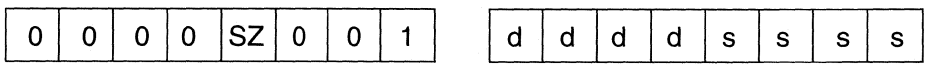
Note: If used with write to PSWL, takes precedence to flag updates

Sizes: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

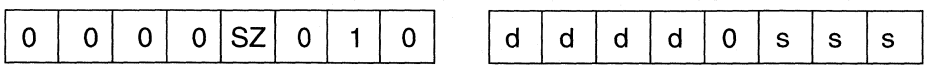
ADD Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:



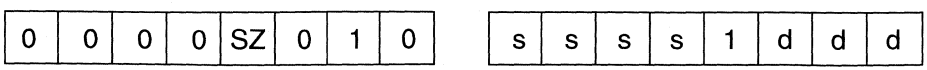
ADD Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:



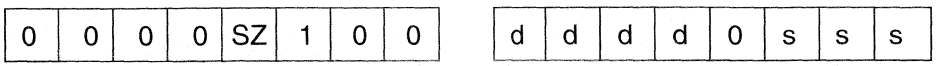
ADD [Rd], Rs

Bytes: 2
Clocks: 4
Operation: (WS:Rd) <-- (WS:Rd) + (Rs)
Encoding:



ADD Rd, [Rs+offset8]

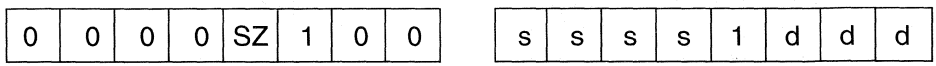
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

ADD [Rd+offset8], Rs

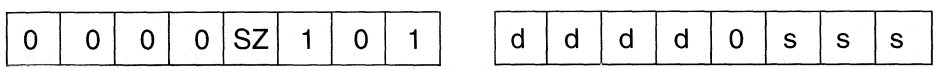
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:



byte 3: offset8

ADD Rd, [Rs+offset16]

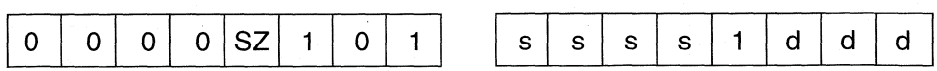
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADD [Rd+offset16], Rs

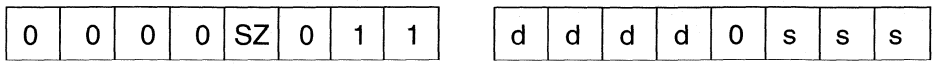
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

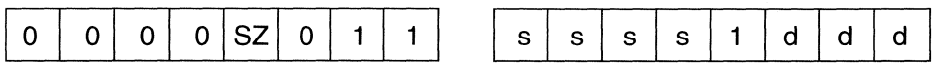
ADD Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



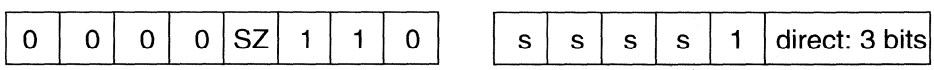
ADD [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



ADD direct, Rs

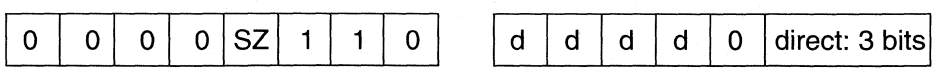
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:



byte 3: lower 8 bits of direct

ADD Rd, direct

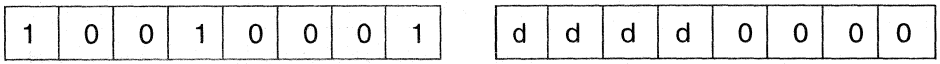
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:



byte 3: lower 8 bits of direct

ADD Rd, #data8

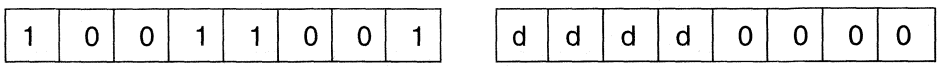
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:



byte 3: #data8

ADD Rd, #data16

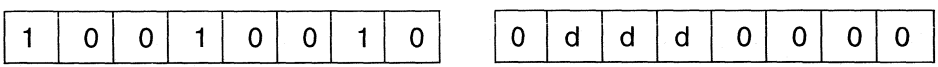
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd], #data8

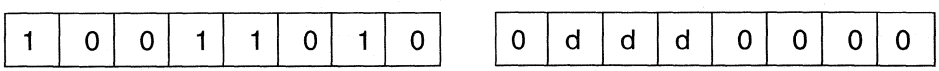
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:



byte 3: #data8

ADD [Rd], #data16

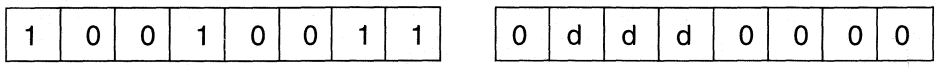
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+], #data8

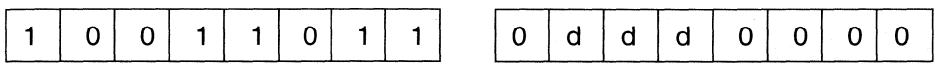
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

ADD [Rd+], #data16

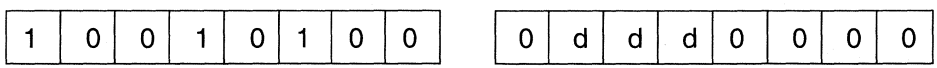
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+offset8], #data8

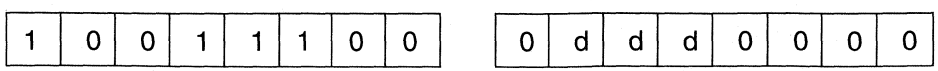
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:



byte 3: offset8
byte 4: #data8

ADD [Rd+offset8], #data16

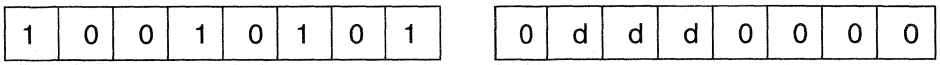
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADD [Rd+offset16], #data8

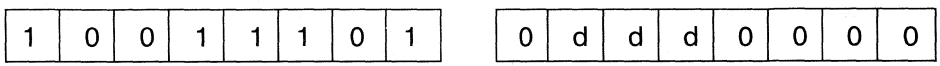
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

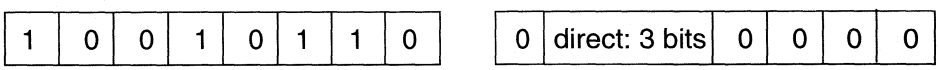
ADD [Rd+offset16], #data16

Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

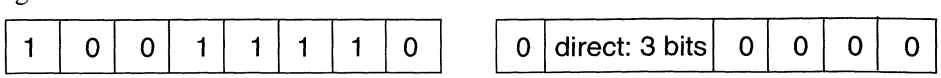
ADD direct, #data8
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

ADD direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ADDC dest, source

Operation: dest <- dest + src + C

Description: Performs a two's complement binary addition of the source operand and the previously generated carry bit with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is one (C=1), the result is greater than the sum of the operands; if it is zero (C=0), the result is the exact sum.

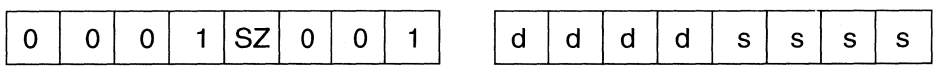
This form of addition is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then ADDC is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

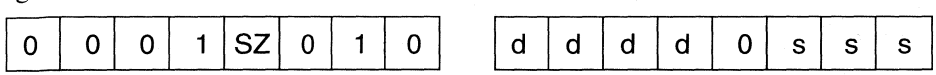
ADDC Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs) + (C)
Encoding:



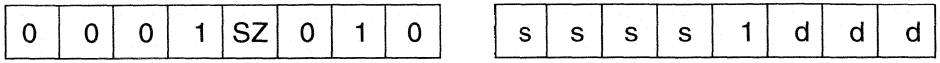
ADDC Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
Encoding:



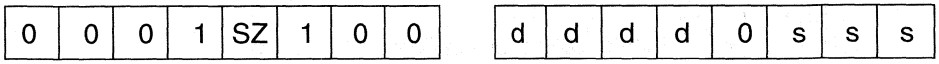
ADDC [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
Encoding:



ADDC Rd, [Rs+offset8]

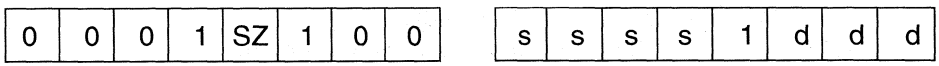
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8) + (C)
Encoding:



byte 3: offset8

ADDC [Rd+offset8], Rs

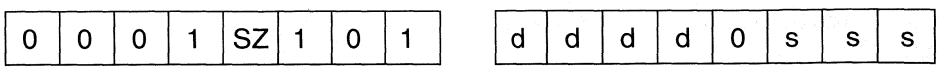
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs) + (C)
Encoding:



byte 3: offset8

ADDC Rd, [Rs+offset16]

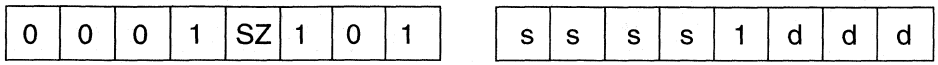
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16) + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC [Rd+offset16], Rs

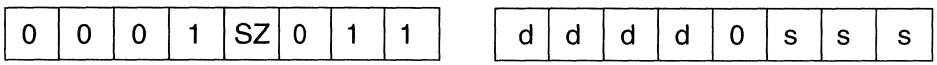
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs) + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

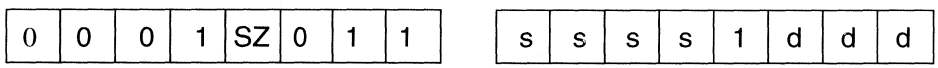
ADDC Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



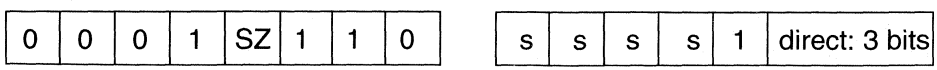
ADDC [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



ADDC direct, Rs

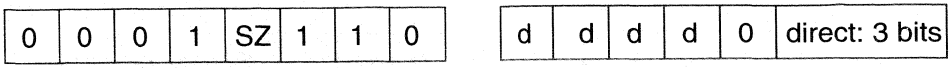
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs) + (C)
Encoding:



byte 3: lower 8 bits of direct

ADDC Rd, direct

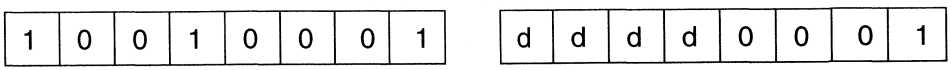
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct) + (C)
Encoding:



byte 3: lower 8 bits of direct

ADDC Rd, #data8

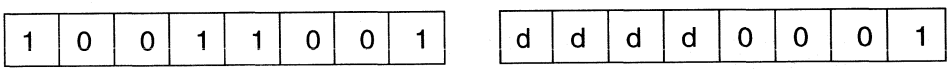
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8 + (C)
Encoding:



byte 3: #data8

ADDC Rd, #data16

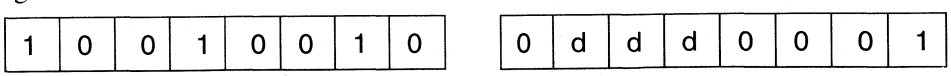
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16 + (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd], #data8

Bytes: 3
Clocks : 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
Encoding:

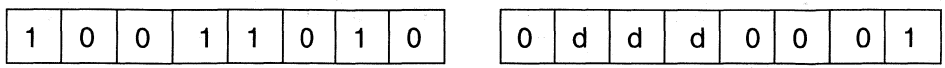


byte 3: #data8

ADDC [Rd], #data16

Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)

Encoding:

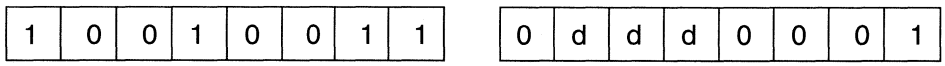


byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+], #data8

Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
(Rd) <-- (Rd) + 1

Encoding:

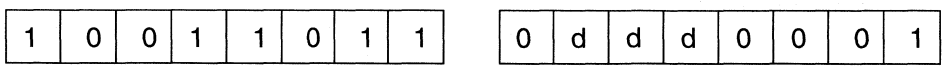


byte 3: #data8

ADDC [Rd+], #data16

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)
(Rd) <-- (Rd) + 2

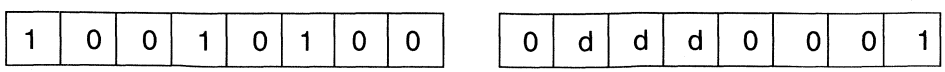
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset8], #data8

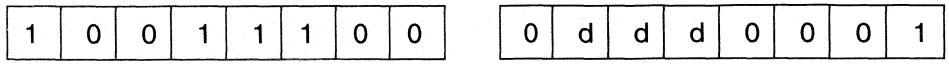
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8 + (C)
Encoding:



byte 3: offset8
byte 4: #data8

ADDC [Rd+offset8], #data16

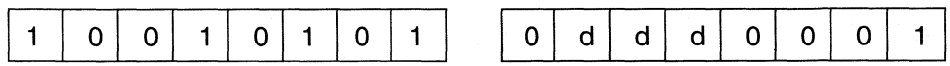
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16 + (C)
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset16], #data8

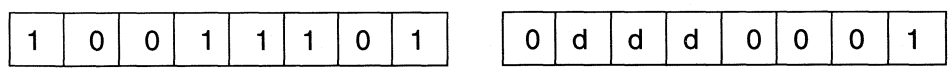
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8 + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

ADDC [Rd+offset16], #data16

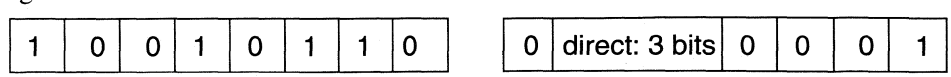
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16 + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

ADDC direct, #data8

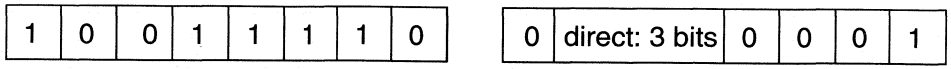
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8 + (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

ADDC direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16 + (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ADDS dest, #value

Operation: dest <- dest + #data4

Description: Four bits of signed immediate data are added to the destination. The immediate data is sign-extended to the proper size, then added to the variable specified by the destination operand, which may be either a byte or a word. The immediate data range is +7 to -8. This instruction is used primarily to increment or decrement pointers and counters.

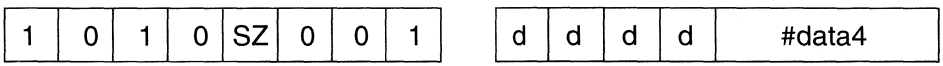
Size: Byte-Byte, Word-Word

Flags Updated: N, Z

(Note: the C and AC flags must not be updated by ADDS since this instruction is used to replace the 80C51 INC and DEC instructions, which do not update the flags.)

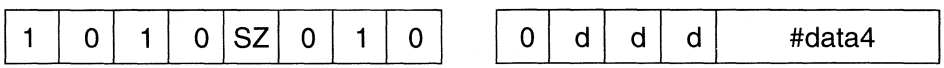
ADDS Rd, #data4

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + #data4
Encoding:



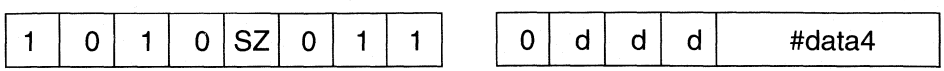
ADDS [Rd], #data4

Bytes: 2
Clocks: 4
Operation:((WS:Rd)) <-- ((WS:Rd)) + #data4
Encoding:



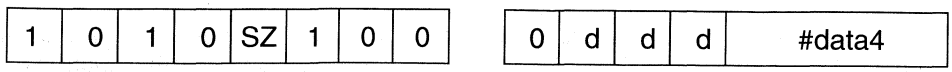
ADDS [Rd+], #data4

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



ADDS [Rd+offset8], #data4

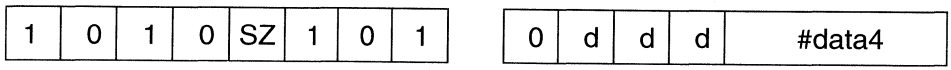
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data4
Encoding:



byte 3: offset8

ADDS [Rd+offset16], #data4

Bytes: 4
Clocks: 6
Operation:((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data4
Encoding:

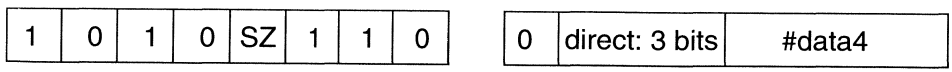


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDS direct, #data4

Bytes: 3
Clocks: 4
Operation:(direct) <-- (direct) + #data4

Encoding:



byte 3: lower 8 bits of direct

Syntax: AND dest, src

Operation: dest <- dest AND src

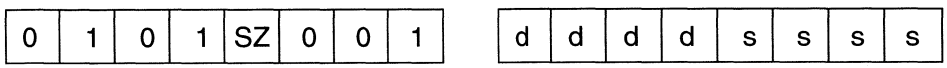
Description: Bitwise logical AND the contents of the source to the destination. The byte or word specified by the source operand is logically ANDed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

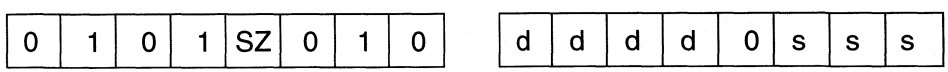
AND Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) • (Rs)
Encoding:



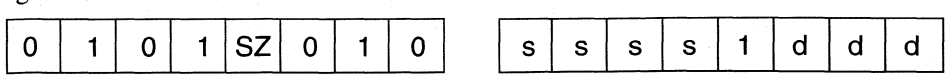
AND Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) • ((WS:Rs))
Encoding:



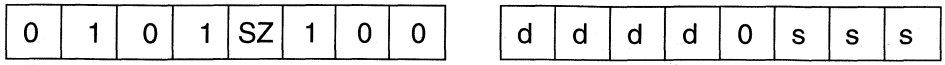
AND [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
Encoding:



AND Rd, [Rs+offset8]

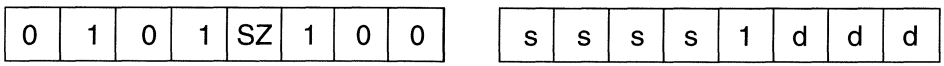
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

AND [Rd+offset8], Rs

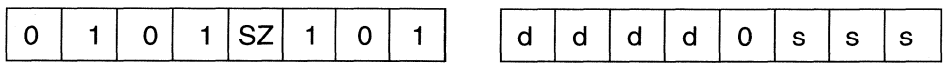
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • (Rs)
Encoding:



byte 3: offset8

AND Rd, [Rs+offset16]

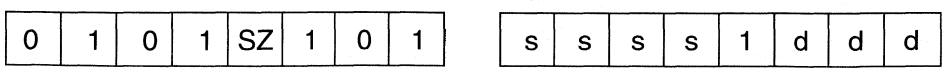
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND [Rd+offset16], Rs

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • (Rs)
Encoding:

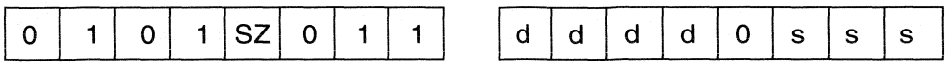


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND Rd, [Rs+]

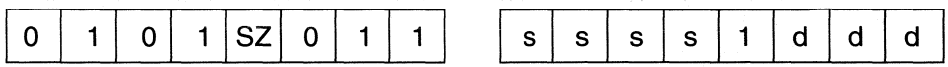
Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) • ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:



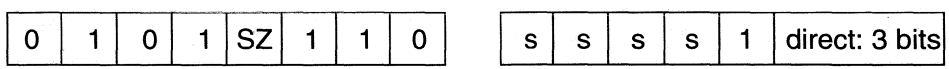
AND [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



AND direct, Rs

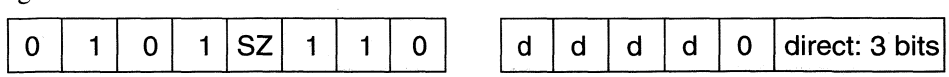
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) • (Rs)
Encoding:



byte 3: lower 8 bits of direct

AND Rd, direct

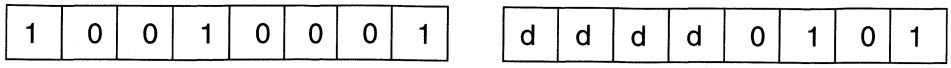
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) • (direct)
Encoding:



byte 3: lower 8 bits of direct

AND Rd, #data8

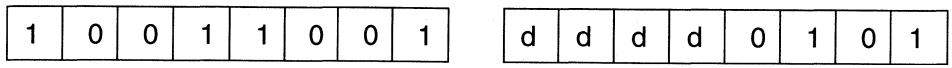
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) • #data8
Encoding:



byte 3: #data8

AND Rd, #data16

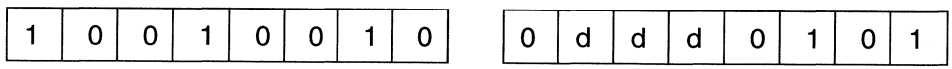
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) • #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd], #data8

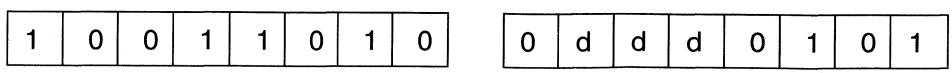
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
Encoding:



byte 3: #data8

AND [Rd], #data16

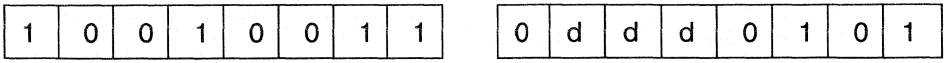
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+], #data8

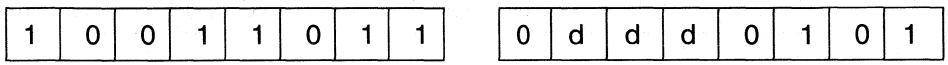
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

AND [Rd+], #data16

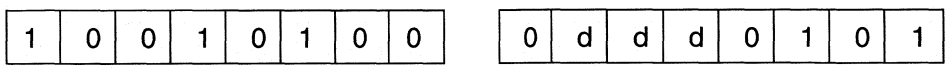
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+offset8], #data8

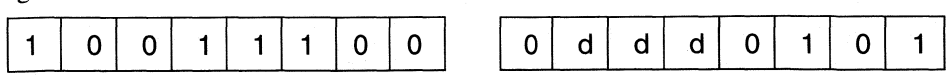
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data8
Encoding:



byte 3: offset8
byte 4: #data8

AND [Rd+offset8], #data16

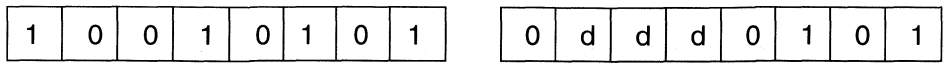
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

AND [Rd+offset16], #data8

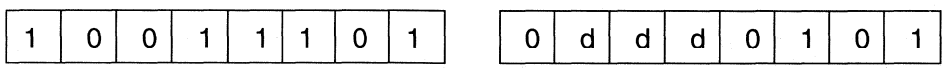
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

AND [Rd+offset16], #data16

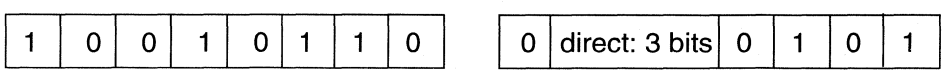
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

AND direct, #data8

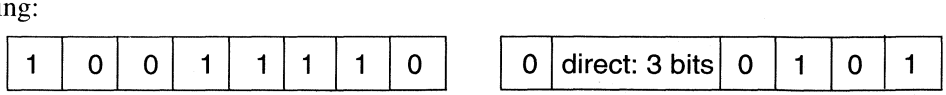
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) • #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

AND direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) • #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ANL C, bit

Operation: C <- C (AND) Bit

Description: Read the specified bit and logically AND it to the Carry flag.

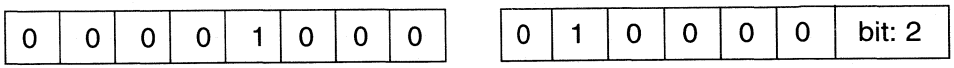
Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: ANL C, /bit

Operation: Carry <- C (AND) $\overline{bit}$

Description: Read the specified bit, complement it, and logically AND it to the Carry flag.

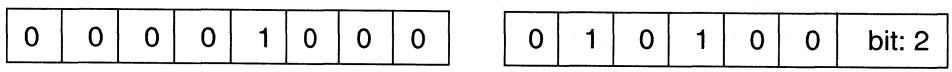
Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: ASL dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.msb)
(dest.bit n+1) <- (dest.bit n)
count = count-1
if sign change during shift,
(V) <- 1
End While

Description:
If the count operand is greater than 0, the destination operand is logically shifted left by the number of bits specified by the count operand. The Low-order bits shifted in are zero-filled and the high-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.

The count operand could be:
- An immediate value (#data4 or #data5)
- A Register (Only 5 bits are used to implement up to 31 bit shifts)

The count is a positive value which may be from 1 to 31 and the destination operand is a signed integer (twos complement form).The destination operand (data size) may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.The count operand is not affected by the operation.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.

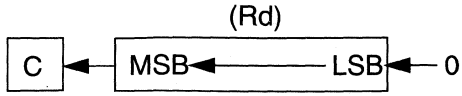
Size: Byte, word, and double word

Flags Updated: C, V, N, Z

Note: The V flag is set if the sign changes at any time during the shift operation and remains set until the end of the shift operation i.e., the V flag does not get cleared even if the sign reverts to its original state because of continued shifts within the same instruction. ASL clears the V flag if the condition to set it does not occur.

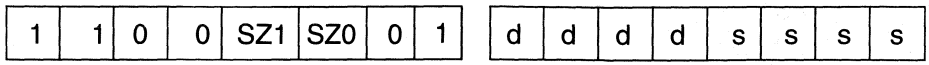
ASL Rd, Rs

Operation:



Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

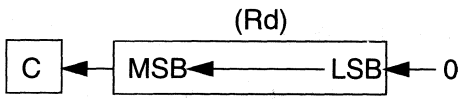
Encoding:



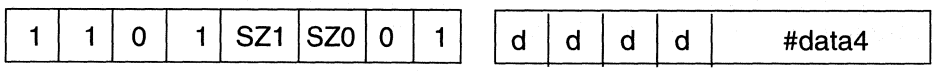
ASL Rd, #data4
Rd,#data5

Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

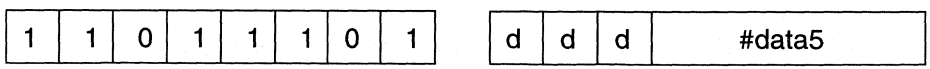
Operation:



Encoding: (for byte and word data sizes)



(for double word data size)



Note: SZ1/SZ0 = 00 : byte operation; SZ1/SZ0 = 10 : word operation; SZ1/SZ0 = 11 : double word operation.

Syntax: ASR dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
dest.msb <- Sign bit
count = count-1
End While

Description:
If the count operand is greater than 0, the destination operand is logically shifted right by the number of bits specified by the count operand. The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed. To preserve the sign of the original operand, the MSBs of the result are sign-extended with the sign bit.

The count operand could be:
- An immediate value (#data4/5)
- A Register (Only 5 bits are used to implement up to 31 bit shifts)

The count operand could be an immediate value or a register. The count is a positive value which may be from 0 to 31 and the destination operand is a signed integer. The count operand is not affected by the operation. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

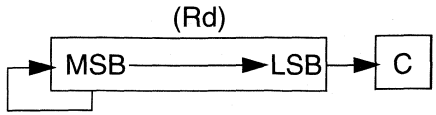
Size: Byte, Word, Double Word

Flags Updated: C, N, Z

ASR Rd, Rs

Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Operation:

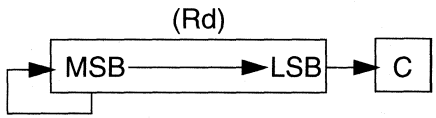


Encoding:

1	1	0	0	SZ1	SZ0	1	0	d	d	d	d	s	s	s	s

ASR Rd, #data4
Rd,#data5

Operation:



Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding: (for byte and word data sizes)

1	1	0	1	SZ1	SZ0	1	0	d	d	d	d	#data4

(for double word data size)

1	1	0	1	SZ1	SZ0	1	0	d	d	d	#data5

Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: BCC rel8

Operation:

(PC) <-- (PC) + 2
if (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) did not generate a carry (the carry flag contains a 0). If Carry is clear, the program execution branches at the location of the PC, plus the specified displacement, rel8. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

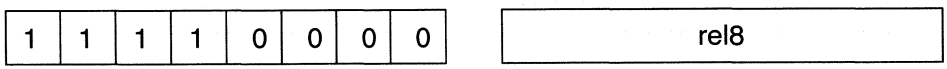
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6 (t) / 3 (nt)

Encoding:



Syntax: BCS rel8

Operation:

(PC) <-- (PC) + 2
if (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) generated a carry (the carry flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

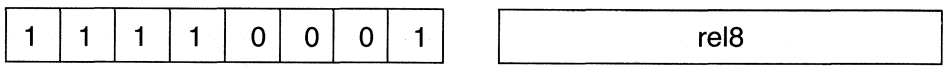
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BEQ rel8

Operation:

(PC) <-- (PC) + 2
if (Z) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a result of zero (the Z flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

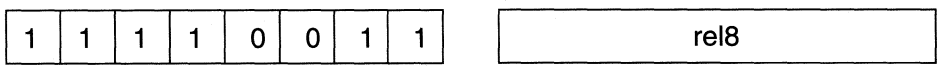
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BG rel8

Operation: (PC) <-- (PC) + 2
if (Z) OR (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

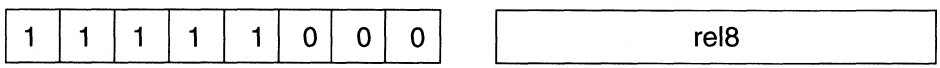
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BGE rel8

Operation: (PC) <-- (PC) + 2
if (N) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

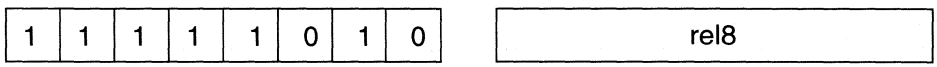
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BGT **Branch if greater than (signed)**

Syntax: BGT rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

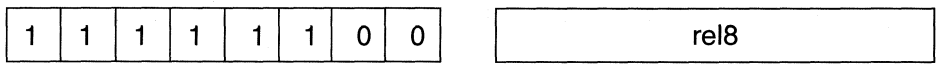
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BKPT

Operation:
(PC) <-- (PC) + 1
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (bkpt vector)
(PC.15-0) <-- code memory (bkpt vector)
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes a breakpoint trap. The breakpoint trap acts like an immediate interrupt, using a vector to call a specific piece of code that will be executed in system mode. This instruction is intended for use in emulator systems to provide a simple method of implementing hardware breakpoints.

For a breakpoint to work properly under all conditions, it must have an instruction length no greater than the smallest other instruction on the processor, in this case the one byte NOP. This requirement exists because a breakpoint may be inserted in place of a NOP that is followed by another instruction that is branched to or otherwise executed without going through the breakpoint. If the breakpoint instruction were longer than the NOP, it would corrupt the next instruction in sequence if that instruction were executed.

The opcode for the breakpoint instruction is specifically assigned to be all ones (FFh). This is so that un-programmed EPROM code memory will contain breakpoints. Similarly, the NOP instruction is assigned to opcode 00 so that both "blank" code states map to innocuous instructions.

Size: None

Flags Updated: none[5]

Bytes: 1
Clocks: 23/19 (PZ)

Encoding:

1	1	1	1	1	1	1	1

5. All flags are affected during the PSW load from the vector table. It is possible that these flags are restored by the debugger, but does not have to be the case.

Syntax: BL rel8

Operation: (PC) <-- (PC) + 2
if (Z) OR (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

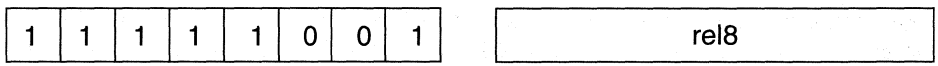
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BLE rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

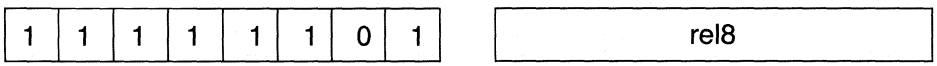
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BLT rel8

Operation: (PC) <-- (PC) + 2
if (N) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

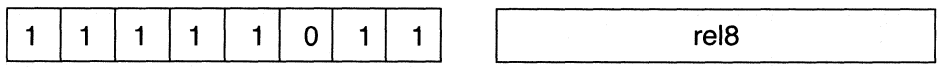
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BMI rel8

Operation: (PC) <-- (PC) + 2
if (N) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is less than 0 (the N flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

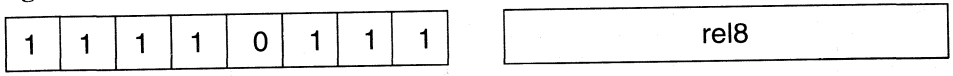
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BNE rel8

Operation: (PC) <-- (PC) + 2
if (Z) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a non-zero result (the Z flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

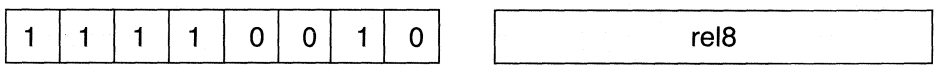
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BNV rel8

Operation: (PC) <-- (PC) + 2
if (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) did not generate an overflow (The V flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

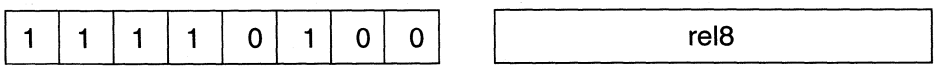
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BOV rel8

Operation:
(PC) <-- (PC) + 2
if (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) generated an overflow (the V flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

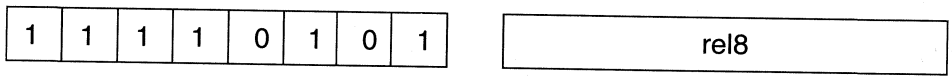
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BPL rel8

Operation: (PC) <-- (PC) + 2
if (N) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is greater than 0 (the N flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

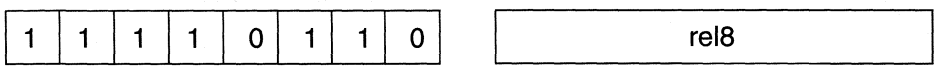
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BR rel8

Operation:
(PC) <-- (PC) + 2
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

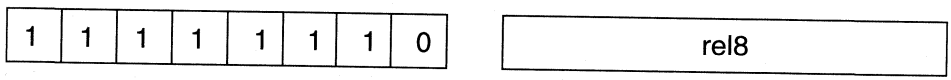
Note: Refer to section 6.3 for details of branch range

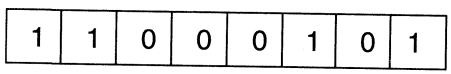
Size: None

Flags Updated: none

Bytes: 2
Clocks: 6

Encoding:





Syntax: CALL rel16

Operation:
(PC) <-- (PC) + 3
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +65,534 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory. The 24-bit return address is saved on the stack.

Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Note: Refer to section 6.3 for details of branch range

Size: None

Flags Updated: none

Bytes: 3
Clocks: 7/4(PZ)

Encoding:

byte 2: upper 8 bits of rel16
byte 3: lower 8 bits of rel16

Syntax: CALL [Rs]

Operation:
(PC) <-- (PC) + 2
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC.15-1) <-- (Rs.15-1)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address contained in the operand register, anywhere within the 64K page following the CALL instruction.The return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned, as CALL or branch will force PC.bit0 to 0.

Note:
(1) Since the PC always points to the instruction following the CALL instruction and if that happens to be on a different page, then the called routine should be located in that page (64K)

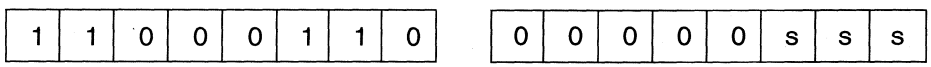
(2) if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 8/5(PZ)

Encoding:



Syntax: CJNE dest, src, rel8

Operation: (PC) <-- (PC) + # of instruction bytes
(dest) - (direct) (result not stored)
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: The byte or word specified by the source operand is compared to the variable specified by the destination operand and the status flags are updated. Jump to the specified address if the values are not equal. The source and destination data are not affected by the operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

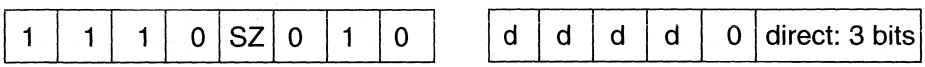
Size: Byte-Byte, Word-Word

Flags Updated: C, N, Z

(Note: this particular type of compare must not update the V or AC flags to duplicate the 80C51 function.)

CJNE Rd, direct, rel8

Bytes: 4
Clocks: 10t/7nt
Encoding:

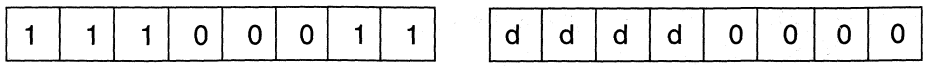


byte 3: lower 8 bits of direct
byte 4: rel8

CJNE Rd, #data8, rel8

Bytes: 4
Clocks: 9t/6nt

Encoding:

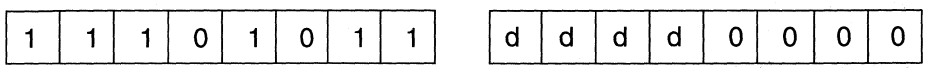


byte 3: rel8
byte 4: data#8

CJNE Rd, #data16, rel8

Bytes: 5
Clocks: 9t/6nt

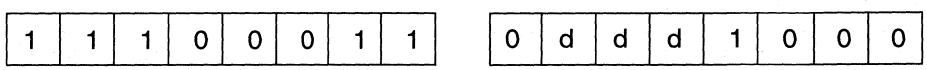
Encoding:



byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CJNE [Rd], #data8, rel8

Bytes: 4
Clocks: 10t/7nt
Encoding:

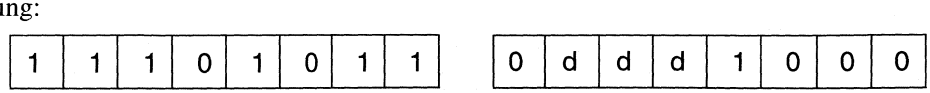


byte 3: rel8
byte 4: #data8

CJNE [Rd], #data16, rel8

Bytes: 5
Clocks: 10t/7nt

Encoding:



byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: CLR bit

Operation: (bit) <-- 0

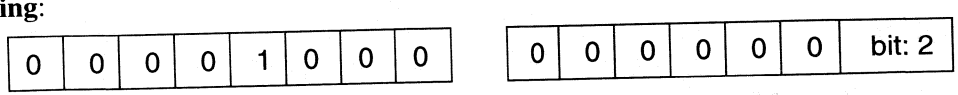
Description: Writes a 0 (clears) to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: CMP dest, src

Operation: dest - src

Description: The byte or word specified by the source operand is compared to the specified destination operand by performing a twos complement binary subtraction of src from dest. The flags are set according to the rules of subtraction. The source and destination data are not affected by the operation.

Size: byte-byte, word-word

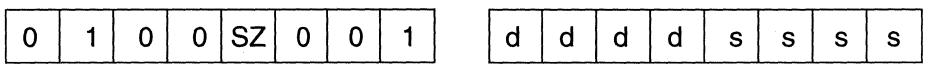
Flags Updated: C, AC, V, N, Z

CMP Rd, Rs

Operation: (Rd) - (Rs)

Bytes: 2
Clocks: 3

Encoding:

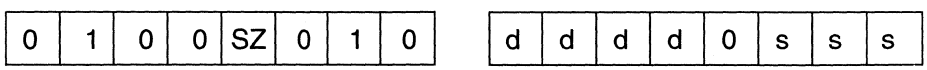


CMP Rd, [Rs]

Operation: (Rd) - ((WS:Rs))

Bytes: 2
Clocks: 4

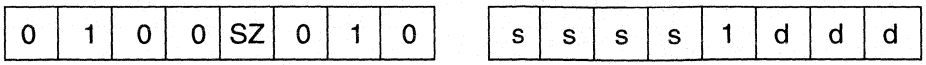
Encoding:



CMP [Rd], Rs

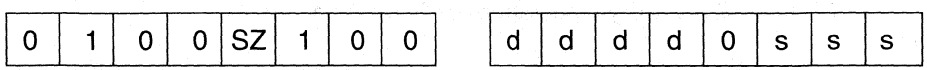
Operation: ((WS:Rd)) - (Rs)

Bytes: 2
Clocks: 4
Encoding:



CMP Rd, [Rs+offset8]

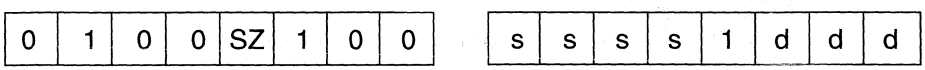
Bytes: 3
Clocks: 6
Operation: (Rd) - ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

CMP [Rd+offset8], Rs

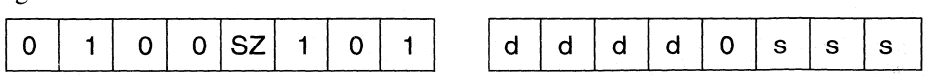
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) - (Rs)
Encoding:



byte 3: offset8

CMP Rd, [Rs+offset16]

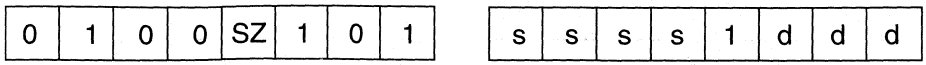
Bytes: 4
Clocks: 6
Operation: (Rd) - ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP [Rd+offset16], Rs

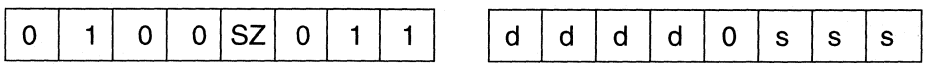
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) - (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

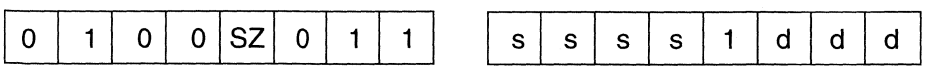


CMP [Rd+], Rs

Bytes: 2
Clocks: 5

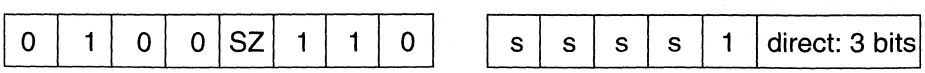
Operation: ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:



CMP direct, Rs

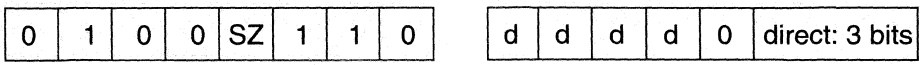
Bytes: 3
Clocks: 4
Operation: (direct) - (Rs)
Encoding:



byte 3: lower 8 bits of direct

CMP Rd, direct

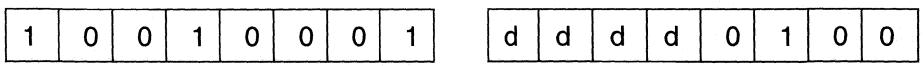
Bytes: 3
Clocks: 4
Operation: (Rd) - (direct)
Encoding:



byte 3: lower 8 bits of direct

CMP Rd, #data8

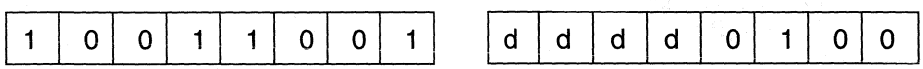
Bytes: 3
Clocks: 3
Operation: (Rd) - #data8
Encoding:



byte 3: #data8

CMP Rd, #data16

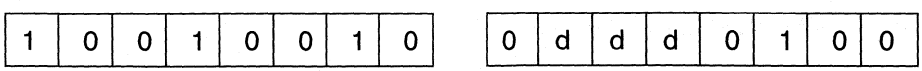
Bytes: 4
Clocks: 3
Operation: (Rd) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd], #data8

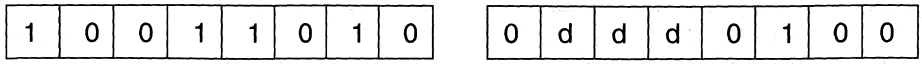
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) - #data8
Encoding:



byte 3: #data8

CMP [Rd], #data16

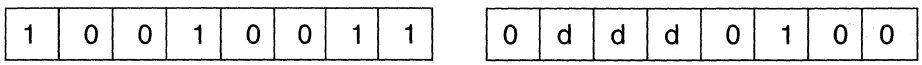
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+], #data8

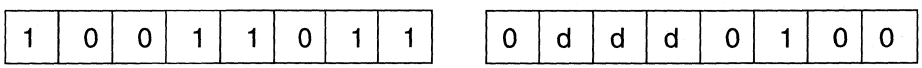
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

CMP [Rd+], #data16

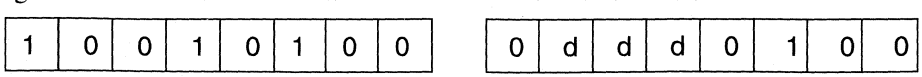
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+offset8], #data8

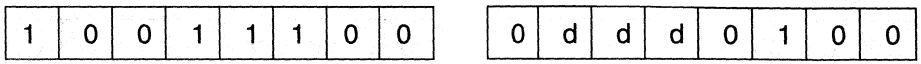
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) - #data8
Encoding:



byte 3: offset8
byte 4: #data8

CMP [Rd+offset8], #data16

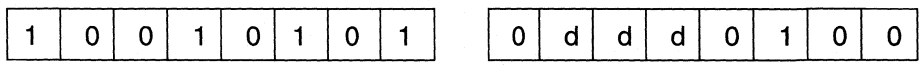
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) - #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CMP [Rd+offset16], #data8

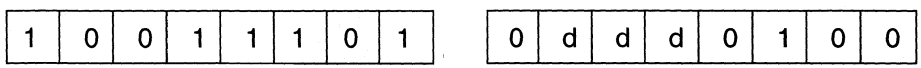
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) - #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

CMP [Rd+offset16], #data16

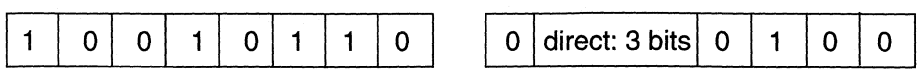
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) - #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

CMP direct, #data8

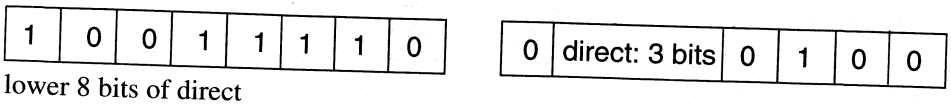
Bytes: 4
Clocks: 4
Operation: (direct) - #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

CMP direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) - #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: CPL Rd

Operation: Rd <-- ($\overline{Rd}$)

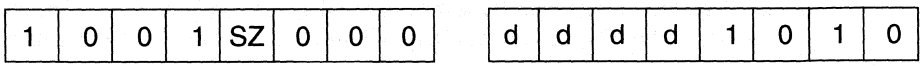
Description: Performs a ones complement of the destination operand specified by the register Rd. The result is stored back into Rd. The destination may be either a byte or a word.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 3

Encoding:



Syntax: DA Rd

Operation: if (Rd.3-0) > 9 or (AC) = 1
then (Rd.3-0) <-- (Rd.3-0) + 6
if (Rd.7-4) > 9 or (C) = 1
then (Rd.7-4) <-- (Rd.7-4) + 6

Description: Adjusts the destination register to BCD format (binary-coded decimal) following an ADD or ADDC operation on BCD values. This operation may only be done on a byte register.

If the lower 4 bits of the destination value are greater than 9, or if the AC flag is set, 6 is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value.

If the upper 4 bits of the destination value are greater than 9, or if the carry flag was set by the add to the lower bits, 60 hex is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value. Carry will never be cleared by the DA instruction if it was already set.

Size: Byte

Flags Updated: C, N, Z

The carry flag may be set but not cleared. See the description of the carry flag update above.

Bytes: 2
Clocks: 4

Encoding:

1	0	0	1	0	0	0	0

d	d	d	d	1	0	0	0

Note: Please refer to the table on the next page.

The following table shows the possible actions that may occur during the DA instruction, related to the input conditions.

Table 6.6

Low nibble (bits 3-0)	AC	Carry to high nibble	High nibble (bits 7-4)	Initial C flag	Number added to value	Resulting C flag
0 - 9	0	0	0 - 9	0	00	0
A - F	0	1	0 - 8	0	06	0
0 - 3 *	1	0	0 - 9	0	06	0
0 - 9	0	0	A - F	0	60	1
A - F	0	1	9 - F	0	66	1
0 - 3 *	1	0	A - F	0	66	1
0 - 9	0	0	0 - 2 **	1	60	1
A - F	0	1	0 - 2 **	1	66	1
0 - 3 *	1	0	0 - 3 ***	1	66	1

: The largest digit that could result from adding two BCD digits that caused the AC flag to be set is 3. This is with an ADDC instruction where 9 + 9 + 1 (the carry flag) = 13 hex.

** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with no carry from the bottom nibble (the AC flag = 0) is 2. For instance, 98 hex + 97 hex = 12F hex.

*** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with a carry from the bottom nibble (the AC flag = 1) is 3. For instance, 99 hex + 99 hex = 132 hex.

DIV.w	**16x8**	**Signed Division**
DIV.d	**32x16**	**Signed Division**
DIVU.b	**8x8**	**Unsigned Division**
DIVU.w	**16x8**	**Unsigned Division**
DIVU.d	**32x16**	**Unsigned Division**

Description: The byte or word specified by the source operand is divided into the variable specified by the destination operand.

For DIVU.b, the destination operand can be any byte register that is the least significant byte of a word register. For DIV.w and DIVU.w, the destination operand must be a word register, and for DIV.d and DIVU.d, the destination operand must identify a word register that is the low-word of a double-word register (see note below). The result is stored in the destination register as the quotient (8 bits for DIVU.b, DIVU.w, DIV.w, and DIVU.w, and 16-bits for DIV.d and DIVU.d) in the least significant half and the remainder (same size as the quotient), in the most significant half (except for DIVU.b which stores the quotient in the destination as identified by the lower half of a word register and the remainder at upper half of the same word register).

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte-Byte, Word-Byte, Double word-Word

Flags Updated: C, V, N, Z

The carry flag is always cleared. The V flag is set in the following cases, otherwise it is cleared:

- DIVU.b: V is set if a divide by 0 occurred. A divide by 0 also causes a hardware trap to be generated.
- DIV.w, DIVU.w: V is set if the result of the divide is larger than 8 bits (the result does not fit in the destination).
- DIV.d, DIVU.d: V is set if the result of the divide is larger than 16 bits (the result does not fit in the destination).

The Z, and N flags are set based on the quotient (integer) portion of the result only and not on the remainder.

Examples:

a) DIVU.b R4L, R4H - will store the result of the division of R4L by R4H in R4L and R4H (quotient in register R4L, remainder in register R4H).

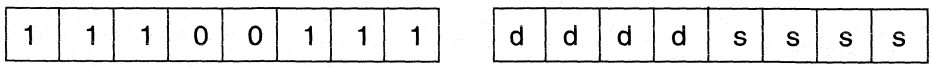
b) DIV.w R0, R2L - will store the result of word register R0 divided by byte register R2L in word register R0 (quotient in register R0L, remainder in register R0H).

c) DIV.d R4,R2 - will store the result of double-word register R5:R4 divided by word register R2 in double-word register R5:R4 (quotient in R4, remainder in R5)

Note: For all divides except DIVU.b, the destination register size is the same as indicated by the instruction (by the ".b", ".w", or ".d") and the source register is half that size.

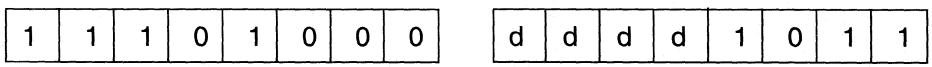
DIV.w Rd, Rs
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Clocks: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:



DIV.w Rd, #data8
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

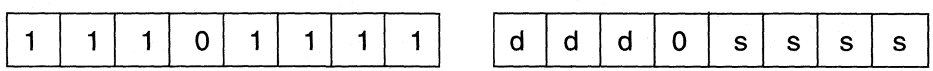
Bytes: 3
Clocks: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:



byte 3: #data8

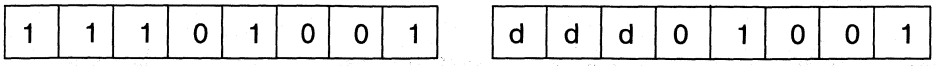
DIV.d Rd, Rs
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Clocks: 24
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (signed divide)
(Rd+1) <-- 16-bit remainder of (Rd) / (Rs)
Encoding:



DIV.d Rd, #data16
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

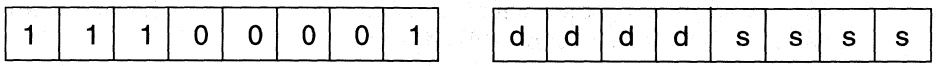
Bytes: 4
Clocks: 24
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (signed divide)
(Rd+1)<-- 16-bit remainder of (Rd) / #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

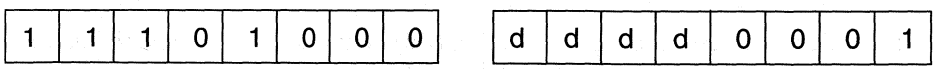
DIVU.b Rd, Rs
(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (RdL) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (RdL) / (Rs)
Encoding:



DIVU.b Rd, #data8
(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

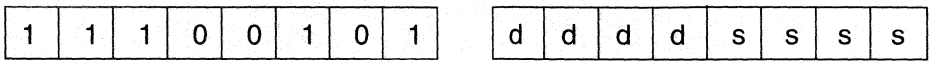
Bytes: 3
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (RdL) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (RdL) / #data8
Encoding:



byte 3: #data8

DIVU.w Rd, Rs
(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

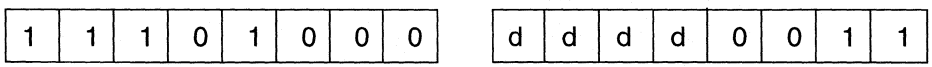
Bytes: 2
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:



DIVU.w Rd, #data8
(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 3
Clocks: 12

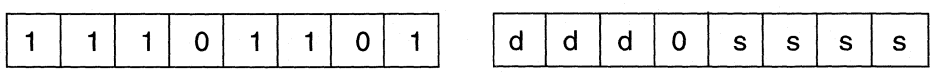
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:



byte 3: #data8

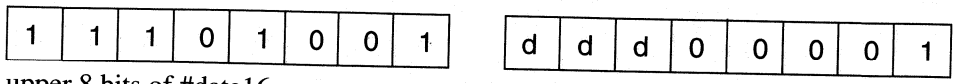
DIVU.d Rd, Rs
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Clocks: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (unsigned divide)
(Rd+1)<-- 16-bit remainder of (Rd) / (Rs)
Encoding:



DIVU.d Rd, #data16
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Clocks: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (unsigned divide)
(Rd+1)<-- 16-bit remainder of (Rd) / #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

Syntax: DJNZ dest, rel8

Operation: (PC) <-- (PC) + 3
(dest) <-- (dest) - 1
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: Controls a loop of instructions. The parameters are: a condition code (Z), a counter (register or memory), and a displacement value. The instruction first decrements the counter by one, tests the condition if the result of decrement is 0 (for termination of the loop); if it is false, execution continues with the next instruction. If true, execution branches to the location indicated by the current value of the PC plus the sign extended displacement. The value in the PC is the address of the instruction following DJNZ.

The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.The destination operand could be byte or word.

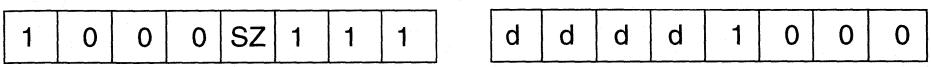
Note: Refer to section 6.3 for details of jump range

Size: Byte, Word

Flags Updated: N, Z

DJNZ Rd, rel8

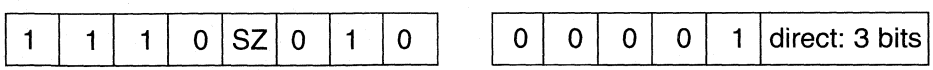
Bytes: 3
Clocks: 8t/5nt
Encoding:



byte 3: rel8

DJNZ direct, rel8

Bytes: 4
Clocks: 9t/5nt
Encoding:



byte 3: lower 8 bits of direct
byte 4: rel8

Syntax: FCALL addr24

Operation:
(PC) <-- (PC) + 4
(SP) <-- (SP) - 4
((SP)) <-- (PC)
(PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space. The 24-bit return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned as CALL or branch will force PC.bit0 to 0.

Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 4
Clocks: 12/8(PZ)

Encoding:

1	1	0	0	0	1	0	0

address: middle 8 bits (bits 15-8)

byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

Syntax: FJMP addr24

Operation: (PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space.
Note: The target address must be word aligned as JMP always forces PC to an even address.

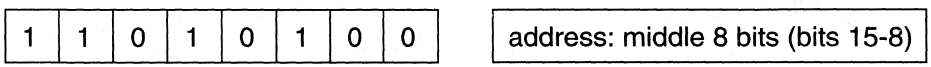
Note: if the XA is in page 0 mode, only 16-bits of the address will be used.

Size: None

Flags Updated: none

Bytes: 4
Clocks: 6

Encoding:



byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

Syntax: JB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0

Description: If the specified bit is a one, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is clear, the instruction following JB is executed.The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

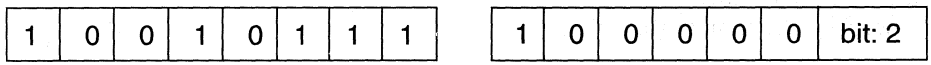
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 10t/6nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JBC bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0; (bit) <-- 0

Description: If the bit specified is set, branch to the address pointed to by the PC plus the specified displacement. The specified bit is then cleared allowing implementation of semaphore operations. If the specified bit is clear, the instruction following JBC is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

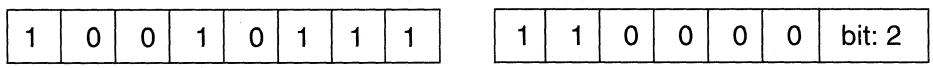
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 11t/7nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JMP rel16

Operation: (PC) <-- (PC) + 3
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Jumps unconditionally. The branch range is +65,535 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory.

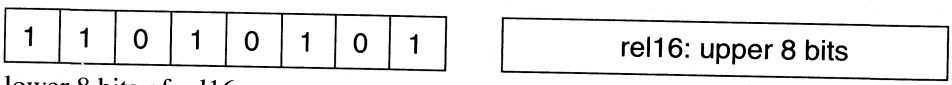
Note: Refer to section 6.3 for details of jump range

Size: None

Flags Updated: none

Bytes: 3
Clocks: 6

Encoding:



byte 3: lower 8 bits of rel16

Syntax: JMP [Rs]

Operation: (PC) <-- (PC) + 2
(PC.15-1) <-- (Rs.15-1) (note that PC.23-16 is not affected)
(PC.0) <-- 0

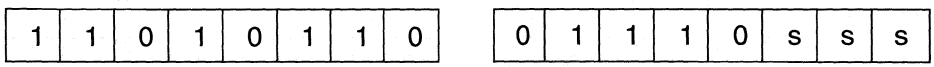
Description: Causes an unconditional branch to the address contained in the operand word register, anywhere within the 64K code page following the JMP instruction.The value of the PC used in the target address calculation is the address of the instruction following the JMP instruction.
The target address must be word aligned as JMP will force PC.bit0 to 0.

Size: none

Flags Updated: none

Bytes: 2
Clocks: 7

Encoding:



Syntax: JMP [A+DPTR]

Operation: (PC) <-- (PC) + 2
(PC15-1) <-- (A) + (DPTR)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address formed by the sum of the 80C51 compatibility registers A and DPTR, anywhere within the 64K code page following the JMP instruction. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Note: The target address must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Clocks: 5

Note: A and DPTR are pre-defined registers used for 80C51 code translation.

Encoding:

1	1	0	1	0	1	1	0

0	1	0	0	0	1	1	0

Syntax: JMP [[Rs+]]

Operation: (PC) <-- (PC) + 2
(PC.15-0) <-- code memory ((WS:Rs))
(PC.0) <-- 0
(Rs) <-- (Rs) + 2

Description: Causes an unconditional branch to the address contained in memory at the address pointed to by the register specified in the instruction. The specified register is post-incremented.

This 2-byte instruction may be used to compress code size by using it to index through a table of procedure addresses that are accessed in sequence. Each procedure would end with another JMP [[R+]] that would immediately go to the next procedure whose address is in the table.

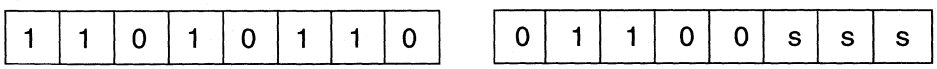
The procedures must be located in the same 64K address page of the executed "Jump Double-indirect" instruction (although the table could be in any page). This instruction can result in substantial code compression and hence cost reduction through smaller memory requirements. The register pointer (index to the table) being automatically post-incremented after the execution of the instruction. The 24-bit address is identified by combining the low order 16-bit of the PC and either of high 8-bits of PC or the contents of a byte-size CS register as chosen by the program through a segment select Special Function Register (SFR).

Note: The subroutine addresses must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Clocks: 8

Encoding:



Syntax: JNB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 0 then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: If the specified bit is a zero, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is set, the instruction following JB is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

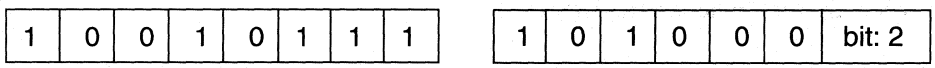
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 10t/6nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JNZ rel8

Operation: (PC) <-- (PC) + 2
if (A) not equal to 0, then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are not zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

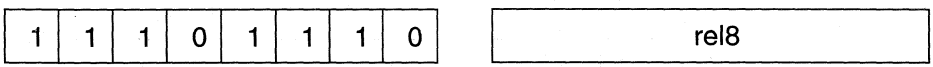
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: JZ rel8

Operation: (PC) <-- (PC) + 2
If (A) = 0 then
(PC.15-0) <-- (PC + rel8*2);
(PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

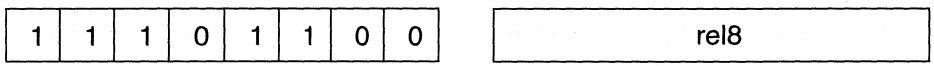
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: LEA Rd, Rs+offset8/16

Operation: (Rd) <-- (Rs)+offset8/16

Description: The word specified by the source operand is added to the offset value and the result is stored into the register specified by the destination operand. The source and destination operands are both registers. The offset value is an immediate data field of either 8 or 16 bits in length. The source data is not affected by the operation.

This instruction mimics the address calculation done during other instructions when the register indirect with offset addressing mode is used, allowing the resulting address to be saved for other purposes.

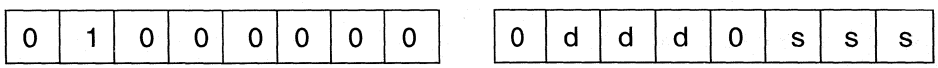
Note: The result of this operation is always a word since it duplicates the calculation of the indirect with offset addressing mode.

Size: Word-Word

Flags Updated: none

LEA Rd, Rs+offset8

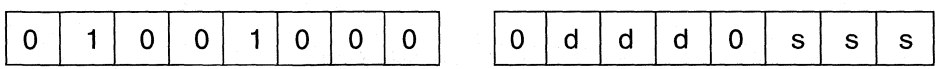
Bytes: 3
Clocks: 3
Encoding:



byte 3: offset8

LEA Rd, Rs+offset16

Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rs)+offset16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

Syntax: LSR dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
(dest.msb) <- 0
count = count-1
End While

Description: If the count operand is greater than the variable specified by the destination operand is logically shifted right by the number of bits specified by the count operand. The MSBs of the result are filled with zeroes.The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.The count operand is a positive value which may be from 0 to 31. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register. The count is not affected by the operation.

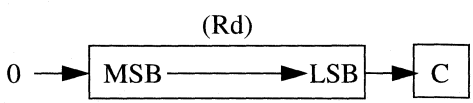
Note:
- For Logical Shift Left, use ASL ignoring the N flag.
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte, Word, Double Word

Flags Updated: C, N, Z (N = 0 after an LSR unless count = 0, then it is unchanged)

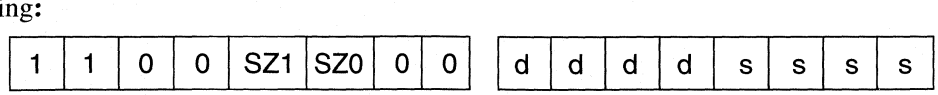
LSR Rd, Rs (Rs = Byte-register)

Operation:



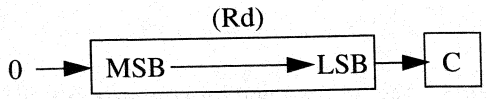
Bytes: 2
Clocks: For 8/16 bit shifts --> 4+1 for each 2 bits of shift
For 32 bit shifts --> 6+1 for each 2 bits of shift

Encoding:



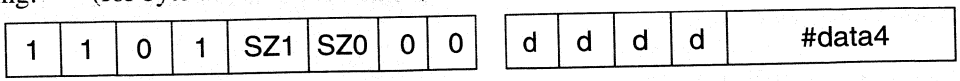
LSR Rd, #data4
Rd, #data5

Operation:

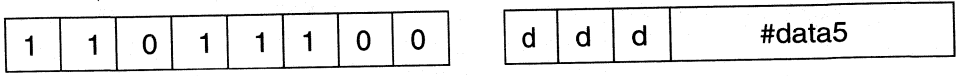


Bytes: 2
Clocks: For 8/16 bit shifts --> 4+1 for each 2 bits of shift
For 32 bit shifts --> 6+1 for each 2 bits of shift

Encoding: (for byte and word data sizes)



(for double word data size)



Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 01: reserved; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: MOV dest, src

Operation: dest <- src

Description: The byte or word specified by the source operand is copied into the variable specified by the destination operand. The source data is not affected by the operation.

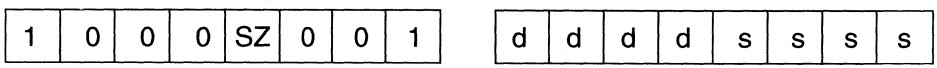
Source and destination operands may be a register in the register file, an indirect address specified by a pointer register, an indirect address specified by a pointer register added to an immediate offset of 8 or 16 bits, or a direct address. Source operands may also be specified as immediate data contained within the instruction. Auto-increment of the indirect pointers is available for simple indirect (not offset) addressing.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

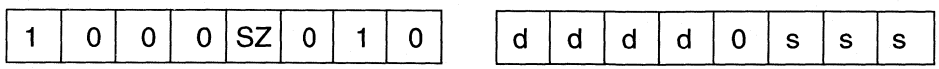
MOV Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rs)
Encoding:



MOV Rd, [Rs]

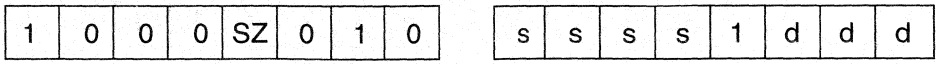
Bytes: 2
Clocks: 3
Operation: (Rd) <-- ((WS:Rs))
Encoding:



MOV [Rd], Rs

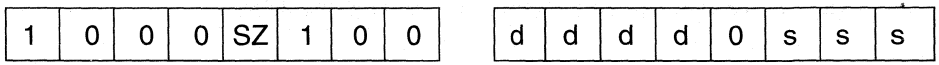
Bytes: 2
Clocks: 3

Operation: ((WS:Rd)) <-- (Rs)
Encoding:



MOV Rd, [Rs+offset8]

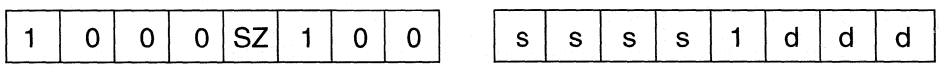
Bytes: 3
Clocks: 5
Operation: (Rd) <-- ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

MOV [Rd+offset8], Rs

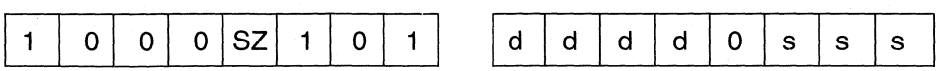
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- (Rs)
Encoding:



byte 3: offset8

MOV Rd, [Rs+offset16]

Bytes: 4
Clocks: 5
Operation: (Rd) <-- ((WS:Rs)+offset16)
Encoding:

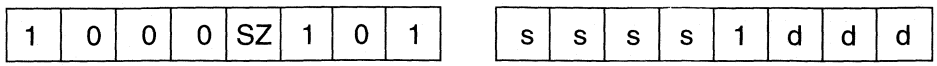


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV [Rd+offset16], Rs

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- (Rs)

Encoding:

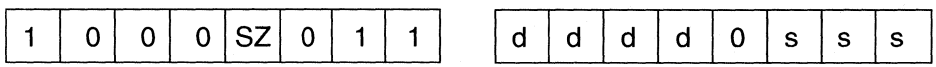


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV Rd, [Rs+]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:

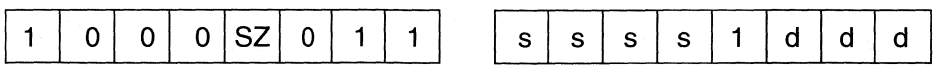


MOV [Rd+], Rs

Bytes: 2
Clocks: 4

Operation: ((WS:Rd)) <-- (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

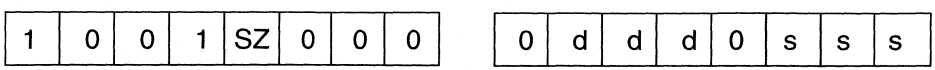
Encoding:



MOV [Rd+], [Rs+]

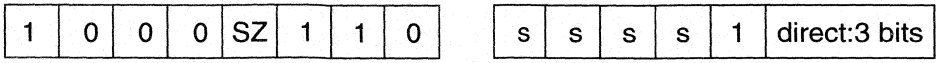
Bytes: 2
Clocks: 6
Operation: ((WS:Rd)) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:



MOV direct, Rs

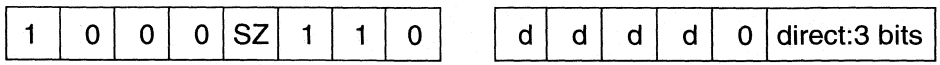
Bytes: 3
Clocks: 4
Operation: (direct) <-- (Rs)
Encoding:



byte 3: lower 8 bits of direct

MOV Rd, direct

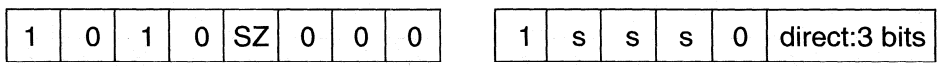
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct

MOV direct, [Rs]

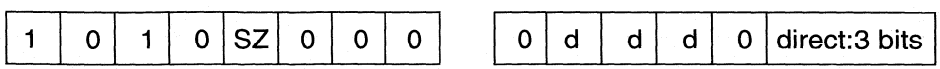
Bytes: 3
Clocks: 4
Operation: (direct) <-- ((WS:Rs))
Encoding:



byte 3: lower 8 bits of direct

MOV [Rd], direct

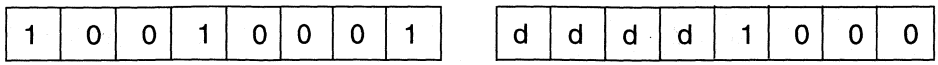
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct

MOV Rd, #data8

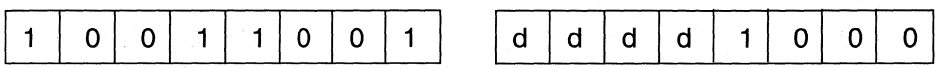
Bytes: 3
Clocks: 3
Operation: (Rd) <-- #data8
Encoding:



byte 3: #data8

MOV Rd, #data16

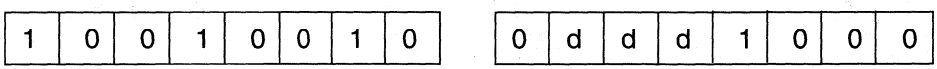
Bytes: 4
Clocks: 3
Operation: (Rd) <-- #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd], #data8

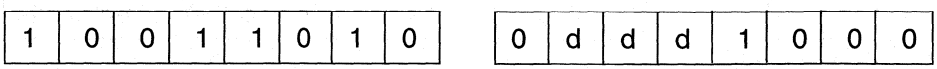
Bytes: 3
Clocks: 3
Operation: ((WS:Rd)) <-- #data8
Encoding:



byte 3: #data8

MOV [Rd], #data16

Bytes: 4
Clocks: 3
Operation: ((WS:Rd)) <-- #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

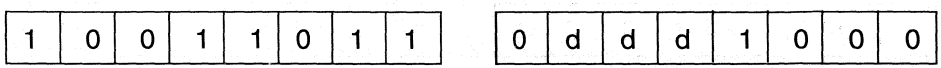
MOV [Rd+], #data8

Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- #data8
(Rd) <-- (Rd) + 1
Encoding:

byte 3: #data8

MOV [Rd+], #data16

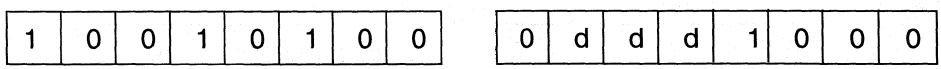
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+offset8], #data8

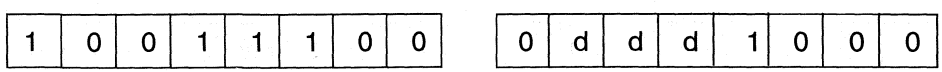
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- #data8
Encoding:



byte 3: offset8
byte 4: #data8

MOV [Rd+offset8], #data16

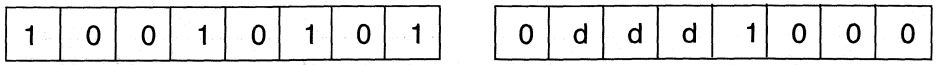
Bytes: 5
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV [Rd+offset16], #data8

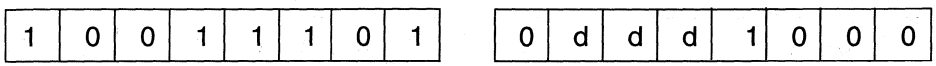
Bytes: 5
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

MOV [Rd+offset16], #data16

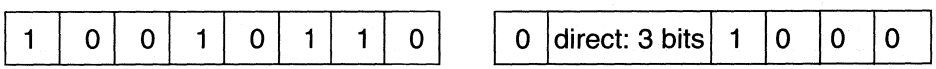
Bytes: 6
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

MOV direct, #data8

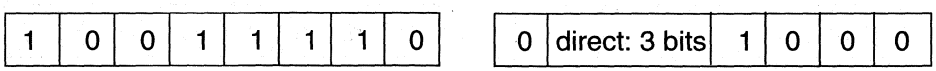
Bytes: 4
Clocks: 3
Operation: (direct) <-- #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

MOV direct, #data16

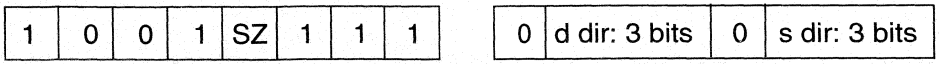
Bytes: 5
Clocks: 3
Operation: (direct) <-- #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV direct, direct

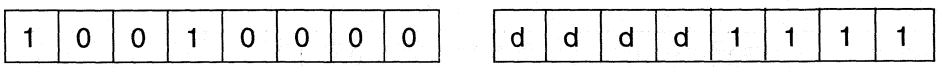
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct (dest)
byte 4: lower 8 bits of direct (src)

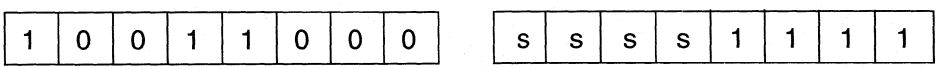
MOV Rd, USP (move from user stack pointer)

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (USP)
Encoding:



MOV USP, Rs (move to user stack pointer)

Bytes: 2
Clocks: 3
Operation: (USP) <-- (Rs)
Encoding:



Syntax: MOV C, bit

Operation: (C) <-- (bit)

Description: Copies the specified bit to the carry flag.

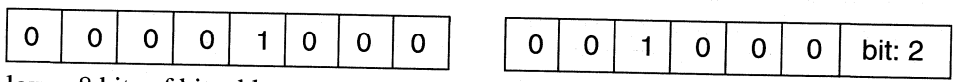
Size: Bit

Flags Updated: none

Note: C is written as the destination of the move, not as a status flag

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: MOV bit, C

Operation: (bit) <-- (C)

Description: Copies the carry flag to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: MOVC Rd, [Rs+]

Operation: (Rd) <-- code memory ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

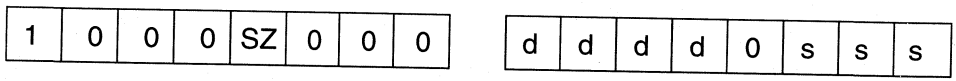
Description: Contents of code memory are copied to an internal register. The byte or word specified by the source operand is copied to the variable specified by the destination operand. In the case of MOVC, the pointer segment selection gives the choices of PC_{23-16} or CS segment (current *working segment* referred here as WS), rather than DS or ES as is used for all other instructions.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4

Encoding:



Syntax: MOVC A, [A+DPTR]

Operation: PC <- PC+2
(A) <-- code memory (PC.23-16:(A) + (DPTR))

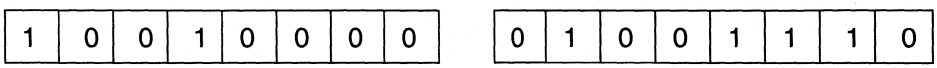
Description: The byte located at the code memory address formed by the sum of A and the DPTR is copied to the A register. The A and DPTR registers are pre-defined registers used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Clocks: 6

Encoding:



Syntax: MOVC A, [A+PC]

Operation: PC <- PC+2
(A) <-- code memory [PC.23-16: (A +PC.15-0)]

Note: Only 16-bits of A+PC are used

Description: The byte located at the code memory address formed by the sum of A and the current Program Counter value is copied to the A register. The A register is a pre-defined register used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Clocks: 6

Encoding:

1	0	0	1	0	0	0	0

0	1	0	0	1	1	0	0

Syntax: MOVS dest, #data

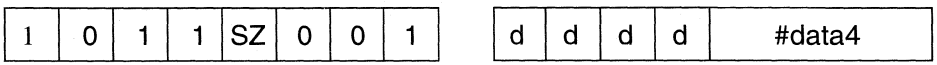
Description: Four bits of signed immediate data are moved to the destination. The immediate data is sign-extended to the proper size, then moved to the variable specified by the destination operand, which may be a byte or a word. The immediate data range is +7 to -8. This instruction is used to save time and code space for the many instances where a small data constant is moved to a destination.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

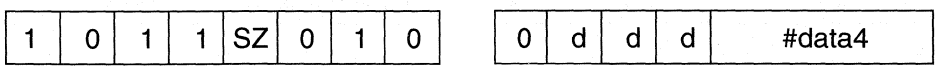
MOVS Rd, #data4

Bytes: 2
Clocks: 3
Operation: (Rd) <-- sign-extended #data4
Encoding:



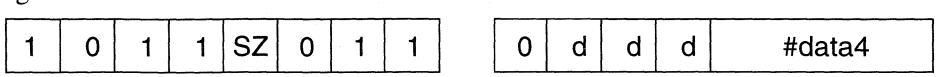
MOVS [Rd], #data4

Bytes: 2
Clocks: 3
Operation: ((WS:Rd)) <-- sign-extended #data4
Encoding:



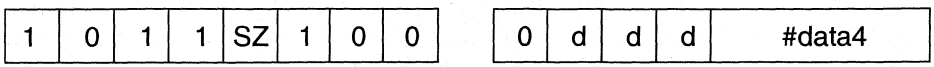
MOVS [Rd+], #data4

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- sign-extended #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



MOVS [Rd+offset8], #data4

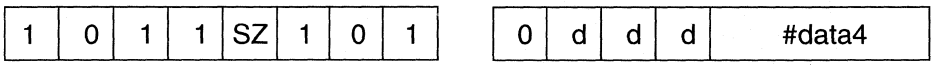
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- sign-extended #data4
Encoding:



byte 3: offset8

MOVS [Rd+offset16], #data4

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- sign-extended #data4
Encoding:

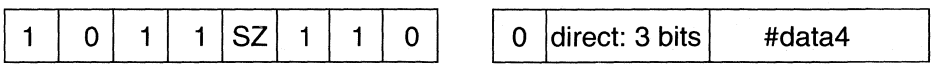


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOVS direct, #data4

Bytes: 3
Clocks: 3
Operation: (direct) <-- sign-extended #data4

Encoding:



byte 3: lower 8 bits of direct

Syntax: MOVX dest, src

Description: Move external data to or from an internal register. The byte or word specified by the source operand is copied into the variable specified by the destination operand. This instruction allows access to data external to the microcontroller in the address range of 0 to 64K. The standard indirect move may access external data only above the boundary where internal data RAM ends, whereas MOVX always forces an external access. MOVX only operates on the first 64K of external data memory. This instruction is included to allow compatibility with 80C51 code.

Note that in the 80C51 MOVX instruction using @Ri as a pointer (where i could be 0 or 1), the pointer was eight bits in length and the upper address lines were not driven on the external bus. The XA always drives all of the enabled external bus address lines. The use of the pointer depends on whether compatibility mode is in use. If CM = 0 (compatibility mode off, the default), 16 bits of R0 or R1 are used as the address within data segment 0. If CM = 1 (compatibility mode on), 8 bits of R0L or R0H are used as the bottom eight bits of the address, while the remainder of the address bits, including those corresponding to the data segment are 0.

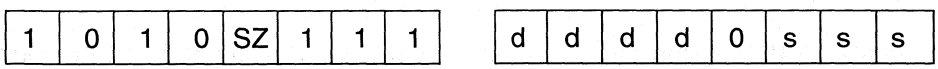
Size: Byte-Byte, Word-Word

Flags Updated: N, Z

MOVX Rd, [Rs]

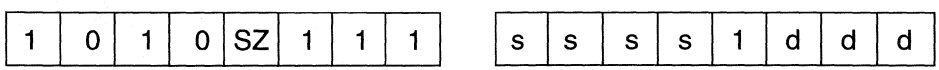
Bytes: 2
Clocks: 6
Operation: (Rd) <-- external data memory ((Rs))

Encoding:



MOVX [Rd], Rs

Bytes: 2
Clocks: 6
Operation: external data memory ((Rd)) <-- (Rs)
Encoding:



Description: The byte or word specified by the source operand is multiplied by the variable specified by the destination operand.

The destination operand must be the first half of a double size register (word for a byte multiply and double word for a word multiply). The result is stored in the double size register.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, and R7:R6).

Size: Byte-Byte, Word-Word

Flags Updated: C, V, N, Z

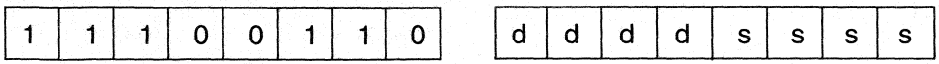
The carry flag is always cleared by a multiply instruction. The V flag is set in the following cases, otherwise it is cleared:
- MULU.b: V is set if the result of the multiply is greater than FFh (the upper byte is not equal to 0).
- MULU.w: V is set if the result of the multiply is greater than FFFFh (the upper word is not equal to 0).
- MUL.w: V is set if the absolute value of the result of the multiply is greater than 7FFFh (the upper word is not a sign extension of the lower word).

Examples:
a) MUL.w R0,R5 stores the product of word register 0 and word register 5 in double word register 0 (least significant word in word register R0, most significant word in word register R1).

b) MULU.b R4L, R4H will store the MS byte of the product of R4L and R4H in R4H and the LS byte in R4L.

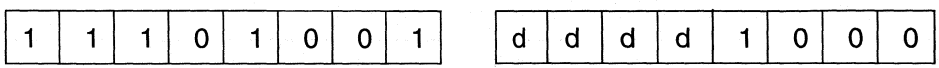
MUL.w Rd, Rs
(signed 16 bits * 16 bits --> 32 bits)

Bytes: 2
Clocks: 12
Operation: (Rd+1)<-- Most significant word of (Rd) * (Rs) (signed multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:



MUL.w Rd, #data16
(signed 16 bits * 16 bits --> 32 bits)

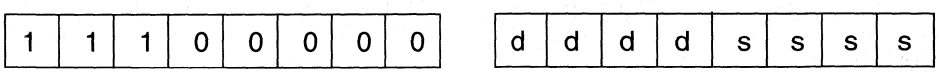
Bytes: 4
Clocks: 12
Operation: (Rd+1)<-- Most significant word of (Rd) * #data16 (signed multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

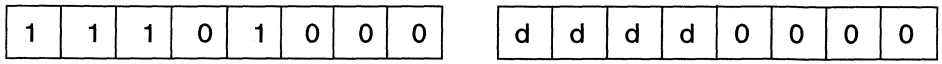
MULU.b Rd, Rs
(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 2
Clocks: 12
Operation: (RdH) <-- Most significant byte of (RdL) * (Rs) (unsigned multiply)
(RdL) <-- Least significant byte of (RdL) * (Rs)
Encoding:



MULU.b Rd, #data8
(unsigned 8 bits * 8 bits --> 16 bits)

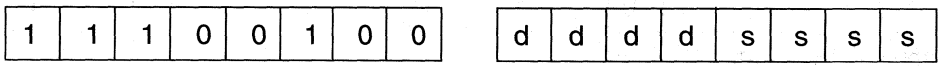
Bytes: 3
Clocks: 12
Operation: (RdH) <-- Most significant byte of (RdL) * #data8 (unsigned multiply)
(RdL) <-- Least significant byte of (RdL) * #data8
Encoding:



byte 3: #data8

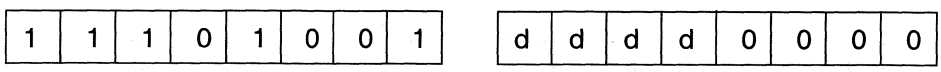
MULU.w Rd, Rs
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 2
Clocks: 12
Operation: (Rd+1)<-- Most significant word of (Rd) * (Rs) (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:



MULU.w Rd, #data16
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 4
Clocks: 12
Operation: (Rd+1)<-- Most significant word of (Rd) * #data16 (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

Syntax: NEG Rd

Operation: Rd <-- ($\overline{Rd}$) + 1

Description: The destination register is negated (twos complement). The destination may be a byte or a word.

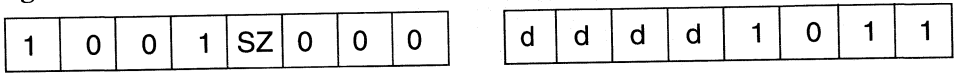
Size: Byte, Word

Flags Updated: V, N, Z

The V flag is set if a twos complement overflow occurred: the original value = result = 8000 hex for a word operation or 80 hex for a byte operation.

Bytes: 2
Clocks: 3

Encoding:



Syntax: NOP

Operation: PC <- PC + 1

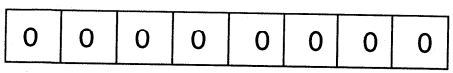
Description: Execution resumes at the following instruction. This instruction is defined as being one byte in length in order to allow it to be used to force word alignment of instructions that are branch targets, or for any other purpose. It may also be used to as a delay for a predictable amount of time.

Size: None

Flags Updated: none

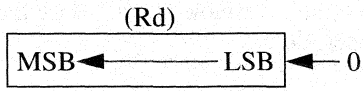
Bytes: 1
Clocks: 3

Encoding:



Syntax: NORM Rd, Rs

Operation:



Description: Logically shifts left the contents of the destination until the MSB is set, storing the number of shifts performed in the count (source) register. The data size may be 8, 16, or 32 bits.

If the destination value already has the MSB set, the count returned will be 0. If the destination value is 0, the count returned will be 0, the N flag will be cleared, and the Z flag will be set. For all other conditions, the N flag will be 1 and the Z flag will be 0.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
The last pair, i.e, R7:R6 is probably not a good idea as R7 is the current stack pointer.

Size: Byte, Word, Double Word

Flags Updated: N, Z

Bytes: 2
Clocks: For 8 or 16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding:

1	1	0	0	SZ1	SZ0	1	1	d	d	d	d	s	s	s	s

Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 01: reserved; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: OR dest, src

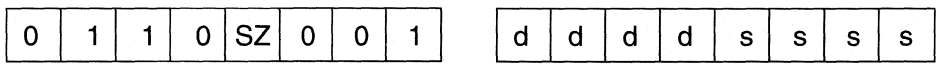
Description: Bitwise logical OR the contents of the source to the destination. The byte or word specified by the source operand is logically ORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

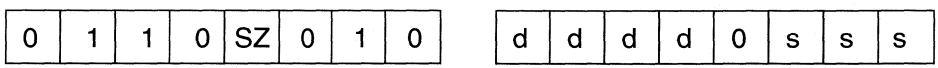
OR Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:



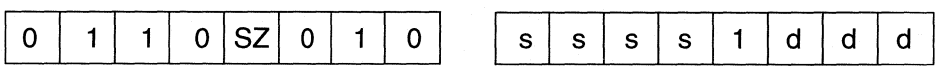
OR Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:



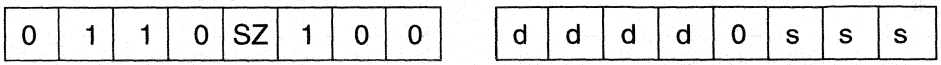
OR [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
Encoding:



OR Rd, [Rs+offset8]

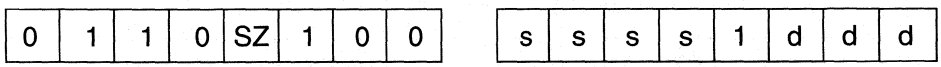
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

OR [Rd+offset8], Rs

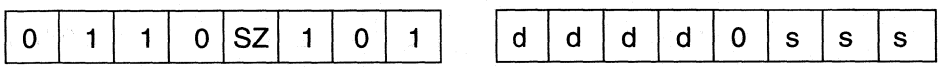
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:



byte 3: offset8

OR Rd, [Rs+offset16]

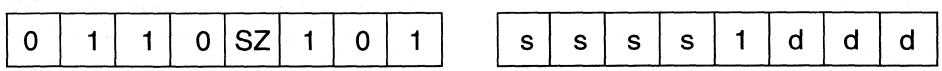
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

OR [Rd+offset16], Rs

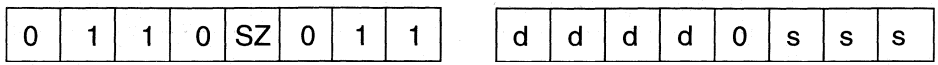
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

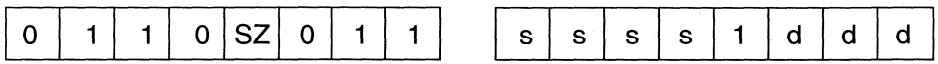
OR Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



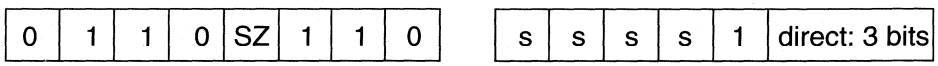
OR [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



OR direct, Rs

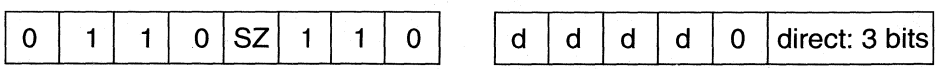
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:



byte 3: lower 8 bits of direct

OR Rd, direct

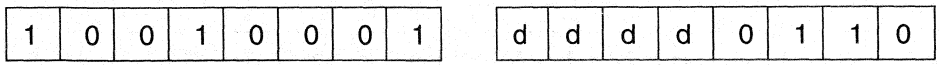
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:



byte 3: lower 8 bits of direct

OR Rd, #data8

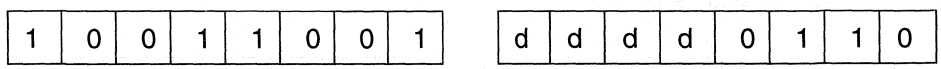
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:



byte 3: #data8

OR Rd, #data16

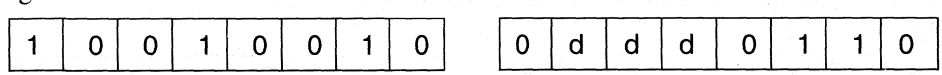
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd], #data8

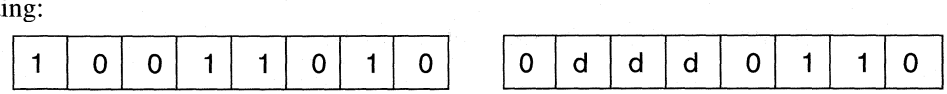
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:



byte 3: #data8

OR [Rd], #data16

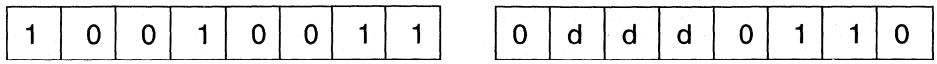
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+], #data8

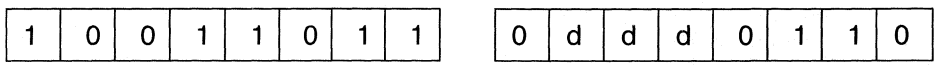
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

OR [Rd+], #data16

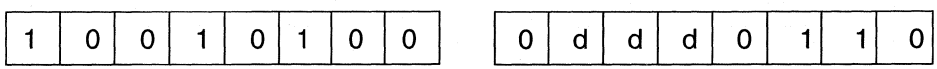
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+offset8], #data8

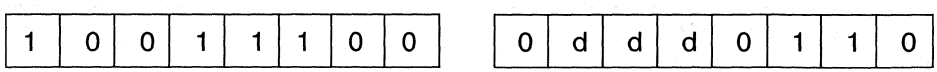
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:



byte 3: offset8
byte 4: #data8

OR [Rd+offset8], #data16

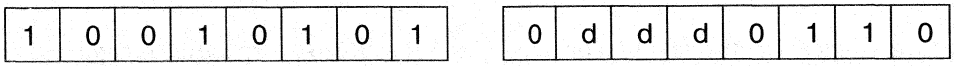
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

OR [Rd+offset16], #data8

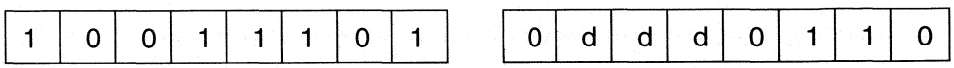
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

OR [Rd+offset16], #data16

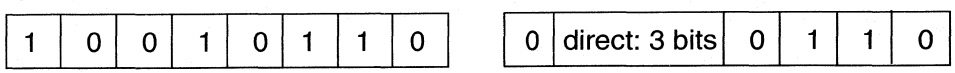
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

OR direct, #data8

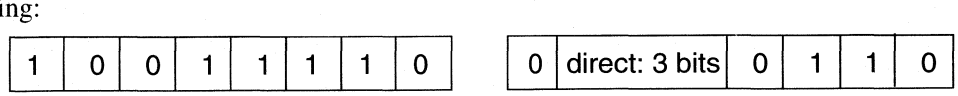
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

OR direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ORL C, bit

Operation:(C) <-- (C) + (bit)

Description: Logical (inclusive) OR a bit to the Carry flag. Read the specified bit and logically OR it to the Carry flag.
(C is written as the destination of the ORL, not as a status flag)

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: ORL C, /bit

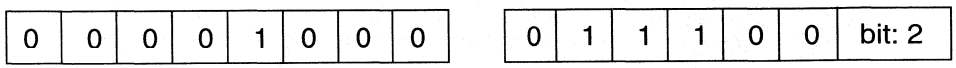
Operation: (C) <-- (C) + ($\overline{bit}$)

Description: Logically OR the complement of a bit to the Carry flag. Read the specified bit, complement it, and logically OR it to the Carry flag.
(C is written as the destination of the move, not as a status flag)

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: POP dest

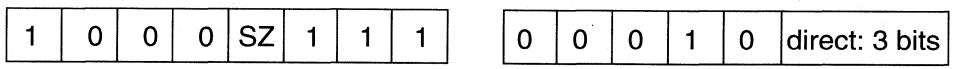
Description: The stack is popped and the data written to the specified directly addressed location. The data size may be byte or word. POP uses the current stack pointer, while POPU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

POP direct

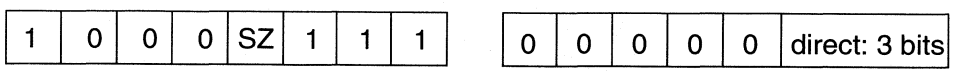
Bytes: 3
Clocks: 5
Operation: (direct) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:



byte 3: 8 bits of direct

POPU direct

Bytes: 3
Clocks: 5
Operation: (direct) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:



byte 3: 8 bits of direct

POP **Pop Multiple**
POPU **Pop User Multiple**

Syntax: POP Rlist
POPU Rlist

Description: Pop the specified registers (one or more) from the stack. The stack is popped (from 1 to 8 times) and the data stored in the specified registers. Any combination of word registers in the group R0 to R7 may be popped in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be popped in a single instruction in a byte operation. POP uses the current stack pointer, while POPU forces an access to the user stack.

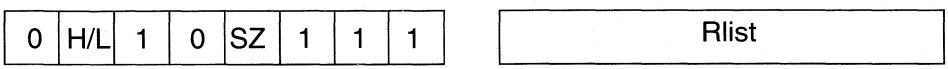
Note: Rlist is a bit map that represents each register to be popped. The registers are in the order R7, R6, R5,....., R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The pop order is from right to left, i.e., the register specified by the rightmost one in Rlist will be popped first, etc. The order must be the reverse of that used by the preceding PUSH instruction. Note that if the same register list is used first with a PUSH, then with a POP, the original register contents will be restored. The order in which the registers are called out in the source code is not important because the Rlist operand is encoded as a fixed order bit map (see below).

Size: Byte, Word

Flags Updated: none

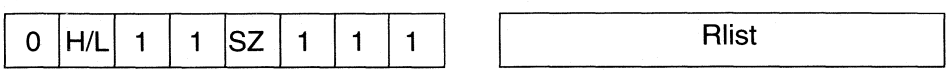
POP Rlist

Bytes: 2
Clocks: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:



POPU Rlist

Bytes: 2
Clocks: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:



Rlist bit definitions for a byte POP from register(s) in the upper register group (R4L through R7H):

R7H	R7L	R6H	R6L	R5H	R5L	R4H	R4L

Rlist bit definitions for a byte POP from register(s) in the lower register group (R0L through R3H):

R3H	R3L	R2H	R2L	R1H	R1L	R0H	R0L

Rlist bit definitions for a word POP from any register(s) (R0 through R7):

R7	R6	R5	R4	R3	R2	R1	R0

PUSH **Push**
PUSHU **Push User**

Syntax: PUSH src
PUSHU src

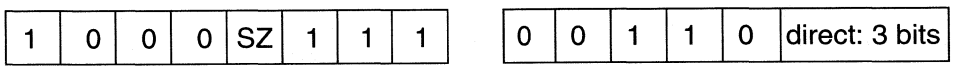
Description: The specified directly addressed data is pushed onto the stack. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

PUSH direct

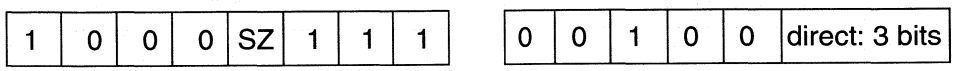
Bytes: 3
Clocks: 5
Operation: (SP) <-- (SP) - 2
((SP)) <-- (direct)
Encoding:



byte 3: 8 bits of direct

PUSHU direct

Bytes: 3
Clocks: 5
Operation: (USP) <-- (USP) - 2
((USP)) <-- (direct)
Encoding:



byte 3: 8 bits of direct

Syntax: PUSH Rlist
PUSHU Rlist

Description: Push the specified registers (one or more) onto the stack. The specified registers are pushed onto the stack. Any combination of word registers in the group R0 to R7 may be pushed in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be pushed in a single instruction in a byte operation. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

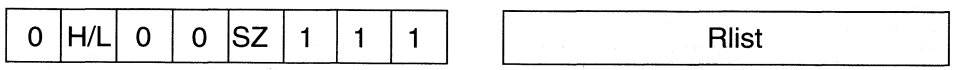
Note: Rlist is a bit map that represents each register to be pushed. The registers are in the order R7, R6, R5,....., R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The push order is from left to right, i.e., the register specified by the leftmost one in Rlist will be pushed first, etc. The order must be the reverse of that used by the corresponding POP instruction. Note that if the same register list is used first with a PUSH, then with a POP, the original register contents will be restored. The order in which the registers are called out in the source code is not important because the Rlist operand is encoded as a fixed order bit map (see below).

Size: Byte, Word

Flags Updated: none

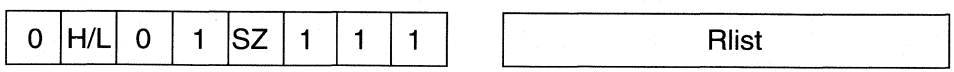
PUSH Rlist

Bytes: 2
Clocks: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(SP) <-- (SP) - 2
((SP)) <-- (Ri)
Encoding:



PUSHU Rlist

Bytes: 2
Clocks: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(USP) <-- (USP) - 2
((USP)) <-- (Ri)
Encoding:



Rlist bit definitions for a byte PUSH from register(s) in the upper register group (R4L through R7H):

R7H	R7L	R6H	R6L	R5H	R5L	R4H	R4L

Rlist bit definitions for a byte PUSH from register(s) in the lower register group (R0L through R3H):

R3H	R3L	R2H	R2L	R1H	R1L	R0H	R0L

Rlist bit definitions for a word PUSH from any register(s) (R0 through R7):

R7	R6	R5	R4	R3	R2	R1	R0

Syntax: RESET

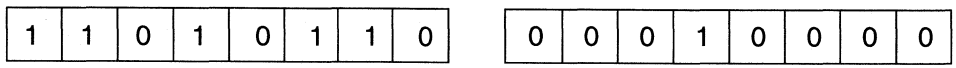
Operation: (PC) <-- vector(0)
(PSW) <-- vector(0)
(SFRs) <-- reset values (refer to the description of reset for details)

Description: The chip is reset exactly as if the external hardware reset has been asserted with the exception that it does not sample inputs for configuration, e.g., $\overline{EA}$, BUSW, etc. When a RESET instruction is executed, the chip is internally reset, but no external $\overline{RESET}$ pulse is generated. The above inputs which are latched during rising edge of a $\overline{RESET}$ pulse, hence does not affect the chip configuration.

Flags Updated: The entire PSW is set to the value specified in the reset vector.

Bytes: 2
Clocks: 18

Encoding:



Syntax: RET

Operation: (PC) <-- ((SP))
(SP) <-- (SP) + 4

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value (PC_{23-0}). This instruction is used to return from a subroutine that was called with a CALL or Far Call (FCALL).

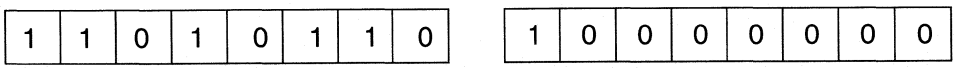
Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 8/6 (PZ)

Encoding:



Syntax: RETI

Operation: (PSW) <-- ((SSP))
(PC.23-0) <-- ((SSP))
(SSP) <-- (SSP) + 6

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value. The Program Status Word is also restored by being popped from the stack.

This instruction is a privileged instruction (limited to system mode) and is used to return from an interrupt/exception. An attempt to use RETI in user mode will generate a trap.

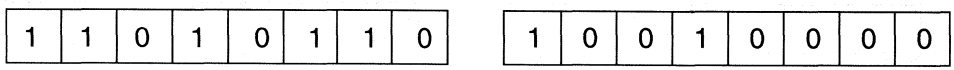
Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: All PSW bits are written by the POP of the PSW value in System mode.

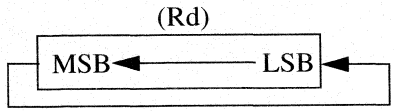
Bytes: 2
Clocks: 10/8 (PZ)

Encoding:



Syntax: RL Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
($dest_0$) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
(count) <- count -1
End While

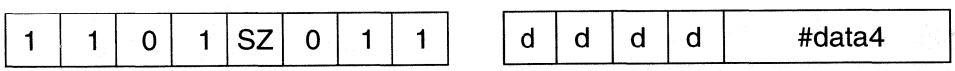
Description: The variable specified by the destination operand is rotated left by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: N, Z

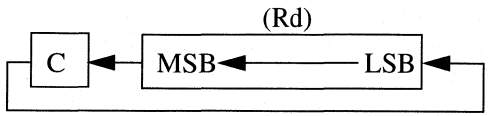
Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

Encoding:



Syntax: RLC Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
(temp) <- (C)
(C) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
($dest_0$) <- (temp)
(count) <- count -1
End While

Description: The variable specified by the destination operand is rotated left through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

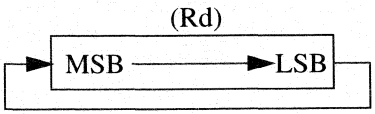
Encoding:

1	1	0	1	SZ	1	1	1

d	d	d	d	#data4

Syntax: RR Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
($dest_{msb}$) <- ($dest_0$)
($dest_{n-1}$) <- ($dest_n$)
(count) <- count -1
End While

Description: If the count operand is greater than 0, the destination operand is rotated right by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15. If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

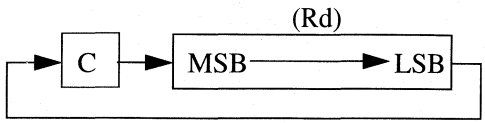
Encoding:

1	0	1	1	SZ	0	0	0

d	d	d	d	#data4

Syntax: RRC Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
(temp) <- (C)
(C) <- ($dest_0$)
($dest_n$) <- ($dest_{n+1}$)
($dest_{msb}$) <- (temp)
(count) <- count -1
End While

Description: If the count operand is greater than 0, the destination operand is rotated right through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.
If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

Encoding:

1	0	1	1	SZ	1	1	1

d	d	d	d	#data4

Syntax: SETB bit

Operation: (bit) <-- 1

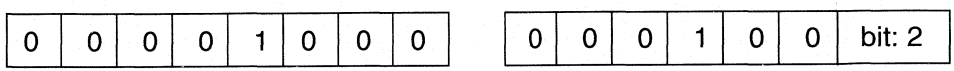
Description: Writes (sets) a 1 to the specified bit.

Size: Bit

Flags Updated:none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: SEXT Rd

Operation: if N = 1
then (Rd) <-- FF in byte mode or FFFF in word mode
if N = 0
then (Rd) <-- 00 in byte mode or 0000 in word mode

Description: Copies the N flag (the sign bit of the last ALU operation) into the destination register. The destination register may be a byte or word register.

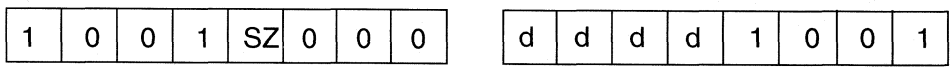
Example:
SEXT.b R1
if the result of the previous operation left the N flag set, then R1 <-- FF

Size: Byte, word

Flags Updated: none

Bytes: 2
Clocks: 3

Encoding:



Syntax: SUB dest, src

Operation: dest <- dest - src

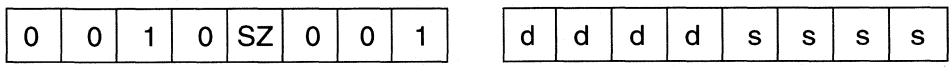
Description: Performs a twos complement binary subtraction of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

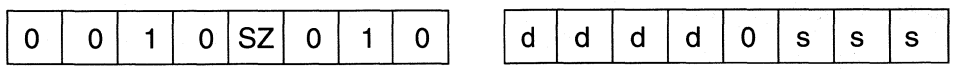
SUB Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) - (Rs)
Encoding:



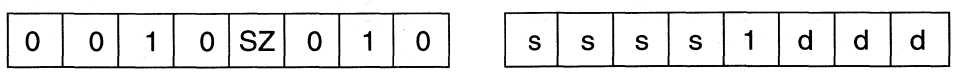
SUB Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs))
Encoding:



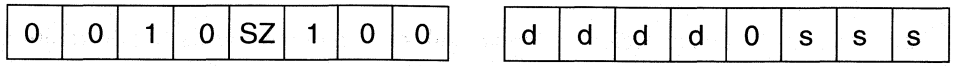
SUB [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
Encoding:



SUB Rd, [Rs+offset8]

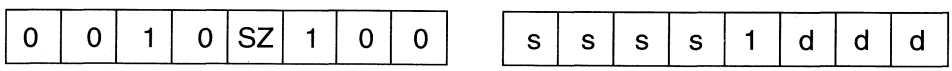
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

SUB [Rd+offset8], Rs

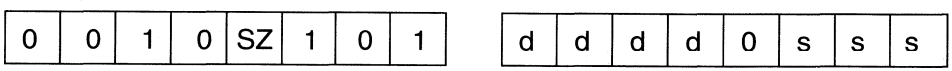
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs)
Encoding:



byte 3: offset8

SUB Rd, [Rs+offset16]

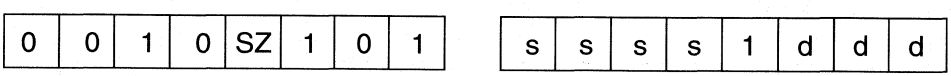
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUB [Rd+offset16], Rs

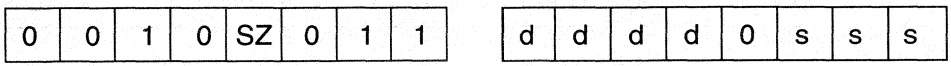
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

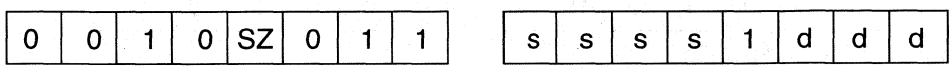
SUB Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



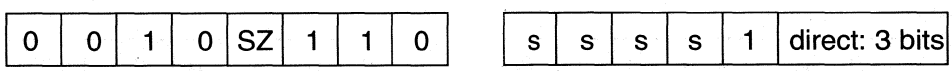
SUB [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



SUB direct, Rs

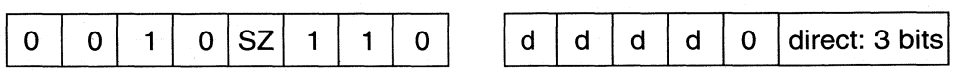
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) - (Rs)
Encoding:



byte 3: lower 8 bits of direct

SUB Rd, direct

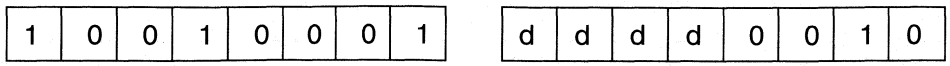
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) - (direct)
Encoding:



byte 3: lower 8 bits of direct

SUB Rd, #data8

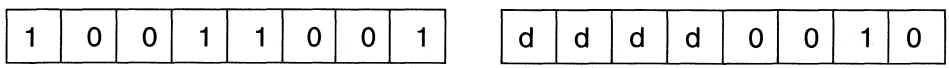
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) - #data8
Encoding:



byte 3: #data8

SUB Rd, #data16

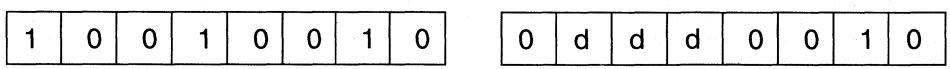
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd], #data8

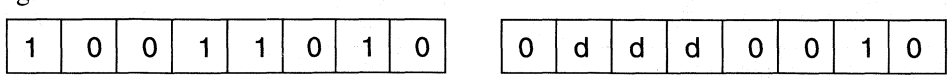
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
Encoding:



byte 3: #data8

SUB [Rd], #data16

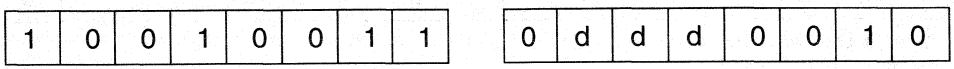
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+], #data8

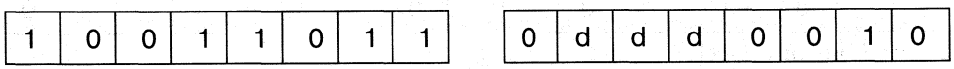
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

SUB [Rd+], #data16

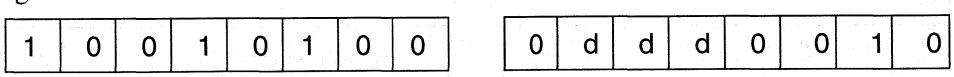
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+offset8], #data8

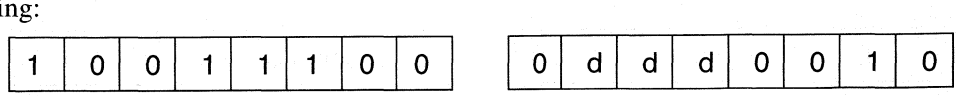
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8
Encoding:



byte 3: offset8
byte 4: #data8

SUB [Rd+offset8], #data16

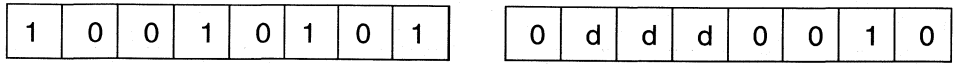
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUB [Rd+offset16], #data8

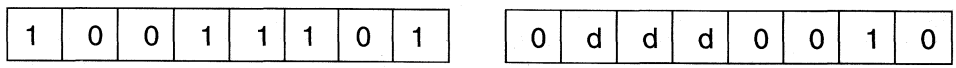
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUB [Rd+offset16], #data16

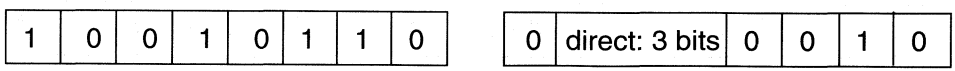
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUB direct, #data8

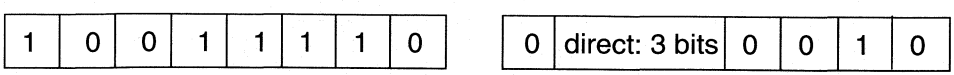
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) - #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

SUB direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) - #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: SUBB dest, src

Operation: dest <- dest - src - C

Description: Performs a twos complement binary addition of the source operand and the previously generated carry bit (borrow) with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is zero (C = 0, i.e., Borrow = 1), the result is exact difference of the operands; if it is one (C = 1, i.e., Borrow = 0), the result is 1 less than the difference in operands.

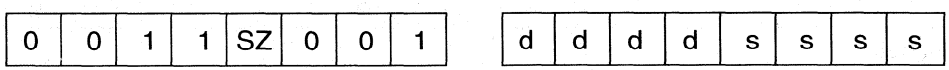
This form of subtraction is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then SUBB is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

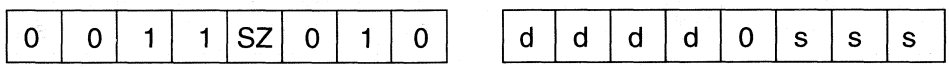
SUBB Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) - (Rs) - (C)
Encoding:



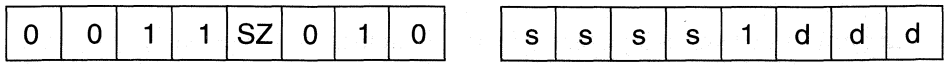
SUBB Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
Encoding:



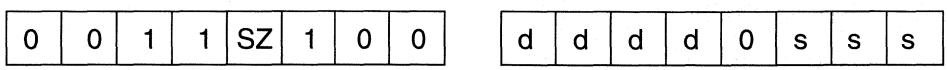
SUBB [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
Encoding:



SUBB Rd, [Rs+offset8]

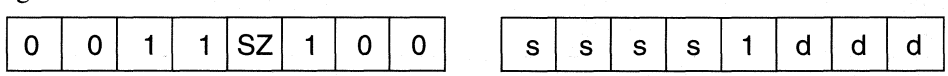
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8) - (C)
Encoding:



byte 3: offset8

SUBB [Rd+offset8], Rs

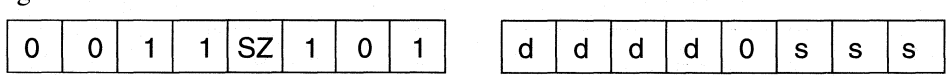
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs) - (C)
Encoding:



byte 3: offset8

SUBB Rd, [Rs+offset16]

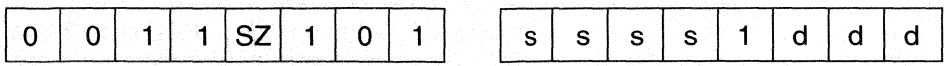
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16) - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUBB [Rd+offset16], Rs

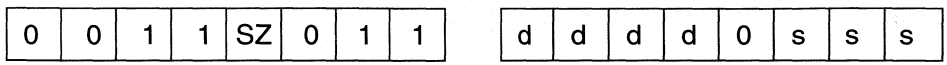
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs) - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

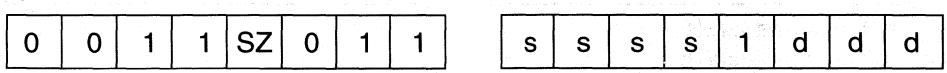
SUBB Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



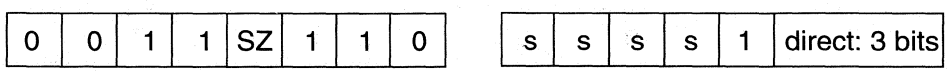
SUBB [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



SUBB direct, Rs

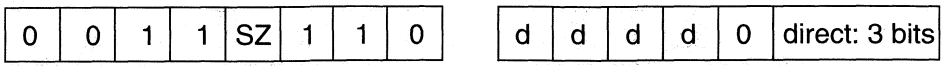
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) - (Rs) - (C)
Encoding:



byte 3: lower 8 bits of direct

SUBB Rd, direct

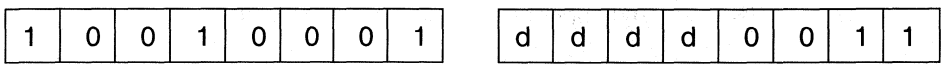
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) - (direct) - (C)
Encoding:



byte 3: lower 8 bits of direct

SUBB Rd, #data8

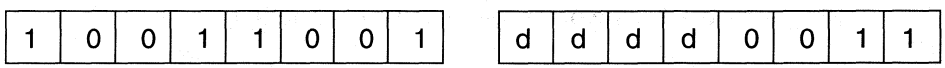
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) - #data8 - (C)
Encoding:



byte 3: #data8

SUBB Rd, #data16

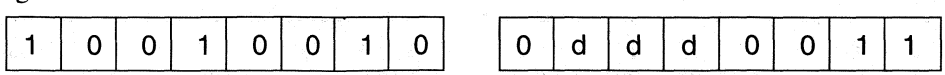
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd], #data8

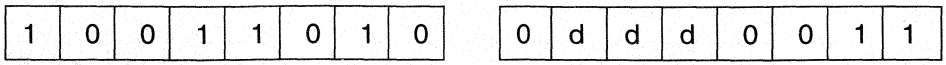
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
Encoding:



byte 3: #data8

SUBB [Rd], #data16

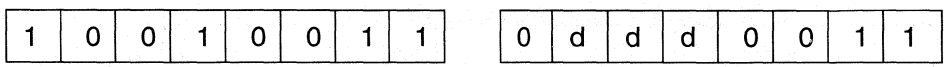
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+], #data8

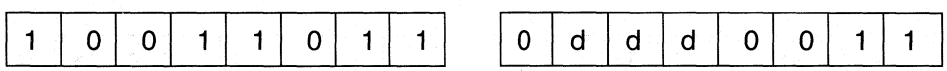
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

SUBB [Rd+], #data16

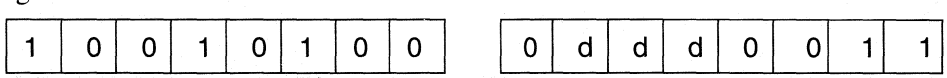
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+offset8], #data8

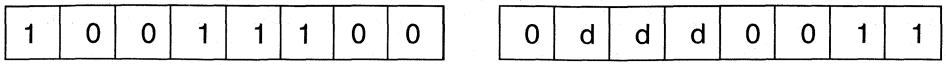
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8 - (C)
Encoding:



byte 3: offset8
byte 4: #data8

SUBB [Rd+offset8], #data16

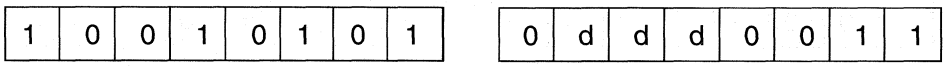
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16 - (C)
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUBB [Rd+offset16], #data8

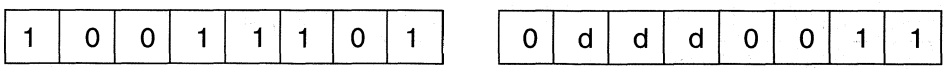
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8 - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUBB [Rd+offset16], #data16

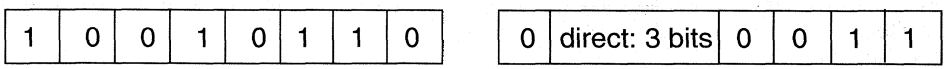
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUBB direct, #data8

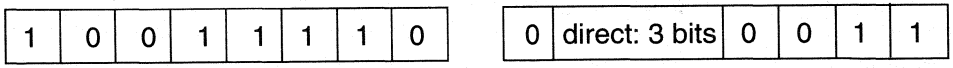
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) - #data8 - (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

SUBB direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) - #data16 - (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: TRAP #data4

Operation:
(PC) <-- (PC) + 2
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (trap vector (#data4))
(PC.15-0) <-- code memory (trap vector (#data4))
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes the specified software trap. The invoked routine is determined by branching to the specified vector table entry point. The RETI, return from interrupt, instruction is used to resume execution after the trap routine has been completed. A trap acts like an immediate interrupt, using a vector to call one of several pieces of code that will be executed in system mode. This may be used to obtain system services for application code, such as altering the data segment register. This is described in more detail in the section on interrupts and exceptions.

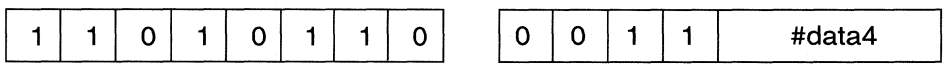
Note: The address of the exception handling routine must be word aligned as the PC is forced to an even address before vectoring to the service routine.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 23/19 (PZ)

Encoding:



Syntax: XCH dest, src

Operation: dest <--> src

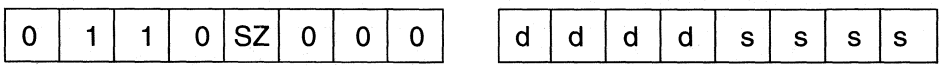
Description: The data specified by the source and destination operands is exchanged.

Size: Byte-Byte, word-word.

Flags Updated: none

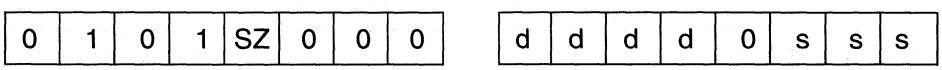
XCH Rd, Rs

Bytes: 2
Clocks: 5
Operation: (Rd) <--> (Rs)
Encoding:



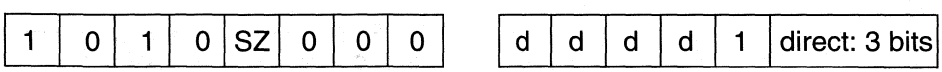
XCH Rd, [Rs]

Bytes: 2
Clocks: 6
Operation: (Rd) <--> ((WS:Rs))
Encoding:



XCH Rd, direct

Bytes: 3
Clocks: 6
Operation: (Rd) <--> (direct)
Encoding:



byte 3: lower 8 bits of direct

Syntax: XOR dest, src

Operation: dest <- dest (XOR) src

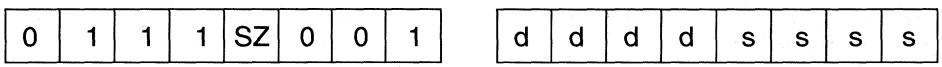
Description: The byte or word specified by the source operand is bitwise logically XORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

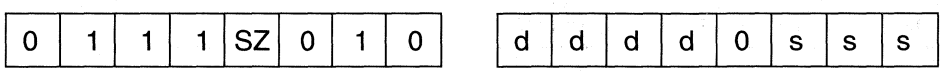
XOR Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) (Rs)
Encoding:



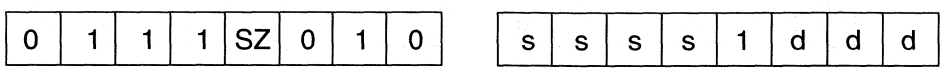
XOR Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
Encoding:



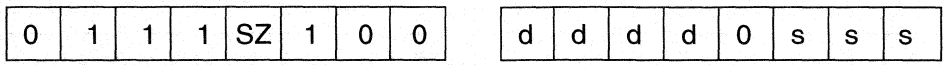
XOR [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
Encoding:



XOR Rd, [Rs+offset8]

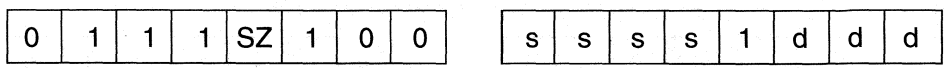
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

XOR [Rd+offset8], Rs

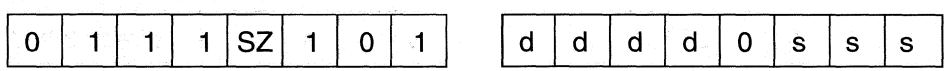
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) (Rs)
Encoding:



byte 3: offset8

XOR Rd, [Rs+offset16]

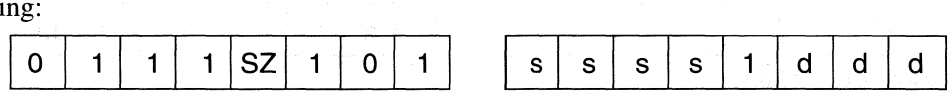
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

XOR [Rd+offset16], Rs

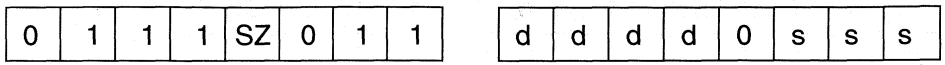
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

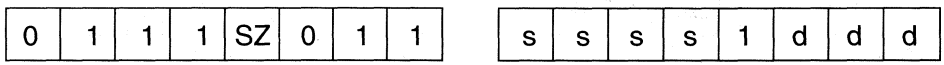
XOR Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



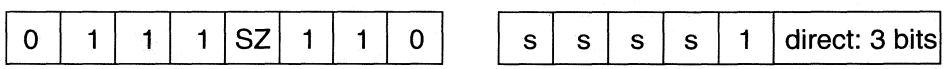
XOR [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



XOR direct, Rs

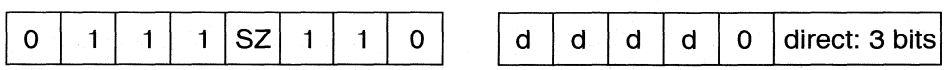
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) (XOR) (Rs)
Encoding:



byte 3: lower 8 bits of direct

XOR Rd, direct

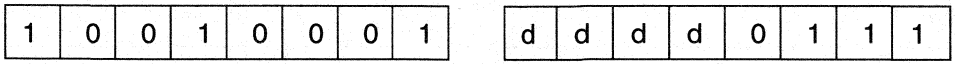
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) (XOR) (direct)
Encoding:



byte 3: lower 8 bits of direct

XOR Rd, #data8

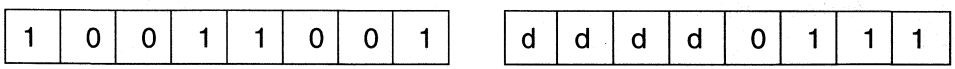
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) #data8
Encoding:



byte 3: #data8

XOR Rd, #data16

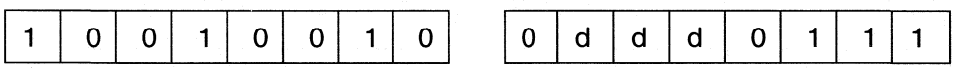
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd], #data8

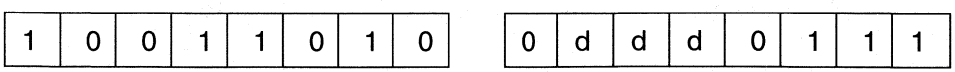
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
Encoding:



byte 3: #data8

XOR [Rd], #data16

Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
Encoding:

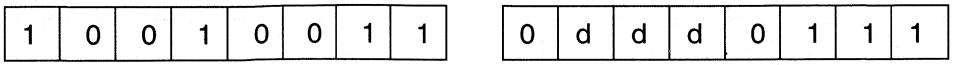


byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+], #data8

Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
(Rd) <-- (Rd) + 1

Encoding:

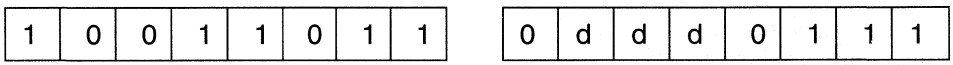


byte 3: #data8

XOR [Rd+], #data16

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
(Rd) <-- (Rd) + 2

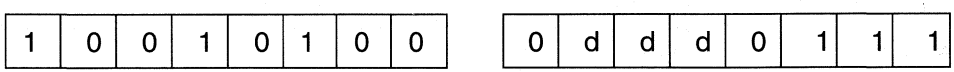
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+offset8], #data8

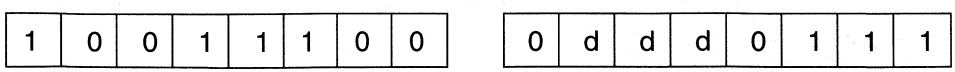
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data8
Encoding:



byte 3: offset8
byte 4: #data8

XOR [Rd+offset8], #data16

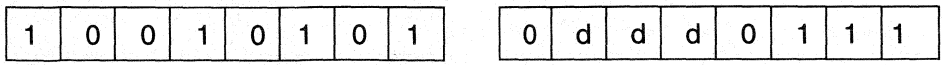
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

XOR [Rd+offset16], #data8

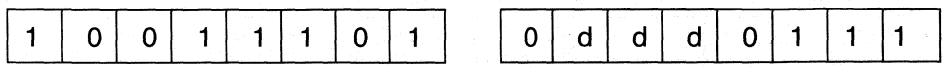
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

XOR [Rd+offset16], #data16

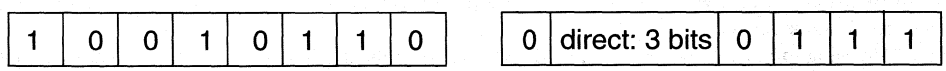
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

XOR direct, #data8

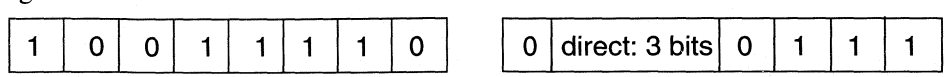
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) (XOR) #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

XOR direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) (XOR) #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

6.6 Summary Of Illegal Operand Combinations On The XA

All but one case are instructions that specify or imply 2 write operations to a single register file location within a single instruction. The other case is a possible corruption of the source register data by an auto-increment before it is read. These conditions are not detected by XA hardware. The instruction/operand combinations indicated should not be used when writing XA code.

Instruction(s) affected	Reason for illegal combination
(any op) Rx, [Rx+]	Auto-increment plus explicit write [1]
mov [Rx+], [Rx+]	Double auto-increment of one register [2]
(any op) [Rx+], Rx	Auto-increment write may corrupt the source register before it is read [3]
NORM Rx, Rx	Result and shift count stored in the same register [4]
XCH Rx, Rx	Double write of a single register [4]
(any op) [Rx+], Ry	Auto-increment plus indirect write to same register [5]
(any op) [Rx+], [Ry+]	Auto-increment plus indirect write to same register [5]
(any op) [Rx+], #data	Auto-increment plus indirect write to same register [5]
XCH Rx, [Rx]	Indirect write plus explicit write to the same register [6]
XCH Rx, direct	Direct write plus explicit write to the same register [7]
POP R7	Stack pointer auto-increment plus explicit write to R7/SP [8]

NOTES:

1 This addressing mode is illegal when the source and destination are the same register. This would cause both a data write and an auto-increment operation to the same register.

2 This instruction is illegal when the source and destination pointer registers are the same register. This would cause two auto-increment operations to the same register.

3 This instruction is illegal when the source and destination are the same register. The source register would be auto-incremented and read at the same time, with an undefined result.

4 This instruction is illegal when the source and destination are the same register. This would cause two writes to the same register.

5 This addressing mode is illegal when the indirect address of the destination points to the pointer register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause both a data write and an auto-increment operation to the same register.

6 This instruction is illegal when the indirect address of the source operand points to the destination register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause two writes to the same register.

7 This instruction is illegal when the direct address of the source operand points to the destination register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause two writes to the same register.

8 A POP to R7 (the stack pointer) would cause both a data write and an auto-increment operation to the same register.

7 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices that are to be accessed, then generates a strobe for the required operation, with data passing in or out on the data bus. Typical bus operations are code read, data read, and data write. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

The following discussion is based on the standard version of the XA external bus. Some specific XA derivatives may have a different implementation of the external bus, or no external bus at all.

7.1 External Bus Signals

For flexibility, the standard XA external bus supports 8 or 16-bit data transfers and a user selectable number of address bits. The maximum number of address lines varies by derivative but may be up to 24. A standard set of bus control signals coordinates activity on the bus. These are described in the following sections.

7.1.1 $\overline{\text{PSEN}}$ - Program Store Enable

The program store enable signal is used to activate an external code memory, such as an EPROM. This active low signal is typically connected to the Output Enable ($\overline{\text{OE}}$) pin of an external EPROM. $\overline{\text{PSEN}}$ remains high when a code read is not in progress.

7.1.2 $\overline{\text{RD}}$ - Read

The bus read signal is also active low. Activity of this signal indicates data read operations on the external bus. $\overline{\text{RD}}$ is typically connected to the pin of the same name on an external peripheral device.

7.1.3 $\overline{\text{WRL}}$ - Write Low Byte

$\overline{\text{WRL}}$ is the external bus data write strobe. It is typically connected to the $\overline{\text{WR}}$ pin of an external peripheral device. When the XA external bus is used in the 16-bit mode, this strobe applies only to the lower data byte, allowing byte writes on the 16-bit bus. The $\overline{\text{WRL}}$ signal is active low.

7.1.4 $\overline{\text{WRH}}$ - Write High Byte

For a 16-bit data bus, a signal similar to $\overline{\text{WRL}}$, but for the upper data byte is needed. The active low signal $\overline{\text{WRH}}$ serves this purpose.

7.1.5 ALE - Address Latch Enable

Since a portion of the XA external bus is used for multiplexed address and data information, that part of the address must be latched outside of the XA so that it will remain constant during the

subsequent read or write operation. The active high ALE signal directs the external latch to allow information to be stored for a data address or a code address. The external latch must close and retain this address when the ALE signal ends, by going low (inactive).

7.1.6 Address Lines

Some of the address lines used by the external bus interface are driven during a complete bus operation and do not need to be latched. In the standard XA bus interface, the lower four address lines are always driven and unlatched in this manner. This is done specifically as part of the optimization of the bus for fetching instructions from external code memory at high speed. This feature will be explained in detail in a later section.

7.1.7 Multiplexed Address and Data Lines

The part of the bus that is used for data transfer is also used for address output from the XA. Prior to asserting the strobe for the bus operation about to be performed, the XA outputs the address for the operation. On the multiplexed portion of the bus, this address is captured by an external latch, as commanded by the ALE signal. After that is done, this part of the bus is free to be used for data transfer either into or out of the XA. The control signals $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, and $\overline{\text{WRH}}$ determine what type of bus operation takes place.

7.1.8 WAIT - Wait

The WAIT input allows wait states to be inserted into any external bus operation. If WAIT is asserted (high) after a bus control strobe ($\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, or $\overline{\text{WRH}}$) is driven by the XA, that bus operation is stretched, and that control strobe continues to be driven by the XA until WAIT goes low again. For this feature to be used, an external circuit must be present to generate the WAIT signal at the appropriate times.

The XA has an internal bus configuration feature that allows programming the various types of external bus cycles to different lengths, so that in most applications the WAIT line will not be needed. This feature will be explained in detail in a later section.

7.1.9 $\overline{\text{EA}}$ - External Access

The $\overline{\text{EA}}$ input determines whether the XA operates in single-chip mode, or begins running code from the internal program memory after reset. If $\overline{\text{EA}}$ is low as Reset goes high, the first code fetch (and all others after that) is made off-chip. If $\overline{\text{EA}}$ is high as Reset goes high, the XA will execute the on-chip code first, but will still attempt to execute instructions from external memory at addresses above the limit of on-chip code. The level on the $\overline{\text{EA}}$ pin is latched as reset goes high, so whatever mode is selected remains valid until the next reset.

On some XA derivatives, the pin used for the $\overline{\text{EA}}$ function may be shared with another function that becomes active after the XA begins code execution.

7.1.10 BUSW - Bus Width

The external XA bus may be configured to be 8 or 16 bits in width. The XA allows the bus width to be programmed in 2 ways. In a system where instructions are initially fetched from on-chip code memory, the user program can configure the external bus size (and many other aspects of the bus) prior to the bus actually being used.

When the initial code fetches must be done using off-chip code memory, however, the XA must know the bus width before the first off-chip code fetch can begin.

On some XA derivatives, the BUSW function may share a pin with some other function. In this case, the level on the BUSW pin is latched as Reset is released and that selection is kept until the next Reset. The secondary function on that pin will be active after Reset when the processor begins executing code normally.

Unlike the $\overline{EA}$ function, the bus width set by the BUSW pin at reset may be over-ridden by a user program, making setting by use of the BUSW pin unnecessary in most systems. Settings in the Bus Configuration Register allow changing the bus size under program control. This feature is covered in more detail in the next section.

7.2 Bus Configuration

The standard XA external bus has a number of configuration options. In addition to the data bus width selection discussed previously, the number of address lines used for external accesses is programmable, as is the bus timing.

7.2.1 8-Bit and 16-Bit Data Bus Widths

The standard XA external bus allows both 8-bit and 16-bit bus widths. BUSW=0 selects an 8-bit bus and BUSW=1 selects a 16-bit bus. On power-up, the XA defaults to the 16-bit bus (due to an on-chip weak pull-up on BUSW). The bus width is determined by the value of the BUSW pin as Reset is released, unless a user program overrides that setting by writing to the Bus Configuration Register (BCR), shown in Figure7.1.

BCR	-	-	-	WAITD	BUSD	BC2	BC1	BC0

WAITD: WAIT disable. Causes the XA external bus interface to ignore the value on the WAIT input. This allows tying the WAIT input high for applications that use internal code and do not need the WAIT function.

BUSD: Bus disable. Causes XA external bus functions to be disabled permanently. The primary purpose of this is to allow prevention of inadvertent activation of the bus by an instruction pre-fetch when the XA is executing code near the end of the on-chip code memory.

BC2 - BC0: These bits select the XA external bus configuration, specifically the number of data bits and the number of address lines. The supported combinations are shown below.

000 : 8-bit data bus, 12 address lines
001 : 8-bit data bus, 16 address lines
010 : 8-bit data bus, 20 address lines
011 : 8-bit data bus, 24 address lines
100 : < function reserved >
101 : < function reserved >
110 : 16-bit data bus, 20 address lines
111 : 16-bit data bus, 24 address lines

"-" Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Figure 7.1 Bus Configuration Register (BCR)

Figures 7.2 and 7.3 show the address and data functions present on XA bus related pins when used with each available bus width.

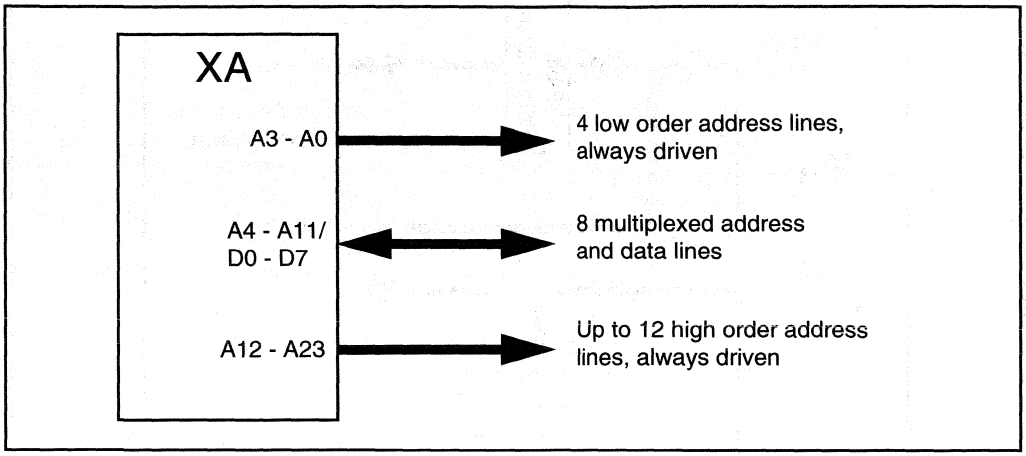


Figure 7.2 8-Bit External Bus Configuration

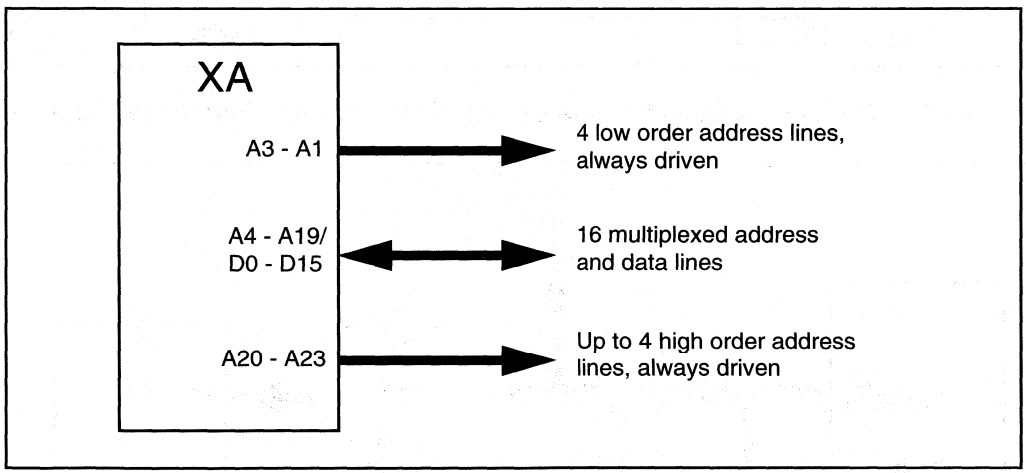


Figure 7.3 16-Bit External Bus Configuration

7.2.2 Typical External Device Connections

Many possibilities exist for connecting and using external devices with the XA bus. The bus will support EPROMs, RAMs, and other memory devices, as well as peripheral devices such as UARTs, and parallel port expanders. The following diagrams show a generalized connection of devices for 8-bit and 16-bit XA bus modes.

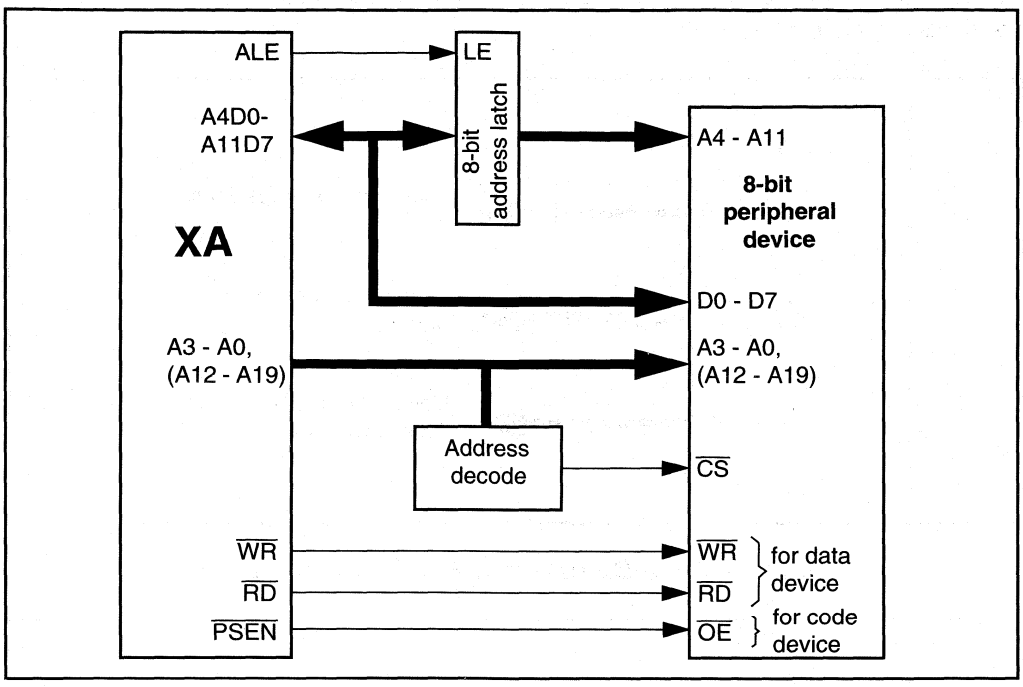


Figure 7.4 Typical XA External Bus Connections for 8-Bit Peripheral Devices

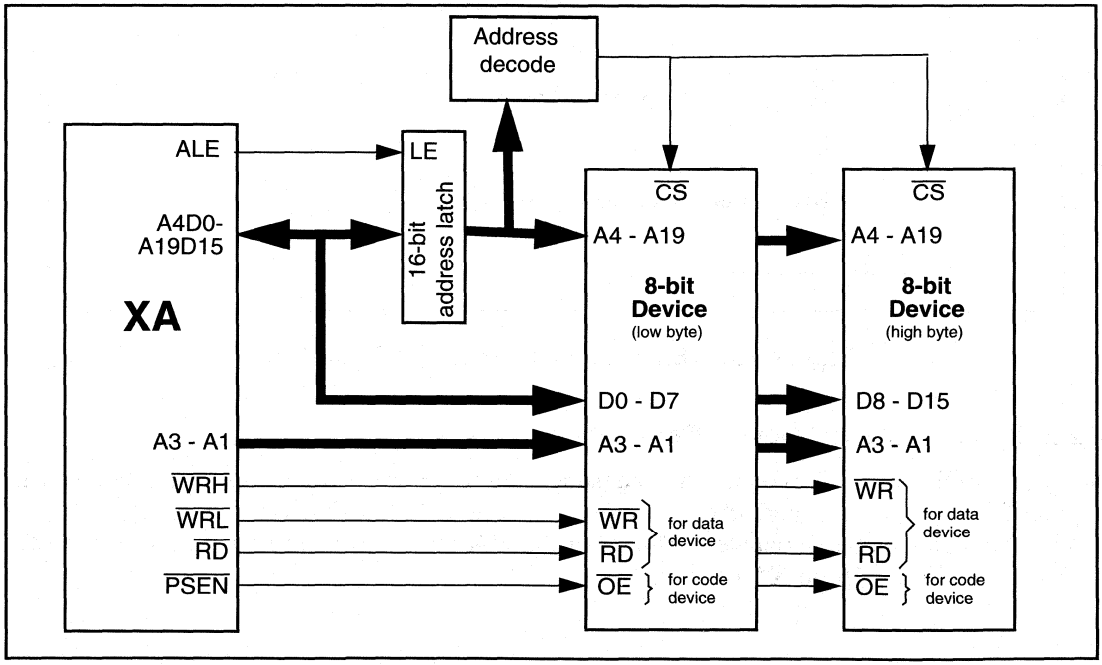


Figure 7.5 Typical XA External Bus Connections for 16-Bit Peripheral Devices

7.3 Bus Timing and Sequences

The standard XA external bus allows programming the widths of the bus control signals ALE, PSEN, WRL, WRH, and RD. There is also an option to extend the data hold time after a write operation. The combinations available will allow interfacing most devices to the XA directly without the need for special buffers or a WAIT state generator. Note that there is always a "rest clock" after any type of bus cycle except part of a burst mode code read. That is, when a bus cycle is completed and the bus strobe de-asserted, no new bus cycle will be begun until one clock has passed with no bus activity.

7.3.1 Code Memory

Interfacing with external code memory, typically in the form of EPROMs, is enabled by the $\overline{\text{PSEN}}$ control signal. If the XA is configured to execute internal code memory at reset, by the setting of the $\overline{\text{EA}}$ pin, it will automatically begin to fetch external code if the program crosses the boundary from internal to external code space. The location of this boundary varies for different XA derivatives, depending on the size of the internal code memory for each part.

Since the XA employs a pre-fetch queue in order to optimize instruction execution times, fetching of external instructions may begin before program execution actually crosses the on/off-chip code memory boundary. If a branch or subroutine return is located near the end of on-chip code memory, the off-chip fetch would be unnecessary, and may in fact cause problems if the XA ports that implement the external bus are being used for other purposes. For this reason, the BUSD (bus disable) bit in the Bus Configuration Register (BCR) is provided to prevent the XA from using the external bus for code or data operations.

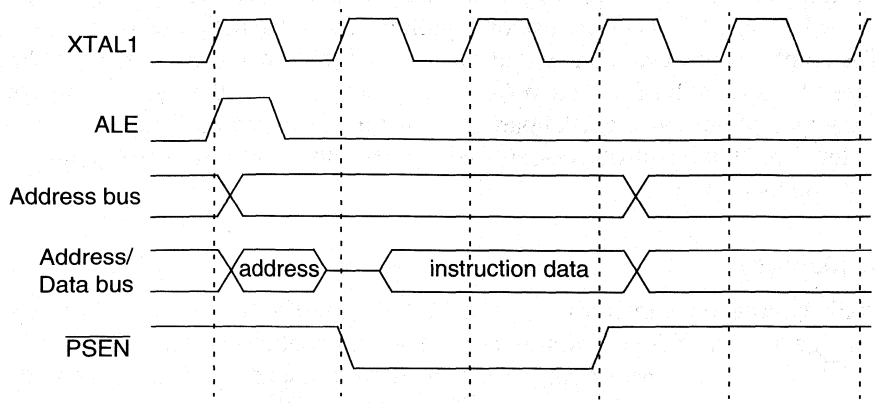
Note also that external code read cycles may sometimes be aborted by the XA. This happens when a code pre-fetch is occurring on the bus and the XA must execute a branch. The instruction data from the code pre-fetch will not be needed, so the bus cycle will be terminated immediately. This may appear as an ALE with no subsequent PSEN strobe, or a PSEN strobe that is shorter than that specified by the bus timing registers.

Code Read with ALE

The classic operation of a multiplexed address and data bus involves the issuance of an address, along with its associated control signal, for every bus cycle. The XA uses the bus control signal ALE to indicate that an address is on the bus that must be latched through the following code or data operation. The following diagram shows a code memory fetch in a cycle using ALE.

Burst Code Read (No ALE)

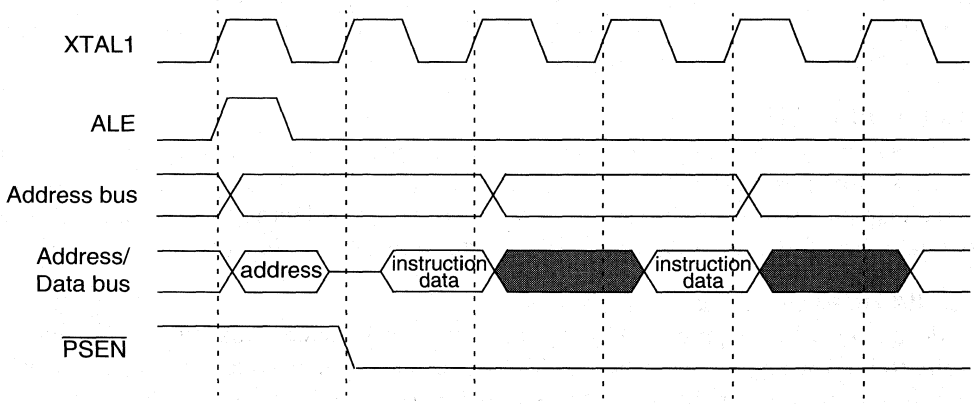
The XA does not always require an ALE cycle for every code fetch. This feature is included specifically to improve performance when the XA executes code from external memory, while increasing the access time available for the external memory device. Because the lower four address lines of the external bus are always driven, not multiplexed, the XA can access up to 16 bytes (or 8 words) of sequential code memory each time an ALE is issued. This type of fast sequential code read is called a burst read. Of course, any type of jump, branch, interrupt, or other change in sequential program flow will require an ALE in order to change the code fetch address in a non-sequential manner. Any data operation (read or write) on the XA external bus also requires an ALE cycle and will cause any subsequent external code fetch to begin with an ALE cycle also.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, CRA1/0 = 01.

Figure 7.6 Typical External Code Read Using ALE

The following diagram shows a typical sequential code fetch where no ALE is issued between code reads. Also note that the $\overline{\text{PSEN}}$ bus control signal does not toggle, but remains asserted throughout the burst code read



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, CR1/0 = 01, CRA1/0 = 00.

Figure 7.7 Burst Mode (Sequential) External Code Read

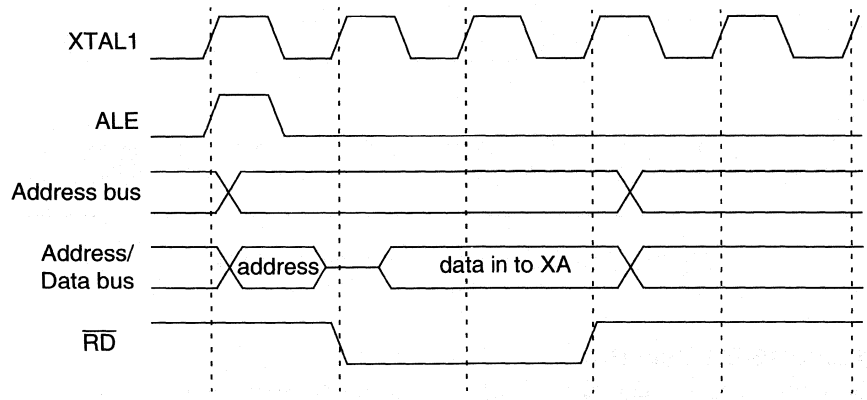
7.3.2 Data Memory

Reads and writes on the XA external bus are controlled through the use of the $\overline{RD}$, $\overline{WRL}$, and $\overline{WRH}$ signals. Since the XA bus supports both 8-bit and 16-bit widths, as well as byte and word read and write operations, several different versions of the basic bus cycles are possible. These are described in the following sections.

Data memory, like code memory, has a boundary where the internal data memory ends, and above which the XA will switch to the external bus in order to act on data memory. This on/off-chip data memory boundary may be in a different place for various XA derivatives, depending upon the amount of internal data memory built into a specific derivative.

Typical Data Read

A simple byte read on an 8-bit bus or any read on a 16-bit bus both begin with an ALE cycle, where the XA presents the address of the data location that is to be read on the bus. This is followed by the assertion of the $\overline{RD}$ strobe, that causes the external device to present its data on the bus. This process is shown in the diagram below.

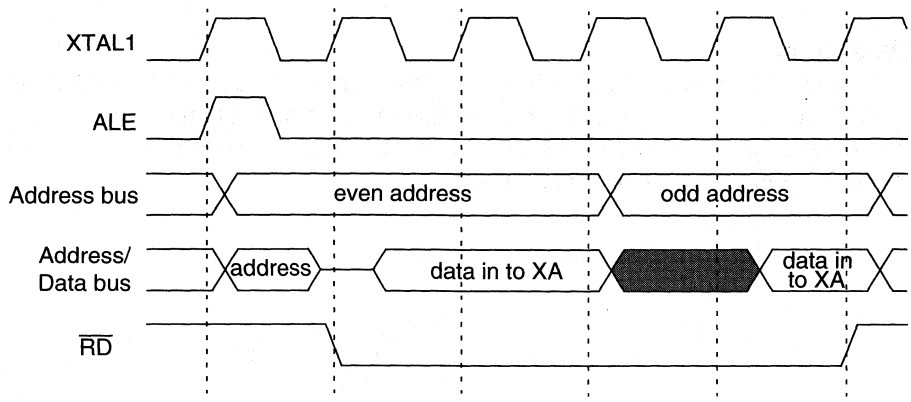


Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DRA1/0 = 01.

Figure 7.8 Typical External Data Read

Word Read on an 8-Bit Data Bus

When the XA external bus is configured for an 8-bit data width, a word read operation is automatically performed as two byte reads at sequential addresses. Since the XA CPU requires word operations to be performed at even addresses, the second half of any word read on a byte-wide bus always uses the same upper address latched by ALE. for this operation, the low order byte first is read at the even byte address, then the high order byte is read at the next (odd) address. So, only one ALE is required in this case. The diagram below shows this sequence.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DRA1/0 = 01, DR1/0 = 01.

Figure 7.9 Word Read on 8-Bit Data Bus

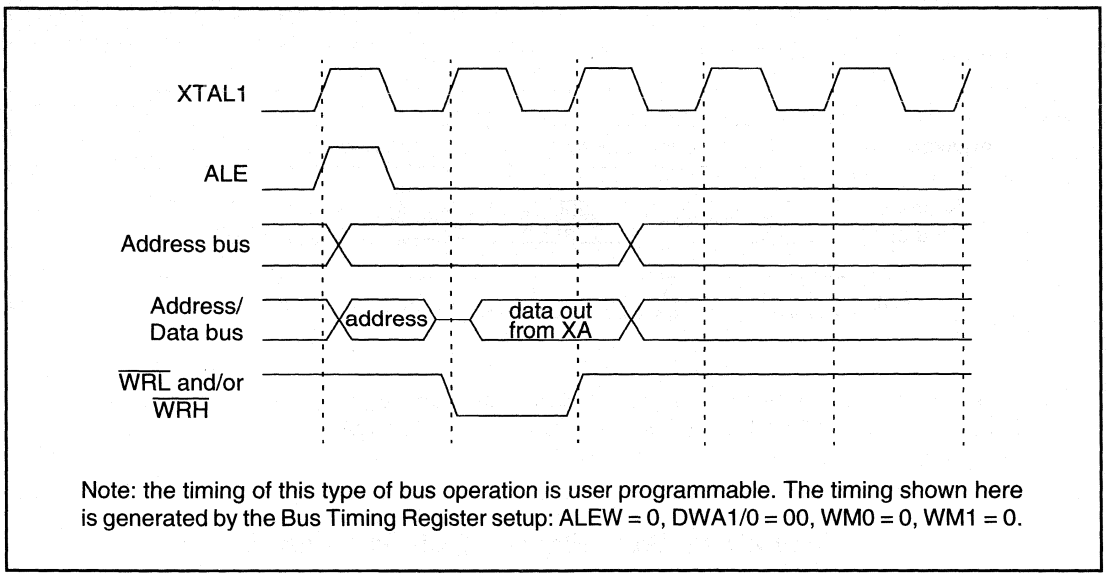
Byte Read on a 16-Bit Data Bus

When an instruction causes a read of one byte of data from the external bus, when it is configured for 16-bit width, a simple read operation is performed. This results in 16 bits of data being received by the XA, which uses only the byte that was requested by the program. There is no way to distinguish a byte read from a word read on the external bus when it is configured for a 16-bit width.

Typical Data Write

A data write operation begins with an ALE cycle, like a read operation, followed by the assertion of one or both of the write strobes, $\overline{WRL}$ and $\overline{WRH}$. This simple bus cycle applies to byte writes on an 8-bit data bus and all writes on a 16-bit data bus.

A byte write on an 8-bit data bus will always use only the $\overline{WRL}$ strobe. A byte write on a 16-bit data bus will always use either the $\overline{WRL}$ or $\overline{WRH}$ strobe, depending on whether the byte is at an even or odd address. A word write on a 16-bit bus requires the assertion of both the $\overline{WRL}$ and $\overline{WRH}$ strobes. The simple data write cycle is shown below.

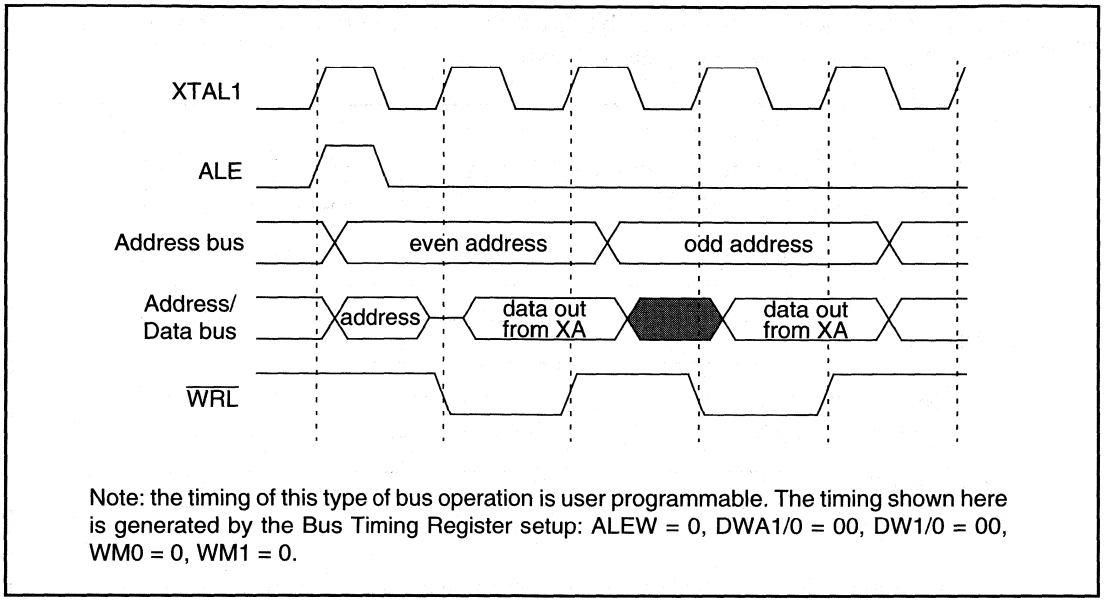


Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DWA1/0 = 00, WM0 = 0, WM1 = 0.

Figure 7.10 Typical External Data Write

Word Write on an 8-Bit Data Bus

When a word write operation is done with the bus configured to an 8-bit width, the XA automatically performs two byte writes. First, the low order byte is written (at the even byte address), then the high order byte is written at the next (odd) address. As with a word read on an 8-bit bus, this requires only a single ALE cycle at the beginning of the process. This sequence is shown in the following diagram.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DWA1/0 = 00, DW1/0 = 00, WM0 = 0, WM1 = 0.

Figure 7.11 Word Write on 8-Bit Data Bus

External Bus Signal Timing Configuration

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, data read and write cycle lengths, and data hold time. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

Programmable bus timing is controlled by settings found in the Bus Timing Register SFRs, named BTRH, and BTRL, shown in Figures 7.12 and 7.13.

BTRH	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0

DW1, DW0: Data Write without ALE. Applies only to the second half of a 16-bit write operation when the bus is configured to 8 bits.
00 : Data write cycle is 2 clock in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DWA1, DWA0: Data Write with ALE. Selects the length (in CPU clocks) of the entire data write cycle, including ALE.
00 : Data write cycle is 2 clocks in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DR1, DR0: Data Read without ALE. Applies only to the second half of a 16-bit read operation when the bus is configured to 8 bits.
00 : Data read cycle is 1 clock in duration.
01 : Data read cycle is 2 clocks in duration.
10 : Data read cycle is 3 clocks in duration.
11 : Data read cycle is 4 clocks in duration.

DRA1, DRA0: Data Read with ALE. Selects the length (in CPU clocks) of the entire data read cycle, including ALE.
00 : Data read cycle is 2 clocks in duration.
01 : Data read cycle is 3 clocks in duration.
10 : Data read cycle is 4 clocks in duration.
11 : Data read cycle is 5 clocks in duration.

Notes:
- See text regarding disallowed bus timing combinations.
- The bit pairs DW1:0, DWA1:0, DR1:0, DRA1:0, CR1:0, and CRA1:0 determine the length of entire bus cycles of different types. Bus cycles with an ALE begin when ALE is asserted. Bus cycles without an ALE begin when the bus strobe is asserted or when the address changes (in the case of burst mode code reads). Bus cycles end either when the bus strobe is de-asserted or when data hold time is completed (in the case of a data write with extra hold time, see bit WM0).

Figure 7.12 Bus Timing Register High Byte (BTRH)

BTRL	WM1	WM0	ALEW	-	CR1	CR0	CRA1	CRA0

WM1: Write Mode 1. Selects the width of the write pulse.
0 : Write pulse (WR) width is 1 CPU clock.
1 : Write pulse (WR) width is 2 CPU clocks.

WM0: Write Mode 0. Selects the data hold time.
0 : Data hold time is minimum (0 clocks).
1 : Data hold time is 1 CPU clock.

ALEW: ALE width selection. Determines the duration of ALE pulses.
0 : ALE width is one half of one CPU clock.
1 : ALE width is one and a half CPU clocks.

CR1, CR0: Code Read. Selects the length of a code read cycle when ALE is not used.
00 : Code read cycle is 1 clocks in duration.
01 : Code read cycle is 2 clocks in duration.
10 : Code read cycle is 3 clocks in duration.
11 : Code read cycle is 4 clocks in duration.

CRA1, CRA0: Code Read with ALE. Selects the length of a code read cycle when ALE is used prior to $\overline{\text{PSEN}}$ being asserted.
00 : Code read cycle is 2 clocks in duration.
01 : Code read cycle is 3 clocks in duration.
10 : Code read cycle is 4 clocks in duration.
11 : Code read cycle is 5 clocks in duration.

"-" Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Notes:
- See text regarding disallowed bus timing combinations.
- The bit pairs DW1:0, DWA1:0, DR1:0, DRA1:0, CR1:0, and CRA1:0 determine the length of entire bus cycles of different types. Bus cycles with an ALE begin when ALE is asserted. Bus cycles without an ALE begin when the bus strobe is asserted or when the address changes (in the case of burst mode code reads). Bus cycles end either when the bus strobe is de-asserted or when data hold time is completed (in the case of a data write with extra hold time, see bit WM0).

Figure 7.13 Bus Timing Register Low Byte (BTRL)

Disallowed Bus Timing Configurations

Some possible combinations of bus timing register settings do not make sense and the XA cannot produce working bus signals that match those settings. The disallowed combinations occur where the sum of the specified components of a bus cycle exceed the specified length of the entire cycle. Two simple rules define the allowed/disallowed combinations. Violating these rules may result in incomplete bus cycles, for example a data read cycle in which an address and ALE pulse are output, but no read strobe ($\overline{RD}$) is produced.

For data write cycles on the external bus there are two conditions that must be met. The first applies to data write cycles with no ALE:

$$WM1 + WM0 \leq DW1:0$$

This says that the sum of the timing values defined by the WM1 and WM0 fields must be less than or equal to the timing value defined by the DW field. Note that this is the value of the timing durations that they specify. For example, if the WM1 field specifies a 2 clock write pulse and the WM0 field specifies a 1 clock data hold time, those two times together (3 clocks) must be less than or equal to the timing specified by the DW1:0 field. In this case the DW1:0 field must specify a total bus cycle duration of at least 3 clocks. The other rule uses the same structure, as follows.

A second requirement applies to write cycles with ALE:

$$ALEW + WM1 + WM0 \leq DWA1:0$$

The configuration for data read has only one requirement, which applies to data read cycles with ALE:

$$ALEW + 1 \leq DRA1:0$$

The configuration for code read also has only one requirement, which applies to code read cycles with ALE:

$$ALEW + 1 \leq CRA1:0$$

7.3.3 Reset Configuration

Upon reset, at the time of power up or later, the XA bus is initially configured in certain ways. As previously discussed, the pins $\overline{EA}$ and BUSW select whether the XA will begin operation from internal code, and whether the bus will be 8-bits or 16-bits.

The values for the programmable bus timing are also set to a default value at reset. All of the timing values are set to their maximum, providing the slowest bus cycles. This setting allows for the slowest external devices that may be sued with the XA without WAIT generation logic. The user program should set the bus timing to the correct values for the specific application in the system initialization code. Refer to the data sheet for a particular XA derivative for details of the values found in registers and SFRs after reset.

7.4 Ports

I/O ports on any microcontroller provide a connection to the outside world. The capabilities of those I/O ports determine how easily the microcontroller can be interfaced to the various external devices that make up a complete application. The standard XA I/O ports provide a high degree of versatility through the use of programmable output modes and allow easy connection to a wide variety of hardware.

7.4.1 I/O Port Access

The standard on-chip I/O ports of the XA are accessed as SFRs. The SFR names used for these ports begin with port 0, called P0. Port numbers and names go up in sequence from there, to the number of ports on a specific XA derivative. Ports are normally identified by their names in assembler source code, such as: "MOV P1,#0". This instruction causes the value 0 to be written to port 1.

XA I/O ports are typically bit addressable, meaning that individual port bits are readable, writable, and testable. An instruction using a port bit looks like this: "SETB P2.1". This particular example would result in the second lowest bit in port 2 (bit 1) having a 1 written to it.

Reading of a Port Pin Versus the Port Latch

Each I/O port has two important logic values associated with it. The first is the contents of the port latch. When data is written to a port, it is stored in the port latch. The second value is the logic level of the actual port pin, which may be different than the port latch value, especially if a port pin is being used as an input.

When a port is explicitly read by an instruction, the value returned is that from the pin. When a port is read intrinsically, in order to perform some operation and store the value back to the port, the port latch is read. This type of operation is called a read-modify-write.

1) The following instructions cause read-modify-write operations, and read the port latch when a port or port bit is specified as the destination:

```
ADD     Px, ...
ADDC    Px, ...
ADDS    Px, ...
AND     Px, ...
DJNZ    Px, ...
OR      Px, ...
SUB     Px, ...
SUBB    Px, ...
XOR     Px, ...

CLR     Px.y
JBC     Px.y, rel8
MOV     Px.y, C
SETB    Px.y
```

2) The following instruction reads the port pins when a port is specified as the destination operand:

```
CMP     Px, ...
```

3) When a port or port bit is specified as a source in any instruction, the port pin is always read.

Figure 7.14 How ports are read.

7.4.2 Port Output Configurations

Standard XA I/O ports provide several different output configurations. One is the 80C51 type quasi-bidirectional port output. Others are open drain, push-pull, and high impedance (input only). It is important to note that the port configuration applies to a pin even if that pin is part of the external bus. Bus pins should normally be configured to push-pull mode. Also, the port latches for pins that are to be used as part of the external bus must be set to one (which is the reset state). A zero in a port latch will override bus operations and force a zero on the corresponding bus position.

The port configuration is controlled by settings in two SFRs for each port. One bit in each port configuration register is associated with a port pin in the corresponding bit position. These port configuration SFRs are called: PnCFGA and PnCFGB, where "n" is the port number. So, the configuration registers for port 1 are named P1CFGA and P1CFGB. The table below shows the port control bit combinations and the associated port output modes.

Table 7.1

PnCFGB	PnCFGA	Port Output Mode
0	0	Open drain.
0	1	Quasi-bidirectional (default).
1	0	High impedance.
1	1	Push-pull.

7.4.3 Quasi-Bidirectional Output

The default port output configuration for standard XA I/O ports is the quasi-bidirectional output that is common on the 80C51 and most of its derivatives. This output type can be used as both an input and output without the need to reconfigure the port. This is possible because when the port outputs a logic high, it is weakly driven, allowing an external device to pull the pin low. When the pin is pulled low, it is driven strongly and able to sink a fairly large current. These features are somewhat similar to an open drain output except that there are three pullup transistors in the quasi-bidirectional output that serve different purposes.

One of these pullups, called the "very weak" pullup, is turned on whenever the port latch for a particular pin contains a logic 1. The very weak pullup sources a very small current that will pull the pin high if it is left floating.

A second pullup, called the "weak" pullup, is turned on when the port latch for its associated pin contains a logic 1 and the pin itself is a logic 1. This pullup provides the primary source current for a pin that is outputting a 1, and can drive several TTL loads. If a pin that has a logic 1 on it is pulled low by an external device, the weak pullup turns off, and only the very weak pullup remains on. In order to pull the pin low under these conditions, the external device has to sink enough current to overpower the weak pullup and pull the voltage on the port pin below its input threshold.

The third (and final) pullup is referred to as the "strong" pullup. This pullup is included to speed up low-to-high transitions on a port pin when the port latch changes from 0 to 1. When this occurs, the strong pullup turns on for a brief time, two CPU clocks, pulling the port pin high quickly, then turning off again.

The quasi-bidirectional output structure normally provides a means to have mixed inputs and outputs on port pins without the need for special configurations. However, it has several drawbacks that can be problems in certain situations. For one thing, quasi-bidirectional outputs have a very small source current and are therefore not well suited to driving certain types of loads. They are especially unsuited to directly drive the bases of external NPN transistors, a common method of boosting the current of I/O pins.

Also, since the weak pullup turns off when a port pin is actually low, and the strong pullup turns on only for a brief time, it is possible that under certain port loading conditions, the port pin will get "stuck" low and cannot be driven high. This tends to happen when an external device being driven by the port pin has some leakage to ground that is larger than the current supplied by the very weak pullup of the quasi-bidirectional port output. If there is also a fairly large capacitance on the pin, from a combination of the wiring itself and the pin capacitance of the device(s) connected to the pin, the strong pullup may not succeed in pulling the pin high enough while it is turned on. When the strong pullup is then turned off, the leakage of the external device pulls the pin low again, since only the very weak pullup is turned on at that point and the leakage is greater than the very weak pullup source current. These issues are the reason for enhancing the port configurations of the XA.

A diagram of the quasi-bidirectional output structure is shown in the figure below.

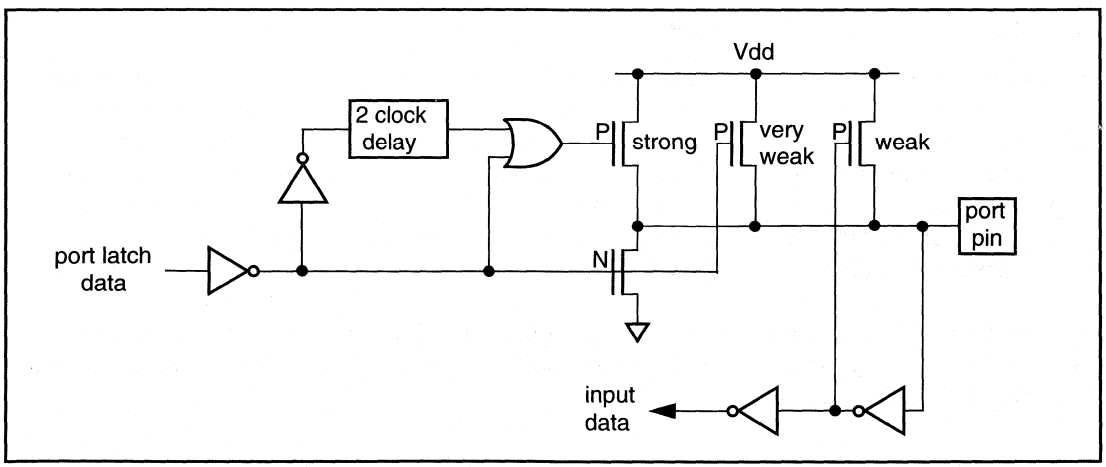


Figure 7.15 Structure of the Quasi-Bidirectional Output Configuration

Open Drain Output

Another port output configuration provided by the standard XA I/O ports is open drain. This configuration turns off all pullups and only drives the pulldown transistor of the port driver when the port latch contains a logic 0. To be used as a logic output, a port configured in this manner must have an external pullup, typically a resistor tied to Vdd. The pulldown for this mode is the same as for the quasi-bidirectional mode.

An advantage of the open drain output is that is may be used to create wired AND logic. Several open drain outputs of various devices can be tied together, and any one of them can drive the wire low, creating a logical AND function without using a logic gate. The figure below show the structure of the open drain output.

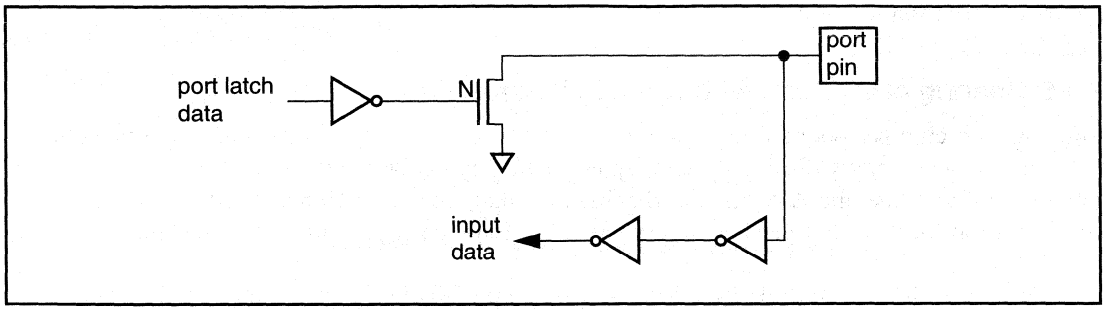


Figure 7.16 Structure of the Open Drain Output Configuration

Push-Pull Output

The push-pull output mode has the same pulldown structure as both the open drain and the quasi-bidirectional output modes, but provides a continuous strong pullup when the port latch contains a logic 1. This mode uses the same pullup as the strong pullup for the quasi-bidirectional mode. The push-pull mode may be used when more source current is needed from a port output. The output structure for this mode is shown below.

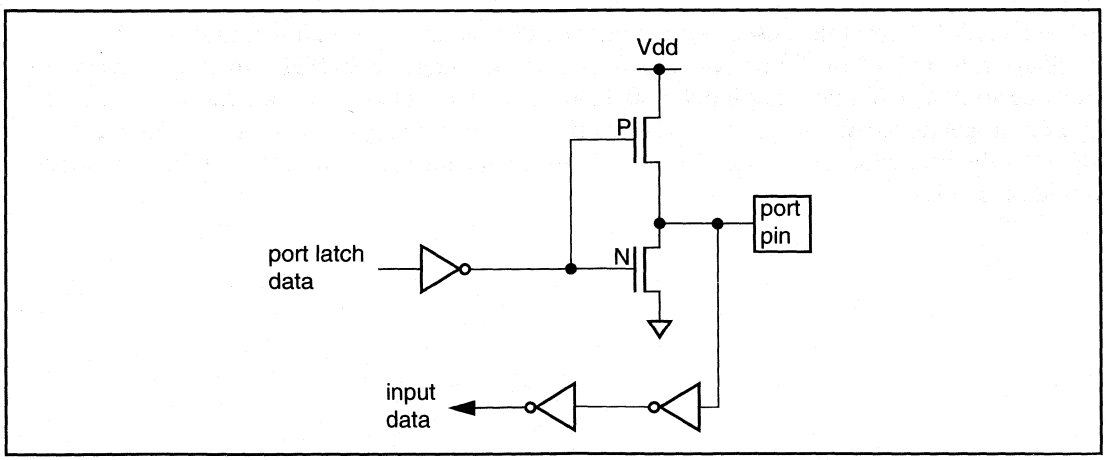


Figure 7.17 Structure of the Push-Pull Output Configuration

High Impedance Output

The final XA port output configuration is called high impedance mode. This mode simply turns all output drivers on a port pin off. Thus, the pin will not source or sink current and may be used effectively as an input-only pin with no internal drivers for an external device to overcome.

7.4.4 Reset State and Initialization

Upon chip reset, all of the port output configurations are set to quasi-bidirectional, and the port latches are written with all ones. The quasi-bidirectional output type is a good default at power-up or reset because it does not source a large amount of current if it is driven by an external device, yet it does not allow the port pin to float. A floating input pin on a CMOS device can cause excess current to flow in the pin's input circuitry, and of course all port pins have input circuits in addition to outputs.

7.4.5 Sharing of I/O Ports with On-Chip Peripherals

Since XA on-chip peripheral devices share device pins with port functions, some care must be taken not to accidentally disable a desired pin function by inadvertently activating another function on the same pin. A peripheral that has an output on a pin will use the I/O port output configuration for that pin (quasi-bidirectional, open drain, push-pull, or high impedance).

The method of sharing multiple functions on a single pin involves a logic AND of all of the functions on a pin. So, if a port latch contains a zero, it will drive that port pin low, and any peripheral output function on that pin is overridden. Conversely, an on-chip peripheral outputting a zero on a pin prevents the contents of the port latch from controlling the output level. It is usually not an issue to avoid turning on an alternate peripheral function on a pin accidentally, since most peripherals must be either explicitly turned on or activated by a write to one of their SFRs. It is more likely that a user program could erroneously write a zero to a port latch bit corresponding to a pin with a peripheral function that is being used and therefore disable that function. The simple rule to follow is: never write a zero to a port bit that is associated with an active on-chip peripheral, or that should only be used an input.

When an XA I/O port pin is used as an input for a peripheral function, it is sampled at the oscillator rate divided by 2. For example, if an XA is running at a 20 MHz clock (giving a 50 ns clock period), an external timer input would have to remain in the same state for at least 100 ns in order to guarantee that it is sampled correctly. This gives a maximum frequency for such inputs as the oscillator rate divided by 4. In this example, the maximum external timer input rate would be 5 MHz.

8 Special Function Register Bus

The Special Function Register Bus or SFR Bus is the means by which all Special Function Registers are connected to the XA CPU so that they may be read and written by user programs. This includes all of the registers contained in peripherals such as Timers and UARTs, as well as some CPU registers such as the PSW. CPU registers communicate functionally with the CPU via direct connections, but read and write operations performed on them are routed through the SFR bus.

The SFR bus provides a common interface for the addition of any new functions to the XA core, thus supplying the means for building a large and varied microcontroller derivative family. This is illustrated in Figure 8.1.

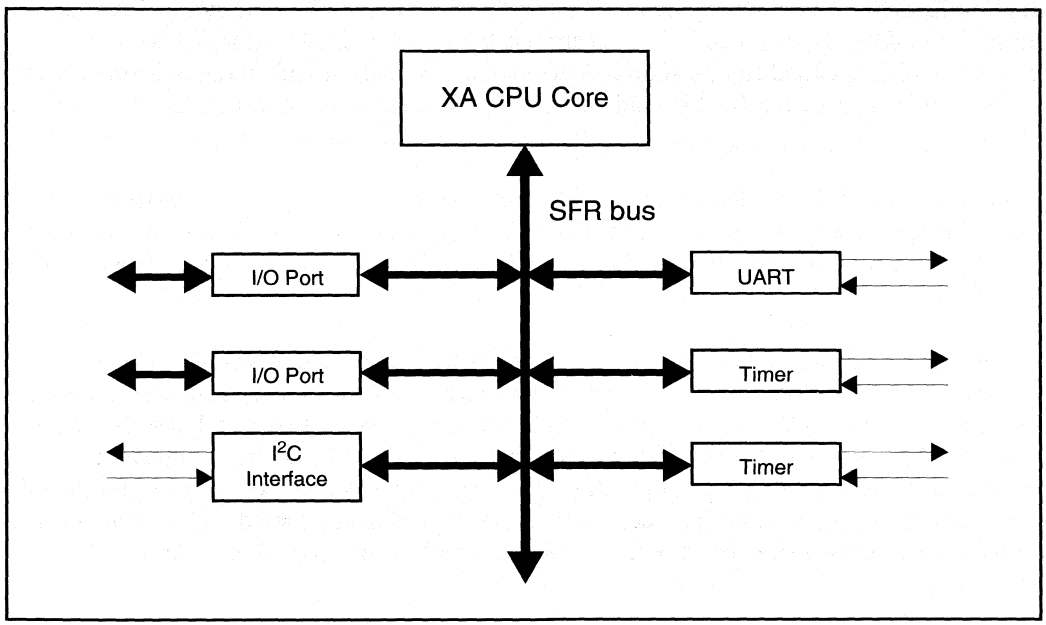


Figure 8.1. Example of peripheral functions connected to the XA SFR bus.

8.1 Implementation and Possible Enhancements

The SFR bus interface is itself not part of the XA CPU core, but a separate functional block. Since the SFR bus controller is a separate block, writes to SFRs may occur simultaneously with the beginning of execution of the next instruction. If the next instruction attempts to access the SFR bus while it is still busy, the instruction execution will stall until the SFR bus becomes available. SFR bus read and write clocks each take 2 CPU clocks to complete. However, the starting time of those 2 clocks has a one clock uncertainty, so the time from the SFR bus controller receiving a request until it is completed can be either 2 or 3 clocks.

The SFR bus implementation on initial XA derivatives is an 8-bit interface. This means that word reads and writes are not allowed. In the future, higher performance XA architecture implementations may expand the capabilities of the SFR bus by supporting 16-bit accesses.

One enhancement to the SFR bus would be to have it divide 16-bit access requests into two 8-bit accesses. This leaves the actual SFR bus width at 8 bits, but allows a user program to act as if it was 16-bits. The highest performance alternative is a full 16-bit SFR bus. This would require extra hardware in the XA to implement, but may eventually become necessary on order to achieve very high performance with some future enhanced XA core implementation.

8.2 Read-Modify-Write Lockout

Some of the SFRs that are accessed via the SFR bus contain interrupt flags and other status bits that are set directly by the peripheral device. When a read-modify-write operation is done on such an SFR, there is a possibility that a peripheral write to a flag bit in the same SFR could occur in the middle of this process. A standard mechanism is defined for the XA to deal with such cases, which is called Read-Modify-Write lockout. A read- modify-write is defined as an operation where a particular SFR is read, altered and written during the execution of a single XA instruction.

The instructions that fit this description are those that write to bits in SFRs and those that modify an entire SFR, except for the MOV instruction. This happens to be the same operations as those that read port latches rather than port pins as specified in Chapter 7, only the SFRs involved are different.

The mechanism used throughout XA peripherals to avoid losing status flags during a read-modify-write operation first involves detecting that such an operation is in progress. A signal from the CPU to the peripherals indicates such a condition. When a peripheral detects this, it prevents the CPU write to just those status flags that the peripheral has already updated since the beginning of the read-modify-write operation. This basically makes it look as if the peripheral flag update happened just after the read-modify-write operation completed, rather than during it. Once the read-modify-write operation is completed, a CPU write may affect all bits in these SFRs.

9 80C51 Compatibility

Many architectural decisions and features were guided by the goal of 80C51 compatibility when the XA core specification was written. The processor's memory configuration, memory addressing modes, instruction set, and many other things had to be taken into account.

9.1 Compatibility Considerations

Source code compatibility of the XA to the 80C51 was chosen as a goal for many reasons. Complete compatibility with an existing processor is not possible if the new processor is to have substantially higher performance.

The XA architecture makes use of a number of rules for 80C51 compatibility. An 80C51 to XA source code translator program is intended to be the means of providing compatibility between the architectures. For the translator software to be fairly simple, a one-to-one translation for all 80C51 instructions is a major consideration. The XA instruction set includes many instructions that are more powerful than 80C51 instructions and yet perform roughly the same function. 80C51 instruction can therefore be translated into those XA instructions. When this is not the case, an 80C51 instruction may be included in its original form in the XA. The XA memory map and memory addressing modes are also a superset of the 80C51, making source code translation easy to accomplish. Other CPU features are made compatible to the extent that such is possible. In rare cases, when this compatibility could not be provided for some important reason, the changes were kept to the minimum while maintaining the XA goals of high performance and low cost.

9.1.1 Compatibility Mode, Memory Map, and Addressing

Specific XA registers are reserved for use as 80C51 registers when translating code. The A register, the B register, and the data pointer all map to a pre-determined place in the XA register file (see figure 9.1). The accumulator (A) is the only one of these that required special hardware support in the XA, because the accumulator can be read or tested directly by certain instructions and in order to generate the parity flag.

The 4 banks of 8 byte registers that are found in the 80C51 are duplicated in the XA. The only difference is that in the XA, these registers do not normally overlap the lower 32 bytes of data memory space as they do in the 80C51. To allow code translation, a special 80C51 compatibility mode causes the XA register file to copy the 80C51 mapping to data memory. This mode is activated by the CM bit in the System Configuration Register (SCR).

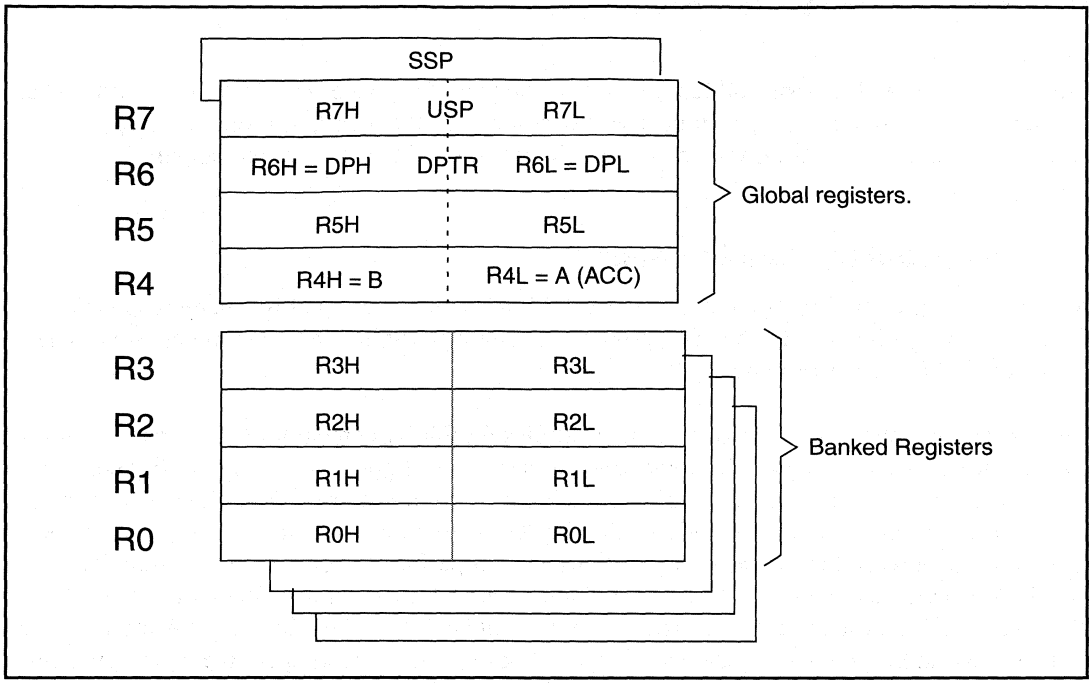


Figure 9.1. XA Register File

Other important registers of the 80C51 are provided in other ways. The program status word (PSW) of the XA is slightly different than the 80C51 PSW, so a special SFR address is reserved to provide an 80C51 compatible "view" of the PSW for use by translated code. This alternate PSW, called PSW51, is shown in the figure 9.2. The F0 flag and the F1 flag are simply readable

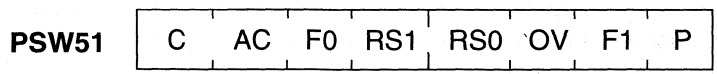


Figure 9.2. PSW CPU status flags

and writable bits. The P flag provides an even parity bit for the 80C51 A register and always reflects the current contents of that register. Note that the P flag, the F0 flag, and the F1 flag only appear in the PSW51 register.

The 80C51 indirect data memory access mode, using R0 or R1 as pointers, requires special support on the XA, where pointers are normally 16 bits in length. The 80C51 compatibility mode also causes the XA to mimic the 80C51 indirect scheme, using the first two bytes of the register file as indirect pointers, each zero extended to make a 16-bit address. Due to this and the previously mentioned register overlap to memory feature, the compatibility mode must be turned on in order to execute most translated 80C51 code on the XA. Other than the two aforementioned effects, nothing else about XA functioning is affected by the compatibility mode.

The 80C51 mapped the special function registers (SFRs) into the direct address space, from address 80 hex to FF hex. SFRs were only accessed by instruction that contain the entire SFR address, so translation to the XA is fairly simple. Since references to SFRs are normally done by their name in 80C51 source code, the translation just copies the name into the XA code output. If an SFR happened to be referred to by its address, its name must be found so that it can be inserted into the XA code. This would require that an SFR table be available for the 80C51 derivative for which the code was originally written.

The XA has another mode which may be useful for translated 80C51 code. In order to save stack space as well as speed up execution, a Page Zero (PZ) mode causes return addresses on the stack to be saved as 16 bits only, instead of the usual 24 bits (which occupy 32 bits due to word alignment on the XA stack). All other program and data addresses are also forced to be 16-bits. If an entire 80C51 application program is translated to the XA, it will very likely fit within this 64K limit, allowing the use of this mode.

Other aspects of the processor stack have been altered on the XA. For one, the standard direction of stack growth for 16 bit processors has been adopted. So, the XA stack grows downward, from higher to lower addresses in data memory. The stack can now be nearly 64K in size if necessary, and begin anywhere in its data segment so may be easily moved to a new location for translated 80C51 applications. This stack direction change is important to match the stack contents to normal data memory accesses on the XA.

80C51 code translated to run on the XA will also tend to use more stack space for two reasons. First, the PSW is automatically saved during interrupt and exception processing on the XA. The original 80C51 code should have also saved the PSW explicitly, but the XA PSW is 16 bits in length. Secondly, the initial implementation of the XA allows only word writes to the stack. Both byte and word operations may be performed, but both types of operations use 16 bits of stack space.

The tendency for stack size increase, in addition to the stack growth direction will require some changes to be made if a complete 80C51 application program is translated to run on the XA.

9.1.2 Interrupt and Exception Processing

Interrupt handling on the XA is inherently much more powerful than it was on the 80C51. Along with this added power and flexibility comes some difference that must be taken into account for 80C51 code conversion.

Previously noted was the fact that the XA automatically saves the PSW during interrupt processing. If an 80C51 program relied on this not being the case somehow, it would not work without alteration. This type of reliance is not found in code using common programming practices and should be very rare.

The XA allows up to 15 interrupt priority levels, compared to only 2 in the standard 80C51, although up to 4 levels are available in a few of the newer 80C51 variations. These priorities are stored as 4-bit values, with the priority for 2 interrupts found in the same SFR byte. This is

different (and much more powerful) than any 80C51 derivative, and will require minor changes to code that is translated.

The method of entering an interrupt routine in the XA uses a vector table stored in low addresses of the code memory. Each interrupt or exception source has a vector which consists of the address of the handler routine for that event and a new PSW value that is loaded when the vector is taken. This differs from the 80C51 approach of fixed addresses for the interrupt service routines, and again is a much more flexible and powerful method. So, if a complete 80C51 application program is converted for the XA, the interrupt service routines must be re-located above the XA vector table and the new address stored in the table, a very simple process.

9.1.3 On-Chip Peripherals

Compatibility with standard on-chip peripherals found in the 80C51 has been kept in the XA whenever possible and reasonable, but not to the extent that some enhancements are not made. The set of standard peripheral devices includes the UART, Timers 0 and 1, and Timer 2 from the 80C52.

The XA UART has been enhanced in a way that does not affect translated 80C51 code. Some additional features are added through the use of a new SFR, such as framing error detection, overrun detection, and break detection.

Timers 0 and 1 remain the same except for one difference in the function, and a difference in timing. The functional change was to remove the 8048 timer mode (mode 0) and replace it with something much more useful: a 16-bit auto-reload mode. Sixteen bit reload registers (formed by RTHn and RTLn) had to be added to Timers 0 and 1 to support the new mode 0. In mode 2, RTLn also replaces THn as the 8-bit reload register.

The relationship of all timer count rates to the microcontroller oscillator has also been changed. This adds flexibility since this is now a programmable feature, allowing oscillator divided by 4, 16, or 64 to be used as the base count rate for all of the timers. Since XA performance is much higher (on a clock-by clock basis), an application converted to the XA from the 80C51 would likely not use the same oscillator frequency anyway.

9.1.4 Bus Interface

The customary 80C51 bus control signals are all found on the standard external XA bus. To provide the best performance, the details of some of these signals have changed somewhat, and a few new ones have been added. In addition to the well known ALE, $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{EA}}$, there are now also WAIT and $\overline{\text{WRH}}$. The WAIT signal causes wait states to be inserted into any XA bus clock as long as it is asserted. The $\overline{\text{WRH}}$ signal is used to distinguish writes to the high order byte when the XA bus is configured to be 16 bits wide.

The multiplexed address/data bus has undergone some renovations on the XA as well. To get the most performance in a system executing code from the external bus, the XA separates the 4 least significant address lines on to their own pins. Since these lines normally change the most often, an ALE clock would be required on every external code fetch if these lines were multiplexed as they are on the 80C51. The 80C51 had time to do this since its performance was not that high.

The XA, however, uses only as many clocks as are needed to execute each instruction, so an ALE for every fetch would slow things down considerably. With this change, up to 16 bytes (or 8 words) of code may be accessed without the need to insert an ALE cycle on the XA bus.

The number of XA clocks used for each type of bus cycle (code read, data read, or data write) can also be programmed, so that slower peripheral devices can work with the XA without the need for an external WAIT state generator.

Due to the various changes to the bus just mentioned, an XA device cannot be completely pin compatible with an 80C51 derivative if the external bus is used. The changes to application hardware needed are relatively small and easy to make.

9.1.5 Instruction Set

The simplest goal of the XA for instruction set compatibility was to have every 80C51 instruction translate to one XA instruction. That has been achieved but for a single exception. The 80C51 instruction, XCHD or exchange digits, cannot be translated in that manner. XCHD is an instruction that is rarely used on the 80C51 and could not be implemented on the XA, due to its internal architecture, without adding a great deal of extra circuitry. So, if this instruction is encountered when 80C51 source code is being translated, a sequence of XA instructions is used to duplicate the function:

```
PUSH    R4H          ; Save temporary register.
MOV     R4H,(Ri)     ; Get second operand.
RR      R4H,#4       ; Swap one byte.
RR      R4L,#4       ; Swap second byte (the "A" register).
RL      R4,#4        ; Swap word.
                     ; Result is swapped nibbles in A and R4H.
MOV     (Ri),R4H     ; Store result.
POP     R4H          ; Restore temporary register.
```

If the application requires this sequence to not be interruptible, some additional instruction must be added in order to disable and re-enable interrupts. The table at the end of this section shows all of the other XA code replacements for 80C51 instructions.

The XA instruction set is much more powerful than the 80C51 instruction set, and as a direct consequence, the average number of bytes in an instruction is higher on the XA. In code written for the XA, the capability of a single instruction is high, so the size of an entire XA program will normally be smaller than the same program written for an 80C51. Of course, this depends on how much the application can take advantage of XA features. When code is translated from 80C51 source, however, the size change can be an issue.

In the case of a jump table, where the JMP @A+DPTR instruction is used to jump into a table of other jumps composed of the 80C51 AJMP instruction, the XA cannot always duplicate the function of the jumps in the table with instructions that are 2 bytes in length, as in the case of the AJMP instruction. An adjustment to the calculation of the table index will be required to make the translated code work properly. For a data table, accessed using MOVC @A+PC, the distance to the table may change, requiring a similar index adjustment.

Since the XA optimizes the timing of each instruction, there will be very little correspondence to the original 80C51 timing for the same code prior to translation to the XA. If the exact timing of a sequence of instructions is important to the application, the translated code must be altered, perhaps by adding NOPs or delay loops, to provide the necessary timing.

To show how a simple 80C51 to XA source code translator might work, a subroutine was extracted from a working 80C51 program and translated using the table at the end of this document and the other rules presented here. The original 80C51 source code was:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal:  MOV     Temp2,A          ; Save step target for later use.
          MOV     B,#Steplow       ; Get low byte of step increment.
          MUL     AB               ; Multiply this by the step target.
          MOV     StepResult,B     ; Save high byte as partial result.
          MOV     Temp1,A          ; Save low byte to use for rounding.

          MOV     A,Temp2          ; Get back the step target.
          MOV     B,#StepHigh      ; Get high byte of step increment,
          MUL     AB               ; and multiply the two.

          ADD     A,StepResult     ; Add the two partial results.
          JNB     Temp1.7,Exit     ; Least significant byte > 80h?
          INC     A                ; If so, round up the final result.
Exit:     ADD     A,#MotorBot      ; Add in the 0 step displacement.
          MOV     StepResult,A     ; Save final step target.
          RET
```

The same code as translated for the XA is as follows:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal:  MOV     Temp2,R4L        ; Save step target for later use.
          MOV     R4H,#Steplow     ; Get low byte of step increment.
          MULU.b  R4,R4H           ; Multiply this by the step target.
          MOV     StepResult,R4H   ; Save high byte as partial result.
          MOV     Temp1,R4L        ; Save low byte to use for rounding.

          MOV     R4L,Temp2        ; Get back the step target.
          MOV     R4H,#StepHigh    ; Get high byte of step increment,
          MULU.b  R4,R4H           ; and multiply the two.

          ADD     R4L,StepResult   ; Add the two partial results.
          JNB     Temp1.7,Exit     ; Least significant byte > 80h?
          ADDS    R4L,#1           ; If so, round up the final result.
Exit:     ADD     R4L,#MotorBot    ; Add in the 0 step displacement.
          MOV     StepResult,R4    ; Save final step target.
          RET
```

In this case, the translated code actually changed very little. Primarily, the 80C51 register names have been replaced by the new ones reserved for them in the XA. The increment (INC) instruction became a short add (ADDS), and the mnemonic for multiply (MUL) changed to MULU8.

Some basic statistical information about these code samples may be found in table 9.1. These statistics show a large performance increase for the XA code. This is significant because the code is only simple translated 80C51 code and therefore does not take any advantage of the XA's unique features.

Table 9.1: 80C51 to XA Code Translation Statistics

Statistic	80C51 code	XA translation	Comments
Code bytes	28	40	- one NOP added for branch alignment on XA
Clocks to execute	300	78	- includes XA pre-fetch queue analysis, raw execution is 66 clocks
Time to execute @ 20MHz	15 µsec	3.9 µsec	- a nearly 4x improvement without any optimization

9.2 Code Translation

Table 9.2 shows every 80C51 instruction type and the XA instruction that replaces it. An actual 80C51 to XA source code translator can make use of this table, but must also flag the compatibility exceptions noted in this section, so that any necessary adjustments may be made to the resulting XA source code.

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Arithmetic operations	
ADD A, Rn	ADD.b R, R
ADD A, #data8	ADD.b R, #data8
ADD A,dir8	ADD.b R, direct
ADD A, @Ri	ADD.b R, [R]
ADDC A, Rn	ADDC.bR, R
ADDC A, #data8	ADDC.bR, #data8
ADDC A,dir8	ADDC.bR, direct
ADDC A, @Ri	ADDC.bR, [R]
SUBB A, Rn	SUBB.bR, R
SUBB A, #data8	SUBB.bR, #data8
SUBB A, dir8	SUBB.bR, direct
SUBB A, @Ri	SUBB.bR, [R]
INC Rn	ADDS.bR, #1
INC dir8	ADDS.bdirect, #1
INC @Ri	ADDS.b[R], #1
INC A	ADDS.bR, #1
INC DPTR	ADDS.wR, #1
DEC Rn	ADDS.bR, #-1
DEC dir8	ADDS.bdirect, #-1
DEC @Ri	ADDS.b[R], #-1
DEC A	ADDS.bR, #-1
MUL AB	MULU.bR, R
DIV AB	DIVU.b R, R
DA A	DA R

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Logical operations	
ANL A, Rn	AND.b R, R
ANL A, #data8	AND.b R, #data8
ANL A, dir8	AND.b R, direct
ANL A, @Ri	AND.b R, [R]
ANL dir8, A	AND.b direct, R
ANL dir8, #data8	AND.b direct, #data8
ORL A, Rn	OR.b R, R
ORL A, #data8	OR.b R, #data8
ORL A, dir8	OR.b R, direct
ORL A, @Ri	OR.b R, [R]
ORL dir8, A	OR.b direct, R
ORL dir8, #data8	OR.b direct, #data8
XRL A, Rn	XOR.b R, R
XRL A, #data8	XOR.b R, #data8
XRL A, dir8	XOR.b R, direct
XRL A, @Ri	XOR.b R, [R]
XRL dir8, A	XOR.b direct, R
XRL dir8, #data8	XOR.b direct, #data8
CLR A	MOVS R, #0
CPL A	CPL.b R
SWAP A	RL.b R, #4
RL A	RL.b R, #1
RLC A	RLC.b R, #1
RR A	RR.b R, #1
RRC A	RRC.b R, #1
CLR C	CLR bit
CLR bit	CLR bit
SETB C	SETB bit
SETB bit	SETB bit
CPL C	XOR.b PSWL, #data8
CPL bit	XOR.b direct, #data8
ANL C, bit	AND C, bit
ANL C, /bit	AND C, /bit
ORL C, bit	OR C, bit
ORL C, /bit	OR C, /bit
MOV C, bit	MOV C, bit
MOV bit, C	MOV bit, C

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Data transfer	
MOV A, Rn	MOV.b R, R
MOV A, #data8	MOV.b R, #data8
MOV A, dir8	MOV.b R, direct
MOV A, @Ri	MOV.b R, [R]
MOV Rn, A	MOV.b R, R
MOV Rn, #data8	MOV.b R, #data8
MOV Rn, dir8	MOV.b R, direct
MOV dir8, A	MOV.b direct, R
MOV dir8, #data8	MOV.b direct, #data8
MOV dir8, Rn	MOV.b direct, R
MOV dir8, dir8	MOV.b direct, direct
MOV dir8, @Ri	MOV.b direct, [R]
MOV @Ri, A	MOV.b [R], R
MOV @Ri, dir8	MOV.b [R], direct
MOV @Ri, #data8	MOV.b [R], #data8
MOV DPTR, #data16	MOV.w R, #data16
XCH A, Rn	XCH.b R, R
XCH A, dir8	XCH.b R, direct
XCH A, @Ri	XCH.b R, R
XCHD A, @Ri	a sequence (see text)
PUSH dir8	PUSH.bdirect
POP dir8	POP.b direct
MOVX A, @Ri	MOVX.bR, [R]
MOVX A, @DPTR	MOVX.bR, [R]
MOVX @Ri, A	MOVX.b[R], R
MOVX @DPTR, A	MOVX.b[R], R
MOVC A, @A+DPTR	MOVC.bA, [A+DPTR]
MOVC A, @A+PC	MOVC.bA, [A+PC]

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Relative branches	
SJMP rel8	BR rel8
CJNE A, dir8, rel CJNE A, #data8, rel CJNE Rn, #data8, rel CJNE @Ri, #data8, rel	CJNE.b R, direct, rel CJNE.b R, #data8, rel CJNE.b R, #data8, rel CJNE.b [R], #data8, rel
DJNZ Rn, rel DJNZ dir8, rel	DJNZ.b R, rel DJNZ.b direct, rel
JZ rel JNZ rel JC rel JNC rel	JZ rel JNZ rel BCS rel BCC rel
Jumps, Calls, Returns, and Misc.	
NOP	NOP
AJMP addr11 LJMP addr16 JMP @A+DPTR	JMP rel16 JMP rel16 JUMP [A+DPTR]
ACALL addr11 LCALL addr16	CALL rel16 CALL rel16
RET RETI	RET RETI

9.3 New Instructions on the XA

While the XA instructions that are similar to 80C51 instructions have a larger addressing range, more status flags, etc., the XA also has many entirely new instructions and addressing modes that make writing new code for the XA much easier and more efficient. The new addressing modes also make the XA work very well with high level language compilers. A complete list of the new XA instructions and addressing modes is shown in Table 9.3.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
alu.w ..., ...	All of the 80C51 arithmetic and logic instructions with a 16-bit data size.
SUBB R,...	Subtract (without borrow), all addressing modes.
alu [R], R	Arithmetic and logic operations (ADD, ADDC, SUB, SUBB, CMPAND, OR, XOR, and MOV) from a register to an indirect address.
alu R, [R+]	Arithmetic and logic operations from an indirect address to a register, with the indirect pointer automatically incremented.
alu R,[R+offset8/16]	Arith/Logic operations from an indirect offset address (with 8 or 16-bit offset) to a register.
alu direct, R	The 80C51 has only MOV direct, R.
alu [R], R	The 80C51 has only MOV [R], R.
alu [R+], R	Arith/Logic operations from a register to an indirect address, with the indirect pointer automatically incremented.
alu [R+offset8/16], R	Arith/Logic operations from a register to an indirect offset address (with 8 or 16-bit offset).
alu direct, #data8/16	Arith/Logic operations to a direct address with 8 or 16-bit immediate data.
alu [R], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data.
alu [R+], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data with the indirect pointer automatically incremented.
alu [R+offset8/16], #data8/16	Arith/Logic operations to an indirect offset address (with 8 or 16-bit offset), with 8 or 16-bit immediate data.
MOV direct, [R]	Move data from an indirect to a direct address.
ADDS R, #data4	The 80C51 can only increment or decrement a register by 1. ADDS has a range of +7 to -8.
ADDS [R], #data4	Add a short value to an indirect address.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
ADDS [R+], #data4	Add a short value to an indirect offset address, with the indirect pointer automatically incremented.
ADDS [R+offset8/16], #data4	Add a short value to an indirect offset address (with 8 or 16-bit offset).
ADDS direct, #data4	Add a short value to a direct address.
MOVS ..., #data4	Move short data to destination using any of the same addressing modes as ADDS.
ASL R, R	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from register.
ASR R, R	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from register.
LSR R, R	Logical shift right a byte, word, or double word, up to 31 places, shift count read from register.
ASL R, #DATA4/5	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from instruction.
ASR R, #DATA4/5	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
LSR R, #DATA4/5	Logical shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
DIV R, R	Signed divide of 32 bits register by 16 bit register, or 16 bit register by 8 bit register.
DIVU R, R	Unsigned divide of 32 bit register by 16 bit register, or 16 bit register by 8 bit register.
MUL R, R	Signed multiply of 16 bit register by 16 bit register, or 8 bit register by 8 bit register.
MULU R, R	Unsigned multiply of 16 bit register by 16 bit register.
DIV R, #data8/16	Signed divide of 32 bits register by 16 bit immediate, or 16 bit register by 8 bit immediate.
DIVU R, #data8/16	Unsigned divide of 32 bit register by 16 bit immediate, or 16 bit register by 8 bit immediate.
MUL R, #data8/16	Signed multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
MULU R, #data8/16	Unsigned multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.
LEA R, R+offset8/16	Load effective address, duplicates the offset8 or 16-bit addressing mode calculation but saves the address in a register.
NEG R	Negate, performs a twos complement operation on a register.
SEXT R	Sign extend, copies the sign flag from the last operation into an 8 or 16-bit register.
NORM R, R	Normalize. Shifts a byte, word, or double word register left until the MSB becomes a 1. The number of shifts used is stored in a register.
RL, RR, RLC, RRC R,#data4	All of the 80C51 rotate modes with 16-bit data size and a variable number of bit positions (up to 15 places).
MOV [R+], [R+]	Block move. Move data from an indirect address to another indirect address, incrementing both pointers.
MOV R, USP and USP, R	Allows system code to move a value to or from the user stack pointer. Handy in multi-tasking applications.
MOVC R, [R+]	Move data from an indirect address in the code space to a register, with the indirect pointer automatically incremented.
PUSH and POP Rlist	PUSH and POP up to 8 word registers in one instruction.
PUSHU and POPU Rlist or direct	Allows system code to write to or read the user stack. Handy in multi-tasking applications.
conditional branches	A complete set of conditional branches, including BEQ, BNE, BG, BGE, BGT, BL, BLE, BMI, BPL, BNV, and BOV.
CALL [R]	Call indirect, to an address contained in a register.
CALL rel16	Call anywhere in a +/- 64K range.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
FCALL addr24	Far call, anywhere within the XA 16Mbyte code address space.
JMP [R]	Jump indirect, to an address contained in a register.
JMP rel16	Jump anywhere in a +/- 64K range.
FJMP addr24	Far jump, anywhere within the XA 16Mbyte code address space.
JMP [[R+]]	Jump double indirect with auto-increment. Used to branch to a sequence of addresses contained in a table.
BKPT	Breakpoint, a debugging feature.
RESET	Allows software to completely reset the XA in one instruction.
TRAP #data4	Call one of up to 16 system services. Acts like an immediate interrupt.

Section 4
XA Family Derivatives

CONTENTS

XA 16-bit microcontroller family
32K–8K/512 OTP/ROM/ROMless, watchdog, 2 UARTs

XA-G1, XA-G2, XA-G3

FAMILY DESCRIPTION

The Philips Semiconductors XA (eXtended Architecture) family of 16-bit single-chip microcontrollers is powerful enough to easily handle the requirements of high performance embedded applications, yet inexpensive enough to compete in the market for high-volume, low-cost applications.

The XA family provides an upward compatibility path for 80C51 users who need higher performance and 64k or more of program memory. Existing 80C51 code can also easily be translated to run on XA microcontrollers.

The performance of the XA architecture supports the comprehensive bit-oriented operations of the 80C51 while incorporating support for multi-tasking operating systems and high-level languages such as C. The speed of the XA architecture, at 10 to 100 times that of the 80C51, gives designers an easy path to truly high performance embedded control.

The XA architecture supports:

- Upward compatibility with the 80C51 architecture
- 16-bit fully static CPU with a 24-bit program and data address range
- Eight 16-bit CPU registers each capable of performing all arithmetic and logic operations as well as acting as memory pointers. Operations may also be performed directly to memory.
- Both 8-bit and 16-bit CPU registers, each capable of performing all arithmetic and logic operations.
- An enhanced instruction set that includes bit intensive logic operations and fast signed or unsigned 16 × 16 multiply and 32 / 16 divide
- Instruction set tailored for high level language support
- Multi-tasking and real-time executives that include up to 32 vectored interrupts, 16 software traps, segmented data memory, and banked registers to support context switching
- Low power operation, which is intrinsic to the XA architecture, includes power-down and idle modes.

More detailed information on the core is available in the XA User Guide.

SPECIFIC FEATURES OF THE XA-G1/G2/G3

- 20-bit address range, 1 megabyte each program and data space. (Note that the XA architecture supports up to 24 bit addresses.)
- 2.7V to 5.5V operation (EPROM and OTP are 5V ± 5%)
- 32K bytes on-chip EPROM/ROM program memory = XA-G37/XA-G33
- 16K bytes on-chip EPROM/ROM program memory = XA-G27/XA-G23
- 8K bytes on-chip EPROM/ROM program memory = XA-G17/XA-G13
- 512 bytes of on-chip data RAM
- Three counter/timers with enhanced features (equivalent to 80C51 T0, T1, and T2)
- Watchdog timer
- Two enhanced UARTs
- Four 8-bit I/O ports with 4 programmable output configurations
- 44-pin PLCC and 44-pin LQFP packages

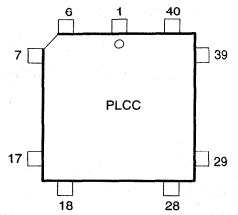
ORDERING INFORMATION

ROMless	ROM	EPROM[1]		TEMPERATURE RANGE °C AND PACKAGE	FREQ (MHz)	DRAWING NUMBER
8k byte program memory						
	P51XAG13KF A	P51XAG17KF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	30	SOT187–2
16k byte program memory						
	P51XAG23KF A	P51XAG27KF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	30	SOT187–2
32k byte program memory (except ROMless devices)						
P51XAG30JB BD	P51XAG33JB BD	P51XAG37JB BD	OTP	0 to +70, Plastic Low Profile Quad Flat Pkg.	25	SOT389–1
P51XAG30JB A	P51XAG33JB A	P51XAG37JB A	OTP	0 to +70, Plastic Leaded Chip Carrier	25	SOT187–2
		P51XAG37JB KA	UV	0 to +70, Ceramic Leaded Chip Carrier	25	1472A
P51XAG30JF BD	P51XAG33JF BD	P51XAG37JF BD	OTP	–40 to +85, Plastic Low Profile Quad Flat Pkg.	25	SOT389–1
P51XAG30JF A	P51XAG33JF A	P51XAG37JF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	25	SOT187–2
P51XAG30KB BD	P51XAG33KB BD	P51XAG37KB BD	OTP	0 to +70, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG30KB A	P51XAG33KB A	P51XAG37KB A	OTP	0 to +70, Plastic Leaded Chip Carrier	30	SOT187–2
		P51XAG37KB KA	UV	0 to +70, Ceramic Leaded Chip Carrier	30	1472A
P51XAG30KF BD	P51XAG33KF BD	P51XAG37KF BD	OTP	–40 to +85, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG30KF A	P51XAG33KF A	P51XAG37KF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	30	SOT187–2

NOTE:
1. OTP = One Time Programmable EPROM. UV = Erasable EPROM.

PIN CONFIGURATIONS

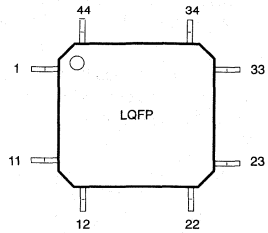
44-Pin PLCC Package



Pin	Function	Pin	Function
1	V_{SS}	23	V_{DD}
2	P1.0/A0/$\overline{WRH}$	24	P2.0/A12D8
3	P1.1/A1	25	P2.1/A13D9
4	P1.2/A2	26	P2.2/A14D10
5	P1.3/A3	27	P2.3/A15D11
6	P1.4/RxD1	28	P2.4/A16D12
7	P1.5/TxD1	29	P2.5/A17D13
8	P1.6/T2	30	P2.6/A18D14
9	P1.7/T2EX	31	P2.7/A19D15
10	$\overline{RST}$	32	$\overline{PSEN}$
11	P3.0/RxD0	33	ALE/$\overline{PROG}$
12	NC	34	NC
13	P3.1/TxD0	35	$\overline{EA}$/V_{PP}/WAIT
14	P3.2/$\overline{INT0}$	36	P0.7/A11D7
15	P3.3/$\overline{INT1}$	37	P0.6/A10D6
16	P3.4/T0	38	P0.5/A9D5
17	P3.5/T1/BUSW	39	P0.4/A8D4
18	P3.6/$\overline{WRL}$	40	P0.3/A7D3
19	P3.7/$\overline{RD}$	41	P0.2/A6D2
20	XTAL2	42	P0.1/A5D1
21	XTAL1	43	P0.0/A4D0
22	V_{SS}	44	V_{DD}

SU00525

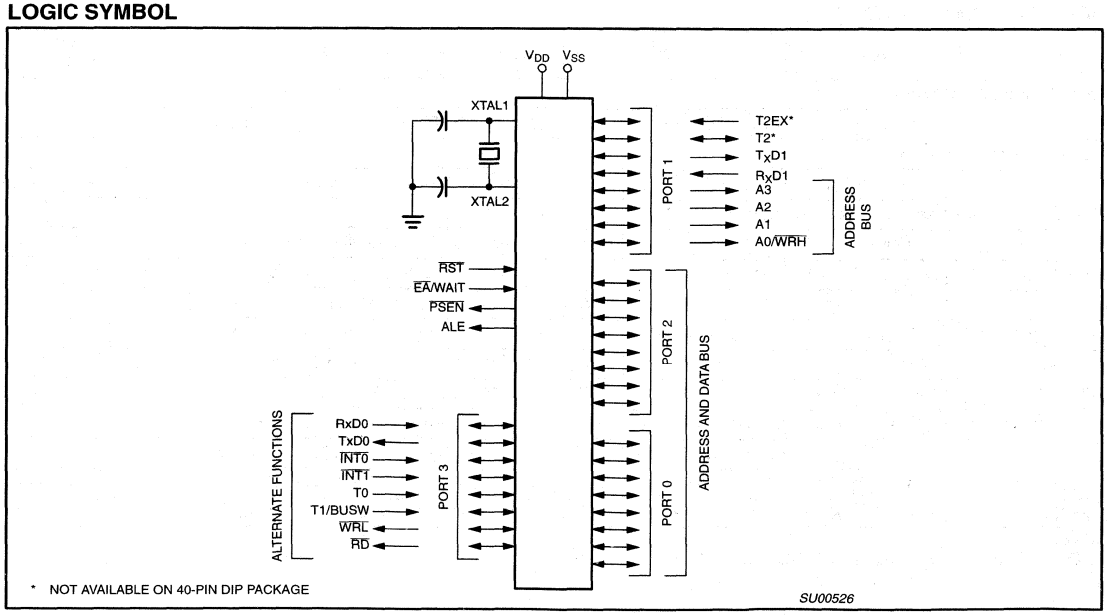
44-Pin LQFP Package



Pin	Function	Pin	Function
1	P1.5/TxD1	23	P2.5/A17D13
2	P1.6/T2	24	P2.6/A18D14
3	P1.7/T2EX	25	P2.7/A19D15
4	$\overline{RST}$	26	$\overline{PSEN}$
5	P3.0/RxD0	27	ALE/$\overline{PROG}$
6	NC	28	NC
7	P3.1/TxD0	29	$\overline{EA}$/V_{PP}/WAIT
8	P3.2/$\overline{INT0}$	30	P0.7/A11D7
9	P3.3/$\overline{INT1}$	31	P0.6/A10D6
10	P3.4/T0	32	P0.5/A9D5
11	P3.5/T1/BUSW	33	P0.4/A8D4
12	P3.6/$\overline{WRL}$	34	P0.3/A7D3
13	P3.7/$\overline{RD}$	35	P0.2/A6D2
14	XTAL2	36	P0.1/A5D1
15	XTAL1	37	P0.0/A4D0
16	V_{SS}	38	V_{DD}
17	V_{DD}	39	V_{SS}
18	P2.0/A12D8	40	P1.0/A0/$\overline{WRH}$
19	P2.1/A13D9	41	P1.1/A1
20	P2.2/A14D10	42	P1.2/A2
21	P2.3/A15D11	43	P1.3/A3
22	P2.4/A16/D12	44	P1.4/RxD1

SU00580

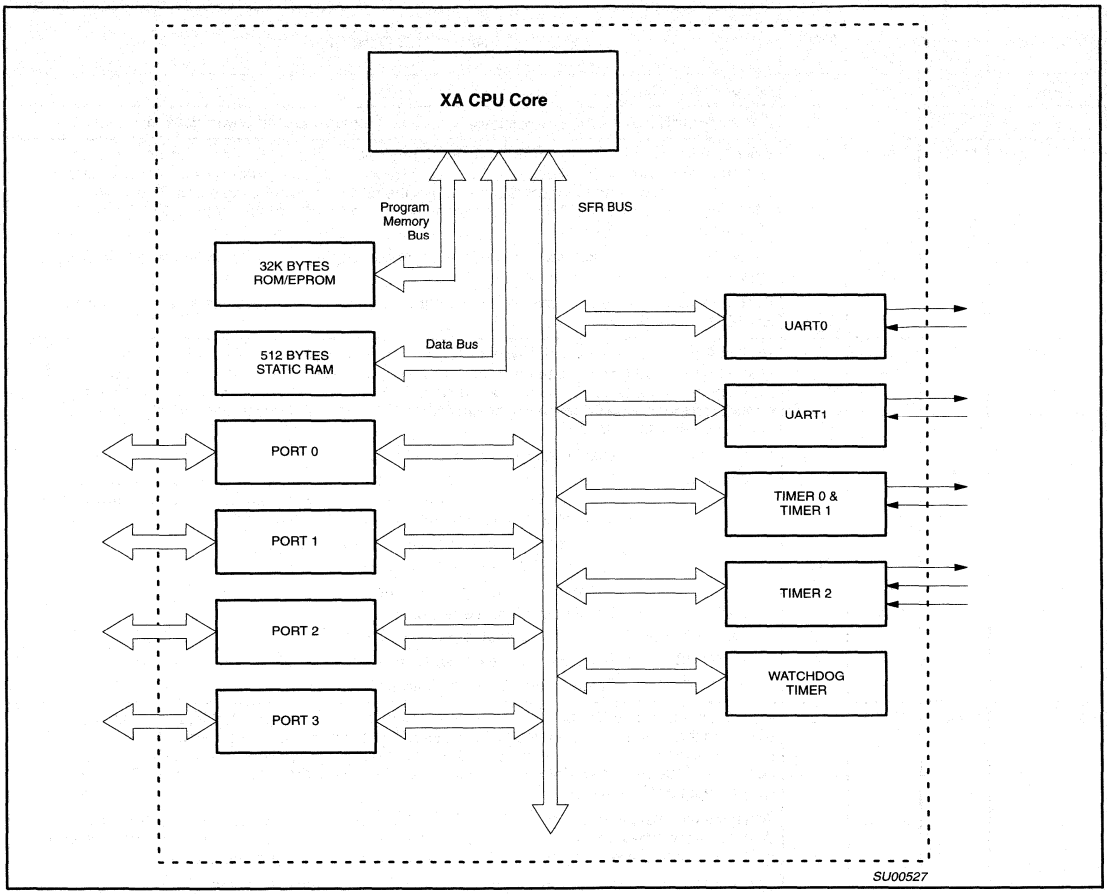
LOGIC SYMBOL



* NOT AVAILABLE ON 40-PIN DIP PACKAGE

SU00526

BLOCK DIAGRAM



PIN DESCRIPTIONS

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
V_{SS}	1, 22	16	I	**Ground:** 0V reference.
V_{DD}	23, 44	17	I	**Power Supply:** This is the power supply voltage for normal, idle, and power down operation.
P0.0 – P0.7	43–36	37–30	I/O	**Port 0:** Port 0 is an 8-bit I/O port with a user-configurable output type. Port 0 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 0 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used, Port 0 becomes the multiplexed low data/instruction byte and address lines 4 through 11. Port 0 also outputs the code bytes during program verification and receives code bytes during EPROM programming.
P1.0 – P1.7	2–9	40–44, 1–3	I/O	**Port 1:** Port 1 is an 8-bit I/O port with a user-configurable output type. Port 1 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 1 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 1 also provides special functions as described below.
	2	40	O	**A0/$\overline{WRH}$:** Address bit 0 of the external address bus when the external data bus is configured for an 8 bit width. When the external data bus is configured for a 16 bit width, this pin becomes the high byte write strobe.
	3	41	O	**A1:** Address bit 1 of the external address bus.
	4	42	O	**A2:** Address bit 2 of the external address bus.
	5	43	O	**A3:** Address bit 3 of the external address bus.
	6	44	I	**RxD1 (P1.4):** Receiver input for serial port 1.
	7	1	O	**TxD1 (P1.5):** Transmitter output for serial port 1.
	8	2	I	**T2 (P1.6):** Timer/counter 2 external count input/clockout.
	9	3	I	**T2EX (P1.7):** Timer/counter 2 reload/capture/direction control
P2.0 – P2.7	24–31	18–25	I/O	**Port 2:** Port 2 is an 8-bit I/O port with a user-configurable output type. Port 2 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. the operation of port 2 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used in 16-bit mode, Port 2 becomes the multiplexed high data/instruction byte and address lines 12 through 19. When the external program/data bus is used in 8-bit mode, the number of address lines that appear on port 2 is user programmable. Port 2 also receives the low-order address byte during program memory verification.
P3.0 – P3.7	11, 13–19	5, 7–13	I/O	**Port 3:** Port 3 is an 8-bit I/O port with a user configurable output type. Port 3 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. the operation of port 3 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 3 pins receive the high order address bits during EPROM programming and verification. Port 3 also provides various special functions as described below.
	11	5	I	**RxD0 (P3.0):** Receiver input for serial port 0.
	13	7	O	**TxD0 (P3.1):** Transmitter output for serial port 0.
	14	8	I	**$\overline{INT0}$ (P3.2):** External interrupt 0 input.
	15	9	I	**$\overline{INT1}$ (P3.3):** External interrupt 1 input.
	16	10	I/O	**T0 (P3.4):** Timer 0 external input, or timer 0 overflow output.
	17	11	I/O	**T1/BUSW (P3.5):** Timer 1 external input, or timer 1 overflow output. The value on this pin is latched as the external reset input is released and defines the default external data bus width (BUSW). 0 = 8-bit bus and 1 = 16-bit bus.
	18	12	O	**$\overline{WRL}$ (P3.6):** External data memory low byte write strobe.
	19	13	O	**$\overline{RD}$ (P3.7):** External data memory read strobe.

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
$\overline{RST}$	10	4	I	**Reset:** A low on this pin resets the microcontroller, causing I/O ports and peripherals to take on their default states, and the processor to begin execution at the address contained in the reset vector. Refer to the section on Reset for details.
ALE/$\overline{PROG}$	33	27	I/O	**Address Latch Enable/Program Pulse:** A high output on the ALE pin signals external circuitry to latch the address portion of the multiplexed address/data bus. A pulse on ALE occurs only when it is needed in order to process a bus cycle. During EPROM programming, this pin is used as the program pulse input.
$\overline{PSEN}$	32	26	O	**Program Store Enable:** The read strobe for external program memory. When the microcontroller accesses external program memory, $\overline{PSEN}$ is driven low in order to enable memory devices. $\overline{PSEN}$ is only active when external code accesses are performed.
$\overline{EA}$/WAIT/ V_{PP}	35	29	I	**External Access/Wait/Programming Supply Voltage:** The $\overline{EA}$ input determines whether the internal program memory of the microcontroller is used for code execution. The value on the $\overline{EA}$ pin is latched as the external reset input is released and applies during later execution. When latched as a 0, external program memory is used exclusively, when latched as a 1, internal program memory will be used up to its limit, and external program memory used above that point. After reset is released, this pin takes on the function of bus Wait input. If Wait is asserted high during any external bus access, that cycle will be extended until Wait is released. During EPROM programming, this pin is also the programming supply voltage input.
XTAL1	21	15	I	**Crystal 1:** Input to the inverting amplifier used in the oscillator circuit and input to the internal clock generator circuits.
XTAL2	20	14	O	**Crystal 2:** Output from the oscillator amplifier.

SPECIAL FUNCTION REGISTERS

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES MSB							LSB	RESET VALUE
BCR	Bus configuration register	46A	—	—	—	WAITD	BUSD	BC2	BC1	BC0	Note 1
BTRH	Bus timing register high byte	469	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0	FF
BTRL	Bus timing register low byte	468	WM1	WM0	ALEW	—	CR1	CR0	CRA1	CRA0	EF
CS	Code segment	443									00
DS	Data segment	441									00
ES	Extra segment	442									00
			33F	33E	33D	33C	33B	33A	339	338	
IEH*	Interrupt enable high byte	427	—	—	—	—	ETI1	ERI1	ETI0	ERI0	00
			337	336	335	334	333	332	331	330	
IEL*	Interrupt enable low byte	426	EA	—	—	ET2	ET1	EX1	ET0	EX0	00
IPA0	Interrupt priority 0	4A0	—	PT0			—	PX0			00
IPA1	Interrupt priority 1	4A1	—	PT1			—	PX1			00
IPA2	Interrupt priority 2	4A2	—	—			—	PT2			00
IPA4	Interrupt priority 4	4A4	—	PTI0			—	PRI0			00
IPA5	Interrupt priority 5	4A5	—	PTI1			—	PRI1			00
			387	386	385	384	383	382	381	380	
P0*	Port 0	430	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FF
			38F	38E	38D	38C	38B	38A	389	388	
P1*	Port 1	431	T2EX	T2	TxD1	RxD1	A3	A2	A1	WRH	FF
			397	396	395	394	393	392	391	390	
P2*	Port 2	432	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	FF

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES MSB							LSB	RESET VALUE
			39F	39E	39D	39C	39B	39A	399	398	
P3*	Port 3	433	RD	WR	T1	T0	INT1	INT0	TxD0	RxD0	FF
P0CFGA	Port 0 configuration A	470									Note 5
P1CFGA	Port 1 configuration A	471									Note 5
P2CFGA	Port 2 configuration A	472									Note 5
P3CFGA	Port 3 configuration A	473									Note 5
P0CFGB	Port 0 configuration B	4F0									Note 5
P1CFGB	Port 1 configuration B	4F1									Note 5
P2CFGB	Port 2 configuration B	4F2									Note 5
P3CFGB	Port 3 configuration B	4F3									Note 5
			227	226	225	224	223	222	221	220	
PCON*	Power control register	404	—	—	—	—	—	—	PD	IDL	00
			20F	20E	20D	20C	20B	20A	209	208	
PSWH*	Program status word (high byte)	401	SM	TM	RS1	RS0	IM3	IM2	IM1	IM0	Note 2
			207	206	205	204	203	202	201	200	
PSWL*	Program status word (low byte)	400	C	AC	—	—	—	V	N	Z	Note 2
			217	216	215	214	213	212	211	210	
PSW51*	80C51 compatible PSW	402	C	AC	F0	RS1	RS0	V	F1	P	Note 3
RTH0	Timer 0 extended reload, high byte	455									00
RTH1	Timer 1 extended reload, high byte	457									00
RTL0	Timer 0 extended reload, low byte	454									00
RTL1	Timer 1 extended reload, low byte	456									00
			307	306	305	304	303	302	301	300	
S0CON*	Serial port 0 control register	420	SM0_0	SM1_0	SM2_0	REN_0	TB8_0	RB8_0	TI_0	RI_0	00
			30F	30E	30D	30C	30B	30A	309	308	
S0STAT*	Serial port 0 extended status	421	—	—	—	—	FE0	BR0	OE0	STINT0	00
S0BUF	Serial port 0 buffer register	460									x
S0ADDR	Serial port 0 address register	461									00
S0ADEN	Serial port 0 address enable register	462									00
			327	326	325	324	323	322	321	320	
S1CON*	Serial port 1 control register	424	SM0_1	SM1_1	SM2_1	REN_1	TB8_1	RB8_1	TI_1	RI_1	00
			32F	32E	32D	32C	32B	32A	329	328	
S1STAT*	Serial port 1 extended status	425	—	—	—	—	FE1	BR1	OE1	STINT1	00
S1BUF	Serial port 1 buffer register	464									x
S1ADDR	Serial port 1 address register	465									00
S1ADEN	Serial port 1 address enable register	466									00
SCR	System configuration register	440	—	—	—	—	PT1	PT0	CM	PZ	00
			21F	21E	21D	21C	21B	21A	219	218	
SSEL*	Segment selection register	403	ESWEN	R6SEG	R5SEG	R4SEG	R3SEG	R2SEG	R1SEG	R0SEG	00
SWE	Software Interrupt Enable	47A	—	SWE7	SWE6	SWE5	SWE4	SWE3	SWE2	SWE1	00

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES MSB							LSB	RESET VALUE
			357	356	355	354	353	352	351	350	
SWR*	Software Interrupt Request	42A	—	SWR7	SWR6	SWR5	SWR4	SWR3	SWR2	SWR1	00
			2C7	2C6	2C5	2C4	2C3	2C2	2C1	2C0	
T2CON*	Timer 2 control register	418	TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2	00
			2CF	2CE	2CD	2CC	2CB	2CA	2C9	2C8	
T2MOD*	Timer 2 mode control	419	—	—	RCLK1	TCLK1	—	—	T2OE	DCEN	00
TH2	Timer 2 high byte	459									00
TL2	Timer 2 low byte	458									00
T2CAPH	Timer 2 capture register, high byte	45B									00
T2CAPL	Timer 2 capture register, low byte	45A									00
			287	286	285	284	283	282	281	280	
TCON*	Timer 0 and 1 control register	410	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00
TH0	Timer 0 high byte	451									00
TH1	Timer 1 high byte	453									00
TL0	Timer 0 low byte	450									00
TL1	Timer 1 low byte	452									00
TMOD	Timer 0 and 1 mode control	45C	GATE	C/T	M1	M0	GATE	C/T	M1	M0	00
			28F	28E	28D	28C	28B	28A	289	288	
TSTAT*	Timer 0 and 1 extended status	411	—	—	—	—	—	T1OE	—	T0OE	00
			2FF	2FE	2FD	2FC	2FB	2FA	2F9	2F8	
WDCON*	Watchdog control register	41F	PRE2	PRE1	PRE0	—	—	WDRUN	WDTOF	—	Note 6
WDL	Watchdog timer reload	45F									00
WFEED1	Watchdog feed 1	45D									x
WFEED2	Watchdog feed 2	45E									x

NOTES:

* SFRs are bit addressable.

1. At reset, the BCR register is loaded with the binary value 0000 0a11, where "a" is the value on the BUSW pin. This defaults the address bus size to 20 bits since the XA-G1/G2/G3 have only 20 address lines.
2. SFR is loaded from the reset vector.
3. All bits except F1, F0, and P are loaded from the reset vector. Those bits are all 0.
4. Unimplemented bits in SFRs are X (unknown) at all times. Ones should not be written to these bits since they may be used for other purposes in future XA derivatives. The reset value shown for these bits is 0.
5. Port configurations default to quasi-bidirectional when the XA begins execution from internal code memory after reset, based on the condition found on the EA pin. Thus all PnCFGA registers will contain FF and PnCFGB registers will contain 00. When the XA begins execution using external code memory, the default configuration for pins that are associated with the external bus will be push-pull. The PnCFGA and PnCFGB register contents will reflect this difference.
6. The WDCON reset value is E6 for a Watchdog reset, E4 for all other reset causes.
7. The XA-G1/G2/G3 implements an 8-bit SFR bus, as stated in Chapter 8 of the *XA User Guide*. All SFR accesses must be 8-bit operations. Attempts to write 16 bits to an SFR will actually write only the lower 8 bits. Sixteen bit SFR reads will return undefined data in the upper byte.

XA-G1/G2/G3 TIMER/COUNTERS

The XA has two standard 16-bit enhanced Timer/Counters: Timer 0 and Timer 1. Additionally, it has a third 16-bit Up/Down timer/counter, T2. A central timing generator in the XA core provides the time-base for all XA Timers and Counters. The timer/event counters can perform the following functions:

- Measure time intervals and pulse duration
- Count external events
- Generate interrupt requests
- Generate PWM or timed output waveforms

All of the timer/counters (Timer 0, Timer 1 and Timer 2) can be independently programmed to operate either as timers or event counters via the C/T bit in the TnCON register. All timers count up unless otherwise stated. These timers may be dynamically read during program execution.

The base clock rate of all of the timers is user programmable. This applies to timers T0, T1, and T2 when running in timer mode (as opposed to counter mode), and the watchdog timer. The clock driving the timers is called TCLK and is determined by the setting of two bits (PT1, PT0) in the System Configuration Register (SCR). The frequency of TCLK may be selected to be the oscillator input divided by 4 (Osc/4), the oscillator input divided by 16 (Osc/16), or the oscillator input divided by 64 (Osc/64). This gives a range of possibilities for the XA timer functions, including baud rate generation, Timer 2 capture. Note that this single rate setting applies to all of the timers.

When timers T0, T1, or T2 are used in the counter mode, the register will increment whenever a falling edge (high to low transition) is detected on the external input pin corresponding to the timer clock. These inputs are sampled once every 2 oscillator cycles, so it can take as many as 4 oscillator cycles to detect a transition. Thus the maximum count rate that can be supported is Osc/4. The duty cycle of the timer clock inputs is not important, but any high or low state on the timer clock input pins must be present for 2 oscillator cycles before it is guaranteed to be "seen" by the timer logic.

Timer 0 and Timer 1

The "Timer" or "Counter" function is selected by control bits C/T in the special function register TMOD. These two Timer/Counters have four operating modes, which are selected by bit-pairs (M1, M0) in the TMOD register. Timer modes 1, 2, and 3 in XA are kept identical to the 80C51 timer modes for code compatibility. Only the mode 0 is replaced in the XA by a more powerful 16-bit auto-reload mode. This will give the XA timers a much larger range when used as time bases.

The recommended M1, M0 settings for the different modes are shown in Figure 2.

SCR Address:440
Not Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	PT1	PT0	CM	PZ

PT1	PT0	OPERATING
		Prescaler selection.
0	0	Osc/4
0	1	Osc/16
1	0	Osc/64
1	1	Reserved
CM		Compatibility Mode allows the XA to execute most translated 80C51 code on the XA. The XA register file must copy the 80C51 mapping to data memory and mimic the 80C51 indirect addressing scheme.
PZ		Page Zero mode forces all program and data addresses to 16-bits only. This saves stack space and speeds up execution but limits memory access to 64k.

SU00589

Figure 1. System Configuration Register (SCR)

TMOD Address:45C
Not Bit Addressable
Reset Value: 00H

MSB							LSB
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
				TIMER 1			TIMER 0

GATE	Gating control when set. Timer/Counter "n" is enabled only while "$\overline{INTn}$" pin is high and "TRn" control bit is set. When cleared Timer "n" is enabled whenever "TRn" control bit is set.
C/$\overline{T}$	Timer or Counter Selector cleared for Timer operation (input from internal system clock.) Set for Counter operation (input from "Tn" input pin).

M1	M0	OPERATING
0	0	16-bit auto-reload timer/counter
0	1	16-bit non-auto-reload timer/counter
1	0	8-bit auto-reload timer/counter
1	1	Dual 8-bit timer mode (timer 0 only)

SU00605

Figure 2. Timer/Counter Mode Control (TMOD) Register

New Enhanced Mode 0

For timers T0 or T1 the 13-bit count mode on the 80C51 (current Mode 0) has been replaced in the XA with a 16-bit auto-reload mode. Four additional 8-bit data registers (two per timer: RTHn and RTLn) are created to hold the auto-reload values. In this mode, the TH overflow will set the TF flag in the TCON register and cause both the TL and TH counters to be loaded from the RTL and RTH registers respectively.

These new SFRs will also be used to hold the TL reload data in the 8-bit auto-reload mode (Mode 2) instead of TH.

The overflow rate for Timer 0 or Timer 1 in Mode 0 may be calculated as follows:

Timer_Rate = Osc / (N * (65536 – Timer_Reload_Value))

where N = the TCLK prescaler value: 4 (default), 16, or 64.

Mode 1

Mode 1 is the 16-bit non-auto reload mode.

Mode 2

Mode 2 configures the Timer register as an 8-bit Counter (TLn) with automatic reload. Overflow from TLn not only sets TFn, but also reloads TLn with the contents of RTLn, which is preset by software. The reload leaves THn unchanged.

Mode 2 operation is the same for Timer/Counter 0.

The overflow rate for Timer 0 or Timer 1 in Mode 2 may be calculated as follows:

Timer_Rate = Osc / (N * (256 – Timer_Reload_Value))

where N = the TCLK prescaler value: 4, 16, or 64.

Mode 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0.

Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. TL0 uses the Timer 0 control bits: C/T, GATE, TR0, INT0, and TF0. TH0 is locked into a timer function and takes over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the "Timer 1" interrupt.

Mode 3 is provided for applications requiring an extra 8-bit timer. When Timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.

TCON Address:410
Bit Addressable
Reset Value: 00H

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

BIT	SYMBOL	FUNCTION
TCON.7	TF1	Timer 1 overflow flag. Set by hardware on Timer/Counter overflow. This flag will not be set if T1OE (TSTAT.2) is set. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.6	TR1	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter 1 on/off.
TCON.5	TF0	Timer 0 overflow flag. Set by hardware on Timer/Counter overflow. This flag will not be set if T0OE (TSTAT.0) is set. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.4	TR0	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter 0 on/off.
TCON.3	IE1	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.2	IT1	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
TCON.1	IE0	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.0	IT0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

SU00604C

Figure 3. Timer/Counter Control (TCON) Register

T2CON Address:418
Bit Addressable
Reset Value: 00H

MSB							LSB
TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2

BIT	SYMBOL	FUNCTION
T2CON.7	TF2	Timer 2 overflow flag. Set by hardware on Timer/Counter overflow. Must be cleared by software. TF2 will not be set when RCLK0, RCLK1, TCLK0, TCLK1 or T2OE=1.
T2CON.6	EXF2	Timer 2 external flag is set when a capture or reload occurs due to a negative transition on T2EX (and EXEN2 is set). This flag will cause a Timer 2 interrupt when this interrupt is enabled. EXF2 is cleared by software.
T2CON.5	RCLK0	Receive Clock Flag.
T2CON.4	TCLK0	Transmit Clock Flag. RCLK0 and TCLK0 are used to select Timer 2 overflow rate as a clock source for UART0 instead of Timer T1.
T2CON.3	EXEN2	Timer 2 external enable bit allows a capture or reload to occur due to a negative transition on T2EX.
T2CON.2	TR2	Start=1/Stop=0 control for Timer 2.
T2CON.1	C/T2	Timer or counter select. 0=Internal timer 1=External event counter (falling edge triggered)
T2CON.0	CP/RL2	Capture/Reload flag. If CP/RL2 & EXEN2=1 captures will occur on negative transitions of T2EX. If CP/RL2=0, EXEN2=1 auto reloads occur with either Timer 2 overflows or negative transitions at T2EX. If RCLK or TCLK=1 the timer is set to auto reload on Timer 2 overflow, this bit has no effect.

SU00606A

Figure 4. Timer/Counter 2 Control (T2CON) Register

New Timer-Overflow Toggle Output

In the XA, the timer module now has two outputs, which toggle on overflow from the individual timers. The same device pins that are used for the T0 and T1 count inputs are also used for the new overflow outputs. An SFR bit (TnOE in the TSTAT register) is associated with each counter and indicates whether Port-SFR data or the overflow signal is output to the pin. These outputs could be used in applications for generating variable duty cycle PWM outputs (changing the auto-reload register values). Also variable frequency (Osc/8 to Osc/8,388,608) outputs could be achieved by adjusting the prescaler along with the auto-reload register values. With a 30.0MHz oscillator, this range would be 3.58Hz to 3.75MHz.

Timer T2

Timer 2 in the XA is a 16-bit Timer/Counter which can operate as either a timer or as an event counter. This is selected by C/T2 in the special function register T2CON. Upon timer T2 overflow/underflow, the TF2 flag is set, which may be used to generate an interrupt. It can be operated in one of three operating modes: auto-reload (up or down counting), capture, or as the baud rate generator (for either or both UARTs via SFRs T2MOD and T2CON). These modes are shown in Table 1.

Capture Mode

In the capture mode there are two options which are selected by bit EXEN2 in T2CON. If EXEN2 = 0, then timer 2 is a 16-bit timer or counter, which upon overflowing sets bit TF2, the timer 2 overflow bit. This will cause an interrupt when the timer 2 interrupt is enabled.

If EXEN2 = 1, then Timer 2 still does the above, but with the added feature that a 1-to-0 transition at external input T2EX causes the current value in the Timer 2 registers, TL2 and TH2, to be captured into registers RCAP2L and RCAP2H, respectively. In addition, the transition at T2EX causes bit EXF2 in T2CON to be set. This will cause an interrupt in the same fashion as TF2 when the Timer 2 interrupt is enabled. The capture mode is illustrated in Figure 7.

Auto-Reload Mode (Up or Down Counter)

In the auto-reload mode, the timer registers are loaded with the 16-bit value in T2CAPH and T2CAPL when the count overflows. T2CAPH and T2CAPL are initialized by software. If the EXEN2 bit in T2CON is set, the timer registers will also be reloaded and the EXF2 flag set when a 1-to-0 transition occurs at input T2EX. The auto-reload mode is shown in Figure 8.

In this mode, Timer 2 can be configured to count up or down. This is done by setting or clearing the bit DCEN (Down Counter Enable) in the T2MOD special function register (see Table 1). The T2EX pin then controls the count direction. When T2EX is high, the count is in the up direction, when T2EX is low, the count is in the down direction.

Figure 8 shows Timer 2, which will count up automatically, since DCEN = 0. In this mode there are two options selected by bit EXEN2 in the T2CON register. If EXEN2 = 0, then Timer 2 counts up to FFFFH and sets the TF2 (Overflow Flag) bit upon overflow. This causes the Timer 2 registers to be reloaded with the 16-bit value in T2CAPL and T2CAPH, whose values are preset by software. If EXEN2 = 1, a 16-bit reload can be triggered either by an overflow or by a 1-to-0 transition at input T2EX. This transition also sets the EXF2 bit. If enabled, either TF2 or EXF2 bit can generate the Timer 2 interrupt.

In Figure 9, the DCEN = 1; this enables the Timer 2 to count up or down. In this mode, the logic level of T2EX pin controls the direction of count. When a logic '1' is applied at pin T2EX, the Timer 2 will count up. The Timer 2 will overflow at FFFFH and set the TF2 flag, which can then generate an interrupt if enabled. This timer overflow, also causes the 16-bit value in T2CAPL and T2CAPH to be reloaded into the timer registers TL2 and TH2, respectively.

A logic '0' at pin T2EX causes Timer 2 to count down. When counting down, the timer value is compared to the 16-bit value contained in T2CAPH and T2CAPL. When the value is equal, the

timer register is loaded with FFFF hex. The underflow also sets the TF2 flag, which can generate an interrupt if enabled.

The external flag EXF2 toggles when Timer 2 underflows or overflows. This EXF2 bit can be used as a 17th bit of resolution, if needed. the EXF2 flag does not generate an interrupt in this mode. As the baud rate generator, timer T2 is incremented by TCLK.

Baud Rate Generator Mode
By setting the TCLKn and/or RCLKn in T2CON or T2MOD, the Timer 2 can be chosen as the baud rate generator for either or both UARTs. The baud rates for transmit and receive can be simultaneously different.

Programmable Clock-Out

A 50% duty cycle clock can be programmed to come out on P1.6. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed (1) to input the external clock for Timer/Counter 2 or (2) to output a 50% duty cycle clock ranging from 3.58Hz to 3.75MHz at a 30MHz operating frequency.

To configure the Timer/Counter 2 as a clock generator, bit C/T2 (in T2CON) must be cleared and bit T20E in T2MOD must be set. Bit TR2 (T2CON.2) also must be set to start the timer.

The Clock-Out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (TCAP2H, TCAP2L) as shown in this equation:

$$\frac{TCLK}{2 \times (65536 - TCAP2H, TCAP2L)}$$

In the Clock-Out mode Timer 2 roll-overs will not generate an interrupt. This is similar to when it is used as a baud-rate generator. It is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud-rate will be 1/8 of the Clock-Out frequency.

Table 1. Timer 2 Operating Modes

TR2	CP/RL2	RCLK+TCLK	DCEN	MODE
0	X	X	X	Timer off (stopped)
1	0	0	0	16-bit auto-reload, counting up
1	0	0	1	16-bit auto-reload, counting up or down depending on T2EX pin
1	1	0	X	16-bit capture
1	X	1	X	Baud rate generator

TSTAT Address:411
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	—	T1OE	—	T0OE

BIT	SYMBOL	FUNCTION
TSTAT.2	T1OE	When 0, this bit allows the T1 pin to clock Timer 1 when in the counter mode. When 1, T1 acts as an output and toggles at every Timer 1 overflow.
TSTAT.0	T0OE	When 0, this bit allows the T0 pin to clock Timer 0 when in the counter mode. When 1, T0 acts as an output and toggles at every Timer 0 overflow.

SU00612B

Figure 5. Timer 0 And 1 Extended Status (TSTAT)

T2MOD Address:419
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	RCLK1	TCLK1	—	—	T2OE	DCEN

BIT	SYMBOL	FUNCTION
T2MOD.5	RCLK1	Receive Clock Flag.
T2MOD.4	TCLK1	Transmit Clock Flag. RCLK1 and TCLK1 are used to select Timer 2 overflow rate as a clock source for UART1 instead of Timer T1.
T2MOD.1	T2OE	When 0, this bit allows the T2 pin to clock Timer 2 when in the counter mode. When 1, T2 acts as an output and toggles at every Timer 2 overflow.
T2MOD.0	DCEN	Controls count direction for Timer 2 in autoreload mode. DCEN=0 counter set to count up only DCEN=1 counter set to count up or down, depending on T2EX (see text).

SU00610B

Figure 6. Timer 2 Mode Control (T2MOD)

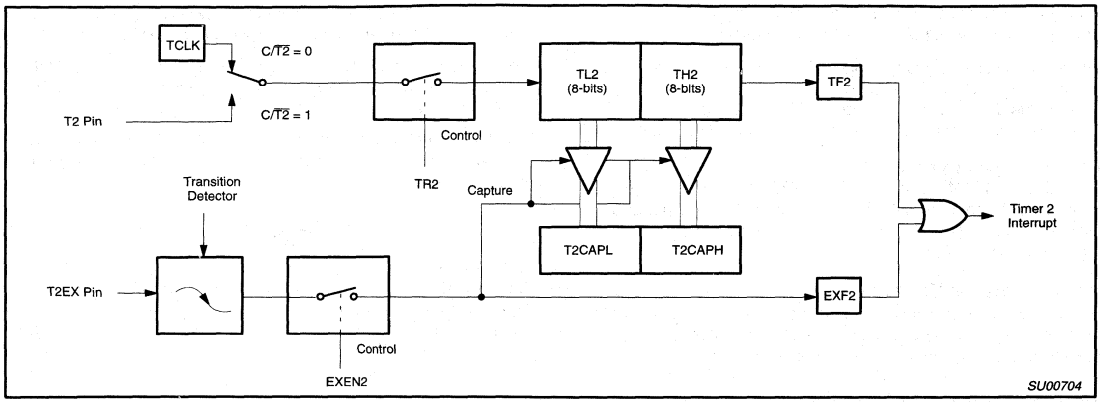


Figure 7. Timer 2 in Capture Mode

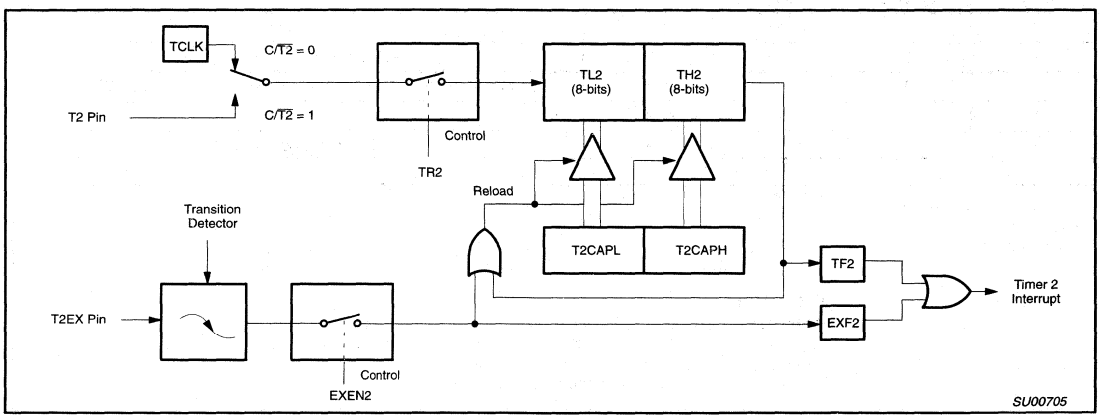


Figure 8. Timer 2 in Auto-Reload Mode (DCEN = 0)

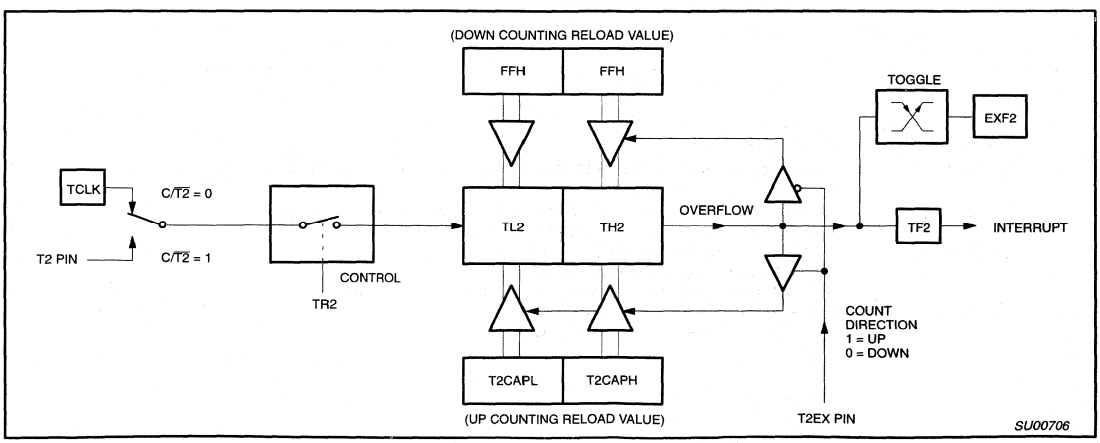


Figure 9. Timer 2 Auto Reload Mode (DCEN = 1)

WATCHDOG TIMER

The watchdog timer subsystem protects the system from incorrect code execution by causing a system reset when the watchdog timer underflows as a result of a failure of software to feed the timer prior to the timer reaching its terminal count. It is important to note that the watchdog timer is running after any type of reset and must be turned off by user software if the application does not use the watchdog function.

Watchdog Function

The watchdog consists of a programmable prescaler and the main timer. The prescaler derives its clock from the TCLK source that also drives timers 0, 1, and 2. The watchdog timer subsystem consists of a programmable 13-bit prescaler, and an 8-bit main timer. The main timer is clocked (decremented) by a tap taken from one of the top 8-bits of the prescaler as shown in Figure 10. The clock source for the prescaler is the same as TCLK (same as the clock source for the timers). Thus the main counter can be clocked as often as once every 64 TCLKs (see Table 2). The watchdog generates an underflow signal (and is autoloaded from WDL) when the watchdog is at count 0 and the clock to decrement the watchdog occurs. The watchdog is 8 bits wide and the autoload value can range from 0 to FFH. (The autoload value of 0 is permissible since the prescaler is cleared upon autoload).

This leads to the following user design equations. Definitions: t_{OSC} is the oscillator period, N is the selected prescaler tap value, W is the main counter autoload value, P is the prescaler value from Table 2, t_{MIN} is the minimum watchdog time-out value (when the autoload value is 0), t_{MAX} is the maximum time-out value (when the autoload value is FFH), t_D is the design time-out value.

$t_{MIN} = t_{OSC} \times 4 \times 32$ (W = 0, N = 4)

$t_{MAX} = t_{OSC} \times 64 \times 4096 \times 256$ (W = 255, N = 64)

$t_D = t_{OSC} \times N \times P \times (W + 1)$

The watchdog timer is not directly loadable by the user. Instead, the value to be loaded into the main timer is held in an autoload register. In order to cause the main timer to be loaded with the appropriate value, a special sequence of software action must take place. This operation is referred to as feeding the watchdog timer.

To feed the watchdog, two instructions must be sequentially executed successfully. No intervening SFR accesses are allowed, so interrupts should be disabled before feeding the watchdog. The instructions should move A5H to the WFEED1 register and then 5AH to the WFEED2 register. If WFEED1 is correctly loaded and WFEED2 is not correctly loaded, then an immediate watchdog reset will occur. The program sequence to feed the watchdog timer or cause new WDCON settings to take effect is as follows:

```
clr     ea              ; disable global interrupts.
mov.b   wfeed1,#A5h     ; do watchdog feed part 1
mov.b   wfeed2,#5Ah     ; do watchdog feed part 2
setb    ea              ; re-enable global interrupts.
```

This sequence assumes that the XA interrupt system is enabled and there is a possibility of an interrupt request occurring during the feed sequence. If an interrupt was allowed to be serviced and the service routine contained any SFR access, it would trigger a watchdog reset. If it is known that no interrupt could occur during the feed sequence, the instructions to disable and re-enable interrupts may be removed.

The software must be written so that a feed operation takes place every t_D seconds from the last feed operation. Some tradeoffs may need to be made. It is not advisable to include feed operations in minor loops or in subroutines unless the feed operation is a specific subroutine.

To turn the watchdog timer completely off, the following code sequence should be used:

```
mov.b   wdcon,#0        ; set WD control register to clear WDRUN.
mov.b   wfeed1,#A5h     ; do watchdog feed part 1
mov.b   wfeed2,#5Ah     ; do watchdog feed part 2
```

This sequence assumes that the watchdog timer is being turned off at the beginning of initialization code and that the XA interrupt system has not yet been enabled. If the watchdog timer is to be turned off at a point when interrupts may be enabled, instructions to disable and re-enable interrupts should be added to this sequence.

Watchdog Control Register (WDCON)

The reset values of the WDCON and WDL registers will be such that the watchdog timer has a timeout period of $4 \times 4096 \times t_{OSC}$ and the watchdog is running. WDCON can be written by software but the changes only take effect after executing a valid watchdog feed sequence.

Table 2. Prescaler Select Values in WDCON

PRE2	PRE1	PRE0	DIVISOR
0	0	0	32
0	0	1	64
0	1	0	128
0	1	1	256
1	0	0	512
1	0	1	1024
1	1	0	2048
1	1	1	4096

Watchdog Detailed Operation

When external $\overline{\text{RESET}}$ is applied, the following takes place:

- Watchdog run control bit set to ON (1).
- Autoload register WDL set to 00 (min. count).
- Watchdog time-out flag cleared.
- Prescaler is cleared.
- Prescaler tap set to the highest divide.
- Autoload takes place.

When coming out of a hardware reset, the software should load the autoload register and then feed the watchdog (cause an autoload).

If the watchdog is running and happens to underflow at the time the external RESET is applied, the watchdog time-out flag will be cleared.

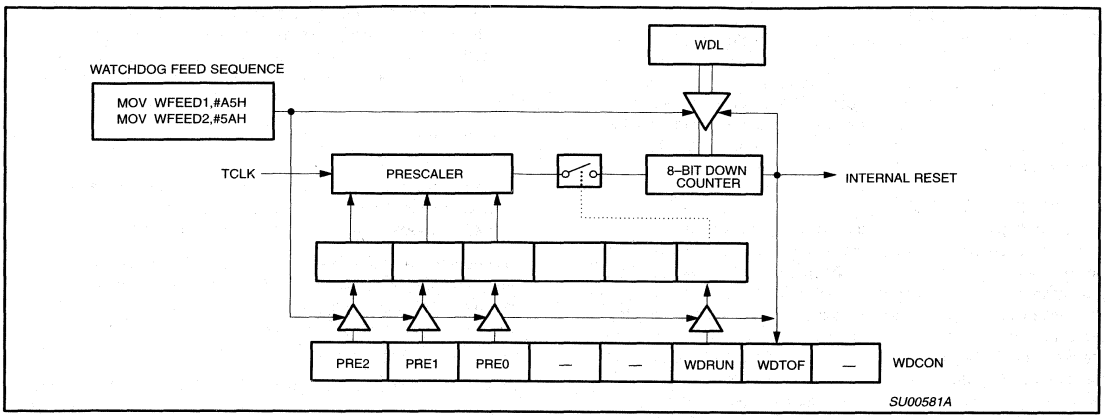


Figure 10. Watchdog Timer in XA-G1/G2/G3

When the watchdog underflows, the following action takes place (see Figure 10):

- Autoload takes place.
- Watchdog time-out flag is set
- Watchdog run bit unchanged.
- Autoload (WDL) register unchanged.
- Prescaler tap unchanged.
- All other device action same as external reset.

Note that if the watchdog underflows, the program counter will be loaded from the reset vector as in the case of an internal reset. The watchdog time-out flag can be examined to determine if the watchdog has caused the reset condition. The watchdog time-out flag bit can be cleared by software.

WDCON Register Bit Definitions

WDCON.7	PRE2	Prescaler Select 2, reset to 1
WDCON.6	PRE1	Prescaler Select 1, reset to 1
WDCON.5	PRE0	Prescaler Select 0, reset to 1
WDCON.4	—	
WDCON.3	—	
WDCON.2	WDRUN	Watchdog Run Control bit, reset to 1
WDCON.1	WDTOF	Timeout flag
WDCON.0	—	

UARTs

The XA-G1/G2/G3 includes 2 UART ports that are compatible with the enhanced UART used on the 8xC51FB. Baud rate selection is somewhat different due to the clocking scheme used for the XA timers.

Some other enhancements have been made to UART operation. The first is that there are separate interrupt vectors for each UART's transmit and receive functions. A break detect function has been added to the UART. This operates independently of the UART itself and provides a start-of-break status bit that the program may test. Finally, an Overrun Error flag has been added to detect missed characters in the received data stream.

Each UART rate is determined by either a fixed division of the oscillator (in UART modes 0 and 2) or by the timer 1 or timer 2 overflow rate (in UART modes 1 and 3).

Timer 1 defaults to clock both UART0 and UART1. Timer 2 can be programmed to clock either UART0 through T2CON (via bits R0CLK and T0CLK) or UART1 through T2MOD (via bits R1CLK and T1CLK). In this case, the UART not clocked by T2 could use T1 as the clock source.

The serial port receive and transmit registers are both accessed at Special Function Register SnBUF. Writing to SnBUF loads the transmit register, and reading SnBUF accesses a physically separate receive register.

The serial port can operate in 4 modes:

Mode 0: Serial I/O expansion mode. Serial data enters and exits through RxDn. TxDn outputs the shift clock. 8 bits are transmitted/received (LSB first). (The baud rate is fixed at 1/16 the oscillator frequency.)

Mode 1: Standard 8-bit UART mode. 10 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receive, the stop bit goes into RB8 in Special Function Register SnCON. The baud rate is variable.

Mode 2: Fixed rate 9-bit UART mode. 11 bits are transmitted (through TxD) or received (through RxD): start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8_n in SnCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8_n. On receive, the 9th data bit goes into RB8_n in Special Function Register SnCON, while the stop bit is ignored. The baud rate is programmable to 1/32 of the oscillator frequency.

Mode 3: Standard 9-bit UART mode. 11 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SnBUF as a destination register. Reception is initiated in Mode 0 by the condition RI_n = 0 and REN_n = 1. Reception is initiated in the other modes by the incoming start bit if REN_n = 1.

Serial Port Control Register

The serial port control and status register is the Special Function Register SnCON, shown in Figure 12. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8_n and RB8_n), and the serial port interrupt bits (TI_n and RI_n).

CLOCKING SCHEME/BAUD RATE GENERATION

The XA UARTS clock rates are determined by either a fixed division (modes 0 and 2) of the oscillator clock or by the Timer 1 or Timer 2 overflow rate (modes 1 and 3).

The clock for the UARTs in XA runs at 16x the Baud rate. If the timers are used as the source for Baud Clock, since maximum speed of timers/Baud Clock is Osc/4, the maximum baud rate is timer overflow divided by 16 i.e. Osc/64.

In Mode 0, it is fixed at Osc/16. In Mode 2, however, the fixed rate is Osc/32.

Pre-scaler for all Timers T0,1,2 controlled by PT1, PT0 bits in SCR	00	Osc/4
	01	Osc/16
	10	Osc/64
	11	reserved

Baud Rate for UART Mode 0:

Baud_Rate = Osc/16

Baud Rate calculation for UART Mode 1 and 3:

Baud_Rate = Timer_Rate/16

Timer_Rate = Osc/(N*(Timer_Range– Timer_Reload_Value))

where N = the TCLK prescaler value: 4, 16, or 64.
and Timer_Range = 256 for timer 1 in mode 2.
65536 for timer 1 in mode 0 and timer 2 in count up mode.

The timer reload value may be calculated as follows:

Timer_Reload_Value = Timer_Range–(Osc/(Baud_Rate*N*16))

NOTES:

1. The maximum baud rate for a UART in mode 1 or 3 is Osc/64.
2. The lowest possible baud rate (for a given oscillator frequency and N value) may be found by using a timer reload value of 0.
3. The timer reload value may never be larger than the timer range.
4. If a timer reload value calculation gives a negative or fractional result, the baud rate requested is not possible at the given oscillator frequency and N value.

Baud Rate for UART Mode 2:

Baud_Rate = Osc/32

Using Timer 2 to Generate Baud Rates

Timer T2 is a 16-bit up/down counter in XA. As a baud rate generator, timer 2 is selected as a clock source for either/both UART0 and UART1 transmitters and/or receivers by setting TCLKn and/or RCLKn in T2CON and T2MOD. As the baud rate generator, T2 is incremented as Osc/N where N = 4, 16 or 64 depending on TCLK as programmed in the SCR bits PT1, and PTO. So, if T2 is the source of one UART, the other UART could be clocked by either T1 overflow or fixed clock, and the UARTs could run independently with different baud rates.

T2CON 0x418		bit5	bit4	
		RCLK0	TCLK0	

T2MOD 0x419		bit5	bit4	
		RCLK1	TCLK1	

Prescaler Select for Timer Clock (TCLK)

SCR 0x440		bit3	bit2	
		PT1	PT0	

SnSTAT Address: S0STAT 421
S1STAT 425
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	FEn	BRn	OEn	STINTn

BIT	SYMBOL	FUNCTION
SnSTAT.3	FEn	Framing Error flag is set when the receiver fails to see a valid STOP bit at the end of the frame. Cleared by software.
SnSTAT.2	BRn	Break Detect flag is set if a character is received with all bits (including STOP bit) being logic '0'. Thus it gives a "Start of Break Detect" on bit 8 for Mode 1 and bit 9 for Modes 2 and 3. The break detect feature operates independently of the UARTs and provides the START of Break Detect status bit that a user program may poll. Cleared by software.
SnSTAT.1	OEn	Overrun Error flag is set if a new character is received in the receiver buffer while it is still full (before the software has read the previous character from the buffer), i.e., when bit 8 of a new byte is received while RI in SnCON is still set. Cleared by software.
SnSTAT.0	STINTn	This flag must be set to enable any of the above status flags to generate a receive interrupt (RIn). The only way it can be cleared is by a software write to this register.

SU00607B

Figure 11. Serial Port Extended Status (SnSTAT) Register (See also Figure 13 regarding Framing Error flag)

INTERRUPT SCHEME

There are separate interrupt vectors for each UART's transmit and receive functions.

Table 3. Vector Locations for UARTs in XA

Vector Address	Interrupt Source	Arbitration
A0H – A3H	UART 0 Receiver	7
A4H – A7H	UART 0 Transmitter	8
A8H – ABH	UART 1 Receiver	9
ACH – AFH	UART 1 Transmitter	10

NOTE:
The transmit and receive vectors could contain the same ISR address to work like a 8051 interrupt scheme

Error Handling, Status Flags and Break Detect
The UARTs in XA has the following error flags; see Figure 11.

Multiprocessor Communications

Modes 2 and 3 have a special provision for multiprocessor communications. In these modes, 9 data bits are received. The 9th one goes into RB8. Then comes a stop bit. The port can be programmed such that when the stop bit is received, the serial port interrupt will be activated only if RB8 = 1. This feature is enabled by setting bit SM2 in SCON. A way to use this feature in multiprocessor systems is as follows:

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the 9th bit is 1 in an address byte and 0 in a data byte. With SM2 = 1, no slave will be interrupted by a data byte. An address byte, however, will interrupt all slaves, so that each slave can examine the received byte and see if it is being addressed. The addressed slave will clear its SM2 bit and prepare to receive the data bytes that will be coming. The slaves that weren't being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

SM2 has no effect in Mode 0, and in Mode 1 can be used to check the validity of the stop bit although this is better done with the Framing Error (FE) flag. In a Mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a valid stop bit is received.

Automatic Address Recognition
Automatic Address Recognition is a feature which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9 bit UART modes, mode 2 and mode 3, the Receive Interrupt flag (RI) will be automatically set when the received byte contains either the "Given" address or the "Broadcast" address. The 9 bit mode requires that the 9th information bit is a 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Figure 14.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the Given slave address or addresses. All of the slaves may be contacted by using the Broadcast address. Two special Function Registers are used to define the slave's address, SADDR, and the address mask, SADEN. SADEN is used to define which bits in the SADDR are to be used and which bits are "don't care". The SADEN mask can be logically ANDed with the SADDR to create the "Given" address which the master will use for addressing each of the slaves. Use of the Given address allows multiple slaves to be recognized while excluding others. The following examples will help to show the versatility of this scheme:

Slave 0	SADDR	=	1100 0000
	SADEN	=	1111 1101
	Given	=	1100 00X0
Slave 1	SADDR	=	1100 0000
	SADEN	=	1111 1110
	Given	=	1100 000X

In the above example SADDR is the same and the SADEN data is used to differentiate between the two slaves. Slave 0 requires a 0 in bit 0 and it ignores bit 1. Slave 1 requires a 0 in bit 1 and bit 0 is ignored. A unique address for Slave 0 would be 1100 0010 since slave 1 requires a 0 in bit 1. A unique address for slave 1 would be 1100 0001 since a 1 in bit 0 will exclude slave 0. Both slaves can be selected at the same time by an address which has bit 0 = 0 (for slave 0) and bit 1 = 0 (for slave 1). Thus, both could be addressed with 1100 0000.

In a more complex system the following could be used to select slaves 1 and 2 while excluding slave 0:

Slave 0	SADDR	=	1100 0000
	SADEN	=	1111 1001
	Given	=	1100 0XX0
Slave 1	SADDR	=	1110 0000
	SADEN	=	1111 1010
	Given	=	1110 0X0X
Slave 2	SADDR	=	1110 0000
	SADEN	=	1111 1100
	Given	=	1110 00XX

In the above example the differentiation among the 3 slaves is in the lower 3 address bits. Slave 0 requires that bit 0 = 0 and it can be uniquely addressed by 1110 0110. Slave 1 requires that bit 1 = 0 and it can be uniquely addressed by 1110 and 0101. Slave 2 requires that bit 2 = 0 and its unique address is 1110 0011. To select Slaves 0 and 1 and exclude Slave 2 use address 1110 0100, since it is necessary to make bit 2 = 1 to exclude slave 2.

The Broadcast Address for each slave is created by taking the logical OR of SADDR and SADEN. Zeros in this result are teated as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR and SADEN are loaded with 0s. This produces a given address of all "don't cares" as well as a Broadcast address of all "don't cares". This effectively disables the Automatic Addressing mode and allows the microcontroller to use standard UART drivers which do not make use of this feature.

SnCON Address: S0CON 420
S1CON 424

Bit Addressable
Reset Value: 00H

MSB							LSB
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	shift register	fosc/ 16
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	fosc/32
1	1	3	9-bit UART	variable

BIT	SYMBOL	FUNCTION
SnCON.5	SM2	Enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1, then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2=1 then RI will not be activated if a valid stop bit was not received. In Mode 0, SM2 should be 0.
SnCON.4	REN	Enables serial reception. Set by software to enable reception. Clear by software to disable reception.
SnCON.3	TB8	The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
SnCON.2	RB8	In Modes 2 and 3, is the 9th data bit that was received. In Mode 1, it SM2=0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
SnCON.1	TI	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the end of the stop bit in the other modes. Must be cleared by software.
SnCON.0	RI	Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the end of the stop bit time in the other modes (except see SM2). Must be cleared by software.

SU00597B

Figure 12. Serial Port Control (SnCON) Register

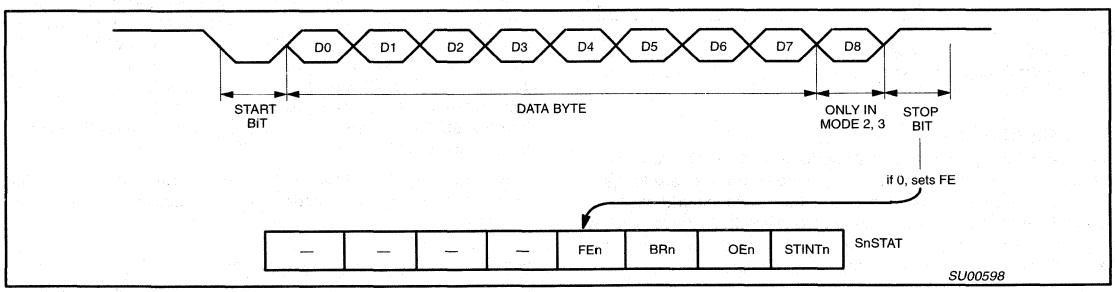


Figure 13. UART Framing Error Detection

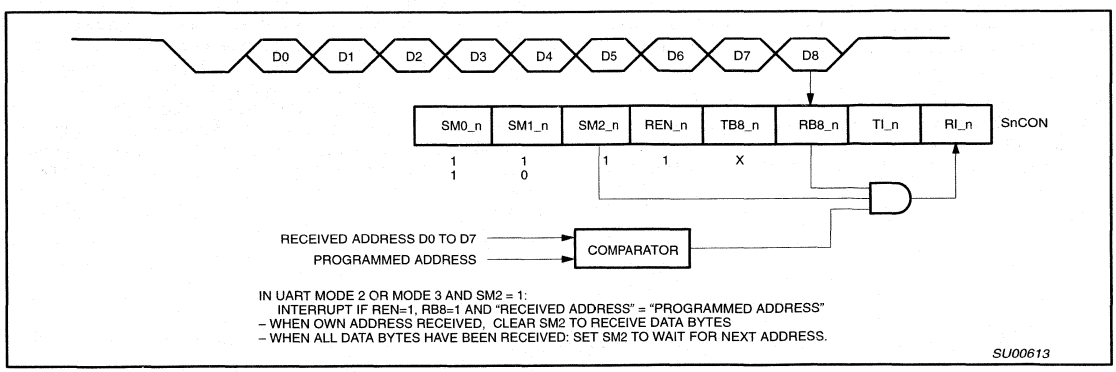


Figure 14. UART Multiprocessor Communication, Automatic Address Recognition

I/O PORT OUTPUT CONFIGURATION

Each I/O port pin can be user configured to one of 4 output types. The types are Quasi-bidirectional (essentially the same as standard 80C51 family I/O ports), Open-Drain, Push-Pull, and Off (high impedance). The default configuration after reset is Quasi-bidirectional. However, in the ROMless mode (the $\overline{EA}$ pin is low at reset), the port pins that comprise the external data bus will default to push-pull outputs.

I/O port output configurations are determined by the settings in port configuration SFRs. There are 2 SFRs for each port, called PnCFGA and PnCFGB, where "n" is the port number. One bit in each of the 2 SFRs relates to the output setting for the corresponding port pin, allowing any combination of the 2 output types to be mixed on those port pins. For instance, the output type of port 1 pin 3 is controlled by the setting of bit 3 in the SFRs P1CFGA and P1CFGB.

Table 4 shows the configuration register settings for the 4 port output types. The electrical characteristics of each output type may be found in the DC Characteristic table.

Table 4. Port Configuration Register Settings

PnCFGB	PnCFGA	Port Output Mode
0	0	Open Drain
0	1	Quasi-bidirectional
1	0	Off (high impedance)
1	1	Push-Pull

NOTE:
Mode changes may cause glitches to occur during transitions. When modifying both registers, WRITE instructions should be carried out consecutively.

EXTERNAL BUS

The external program/data bus allows for 8-bit or 16-bit bus width, and address sizes from 12 to 20 bits. The bus width is selected by an input at reset (see Reset Options below), while the address size is set by the program in a configuration register. If all off-chip code is selected (through the use of the $\overline{EA}$ pin), the initial code fetches will be done with the maximum address size (20 bits).

RESET

The device is reset whenever a logic "0" is applied to $\overline{RST}$ for at least 10 microseconds, placing a low level on the pin re-initializes the on-chip logic. Reset must be asserted when power is initially applied to the XA and held until the oscillator is running.

The duration of reset must be extended when power is initially applied or when using reset to exit power down mode. This is due to the need to allow the oscillator time to start up and stabilize. For most power supply ramp up conditions, this time is 10 milliseconds.

As it is brought high again, an exception is generated which causes the processor to jump to the address contained in the memory location 0000. The destination of the reset jump must be located in the first 64k of code address on power-up, all vectors are 16-bit values and so point to page zero addresses only. After a reset the RAM contents are indeterminate.

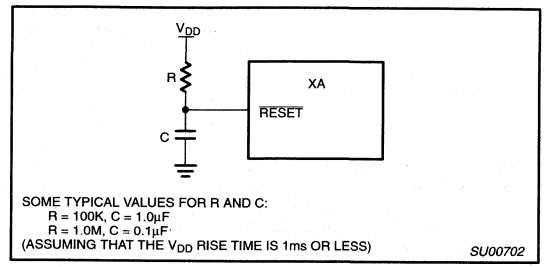


Figure 15. Recommended Reset Circuit

RESET OPTIONS

The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse, and determines whether the device is to begin execution from internal or external code memory. $\overline{EA}$ pulled high configures the XA in single-chip mode. If $\overline{EA}$ is driven low, the device enters ROMless mode. After Reset is released, the $\overline{EA}$/WAIT pin becomes a bus wait signal for external bus transactions.

The BUSW/P3.5 pin is weakly pulled high while reset is asserted, allowing simple biasing of the pin with a resistor to ground to select the alternate bus width. If the BUSW pin is not driven at reset, the weak pullup will cause a 1 to be loaded for the bus width, giving a 16-bit external bus. BUSW may be pulled low with a 2.7K or smaller value resistor, giving an 8-bit external bus. The bus width setting from the BUSW pin may be overridden by software once the user program is running.

Both $\overline{EA}$ and WAIT must be held for three oscillator clock times after reset is deasserted to guarantee that their values are latched correctly.

POWER REDUCTION MODES

The XA-G1/G2/G3 supports Idle and Power Down modes of power reduction. The idle mode leaves some peripherals running to allow them to wake up the processor when an interrupt is generated. The power down mode stops the oscillator in order to minimize power. The processor can be made to exit power down mode via reset or one of the external interrupt inputs. In order to use an external interrupt to re-activate the XA while in power down mode, the external interrupt must be enabled and be configured to level sensitive mode. In power down mode, the power supply voltage may be reduced to the RAM keep-alive voltage (2V), retaining the RAM, register, and SFR values at the point where the power down mode was entered.

INTERRUPTS

The XA-G1/G2/G3 supports 38 vectored interrupt sources. These include 9 maskable event interrupts, 7 exception interrupts, 16 trap interrupts, and 7 software interrupts. The maskable interrupts each have 8 priority levels and may be globally and/or individually enabled or disabled.

The XA defines four types of interrupts:

- **Exception Interrupts** – These are system level errors and other very important occurrences which include stack overflow, divide-by-0, and reset.
- **Event interrupts** – These are peripheral interrupts from devices such as UARTs, timers, and external interrupt inputs.
- **Software Interrupts** – These are equivalent of hardware interrupt, but are requested only under software control.
- **Trap Interrupts** – These are TRAP instructions, generally used to call system services in a multi-tasking system.

Exception interrupts, software interrupts, and trap interrupts are generally standard for XA derivatives and are detailed in the *XA User Guide*. Event interrupts tend to be different on different XA derivatives.

The XA-G1/G2/G3 supports a total of 9 maskable event interrupt sources (for the various XA peripherals), seven software interrupts, 5 exception interrupts (plus reset), and 16 traps. The maskable event interrupts share a global interrupt disable bit (the EA bit in the IEL register) and each also has a separate individual interrupt enable bit (in the IEL or IEH registers). Only three bits of the IPA register values are used on the XA-G1/G2/G3. Each event interrupt can be set to occur at one of 8 priority levels via bits in the Interrupt Priority (IP) registers, IPA0 through IPA5. The value 0 in the IPA field gives the interrupt priority 0, in effect disabling the interrupt. A value of 1 gives the interrupt a priority of 9, the value 2 gives priority 10, etc. The result is the same as if all four bits were used and the top bit set for all values except 0. Details of the priority scheme may be found in the XA User Guide.

The complete interrupt vector list for the XA-G1/G2/G3, including all 4 interrupt types, is shown in the following tables. The tables include the address of the vector for each interrupt, the related priority register bits (if any), and the arbitration ranking for that interrupt source. The arbitration ranking determines the order in which interrupts are processed if more than one interrupt of the same priority occurs simultaneously.

Table 5. Interrupt Vectors

EXCEPTION/TRAPS PRECEDENCE

DESCRIPTION	VECTOR ADDRESS	ARBITRATION RANKING
Reset (h/w, watchdog, s/w)	0000–0003	0 (High)
Breakpoint (h/w trap 1)	0004–0007	1
Trace (h/w trap 2)	0008–000B	1
Stack Overflow (h/w trap 3)	000C–000F	1
Divide by 0 (h/w trap 4)	0010–0013	1
User RETI (h/w trap 5)	0014–0017	1
TRAP 0– 15 (software)	0040–007F	1

EVENT INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY	ARBITRATION RANKING
External interrupt 0	IE0	0080–0083	EX0	IPA0.2–0 (PX0)	2
Timer 0 interrupt	TF0	0084–0087	ET0	IPA0.6–4 (PT0)	3
External interrupt 1	IE1	0088–008B	EX1	IPA1.2–0 (PX1)	4
Timer 1 interrupt	TF1	008C–008F	ET1	IPA1.6–4 (PT1)	5
Timer 2 interrupt	TF2(EXF2)	0090–0093	ET2	IPA2.2–0 (PT2)	6
Serial port 0 Rx	RI.0	00A0–00A3	ERI0	IPA4.2–0 (PRIO)	7
Serial port 0 Tx	TI.0	00A4–00A7	ETI0	IPA4.6–4 (PTIO)	8
Serial port 1 Rx	RI.1	00A8–00AB	ERI1	IPA5.2–0 (PRT1)	9
Serial port 1 Tx	TI.1	00AC–00AF	ETI1	IPA5.6–4 (PTI1)	10

SOFTWARE INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY
Software interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0118–011B	SWE7	(fixed at 7)

ABSOLUTE MAXIMUM RATINGS

PARAMETER	RATING	UNIT
Operating temperature under bias	–55 to +125	°C
Storage temperature range	–65 to +150	°C
Voltage on $\overline{EA}/V_{PP}$ pin to V_{SS}	0 to +13.0	V
Voltage on any other pin to V_{SS}	–0.5 to V_{DD}+0.5V	V
Maximum I_{OL} per I/O pin	15	mA
Power dissipation (based on package heat transfer limitations, not device power consumption)	1.5	W

DC ELECTRICAL CHARACTERISTICS

ROM (G13, G23, G33) and ROMless (G30): 2.7V to 5.5V unless otherwise specified;
EPROM/OTP (G17, G27, G37): V_{DD} = 5.0V ±5% unless otherwise specified;
T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial, unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP	MAX	
Supplies						
I_{DD}	Supply current operating	30 MHz		60	80	mA
I_{ID}	Idle mode supply current	30 MHz		22	30	mA
I_{PD}	Power-down current			5	100	µA
I_{PDI}	Power-down current (–40°C to +85°C)				150	µA
V_{RAM}	RAM-keep-alive voltage	RAM-keep-alive voltage	1.5			V
V_{IL}	Input low voltage		–0.5		$0.22V_{DD}$	V
V_{IH}	Input high voltage, except XTAL1, RST	At 5.0V	2.2			V
		At 3.3V	2			V
V_{IH1}	Input high voltage to XTAL1, RST	For both 3.0 & 5.0V	$0.7V_{DD}$			V
V_{OL}	Output low voltage all ports, ALE, PSEN[3]	I_{OL} = 3.2mA, V_{DD} = 5.0V			0.5	V
		1.0mA, V_{DD} = 3.0V			0.4	V
V_{OH1}	Output high voltage all ports, ALE, PSEN[1]	I_{OH} = –100µA, V_{DD} = 4.5V	2.4			V
		I_{OH} = –15µA, V_{DD} = 2.7V	2.0			V
V_{OH2}	Output high voltage, ports P0–3, ALE, PSEN[2]	I_{OH} = 3.2mA, V_{DD} = 4.5V	2.4			V
		I_{OH} = 1mA, V_{DD} = 2.7V	2.2			V
C_{IO}	Input/Output pin capacitance				15	pF
I_{IL}	Logical 0 input current, P0–3[6]	V_{IN} = 0.45V		–25	–75	µA
I_{LI}	Input leakage current, P0–3[5]	V_{IN} = V_{IL} or V_{IH}			±10	µA
I_{TL}	Logical 1 to 0 transition current all ports[4]	At 5.5V			–650	µA

NOTES:
1. Ports in Quasi bi-directional mode with weak pull-up (applies to ALE, $\overline{PSEN}$ only during RESET).
2. Ports in Push-Pull mode, both pull-up and pull-down assumed to be same strength
3. In all output modes
4. Port pins source a transition current when used in quasi-bidirectional mode and externally driven from 1 to 0. This current is highest when V_{IN} is approximately 2V.
5. Measured with port in high impedance output mode.
6. Measured with port in quasi-bidirectional output mode.
7. Load capacitance for all outputs=80pF.
8. Under steady state (non-transient) conditions, I_{OL} must be externally limited as follows:
   Maximum I_{OL} per port pin: 15mA (*NOTE: This is 85°C specification for V_{DD} = 5V.)
   Maximum I_{OL} per 8-bit port: 26mA
   Maximum total I_{OL} for all output: 71mA
   If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

AC ELECTRICAL CHARACTERISTICS (5V)

V_{DD} = 4.5V to 5.5V; T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial.

SYMBOL	FIGURE	PARAMETER	VARIABLE CLOCK		UNIT
			MIN	MAX	
External Clock					
f_C		Oscillator frequency	0	30	MHz
t_C	22	Clock period and CPU timing cycle	$1/f_C$		ns
t_{CHCX}	22	Clock high time	$t_C * 0.5$		ns
t_{CLCX}	22	Clock low time	$t_C * 0.4$		ns
t_{CLCH}	22	Clock rise time		5	ns
t_{CHCL}	22	Clock fall time		5	ns
Address Cycle					
t_{CRAR}	21	Delay from clock rising edge to ALE rising edge	10	46	ns
t_{LHLL}	16	ALE pulse width (programmable)	$(V1 * t_C) - 6$		ns
t_{AVLL}	16	Address valid to ALE de-asserted (set-up)	$(V1 * t_C) - 12$		ns
t_{LLAX}	16	Address hold after ALE de-asserted	$(t_C/2) - 10$		ns
Code Read Cycle					
t_{PLPH}	16	$\overline{PSEN}$ pulse width	$(V2 * t_C) - 10$		ns
t_{LLPL}	16	ALE de-asserted to $\overline{PSEN}$ asserted	$(t_C/2) - 7$		ns
t_{AVIVA}	16	Address valid to instruction valid, ALE cycle (access time)		$(V3 * t_C) - 36$	ns
t_{AVIVB}	17	Address valid to instruction valid, non-ALE cycle (access time)		$(V4 * t_C) - 29$	ns
t_{PLIV}	16	$\overline{PSEN}$ asserted to instruction valid (enable time)		$(V2 * t_C) - 29$	ns
t_{PXIX}	16	Instruction hold after $\overline{PSEN}$ de-asserted	0		ns
t_{PXIZ}	16	Bus 3-State after $\overline{PSEN}$ de-asserted (disable time)		$t_C - 8$	ns
t_{IXUA}	16	Hold time of unlatched part of address after instruction latched	0		ns
Data Read Cycle					
t_{RLRH}	18	$\overline{RD}$ pulse width	$(V7 * t_C) - 10$		ns
t_{LLRL}	18	ALE de-asserted to $\overline{RD}$ asserted	$(t_C/2) - 7$		ns
t_{AVDVA}	18	Address valid to data input valid, ALE cycle (access time)		$(V6 * t_C) - 36$	ns
t_{AVDVB}	19	Address valid to data input valid, non-ALE cycle (access time)		$(V5 * t_C) - 29$	ns
t_{RLDV}	18	$\overline{RD}$ low to valid data in, enable time		$(V7 * t_C) - 29$	ns
t_{RHDX}	18	Data hold time after $\overline{RD}$ de-asserted	0		ns
t_{RHDZ}	18	Bus 3-State after $\overline{RD}$ de-asserted (disable time)		$t_C - 8$	ns
t_{DXUA}	18	Hold time of unlatched part of address after data latched	0		ns
Data Write Cycle					
t_{WLWH}	20	$\overline{WR}$ pulse width	$(V8 * t_C) - 10$		ns
t_{LLWL}	20	ALE falling edge to $\overline{WR}$ asserted	$(V12 * t_C) - 10$		ns
t_{QVWX}	20	Data valid before $\overline{WR}$ asserted (data setup time)	$(V13 * t_C) - 22$		ns
t_{WHQX}	20	Data hold time after $\overline{WR}$ de-asserted (Note 6)	$(V11 * t_C) - 5$		ns
t_{AVWL}	20	Address valid to $\overline{WR}$ asserted (address setup time) (Note 5)	$(V9 * t_C) - 22$		ns
t_{UAWH}	20	Hold time of unlatched part of address after $\overline{WR}$ is de-asserted	$(V11 * t_C) - 7$		ns
Wait Input					
t_{WTH}	21	WAIT stable after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) asserted		$(V10 * t_C) - 30$	ns
t_{WTL}	21	WAIT hold after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) assertion	$(V10 * t_C) - 5$		ns

NOTES ON PAGE 342.

AC ELECTRICAL CHARACTERISTICS (3V)

V_{DD} = 2.7V to 4.5V; T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial.

SYMBOL	FIGURE	PARAMETER	VARIABLE CLOCK		UNIT
			MIN	MAX	
Address Cycle					
t_{CRAR}	21	Delay from clock rising edge to ALE rising edge	15	60	ns
t_{LHLL}	16	ALE pulse width (programmable)	(V1 * t_C) – 10		ns
t_{AVLL}	16	Address valid to ALE de-asserted (set-up)	(V1 * t_C) – 18		ns
t_{LLAX}	16	Address hold after ALE de-asserted	(t_C/2) – 12		ns
Code Read Cycle					
t_{PLPH}	16	$\overline{PSEN}$ pulse width	(V2 * t_C) – 12		ns
t_{LLPL}	16	ALE de-asserted to $\overline{PSEN}$ asserted	(t_C/2) – 9		ns
t_{AVIVA}	16	Address valid to instruction valid, ALE cycle (access time)		(V3 * t_C) – 58	ns
t_{AVIVB}	17	Address valid to instruction valid, non-ALE cycle (access time)		(V4 * t_C) – 52	ns
t_{PLIV}	16	$\overline{PSEN}$ asserted to instruction valid (enable time)		(V2 * t_C) – 52	ns
t_{PXIX}	16	Instruction hold after $\overline{PSEN}$ de-asserted	0		ns
t_{PXIZ}	16	Bus 3-State after $\overline{PSEN}$ de-asserted (disable time)		t_C – 8	ns
t_{IXUA}	16	Hold time of unlatched part of address after instruction latched	0		ns
Data Read Cycle					
t_{RLRH}	18	$\overline{RD}$ pulse width	(V7 * t_C) – 12		ns
t_{LLRL}	18	ALE de-asserted to $\overline{RD}$ asserted	(t_C/2) – 9		ns
t_{AVDVA}	18	Address valid to data input valid, ALE cycle (access time)		(V6 * t_C) – 58	ns
t_{AVDVB}	19	Address valid to data input valid, non-ALE cycle (access time)		(V5 * t_C) – 52	ns
t_{RLDV}	18	$\overline{RD}$ low to valid data in, enable time		(V7 * t_C) – 52	ns
t_{RHDX}	18	Data hold time after $\overline{RD}$ de-asserted	0		ns
t_{RHDZ}	18	Bus 3-State after $\overline{RD}$ de-asserted (disable time)		t_C – 8	ns
t_{DXUA}	18	Hold time of unlatched part of address after data latched	0		ns
Data Write Cycle					
t_{WLWH}	20	$\overline{WR}$ pulse width	(V8 * t_C) – 12		ns
t_{LLWL}	20	ALE falling edge to $\overline{WR}$ asserted	(V12 * t_C) – 10		ns
t_{QVWX}	20	Data valid before $\overline{WR}$ asserted (data setup time)	(V13 * t_C) – 28		ns
t_{WHQX}	20	Data hold time after $\overline{WR}$ de-asserted (Note 6)	(V11 * t_C) – 8		ns
t_{AVWL}	20	Address valid to $\overline{WR}$ asserted (address setup time) (Note 5)	(V9 * t_C) – 28		ns
t_{UAWH}	20	Hold time of unlatched part of address after $\overline{WR}$ is de-asserted	(V11 * t_C) – 10		ns
Wait Input					
t_{WTH}	21	WAIT stable after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) asserted		(V10 * t_C) – 40	ns
t_{WTL}	21	WAIT hold after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) assertion	(V10 * t_C) – 5		ns

NOTES:

1. Load capacitance for all outputs = 80pF.
2. Variables V1 through V13 reflect programmable bus timing, which is programmed via the Bus Timing registers (BTRH and BTRL). Refer to the *XA User Guide* for details of the bus timing settings.
   - V1) This variable represents the programmed width of the ALE pulse as determined by the ALEW bit in the BTRL register. V1 = 0.5 if the ALEW bit = 0, and 1.5 if the ALEW bit = 1.
   - V2) This variable represents the programmed width of the $\overline{PSEN}$ pulse as determined by the CR1 and CR0 bits or the CRA1, CRA0, and ALEW bits in the BTRL register.
     - For a bus cycle with **no** ALE, V2 = 1 if CR1/0 = 00, 2 if CR1/0 = 01, 3 if CR1/0 = 10, and 4 if CR1/0 = 11. Note that during burst mode code fetches, $\overline{PSEN}$ does not exhibit transitions at the boundaries of bus cycles. V2 still applies for the purpose of determining peripheral timing requirements.
     - For a bus cycle **with** an ALE, V2 = the total bus cycle duration (2 if CRA1/0 = 00, 3 if CRA1/0 = 01, 4 if CRA1/0 = 10, and 5 if CRA1/0 = 11) minus the number of clocks used by ALE (V1 + 0.5).
       Example: If CRA1/0 = 10 and ALEW = 1, the V2 = 4 – (1.5 + 0.5) = 2.

V3) This variable represents the programmed length of an entire code read cycle **with** ALE. This time is determined by the CRA1 and CRA0 bits in the BTRL register. V3 = the total bus cycle duration (2 if CRA1/0 = 00, 3 if CRA1/0 = 01, 4 if CRA1/0 = 10, and 5 if CRA1/0 = 11).

V4) This variable represents the programmed length of an entire code read cycle with **no** ALE. This time is determined by the CR1 and CR0 bits in the BTRL register. V4 = 1 if CR1/0 = 00, 2 if CR1/0 = 01, 3 if CR1/0 = 10, and 4 if CR1/0 = 11.

V5) This variable represents the programmed length of an entire data read cycle with **no** ALE. this time is determined by the DR1 and DR0 bits in the BTRH register. V5 = 1 if DR1/0 = 00, 2 if DR1/0 = 01, 3 if DR1/0 = 10, and 4 if DR1/0 = 11.

V6) This variable represents the programmed length of an entire data read cycle **with** ALE. The time is determined by the DRA1 and DRA0 bits in the BTRH register. V6 = the total bus cycle duration (2 if DRA1/0 = 00, 3 if DRA1/0 = 01, 4 if DRA1/0 = 10, and 5 if DRA1/0 = 11).

V7) This variable represents the programmed width of the $\overline{RD}$ pulse as determined by the DR1 and DR0 bits or the DRA1, DRA0 in the BTRH register, and the ALEW bit in the BTRL register. Note that during a 16-bit operation on an 8-bit external bus, $\overline{RD}$ remains low and does not exhibit a transition between the first and second byte bus cycles. V7 still applies for the purpose of determining peripheral timing requirements. The timing for the first byte is for a bus cycle with ALE, the timing for the second byte is for a bus cycle with no ALE.
- For a bus cycle with **no** ALE, V7 = 1 if DR1/0 = 00, 2 if DR1/0 = 01, 3 if DR1/0 = 10, and 4 if DR1/0 = 11.
- For a bus cycle **with** an ALE, V7 = the total bus cycle duration (2 if DRA1/0 = 00, 3 if DRA1/0 = 01, 4 if DRA1/0 = 10, and 5 if DRA1/0 = 11) minus the number of clocks used by ALE (V1 + 0.5).
  Example: If DRA1/0 = 00 and ALEW = 0, then V7 = 2 – (0.5 + 0.5) = 1.

V8) This variable represents the programmed width of the WRL and/or WRH pulse as determined by the WM1 bit in the BTRL register. V8 1 if WM1 = 0, and 2 if WM1 = 1.

V9) This variable represents the programmed address setup time for a write as determined by the data write cycle duration (defined by DW1 and DW0 or the DWA1 and DWA0 bits in the BTRH register), the WM0 bit in the BTRL register, and the value of V8.
- For a bus cycle **with** an ALE, V9 = the total bus write cycle duration (2 if DWA1/0 = 00, 3 if DWA1/0 = 01, 4 if DWA1/0 = 10, and 5 if DWA1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1).
  Example: If DWA1/0 = 10, WM0 = 1, and WM1 = 1, then V9 = 4 – 1 – 2 = 1.
- For a bus cycle with **no** ALE, V9 = the total bus cycle duration (2 if DW1/0 = 00, 3 if DW1/0 = 01, 4 if DW1/0 = 10, and 5 if DW1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1).
  Example: If DW1/0 = 11, WM0 = 1, and WM1 = 0, then V9 = 5 – 1 – 1 = 3.

V10) This variable represents the length of a bus strobe for calculation of WAIT setup and hold times. The strobe may be $\overline{RD}$ (for data read cycles), $\overline{WRL}$ and/or $\overline{WRH}$ (for data write cycles), or $\overline{PSEN}$ (for code read cycles), depending on the type of bus cycle being widened by WAIT. V10 = V2 for WAIT associated with a code read cycle using $\overline{PSEN}$. V10 = V8 for a data write cycle using $\overline{WRL}$ and/or $\overline{WRH}$. V10 = V7–1 for a data read cycle using $\overline{RD}$. This means that a single clock data read cycle cannot be stretched using WAIT. If WAIT is used to vary the duration of data read cycles, the $\overline{RD}$ strobe width must be set to be at least two clocks in duration. Also see Note 4.

V11) This variable represents the programmed write hold time as determined by the WM0 bit in the BTRL register. V11 = 0 if the WM0 bit = 0, and 1 if the WM0 bit = 1.

V12) This variable represents the programmed period between the end of the ALE pulse and the beginning of the $\overline{WRL}$ and/or $\overline{WRH}$ pulse as determined by the data write cycle duration (defined by the DWA1 and DWA0 bits in the BTRH register), the WM0 bit in the BTRL register, and the values of V1 and V8. V12 = the total bus cycle duration (2 if DWA1/0 = 00, 3 if DWA1/0 = 01, 4 if DWA1/0 = 10, and 5 if DWA1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1), minus the width of the ALE pulse (V1).
Example: If DWA1/0 = 11, WM0 = 1, WM1 = 0, and ALEW = 1, then V12 = 5 – 1 – 1 – 1.5 = 1.5.

V13) This variable represents the programmed data setup time for a write as determined by the data write cycle duration (defined by DW1 and DW0 or the DWA1 and DWA0 bits in the BTRH register), the WM0 bit in the BTRL register, and the values of V1 and V8.
- For a bus cycle **with** an ALE, V13 = the total bus cycle duration (2 if DWA1/0 = 00, 3 if DWA1/0 = 01, 4 if DWA1/0 = 10, and 5 if DWA1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1), minus the number of clocks used by ALE (V1 + 0.5).
  Example: If DWA1/0 = 11, WM0 = 1, WM1 = 1, and ALEW = 0, then V13 = 5 – 1 – 2 – 1 = 1.
- For a bus cycle with **no** ALE, V13 = the total bus cycle duration (2 if DW1/0 = 00, 3 if DW1/0 = 01, 4 if DW1/0 = 10, and 5 if DW1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1).
  Example: If DW1/0 = 01, WM0 = 1, and WM1 = 0, then V13 = 3 – 1 – 1 = 1.

3. Not all combinations of bus timing configuration values result in valid bus cycles. Please refer to the XA User Guide section on the External Bus for details.
4. When code is being fetched for execution on the external bus, a burst mode fetch is used that does not have $\overline{PSEN}$ edges in every fetch cycle. Thus, if WAIT is used to delay code fetch cycles, a change in the low order address lines must be detected to locate the beginning of a cycle. This would be A3–A0 for an 8-bit bus, and A3–A1 for a 16-bit bus. Also, a 16-bit data read operation conducted on a 8-bit wide bus similarly does not include two separate $\overline{RD}$ strobes. So, a rising edge on the low order address line (A0) must be used to trigger a WAIT in the second half of such a cycle.
5. This parameter is provided for peripherals that have the data clocked in on the falling edge of the $\overline{WR}$ strobe. This is not usually the case, and in most applications this parameter is not used.
6. Please note that the XA-G1/G2/G3 requires that extended data bus hold time (WM0 = 1) to be used with external bus write cycles.

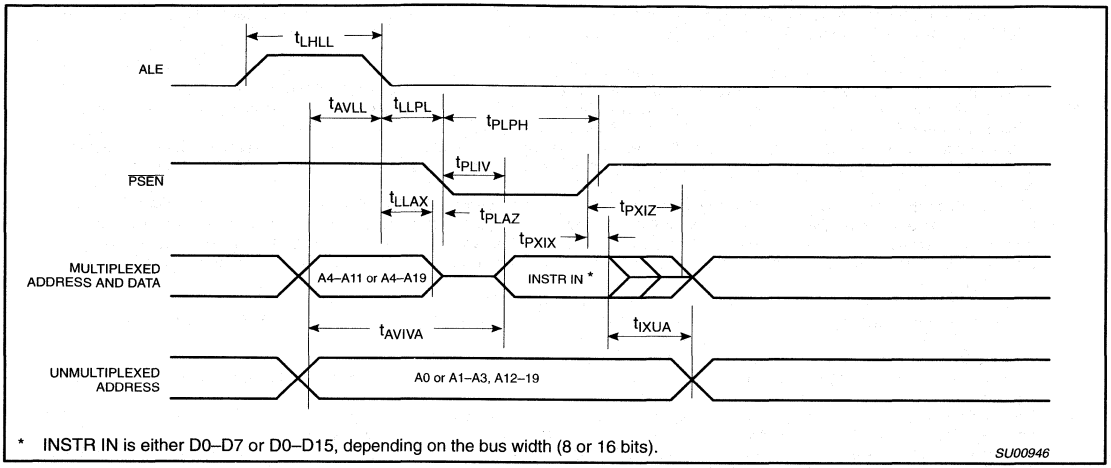


Figure 16. External Program Memory Read Cycle (ALE Cycle)

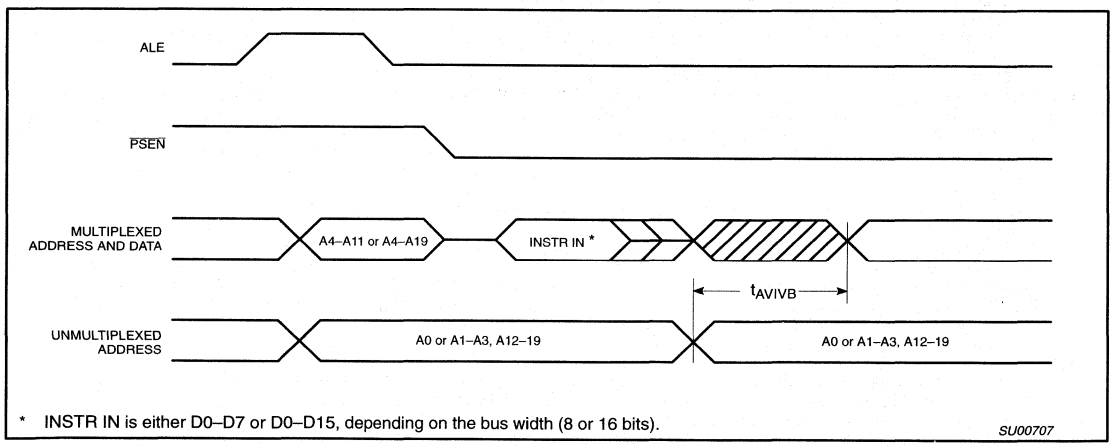


Figure 17. External Program Memory Read Cycle (Non-ALE Cycle)

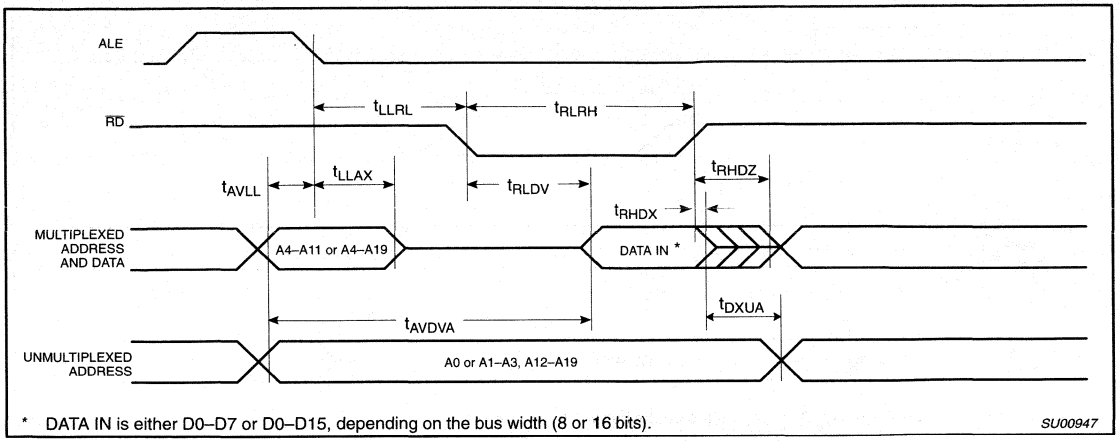


Figure 18. External Data Memory Read Cycle (ALE Cycle)

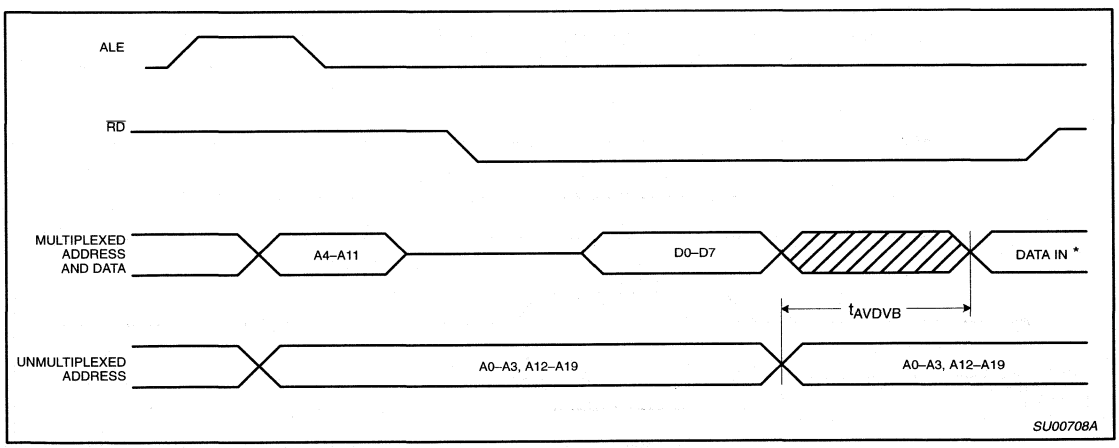


Figure 19. External Data Memory Read Cycle (Non-ALE Cycle)

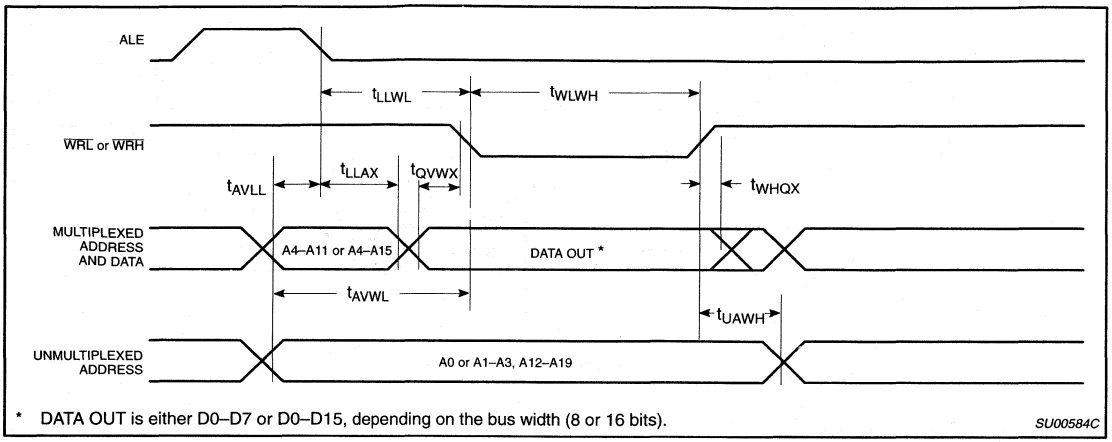


Figure 20. External Data Memory Write Cycle

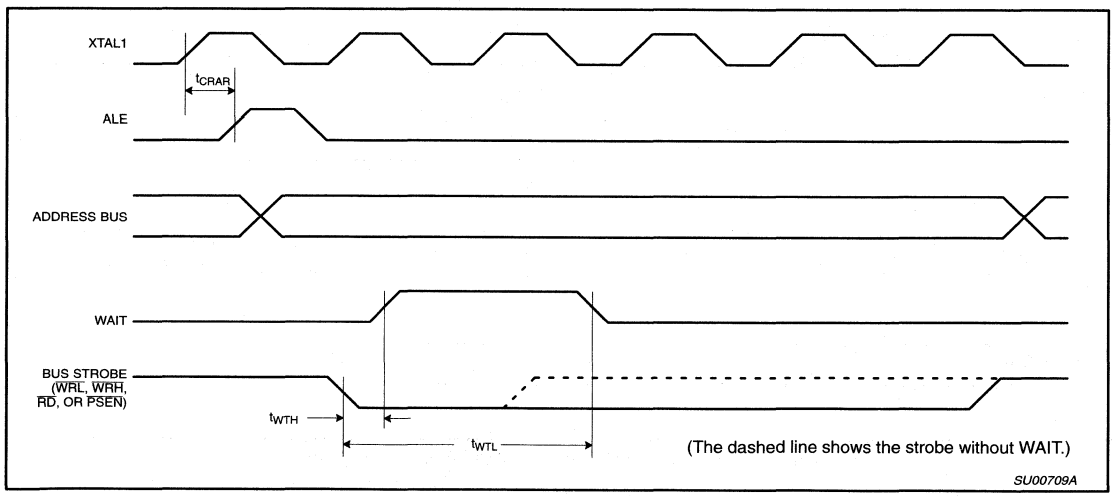


Figure 21. WAIT Signal Timing

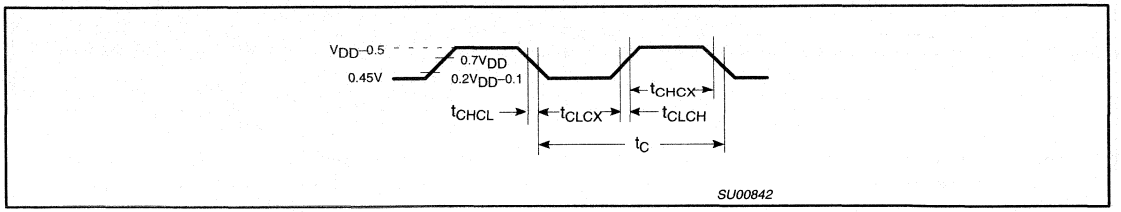


Figure 22. External Clock Drive

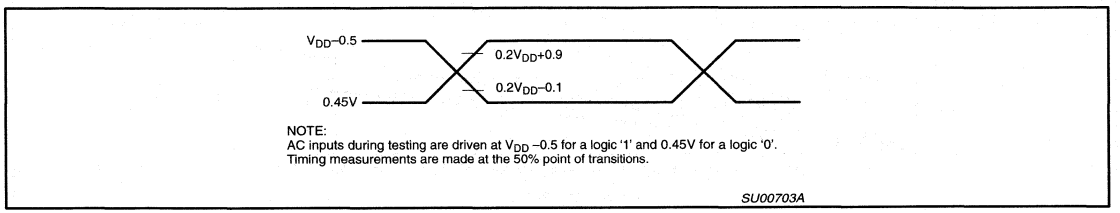


Figure 23. AC Testing Input/Output

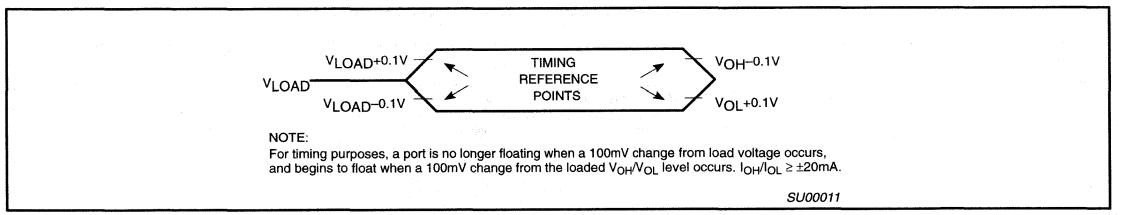


Figure 24. Float Waveform

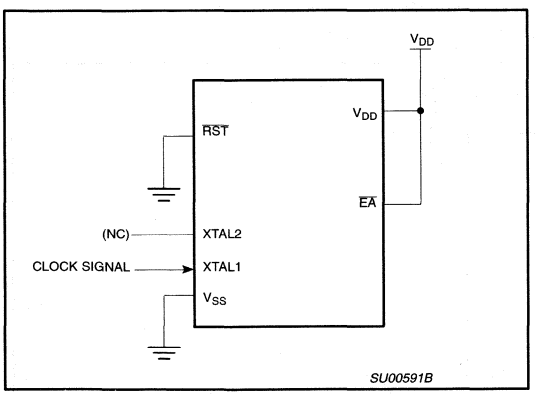


Figure 25. I_{DD} Test Condition, Active Mode
All other pins are disconnected

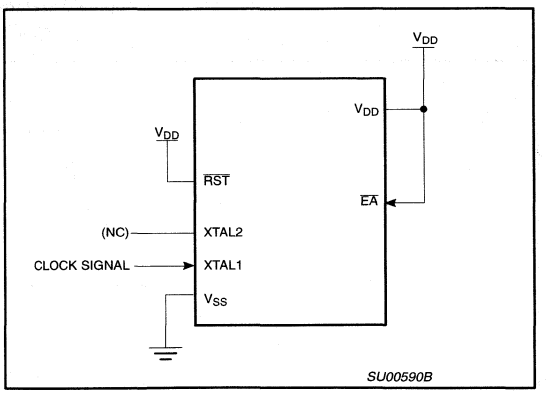


Figure 26. I_{DD} Test Condition, Idle Mode
All other pins are disconnected

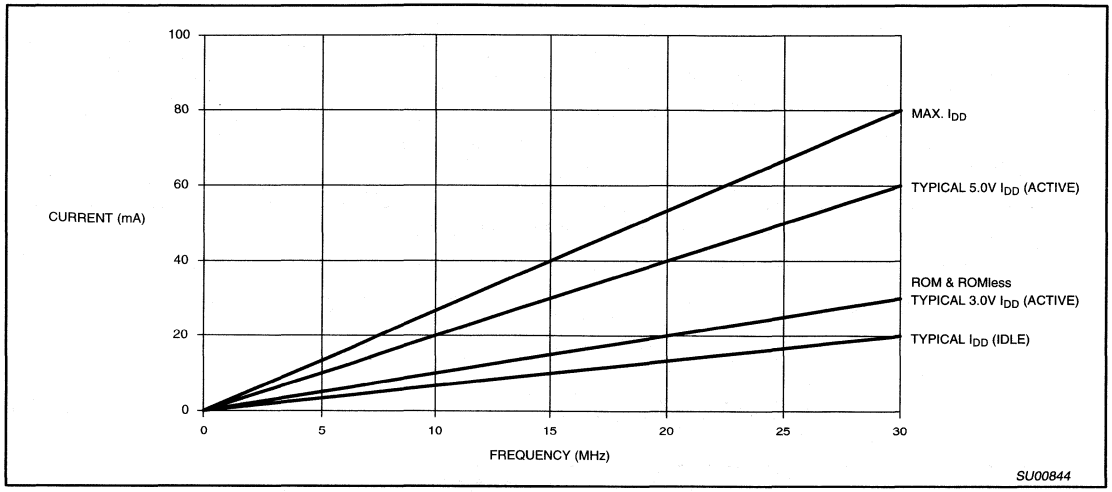


Figure 27. I_{DD} vs. Frequency
Valid only within frequency specification of the device under test.

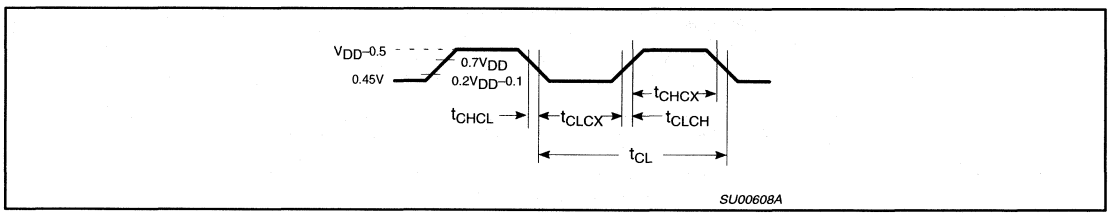


Figure 28. Clock Signal Waveform for I_{DD} Tests in Active and Idle Modes
$t_{CLCH} = t_{CHCL} = 5ns$

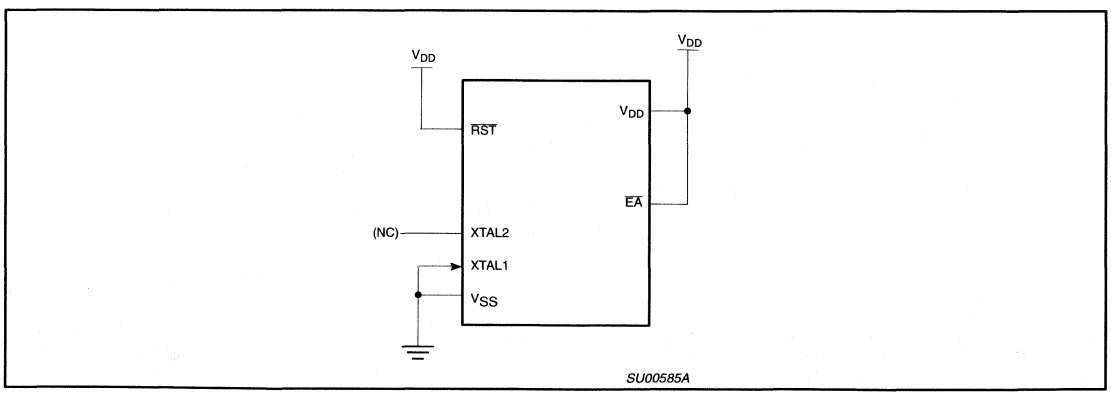


Figure 29. I_{DD} Test Condition, Power Down Mode
All other pins are disconnected. V_{DD}=2V to 5.5V

EPROM CHARACTERISTICS

The XA-G17/G27/G37 is programmed by using a modified Improved Quick-Pulse Programming™ algorithm. This algorithm is essentially the same as that used by the later 80C51 family EPROM parts. However different pins are used for many programming functions.

Detailed EPROM programming information may be obtained from the internet at www.philipsmcu.com/ftp.html.

The XA-G1/G2/G3 contains three signature bytes that can be read and used by an EPROM programming system to identify the device. The signature bytes identify the device as an XA-Gx manufactured by Philips.

Security Bits

With none of the security bits programmed the code in the program memory can be verified. When only security bit 1 (see Table 6) is programmed, MOVC instructions executed from external program memory are disabled from fetching code bytes from the internal memory. All further programming of the EPROM is disabled. When security bits 1 and 2 are programmed, in addition to the above, verify mode is disabled. When all three security bits are programmed, all of the conditions above apply and all external program memory execution is disabled. (See Table 6)

Table 6. Program Security Bits

PROGRAM LOCK BITS				
	SB1	SB2	SB3	PROTECTION DESCRIPTION
1	U	U	U	No Program Security features enabled.
2	P	U	U	MOVC instructions executed from external program memory are disabled from fetching code bytes from internal memory and further programming of the EPROM is disabled.
3	P	P	U	Same as 2, also verify is disabled.
4	P	P	P	Same as 3, external execution is disabled. Internal data RAM is not accessible.

NOTES:
1. P – programmed. U – unprogrammed.
2. Any other combination of the security bits is not defined.

™Trademark phrase of Intel Corporation.

ROM CODE SUBMISSION

When submitting ROM code for the XA-G13, the following must be specified:

1. 8k bytes user ROM data.
2. ROM security bits.

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 1FFFH	DATA	7:0	User ROM Data
2020H	SECURITY BIT	0	ROM Security Bit 1 0 = enable security 1 = disable security
2020H	SECURITY BIT	1	ROM Security Bit 2 0 = enable security 1 = disable security
2020H	SECURITY BIT	3	ROM Security Bit 3 0 = enable security 1 = disable security

When submitting ROM code for the XA-G23, the following must be specified:

1. 16k bytes user ROM data.
2. ROM security bits.

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 3FFFH	DATA	7:0	User ROM Data
4020H	SECURITY BIT	0	ROM Security Bit 1 0 = enable security 1 = disable security
4020H	SECURITY BIT	1	ROM Security Bit 2 0 = enable security 1 = disable security
4020H	SECURITY BIT	3	ROM Security Bit 3 0 = enable security 1 = disable security

When submitting ROM code for the XA-G33, the following must be specified:

1. 32k bytes user ROM data.
2. ROM security bits.

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 7FFFH	DATA	7:0	User ROM Data
8020H	SECURITY BIT	0	ROM Security Bit 1 0 = enable security 1 = disable security
8020H	SECURITY BIT	1	ROM Security Bit 2 0 = enable security 1 = disable security
8020H	SECURITY BIT	3	ROM Security Bit 3 0 = enable security 1 = disable security

XA 16-bit microcontroller
32K/1K OTP/ROM/ROMless, 8-channel 8-bit A/D, low voltage (2.7V–5.5V), I²C, 2 UARTs, 16MB address range

XA-S3

GENERAL DESCRIPTION

The XA-S3 device is a member of Philips Semiconductors' XA (eXtended Architecture) family of high performance 16-bit single-chip microcontrollers.

The XA-S3 device combines many powerful peripherals on one chip. With its high performance A/D converter, timers/counters, watchdog, Programmable Counter Array (PCA), I²C interface, dual UARTs, and multiple general purpose I/O ports, it is suited for general multipurpose high performance embedded control functions.

Specific features of the XA-S3

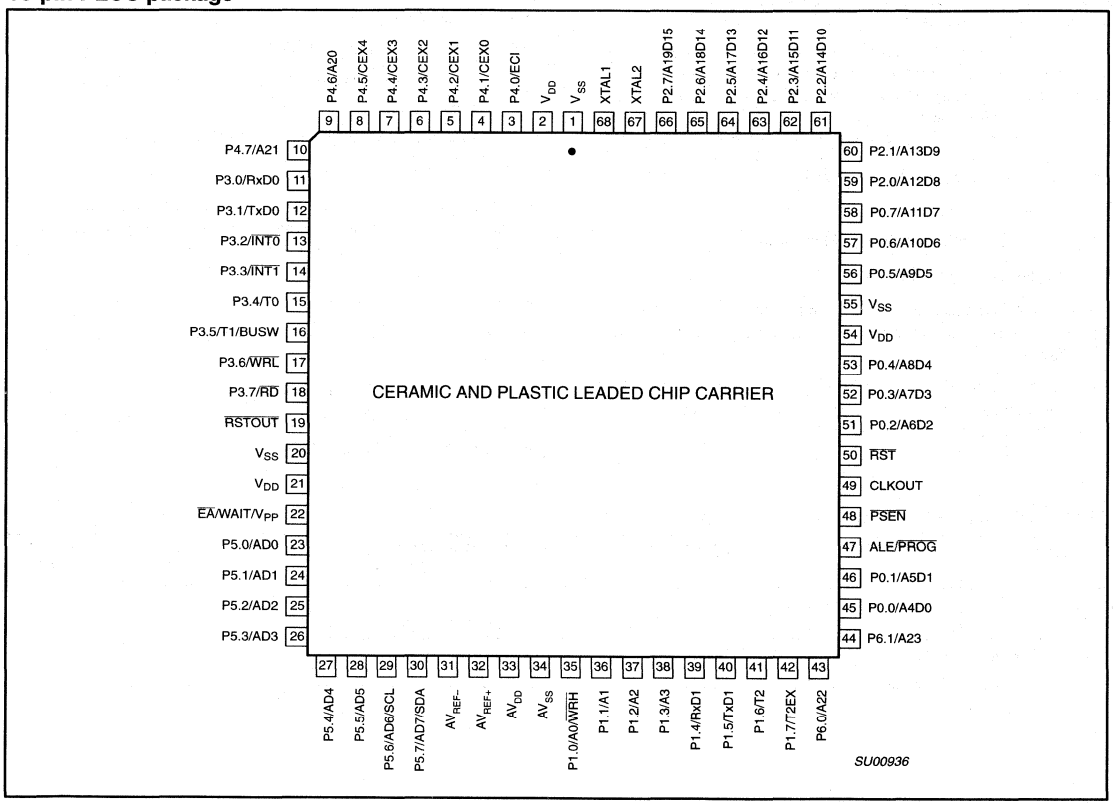
- 2.7 V to 5.5 V operation.
- 32K bytes of on-chip EPROM/ROM program memory.
- 1024 bytes of on-chip data RAM.
- Supports off-chip addressing up to 16 megabytes (24 address lines). A clock output reference is added to simplify external bus interfacing.
- High performance 8-channel 8-bit A/D converter with automatic channel scan and repeated read functions. Completes a conversion in 4.46 microseconds at 30 MHz.
- Three standard counter/timers with enhanced features. All timers have a toggle output capability.
- Watchdog timer.
- 5-channel 16-bit Programmable Counter Array (PCA).
- I²C-bus serial I/O port with byte-oriented master and slave functions.
- Two enhanced UARTs with independent baud rates.
- Seven software interrupts.
- Active low reset output pin indicates all reset occurrences (external reset, watchdog reset and the RESET instruction). A reset source register allows program determination of the cause of the most recent reset.
- 50 I/O pins, each with 4 programmable output configurations.
- 30 MHz operating frequency at 2.7–5.5V V_{DD} over commercial operating conditions.
- Power saving operating modes: Idle and Power-down. Wake-up from power-down via an external interrupt is supported.
- 68-pin PLCC and 80-pin PQFP packages.

ORDERING INFORMATION

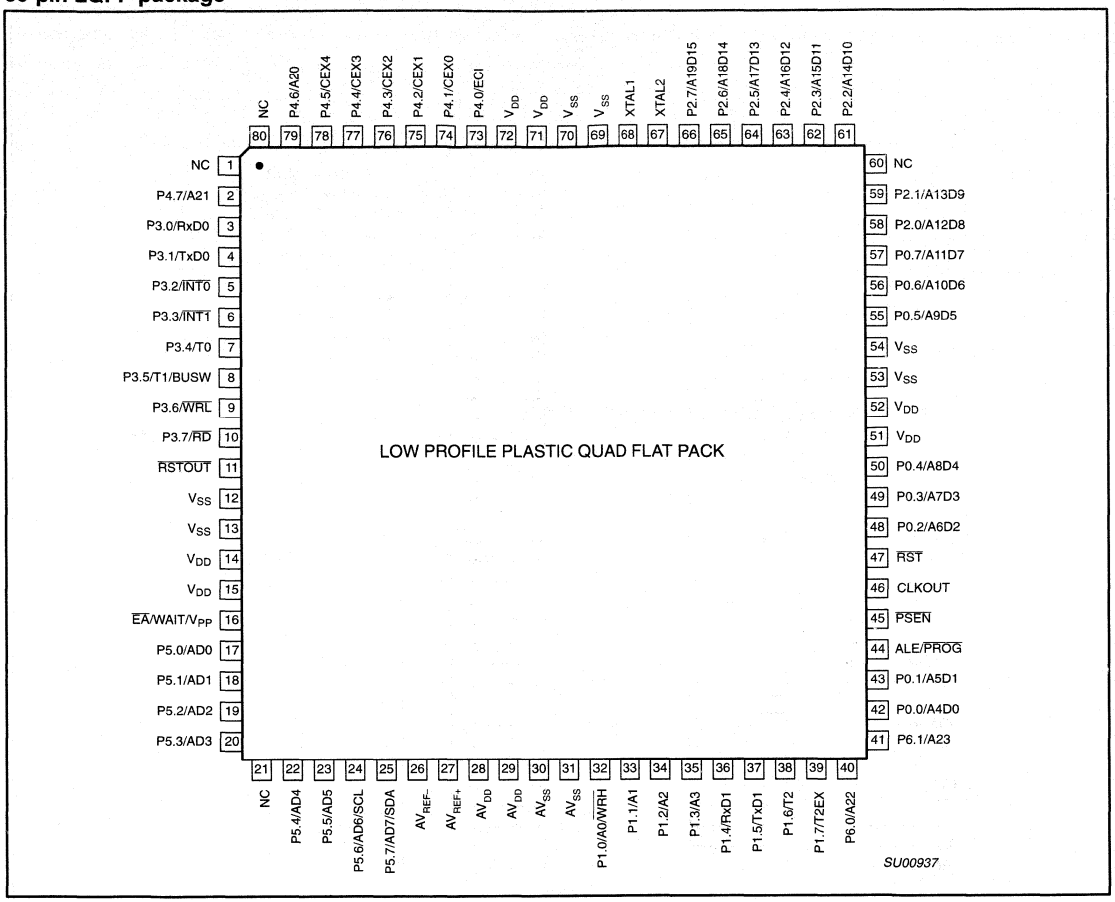
ROMless	ROM	EPROM		TEMPERATURE RANGE (°C) AND PACKAGE	FREQ. (MHz)	DRAWING NUMBER
P51XAS30KBBA	P51XAS33KBBA	P51XAS37KBBA	OTP	0 to +70, 68-pin Plastic Leaded Chip Carrier	30	SOT188-3
P51XAS30KBBD	P51XAS33KBBD	P51XAS37KBBD	OTP	0 to +70, 80-pin Plastic Quad Flat Pack	30	SOT315-1

PIN CONFIGURATIONS

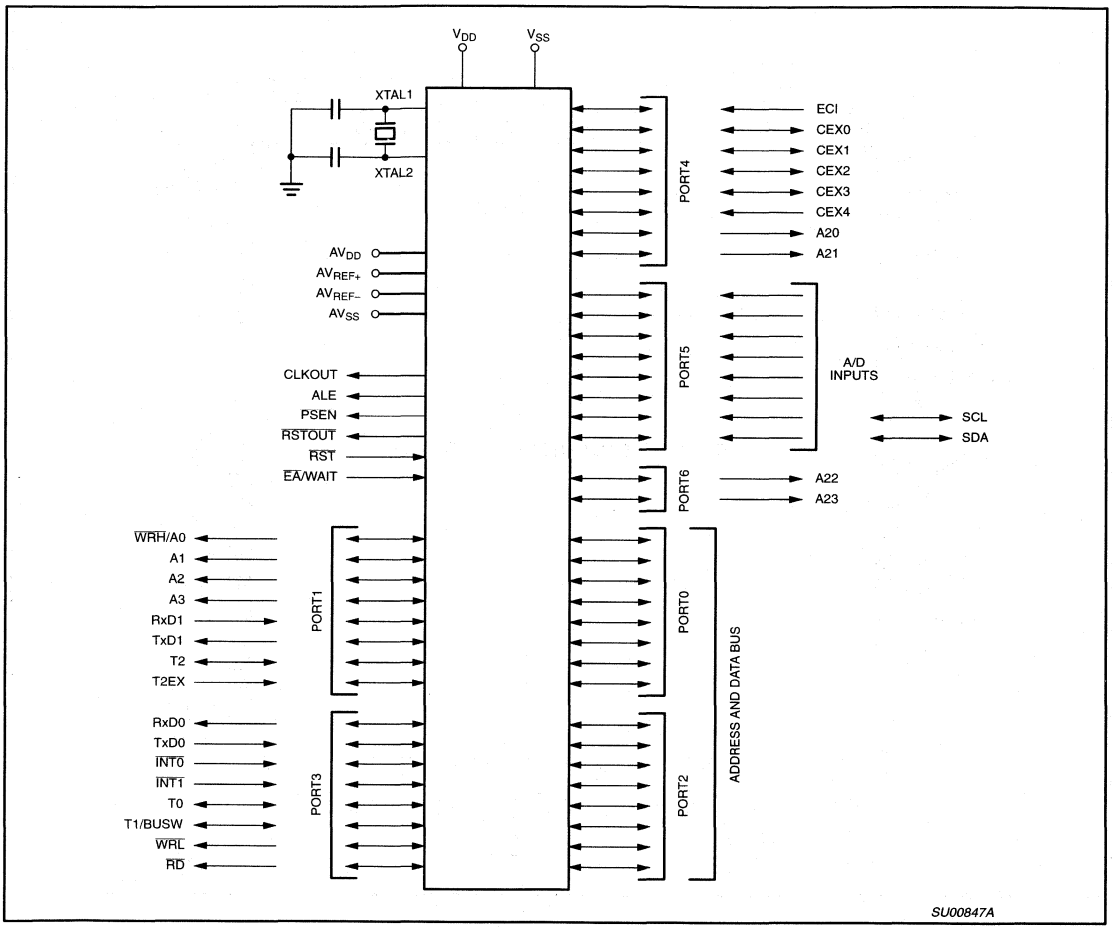
68-pin PLCC package



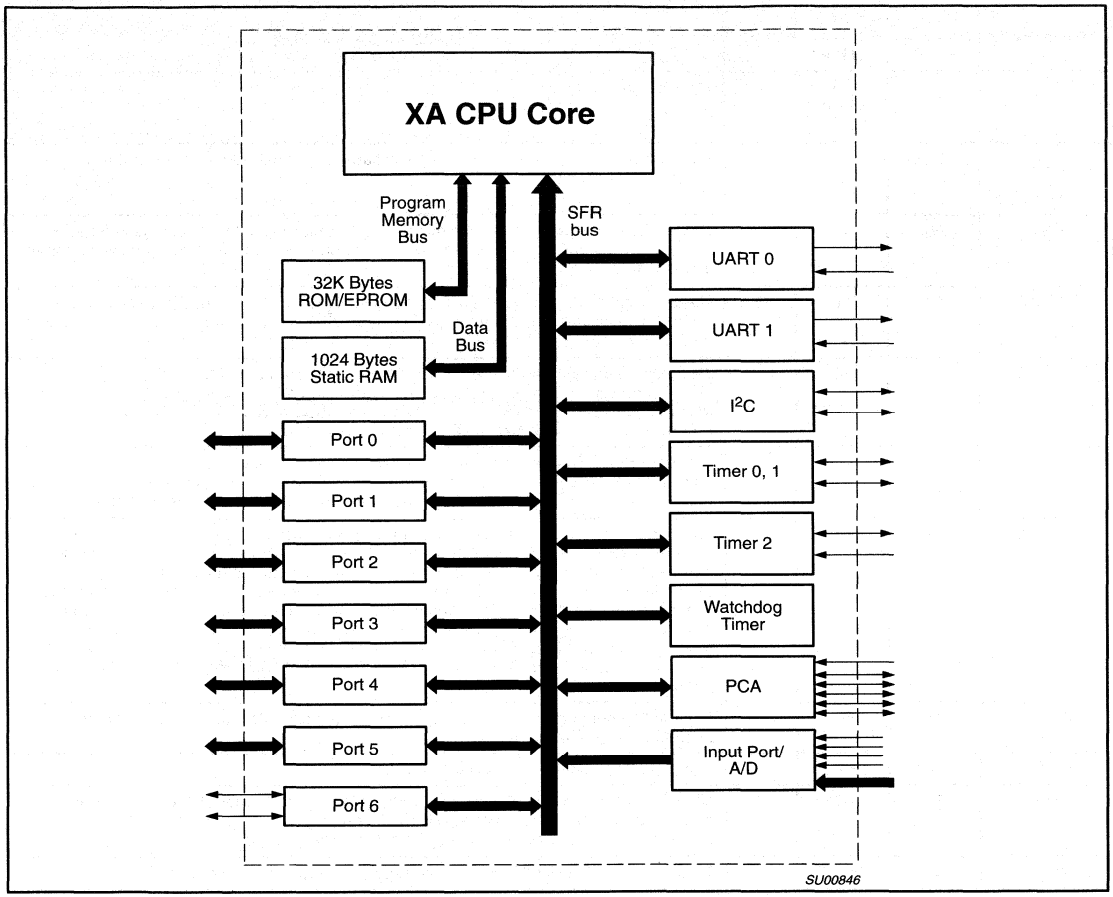
80-pin LQFP package



LOGIC SYMBOL



BLOCK DIAGRAM



PIN DESCRIPTIONS

MNEMONIC	PIN NUMBER		TYPE	NAME AND FUNCTION
	PLCC	LQFP		
V_{SS}	1, 20, 55	12, 13, 53, 54, 69, 70	I	**Ground:** 0V reference.
V_{DD}	2, 21, 54	14, 15, 51, 52, 71, 72	I	**Power Supply:** This is the power supply voltage for normal, idle, and power down operation.
$\overline{RST}$	50	47	I	**Reset:** A low on this pin resets the microcontroller, causing I/O ports and peripherals to take on their default states, and the processor to begin execution at the address contained in the reset vector.
$\overline{RSTOUT}$	19	11	O	**Reset Output:** This pin outputs a low whenever the XA-S3 processor is reset for any reason. This includes an external reset via the $\overline{RST}$ pin, watchdog reset, and the $\overline{RESET}$ instruction.
ALE/$\overline{PROG}$	47	44	I/O	**Address Latch Enable/Program Pulse:** A high output on the ALE pin signals external circuitry to latch the address portion of the multiplexed address/data bus. A pulse on ALE occurs only when it is needed in order to process a bus cycle. During EPROM programming, this pin is used as the program pulse input.
$\overline{PSEN}$	48	45	O	**Program Store Enable:** The read strobe for external program memory. When the microcontroller accesses external program memory, $\overline{PSEN}$ is driven low in order to enable memory devices. $\overline{PSEN}$ is only active when external code accesses are performed.
$\overline{EA}$/WAIT/V_{PP}	22	16	I	**External Access/Bus Wait/Programming Supply Voltage:** The $\overline{EA}$ input determines whether the internal program memory of the microcontroller is used for code execution. The value on the $\overline{EA}$ pin is latched as the external reset input is released and applies during later execution. When latched as a 0, external program memory is used exclusively. When latched as a 1, internal program memory will be used up to its limit, and external program memory used above that point. After reset is released, this pin takes on the function of bus WAIT input. If WAIT is asserted high during an external bus access, that cycle will be extended until WAIT is released. During EPROM programming, this pin is also the programming supply voltage input.
XTAL1	68	68	I	**Crystal 1:** Input to the inverting amplifier used in the oscillator circuit and input to the internal clock generator circuits.
XTAL2	67	67	I	**Crystal 2:** Output from the oscillator amplifier.
CLKOUT	49	46	O	**Clock Output:** This pin outputs a buffered version of the internal CPU clock. The clock output may be used in conjunction with the external bus to synchronize WAIT state generators, etc. The clock output may be disabled by software.
AV_{DD}	33	28, 29	I	**Analog Power Supply:** Positive power supply input for the A/D converter.
AV_{SS}	34	30, 31	I	**Analog Ground.**
AV_{REF+}	32	27	I	**A/D Positive Reference Voltage:** High end reference for the A/D converter.
AV_{REF-}	31	26	I	**A/D Negative Reference Voltage:** Low end reference for the A/D converter.
P0.0 – P0.7	45, 46, 51–53, 56–58	42, 43, 48–50, 55–57	I/O	**Port 0:** Port 0 is an 8-bit I/O port with a user-configurable output type. Port 0 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 0 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used, Port 0 becomes the multiplexed low data/instruction byte and address lines 4 through 11.

MNEMONIC	PIN NUMBER		TYPE	NAME AND FUNCTION
	PLCC	LQFP		
P1.0 – P1.7	35–42	32–39	I/O	**Port 1:** Port 1 is an 8-bit I/O port with a user-configurable output type. Port 1 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 1 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details.
				Port 1 also provides various special functions as described below:
	35	32	O	**A0/$\overline{\text{WRH}}$ (P1.0)** Address bit 0 of the external address bus when the eternal data bus is configured for an 8-bit width. When the external data bus is configured for a 16-bit width, this pin becomes the high byte write strobe.
	36	33	O	**A1 (P1.1):** Address bit 1 of the external address bus.
	37	34	O	**A2 (P1.2):** Address bit 2 of the external address bus.
	38	35	O	**A3 (P1.3):** Address bit 3 of the external address bus.
	39	36	I	**RxD1 (P1.4):** Serial port 1 receiver input.
	40	37	O	**TxD1 (P1.5):** Serial port 1 transmitter output.
	41	38	I/O	**T2 (P1.6):** Timer/counter 2 external count input or overflow output.
	42	39	O	**T2EX (P1.7):** Timer/counter 2 reload/capture/direction control.
P2.0 – P2.7	59–66	58, 59, 61–66	I/O	**Port 2:** Port 2 is an 8-bit I/O port with a user-configurable output type. Port 2 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 2 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used in 16-bit mode, Port 2 becomes the multiplexed high data/instruction byte and address lines 12 through 19. When the external data/address bus is used in 8-bit mode, the number of address lines that appear on Port 2 is user programmable in groups of 4 bits.
P3.0 – P3.7	11–18	3–10	I/O	**Port 3:** Port 3 is an 8-bit I/O port with a user-configurable output type. Port 3 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 3 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details.
				Port 3 also provides the various special functions as described below:
	11	3	I	**RxD0 (P3.0):** Receiver input for serial port 0.
	12	4	O	**TxD0 (P3.1):** Transmitter output for serial port 0.
	13	5	I	**$\overline{\text{INT0}}$ (P3.2):** External interrupt 0 input.
	14	6	I	**$\overline{\text{INT1}}$ (P3.3):** External interrupt 1 input.
	15	7	I/O	**T0 (P3.4):** Timer/counter 0 external count input or overflow output.
	16	8	I/O	**T1 / BUSW (P3.5):** Timer/counter 1 external count input or overflow output. The value on this pin is latched as an external chip reset is completed and defines the default external data bus width.
	17	9	O	**$\overline{\text{WRL}}$ (P3.6):** External data memory low byte write strobe.
	18	10	O	**$\overline{\text{RD}}$ (P3.7):** External data memory read strobe.
P4.0 – P4.7	3–10	73–79, 2	I/O	**Port 4:** Port 4 is an 8-bit I/O port with a user-configurable output type. Port 4 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of Port 4 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details.
				Port 4 also provides various special functions as described below:
	3	73	I	**ECI (P4.0):** PCA External clock input.
	4	74	I/O	**CEX0 (P4.1):** Capture/compare external I/O for PCA module 0.
	5	75	I/O	**CEX1 (P4.2):** Capture/compare external I/O for PCA module 1.
	6	76	I/O	**CEX2 (P4.3):** Capture/compare external I/O for PCA module 2.
	7	77	I/O	**CEX3 (P4.4):** Capture/compare external I/O for PCA module 3.
	8	78	I/O	**CEX4 (P4.5):** Capture/compare external I/O for PCA module 4.
	9	79	O	**A20 (P4.6):** Address bit 20 of the external address bus.
	10	2	O	**A21 (P4.7):** Address bit 21 of the external address bus.

MNEMONIC	PIN NUMBER		TYPE	NAME AND FUNCTION
	PLCC	LQFP		
P5.0 – P5.7	23–30	17–20, 22–25	I/O	**Port 5:** Port 5 is an 8-bit I/O port with a user-configurable output type. Port 5 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of Port 5 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 5 also provides various special functions as described below. Port 5 pins used as A/D inputs must be configured by the user to the high impedance mode.
	23	17	I	**AD0 (P5.0):** A/D channel 0 input.
	24	18	I	**AD1 (P5.1):** A/D channel 1 input.
	25	19	I	**AD2 (P5.2):** A/D channel 2 input.
	26	20	I	**AD3 (P5.3):** A/D channel 3 input.
	27	22	I	**AD4 (P5.4):** A/D channel 4 input.
	28	23	I	**AD5 (P5.5):** A/D channel 5 input.
	29	24	I/O	**AD6/SCL (P5.6):** A/D channel 6 input. I²C serial clock input/output.
	30	25	I/O	**AD7/SDA (P5.7):** A/D channel 7 input. I²C serial data input/output.
P6.0 – P6.7	43, 44	40, 41	I/O	**Port 6:** Port 6 is a 2-bit I/O port with a user-configurable output type. Port 6 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of Port 6 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 6 also provides special functions as described below:
	43	40	O	**A22 (P6.0):** Address bit 22 of the external address bus.
	44	41	O	**A23 (P6.1):** Address bit 23 of the external address bus.

Table 1. Special Function Registers

NAME	DESCRIPTION	SFR Address	BIT FUNCTIONS AND ADDRESSES MSB							LSB	Reset Value
			3F7	3F6	3F5	3F4	3F3	3F2	3F1	3F0	
ADCON#*	A/D control register	43E	–	–	–	–	–	ADMOD	ADSST	ADINT	00h
			3FF	3FE	3FD	3FC	3FB	3FA	3F9	3F8	
ADCS#*	A/D channel select register	43F	ADCS7	ADCS6	ADCS5	ADCS4	ADCS3	ADCS2	ADCS1	ADCS0	00h
ADCFG#	A/D timing configuration	4B9	–	–	–	–	A/D Timing Configuration				0Fh
ADRSH0#	A/D high byte result, channel 0	4B0									xx
ADRSH1#	A/D high byte result, channel 1	4B1									xx
ADRSH2#	A/D high byte result, channel 2	4B2									xx
ADRSH3#	A/D high byte result, channel 3	4B3									xx
ADRSH4#	A/D high byte result, channel 4	4B4									xx
ADRSH5#	A/D high byte result, channel 5	4B5									xx
ADRSH6#	A/D high byte result, channel 6	4B6									xx
ADRSH7#	A/D high byte result, channel 7	4B7									xx
BCR#	Bus configuration register	46A	–	–	CLKD	WAITD	BUSD	BC2	BC1	BC0	Note 1
BTRH	Bus timing register high byte	469	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0	FFh
BTRL	Bus timing register low byte	468	WM1	WM0	ALEW	–	CR1	CR0	CRA1	CRA0	EFh

NAME	DESCRIPTION	SFR Address	BIT FUNCTIONS AND ADDRESSES MSB							LSB	Reset Value
			2D7	2D6	2D5	2D4	2D3	2D2	2D1	2D0	
CCON#*	PCA counter control	41A	CF	CR	–	CCF4	CCF3	CCF2	CCF1	CCF0	00h
CMOD#	PCA mode control	490	–	WDTE	–	–	–	CPS1	CPS0	ECF	00h
CH#	PCA counter high byte	48B									00h
CL#	PCA counter low byte	48A									00h
CCAPM0#	PCA module 0 mode	491	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	00h
CCAPM1#	PCA module 1 mode	492	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	00h
CCAPM2#	PCA module 2 mode	493	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	00h
CCAPM3#	PCA module 3 mode	494	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	00h
CCAPM4#	PCA module 4 mode	495	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	00h
CCAP0H#	PCA module 0 capture high byte	497									xx
CCAP1H#	PCA module 1 capture high byte	499									xx
CCAP2H#	PCA module 2 capture high byte	49B									xx
CCAP3H#	PCA module 3 capture high byte	49D									xx
CCAP4H#	PCA module 4 capture high byte	49F									xx
CCAP0L#	PCA module 0 capture low byte	496									xx
CCAP1L#	PCA module 1 capture low byte	498									xx
CCAP2L#	PCA module 2 capture low byte	49A									xx
CCAP3L#	PCA module 3 capture low byte	49C									xx
CCAP4L#	PCA module 4 capture low byte	49E									xx
CS	Code segment	443									00h
DS	Data segment	441									00h
ES	Extra segment	442									00h
			367	366	365	364	363	362	361	360	
I2CON#*	I²C control register	42C	CR2	ENA	STA	STO	SI	AA	CR1	CR0	00h
I2STAT#	I²C status register	46C	I²C Status Code/Vector					0	0	0	F8h
I2DAT#	I²C data register	46D									xx
I2ADDR#	I²C address register	46E	I²C Slave Address							GC	00h
			33F	33E	33D	33C	33B	33A	339	338	
IEH*	Interrupt enable high byte	427	–	–	–	–	ETI1	ERI1	ETI0	ERI0	00h
			337	336	335	334	333	332	331	330	
IEL#*	Interrupt enable low byte	426	EA	EAD	EPC	ET2	ET1	EX1	ET0	EX0	00h
			377	376	375	374	373	372	371	370	
IELB#*	Interrupt enable B low byte	42E	–	–	EI2	EC4	EC3	EC2	EC1	EC0	00h
IPA0	Interrupt priority A0	4A0	PT0				PX0				00h
IPA1	Interrupt priority A1	4A1	PT1				PX1				00h
IPA2#	Interrupt priority A2	4A2	PPC				PT2				00h
IPA3#	Interrupt priority A3	4A3	–				PAD				00h
IPA4	Interrupt priority A4	4A4	PTI0				PRI0				00h

NAME	DESCRIPTION	SFR Address	BIT FUNCTIONS AND ADDRESSES MSB							LSB	Reset Value
IPA5	Interrupt priority A5	4A5	PTI1				PRI1				00h
IPB0#	Interrupt priority B0	4A8	PC1				PC0				00h
IPB1#	Interrupt priority B1	4A9	PC3				PC2				00h
IPB2#	Interrupt priority B2	4AA	PI2				PC4				00h
			387	386	385	384	383	382	381	380	
P0*	Port 0	430	A11D7	A10D6	A9D5	A8D4	A7D3	A6D2	A5D1	A4D0	FFh
			38F	38E	38D	38C	38B	38A	389	388	
P1*	Port 1	431	T2EX	T2	TxD1	RxD1	A3	A2	A1	A0/WRH	FFh
			397	396	395	394	393	392	391	390	
P2*	Port 2	432	A19D15	A18D14	A17D13	A16D12	A15D11	A14D10	A13D9	A12D8	FFh
			39F	39E	39D	39C	39B	39A	399	398	
P3*	Port 3	433	RD	WRL	T1	T0	INT1	INT0	TxD0	RxD0	FFh
			3A7	3A6	3A5	3A4	3A3	3A2	3A1	3A0	
P4#*	Port 4	434	A21	A20	CEX4	CEX3	CEX2	CEX1	CEX0	ECI	FFh
			3AF	3AE	3AD	3AC	3AB	3AA	3A9	3A8	
P5#*	Port 5	435	AD7/SDA	AD6/SCL	AD5	AD4	AD3	AD2	AD1	AD0	FFh
									3B1	3B0	
P6#*	Port 6	436	–	–	–	–	–	–	A23	A22	FFh
P0CFGA	Port 0 configuration A	470									Note 5
P1CFGA	Port 1 configuration A	471									Note 5
P2CFGA	Port 2 configuration A	472									Note 5
P3CFGA	Port 3 configuration A	473									Note 5
P4CFGA#	Port 4 configuration A	474									Note 5
P5CFGA#	Port 5 configuration A	475									Note 5
P6CFGA#	Port 6 configuration A	476	–	–	–	–	–	–			Note 5
P0CFGB	Port 0 configuration B	4F0									Note 5
P1CFGB	Port 1 configuration B	4F1									Note 5
P2CFGB	Port 2 configuration B	4F2									Note 5
P3CFGB	Port 3 configuration B	4F3									Note 5
P4CFGB#	Port 4 configuration B	4F4									Note 5
P5CFGB#	Port 5 configuration B	4F5									Note 5
P6CFGB#	Port 6 configuration B	4F6	–	–	–	–	–	–			Note 5
			227	226	225	224	223	222	221	220	
PCON*	Power control register	404	–	–	–	–	–	–	PD	IDL	00h
			20F	20E	20D	20C	20B	20A	209	208	
PSWH*	Program status word (high byte)	401	SM	TM	RS1	RS0	IM3	IM2	IM1	IM0	Note 2
			207	206	205	204	203	202	201	200	
PSWL*	Program status word (low byte)	400	C	AC	–	–	–	V	N	Z	Note 2
			217	216	215	214	213	212	211	210	
PSW51*	80C51 compatible PSW	402	C	AC	F0	RS1	RS0	V	F1	P	Note 3

NAME	DESCRIPTION	SFR Address	BIT FUNCTIONS AND ADDRESSES MSB							LSB	Reset Value
RSTSRC#	Reset source register	463	–	–	–	–	–	R_WD	R_CMD	R_EXT	Note 7
RTH0	Timer 0 reload register, high byte	455									00h
RTH1	Timer 1 reload register, high byte	457									00h
RTL0	Timer 0 reload register, low byte	454									00h
RTL1	Timer 1 reload register, low byte	456									00h
			307	306	305	304	303	302	301	300	
S0CON*	Serial port 0 control register	420	SM0_0	SM1_0	SM2_0	REN_0	TB8_0	RB8_0	TI_0	RI_0	00h
			30F	30E	30D	30C	30B	30A	309	308	
S0STAT#*	Serial port 0 extended status	421	–	–	–	ERR0	FE0	BR0	OE0	STINT0	00h
S0BUF	Serial port 0 data buffer register	460									xx
S0ADDR	Serial port 0 address register	461									00h
S0ADEN	Serial port 0 address enable	462									00h
			327	326	325	324	323	322	321	320	
S1CON*	Serial port 1 control register	424	SM0_1	SM1_1	SM2_1	REN_1	TB8_1	RB8_1	TI_1	RI_1	00H
			32F	32E	32D	32C	32B	32A	329	328	
S1STAT#*	Serial port 1 extended status	425	–	–	–	ERR1	FE1	BR1	OE1	STINT1	00h
S1BUF	Serial port 1 data buffer register	464									xx
S1ADDR	Serial port 1 address register	465									00h
S1ADEN	Serial port 1 address enable	466									00h
SCR	System configuration register	440	–	–	–	–	PT1	PT0	CM	PZ	00h
			21F	21E	21D	21C	21B	21A	219	218	
SSEL*	Segment selection register	403	ESWEN	R6SEG	R5SEG	R4SEG	R3SEG	R2SEG	R1SEG	R0SEG	00h
SWE	Software interrupt enable	47A	–	SWE7	SWE6	SWE5	SWE4	SWE3	SWE2	SWE1	00h
			357	356	355	354	353	352	351	350	
SWR*	Software interrupt request	42A	–	SWR7	SWR6	SWR5	SWR4	SWR3	SWR2	SWR1	00h
			2C7	2C6	2C5	2C4	2C3	2C2	2C1	2C0	
T2CON*	Timer 2 control register	418	TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2	00h
			2CF	2CE	2CD	2CC	2CB	2CA	2C9	2C8	
T2MOD*	Timer 2 mode control	419	–	–	RCLK1	TCLK1	–	–	T2OE	DCEN	00h
TH2	Timer 2 high byte	459									00h
TL2	Timer 2 low byte	458									00h
T2CAPH	Timer 2 capture, high byte	45B									00h
T2CAPL	Timer 2 capture, low byte	45A									00h
			287	286	285	284	283	282	281	280	
TCON*	Timer 0 and 1 control register	410	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00h

NAME	DESCRIPTION	SFR Address	BIT FUNCTIONS AND ADDRESSES MSB							LSB	Reset Value
TH0	Timer 0 high byte	451									00h
TH1	Timer 1 high byte	453									00h
TL0	Timer 0 low byte	450									00h
TL1	Timer 1 low byte	452									00h
TMOD	Timer 0 and 1 mode control	45C	GATE	C/T	M1	M0	GATE	C/T	M1	M0	00h
			28F	28E	28D	28C	28B	28A	289	288	
TSTAT*	Timer 0 and 1 extended status	411	–	–	–	–	–	T1OE	–	T0OE	00h
			2FF	2FE	2FD	2FC	2FB	2FA	2F9	2F8	
WDCON*	Watchdog control register	41F	PEW2	PRE1	PRE0	–	–	WDRUN	WDTOF	–	Note 6
WDL	Watchdog timer reload	45F									00h
WFEED1	Watchdog feed 1	45D									xx
WFEED2	Watchdog feed 2	45E									xx

NOTES:

* SFRs are bit addressable.

\# SFRs are modified from or added to XA-G3 SFRs.

1. At reset, the BCR is loaded with the binary value 00000a11, where "a' is the value on the BUSW pin. This defaults the address bus size to 24 bits.
2. SFR is loaded from the reset vector.
3. All bits except F1, F0, and P are loaded from the reset vector. Those bits are all 0.
4. Unimplemented bits in SFRs are X (unknown) at all times. Ones should not be written to these bits since they may be used for other purposes in future XA derivatives. The reset value shown for these bits is 0.
5. Port configurations default to quasi-bidirectional when the XA begins execution from internal code memory after reset, based on the condition found on the $\overline{EA}$ pin. Thus, all PnCFGA registers will contain FF, and PnCFGB register will contain 00 when the XA begins execution using internal code memory. When the XA begins execution using external code memory, the default configuration for pins that are associated with the external bus will be push-pull. The PnCFGA and PnCFGB register contents will reflect this difference.
6. The WDCON reset value is E6 for a Watchdog reset, E4 for all other reset causes.
7. The RSTSRC register reflects the cause of the last XA-S3 reset. One bit will be set to 1, the others will be cleared to 0.
8. The XA guards writes to certain bits (typically interrupt flags) that may be altered directly by a peripheral function. This prevents loss of an interrupt or other status if a bit was written directly by a peripheral action during the time between the read and write portions of an instruction that performs a read-modify-write operation. Examples of such instructions are:

```
and     s0con,#$fb
clr     tr0
setb    ti_0
```

   XA-S3 SFR bits that are guarded in this manner are: ADINT (in ADCON); CF, CCF4, CCF3, CCF2, CCF1, and CCF0 (in CCON); SI (in I2CON); TI_0 and RI_0 (in S0CON); TI_1 and RI_1 (in S1CON); FE0, BR0, and OE0 (in S0STAT); FE1, BR1, and OE1 (in S1STAT); TF2 (in T2CON); TF1, TF0, IE1, and IE0 (in TCON); and WDTOF (in WDCON).
9. The XA-S3 implements an 8-bit SFR bus, as stated in *Chapter 8* of the *XA User Guide*. All SFR accesses must be 8-bit operations. Attempts to write 16 bits to an SFR will actually write only the lower 8 bits. Sixteen bit SFR reads will return undefined data in the upper byte.

FUNCTIONAL DESCRIPTION

Details of XA-S3 functions will be described in the following sections.

Analog to Digital converter

The XA-S3 has an 8-channel, 8-bit A/D converter with 8 sets of result registers, single scan and multiple scan operating modes. The A/D input range is limited to 0 to AV_{DD} (3.3V max.). The A/D inputs are on Port 5. Analog Power and Ground as well as AV_{REF+} and AV_{REF-} must be supplied in order for the A/D converter to be used. Prior to enabling the A/D converter or driving analog signals into the A/D inputs, the port configurations for the pins being used as A/D inputs must be set to the "off" (high impedance, input only) mode.

A/D timing can be adapted to the application clock frequency in order to provide the fastest possible conversion.

A/D converter operation is controlled through the ADCON (A/D Control) register, see Figure 1. Bits in ADCON start and stop the A/D, flag conversion completion, and select the converter operating modes.

A/D Conversion Modes

The A/D converter supports a single scan mode and a continuous scan mode. In either mode, one or more A/D channels may be converted. The ADCS register determines which channels are converted. If the corresponding bit in the ADCS register is set, that channel is selected for conversions, otherwise that channel is skipped. The ADCS register is detailed in Figure 2.

For any A/D conversion, the results are stored in ADRSHn, corresponding to the A/D channel just converted.

A/D conversions are begun by setting the A/D Start and STatus bit in ADCON. In the single scan mode, all of the channels selected by bits in the ADCS register will be converted once. The ADINT flag is set when the last channel is converted. In the continuous scan mode, the A/D converter continuously converts all A/D channels selected by bits in the ADCS register. The ADINT flag is set when all channels have been converted once.

The A/D converter can generate an interrupt when the ADINT flag is set. This will occur if the A/D interrupt is enabled (via the EAD bit in IEL), the interrupt system is enabled (via the EA bit in IEL), and the A/D interrupt priority (specified in IPA3 bits 3 to 0) is higher than the currently running code (PSW bits IM3 through IM0) and any other pending interrupt. ADINT must be cleared by software.

A/D Timing Configuration

The A/D sampling and conversion timing may be optimized for the particular oscillator frequency and input drive characteristics of the application. Because A/D operation is mostly dependent on real-time effects (charging time of sampling capacitors, settling time of the comparator, etc.), A/D conversion times are not necessarily much longer at slower clock frequencies. The A/D timing is controlled by the ADCFG register, as shown in Figure 3 and Table 2.

The primary effect of ADCFG settings is to adjust the A/D sample and hold time to be relatively constant over various clock frequencies. Two settings (value 6 and B) are provided to allow fast conversions with a lower external source driving the A/D inputs. These settings provide double the sample time at the same frequency. Of course, settings intended for lower frequencies may also be used at higher frequencies in order to increase the A/D sampling time, but this method has the side effect of significantly increasing A/D conversion times.

ADCON Address:43Eh
Bit Addressable
Reset Value: 00h

MSB							LSB
—	—	—	—	—	ADMOD	ADSST	ADINT

BIT	SYMBOL	FUNCTION
ADCON.7	—	Reserved for future use. Should not be set to 1 by user programs.
ADCON.6	—	Reserved for future use. Should not be set to 1 by user programs.
ADCON.5	—	Reserved for future use. Should not be set to 1 by user programs.
ADCON.4	—	Reserved for future use. Should not be set to 1 by user programs.
ADCON.3	—	Reserved for future use. Should not be set to 1 by user programs.
ADCON.2	ADMOD	A/D mode select. 1 = continuous scan of selected inputs after a start of the A/D. 0 = single scan of selected inputs after a start of the A/D.
ADCON.1	ADSST	A/D start and status. Setting this bit by software starts the A/D conversion of the selected A/D inputs. ADSST remains set as long as the A/D is in operation. In continuous conversion mode, ADSST will remain set unless the A/D is stopped by software. While ADSST is set, new start commands are ignored. An A/D conversion is progress may be aborted by software clearing ADSST.
ADCON.0	ADINT	A/D conversion complete/interrupt flag. This flag is set when all selected A/D channels are converted in either the single scan or continuous scan modes. Must be cleared by software.

SU00938A

Figure 1. A/D Control Register (ADCON)

ADCS Address:43Fh
Bit Addressable
Reset Value: 00h

MSB							LSB
ADCS7	ADCS6	ADCS5	ADCS4	ADCS3	ADCS2	ADCS1	ADCS0

BIT	SYMBOL	FUNCTION
ADCS.7	ADCS7	A/D channel 7 select bit.
ADCS.6	ADCS6	A/D channel 6 select bit.
ADCS.5	ADCS5	A/D channel 5 select bit.
ADCS.4	ADCS4	A/D channel 4 select bit.
ADCS.3	ADCS3	A/D channel 3 select bit.
ADCS.2	ADCS2	A/D channel 2 select bit.
ADCS.1	ADCS1	A/D channel 1 select bit.
ADCS.0	ADCS0	A/D channel 0 select bit.

SU00939

Figure 2. A/D Channel Select Register (ADCS)

ADCFG Address:4B9h
Not bit Addressable
Reset Value: 00h

MSB					LSB
—	—	—	—	A/D Timing Configuration	

BIT	SYMBOL	FUNCTION
ADCFG.7	—	Reserved for future use. Should not be set to 1 by user programs.
ADCFG.6	—	Reserved for future use. Should not be set to 1 by user programs.
ADCFG.5	—	Reserved for future use. Should not be set to 1 by user programs.
ADCFG.4	—	Reserved for future use. Should not be set to 1 by user programs.
ADCFG.3–0	ADCFG	A/D timing configuration (see text and table).

SU00940

Figure 3. A/D Timing Configuration Register (ADCFG)

Table 2. A/D Timing Configuration

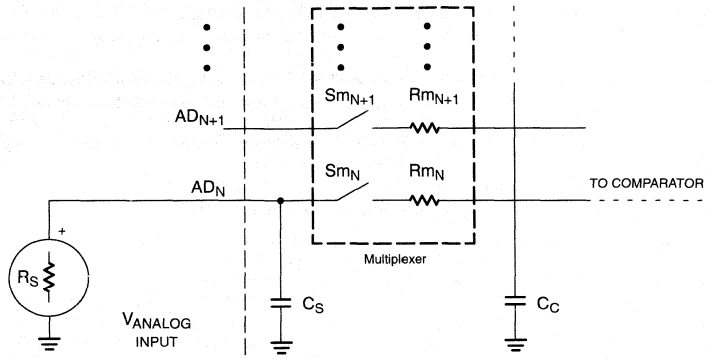
ADCFG.3–0	Max. Oscillator Frequency (MHz)	Conversion Time		Sampling Time (Osc. Clocks)
		Osc. Clocks	µsec at max. Osc.	
0h (0000)	6.66	70	11.11	4
1h (0001)	10	78	7.8	6
2h (0010)	11.11	82	7.38	8
3h (0011)	13.33	98	7.35	8
4h (0100)	16.66	102	6.12	10
5h (0101)	20	106	5.3	12
6h (0110)[1]	20	118	5.9	24
7h (0111)	22.2	102	4.95	14
8h (1000)	23.3	126	5.4	14
9h (1001)	26.6	130	4.88	16
Ah (1010)	30	134	4.46	18
Bh (1011)[1]	30	148	4.93	32
Ch (1100)	32	138	4.31	20
Dh (1101)	33.3	152	4.56	20
Eh (1110)	36.6	172	4.69	22
Fh (1111)	40	176	4.4	24

NOTE:
1. These settings provide additional A/D input sampling time, in order to allow accurate readings with a higher external source impedance.

A/D Inputs

In order to obtain accurate measurements with the A/D Converter, the source drive must be sufficient to adequately charge the sampling capacitor during the sampling time. Figure 4 shows the equivalent resistance and capacitance related to the A/D inputs.

A/D timing configurations indicated in Table 1 allow for full A/D accuracy (according to the A/D specifications) assuming a source resistance of less than or equal to 20kΩ. Larger source resistances may be accommodated by increasing the sampling time with a different A/D timing configuration.



R_m (multiplexer resistance) = 3 kΩ maximum
C_S (pin capacitance) = 10 pF maximum
C_C (sampling capacitor) = 2 pF maximum
R_S (source resistance) = Recommended less than 20kΩ for full specified accuracy. This allows time for the sampling capacitor (C_C) to fully charge while the multiplexer switch is closed. Please note that sampling causes the analog input to present a varying load to the pin.

SU00948

Figure 4. A/D Input: Equivalent Circuit

I2CON Address:42Ch
Bit Addressable
Reset Value: 00h

MSB LSB

CR2	ENA	STA	STO	SI	AA	CR1	CR0

BIT	SYMBOL	FUNCTION
I2CON.7	CR2	I²C Rate Control, with CR1 and CR0. See text and table.
I2CON.6	ENA	Enable I²C port. When ENA = 1, the I²C port is enabled.
I2CON.5	STA	Start flag. Setting STA to 1 causes the I²C interface to attempt to gain mastership of the bus by generating a Start condition.
I2CON.4	STO	Stop flag. Setting STO to 1 causes the I²C interface to attempt to generate a Stop condition.
I2CON.3	SI	Serial Interrupt. SI is set by the I²C hardware when a new I²C state is entered, indicating that software needs to respond. SI causes an I²C interrupt if enabled and of sufficient priority.
I2CON.2	AA	Assert Acknowledge. Setting AA to 1 causes the I²C hardware to automatically generate acknowledge pulses for various conditions (see text).
I2CON.1	CR1	I²C Rate Control, with CR2 and CR0. See text and table.
I2CON.0	CR0	I²C Rate Control, with CR2 and CR1. See text and table.

SU00941

Figure 5. I²C Control Register (I2CON)

I²C Interface

The I²C interface on the XA-S3 is identical to the standard byte-style I²C interface found on devices such as the 8xC552 except for the rate selection. ***The I²C interface conforms to the 100 kHz I²C specification, but may be used at rates up to 400 kHz (non-conforming).***

Important: Before the I²C interface may be used, the port pins P5.6 and 5.7, which correspond to the I²C functions SCL and SDA respectively, must be set to the open drain mode.

The processor interfaces to the I²C logic via the following four special function registers: I2CON (I²C control register), I2STA (I²C status register), I2DAT (I²C data register), and I2ADR (I²C slave address register). The I²C control logic interfaces to the external I²C bus via two port 5 pins: P5.6/SCL (serial clock line) and P5.7/SDA (serial data line).

The Control Register, I2CON

This register is shown in Figure 5. Two bits are affected by the I²C hardware: the SI bit is set when a serial interrupt is requested, and the STO bit is cleared when a STOP condition is present on the I²C bus. The STO bit is also cleared when ENA = "0".

ENA, the I²C Enable Bit

ENA = 0: When ENA is "0", the SDA and SCL outputs are not driven. SDA and SCL input signals are ignored, SIO1 is in the "not addressed" slave state, and the STO bit in I2CON is forced to "0". No other bits are affected. P5.6 and P5.7 may be used as open drain I/O ports.

ENA = 1: When ENA is "1", SIO1 is enabled. The P5.6 and P5.7 port latches must be set to logic 1.

ENA should not be used to temporarily release the I²C-bus since, when ENA is reset, the I²C-bus status is lost. The AA flag should be used instead (see description of the AA flag in the following text).

In the following text, it is assumed the ENA = "1".

STA, the START flag

STA = 1: When the STA bit is set to enter a master mode, the I²C hardware checks the status of the I²C bus and generates a START condition if the bus is free. If the bus is not free, the I²C interface waits for a STOP condition (which will free the bus) and generates a START condition after a delay of a half clock period of the internal serial clock generator.

If STA is set while the I²C interface is already in a master mode and one or more bytes are transmitted or received, the hardware transmits a repeated START condition. STA may be set at any time. STA may also be set when the I²C interface is an addressed slave.

STA = 0: When the STA bit is reset, no START condition or repeated START condition will be generated.

STO, the STOP flag

STO = 1: When the STO bit is set while the I²C interface is in a master mode, a STOP condition is transmitted to the I²C bus. When the STOP condition is detected on the bus, the hardware clears the STO flag. In a slave mode, the STO flag may be set to recover from an error condition. In this case, no STOP condition is transmitted to the I²C bus. However, the hardware behaves as if a STOP condition has been received and switches to the defined "not addressed" slave receiver mode. The STO flag is automatically cleared by hardware.

If the STA and STO bits are both set, then a STOP condition is transmitted to the I²C bus if the interface is in a master mode (in a slave mode, the hardware generates an internal STOP condition which is not transmitted). The I²C interface then transmits a START condition.

STO = 0: When the STO bit is reset, no STOP condition will be generated.

SI, the Serial Interrupt flag

SI = 1: When the SI flag is set, and the EA (interrupt system enable) and EI2 (I²C interrupt enable) bits are also set, an I²C interrupt is requested. SI is set by hardware when one of 25 of the 26 possible I²C interface states is entered. The only state that does not cause SI to be set is state F8H, which indicates that no relevant state information is available.

While SI is set, the low period of the serial clock on the SCL line is stretched, and the serial transfer is suspended. A high level on the SCL line is unaffected by the serial interrupt flag. SI must be reset by software.

SI = 0: When the SI flag is reset, no serial interrupt is requested, and there is no stretching of the serial clock on the SCL line.

AA, the Assert Acknowledge flag

AA = 1: If the AA flag is set, an acknowledge (low level to SDA) will be returned during the acknowledge clock pulse on the SCL line when:

- The "own slave address" has been received.
- The general call address has been received while the general call bit (GC) in I2ADR is set.
- A data byte has been received while the I²C interface is in the master receiver mode.
- A data byte has been received while the I²C interface is in the addressed slave receiver mode.

AA = 0: If the AA flag is reset, a not acknowledge (high level to SDA) will be returned during the acknowledge clock pulse on the SCL line when:

- A data byte has been received while the I²C interface is in the master receiver mode.
- A data byte has been received while the I²C interface is in the addressed slave receiver mode.

When the I²C interface is in the addressed slave transmitter mode, state C8H will be entered after the last serial data byte is transmitted. When SI is cleared, the I²C interface leaves state C8H, enters the not addressed slave receiver mode, and the SDA line remains at a high level. In state C8H, the AA flag can be set again for future address recognition.

When the I²C interface is in the not addressed slave mode, its own slave address and the general call address are ignored. Consequently, no acknowledge is returned, and a serial interrupt is not requested. Thus, the hardware can be temporarily released from the I²C bus while the bus status is monitored. While the hardware is released from the bus, START and STOP conditions are detected, and serial data is shifted in. Address recognition can be resumed at any time by setting the AA flag. If the AA flag is set when the part's own slave address or the general call address has been partly received, the address will be recognized at the end of the byte transmission.

CR0, CR1, and CR2, the Clock Rate Bits

These three bits determine the serial clock frequency when the I²C interface is in a master mode. An I²C rate of 100kHz or lower is typical and can be derived from many oscillator frequencies. The various serial rates are shown in Table 3. A variable bit rate may also be used if Timer 1 is not required for any other purpose while the I²C hardware is in a master mode. The frequencies shown in Table 3 are unimportant when the I²C hardware is in a slave mode. In the slave modes, the hardware will automatically synchronize with the incoming clock frequency.

The I²C Status Register, I2STA

I2STA is an 8-bit read-only special function register. The three least significant bits are always zero. The five most significant bits contain the status code. There are 26 possible status codes. When I2STA contains F8H, no relevant state information is available and no serial interrupt is requested. All other I2STA values correspond to defined hardware interface states. When each of these states is entered, a serial interrupt is requested (SI = "1").

NOTE: A detailed I²C interface description and usage information, including example driver code, will be provided in a separate document.

Table 3. I²C Rate Control

Frequency Select (CR2, CR1, CR0)	Clock Divisor	Example I²C Rates at Specific Oscillator Frequencies					
		8 MHz	12 MHz	16 MHz	20 MHz	24 MHz	30 MHz
0h (0000)	20	(400)[1]	–	–	–	–	–
1h (0001)	40	(200)[1]	(300)[1]	(400)[1]	–	–	–
2h (0010)	68	(116.65)[1]	(176.46)[1]	(235.29)[1]	(294.12)[1]	(352.94)[1]	–
3h (0011)	88	90.91	(136.36)[1]	(181.82)[1]	(227.27)[1]	(272.73)[1]	(340.91)[1]
4h (0100)	160	50	75	100	(125)[1]	(150)[1]	(187.5)[1]
5h (0101)	272	29.41	44.12	58.82	73.53	88.24	(110.29)[1]
6h (0110)	352	22.73	34.09	45.45	56.82	68.18	85.23
7h (0111)	(Timer 1)[2]	(Timer 1)[2]	(Timer 1)[2]	(Timer 1)[2]	(Timer 1)[2]	(Timer 1)[2]	(Timer 1)[2]

NOTES:
1. The XA-S3 I²C interface does not conform to the 400kHz I²C specification (which applies to rates greater than 100kHz) in all details, but may be used with care where higher rates are required by the application.
2. The timer 1 overflow is used to clock the I²C interface. The resulting bit rate is 1/2 of the timer overflow rate.

XA-S3 Timer/Counters

The XA-S3 has three general purpose counter/timers, two of which may also be used as baud rate generators for either or both of the UARTs.

Timer 0 and 1
These are identical to the standard XA-G3 timer 0 and 1.

Timer 2
This is identical to the standard XA-G3 timer 2.

PCA

This is a standard 80C51FC-style PCA counter/timer. The XA uses TCLK (the global peripheral clock which is Osc/4, Osc/16, or Osc/64), Timer 0 overflow, and External (ECI pin). When the ECI input is used, the falling edge clocks the PCA counter. The maximum rate for the counter in this mode on the XA is Osc/4. Each PCA module has its own interrupt (in addition to the standard global PCA interrupt).

Watchdog Timer

This is a standard XA-G3 watchdog timer. This watchdog timer always comes up running at reset. The watchdog acts the same on EPROM, ROM, and ROMless parts, as in the XA-G3.

UARTs

Standard XA-G3 UART0 and UART1 with double buffered transmit register. A flag has been added to SnSTAT that is set if any of the status flags (BRn, FEn, or OEn) is set for the corresponding UART channel. This allows polling for UART errors quickly at the interrupt service routine. Baud rate sources may be timer 1 or timer 2.

Clocking / Baud Rate Generation

Same as for the XA-G3.

I/O Port Output Configuration

Port output configurations are the same as for the XA-G3: open drain, quasi-bidirectional, push-pull, and off.

External Bus

The external bus operates in the same manner as the XA-G3, but all 24 address lines are brought out to the outside world. This allows for a maximum of 16 Mbytes of code memory and 16 Mbytes of data memory.

Clock Output

The CLKOUT pin allows easier external bus interfacing in some situations. This output reflects the X1 clock input to the XA, but is delayed to match the external bus outputs and strobes. The default is for CLKOUT to be on at reset, but it may be turned off via the CLKD bit that has been added to the BCR register.

Reset

Active low reset input, the same as the XA-G3.

The associated $\overline{\text{RSTOUT}}$ pin provides an external indication via an active low open drain output when an internal reset occurs. The $\overline{\text{RSTOUT}}$ pin will be driven low when the $\overline{\text{RST}}$ pin is driven low, when a Watchdog reset occurs or the RESET instruction is executed. This signal may be used to inform other devices in a system that the XA-S3 has been reset.

The latched values of EA and BUSW are NOT automatically updated when an internal reset occurs. $\overline{\text{RSTOUT}}$ may be used to apply an external reset to the XA-S3 in order to update the previously latched $\overline{\text{EA}}$ and BUSW values. However, since $\overline{\text{RSTOUT}}$ reflects ALL reset sources, it cannot simply be fed back into the $\overline{\text{RST}}$ pin without other logic.

The reset source identification register (RSTSRC) indicates the cause of the most recent XA reset. The cause may have been an externally applied reset signal, execution of the RESET instruction, or a Watchdog reset. Figure 6 shows the fields in the RSTSRC register.

Power Reduction Modes

The XA-S3 supports Idle and Power Down modes of power reduction. The idle mode leaves some peripherals running in order to allow them to activate the processor when an interrupt is generated. The power down mode stops the oscillator in order to absolutely minimize power. The processor can be made to exit power down mode via a reset or one of the external interrupt inputs (INT0 or INT1). This will occur if the interrupt is enabled and its priority is higher than that defined by IM3 through IM0. In power down mode, the power supply voltage may be reduced to the RAM keep-alive voltage V_{RAM}. This retains the RAM, register, and SFR contents at the point where power down mode was entered. V_{DD} must be raised to within the operating range before power down mode is exited.

RSTSRC Address:463h
Not bit Addressable
Reset Value: see below

MSB							LSB
—	—	—	—	—	R_WD	R_CMD	R_EXT

BIT	SYMBOL	FUNCTION
RSTSRC.7	—	Reserved for future use. Should not be set to 1 by user programs.
RSTSRC.6	—	Reserved for future use. Should not be set to 1 by user programs.
RSTSRC.5	—	Reserved for future use. Should not be set to 1 by user programs.
RSTSRC.4	—	Reserved for future use. Should not be set to 1 by user programs.
RSTSRC.3	—	Reserved for future use. Should not be set to 1 by user programs.
RSTSRC.2	R_WD	Indicates that the last reset was caused by a watchdog timer overflow.
RSTSRC.1	R_CMD	Indicates that the last reset was caused by execution of the RESET instruction.
RSTSRC.0	R_EXT	Indicates that the last reset was caused by the external $\overline{\text{RST}}$ input.

SU00942

Figure 6. Reset source register (RSTSRC)

INTERRUPTS

XA-S3 interrupt sources include the following:

- External interrupts 0 and 1 (2)
- Timer 0, 1, and 2 interrupts (3)
- PCA: 1 global and 5 channel interrupts (6)
- A/D interrupt (1)
- UART 0 transmitter and receiver interrupts (2)
- UART 1 transmitter and receiver interrupts (2)
- I²C interrupt (1)
- Software interrupts (7)

There are a total of 17 **hardware** interrupt sources, enable bits, priority bit sets, etc.

EXCEPTION/TRAPS PRECEDENCE

DESCRIPTION	VECTOR ADDRESS	ARBITRATION RANKING
Reset (h/w, watchdog, s/w)	0000–0003	0 (High)
Breakpoint	0004–0007	1
Trace	0008–000B	1
Stack Overflow	000C–000F	1
Divide by 0	0010–0013	1
User RETI	0014–0017	1
TRAP 0–15 (software)	0040–007F	1

EVENT INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY	ARBITRATION RANKING
External Interrupt 0	IE0	0080–0083	EX0	IPA0.3–0 (PX0)	2
Timer 0 Interrupt	TF0	0084–0087	ET0	IPA0.7–4 (PT0)	3
External Interrupt 1	IE1	0088–008B	EX1	IPA1.3–0 (PX1)	4
Timer 1 Interrupt	TF1	008C–008F	ET1	IPA1.7–4 (PT1)	5
Timer 2 Interrupt	TF2 (EXF2)	0090–0093	ET2	IPA2.3–0 (PT2)	6
PCA Interrupt	CCF0–CCF4, CF	0094–0097	EPC	IPA2.7–4 (PPC)	7
A/D Interrupt	ADINT	0098–009B	EAD	IPA3.3–0 (PAD)	8
Serial Port 0 Rx	RI_0	00A0–00A3	ERI0	IPA4.3–0 (PRI0)	9
Serial Port 0 Tx	TI_0	00A4–00A7	ETI0	IPA4.7–4 (PTI0)	10
Serial Port 1 Rx	RI_1	00A8–00AB	ERI1	IPA5.3–0 (PRI1)	11
Serial Port 1 Tx	TI_1	00AC–00AF	ETI1	IPA5.7–4 (PTI1)	12
PCA channel 0	CCF0	00C0–00C3	EC0	IPB0.3–0 (PC0)	17
PCA channel 1	CCF1	00C4–00C7	EC1	IPB0.7–4 (PC1)	18
PCA channel 2	CCF2	00C8–00CB	EC2	IPB1.3–0 (PC2)	19
PCA channel 3	CCF3	00CC–00CF	EC3	IPB1.7–4 (PC3)	20
PCA channel 4	CCF4	00D0–00D3	EC4	IPB2.3–0 (PC4)	21
I²C Interrupt	SI	00D4–00D7	EI2	IPB2.7–4 (PI2)	22

SOFTWARE INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY
Software Interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software Interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software Interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software Interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software Interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software Interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software Interrupt 7	SWR7	0118–011B	SWE7	(fixed at 7)

ABSOLUTE MAXIMUM RATINGS

PARAMETER	RATING	UNIT
Operating temperature under bias	–55 to +125	°C
Storage temperature range	–65 to +150	°C
Voltage on $\overline{EA}/V_{PP}$ pin to V_{SS}	0 to +13.0	V
Voltage on any other pin to V_{SS}	–0.5 to V_{DD}+0.5V	V
Maximum I_{OL} per I/O pin	15	mA
Power dissipation (based on package heat transfer, not device power consumption)	1.5	W

DC ELECTRICAL CHARACTERISTICS

V_{DD} = 2.7V to 5.5V, unless otherwise specified.
T_{amb} = 0 to +70°C for commercial unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP	MAX	
I_{DD}	Power supply current, operating	5.0V, 30MHz			100	mA
I_{ID}	Power supply current, Idle mode	5.0V, 30MHz			25	mA
I_{PD}	Power supply current, Power Down mode	5.0V, 3.0V		5	50	µA
V_{RAM}	RAM keep-alive voltage		1.5			V
V_{IL}	Input low voltage		–0.5		0.22 V_{DD}	V
V_{IH}	Input high voltage, except XTAL1, $\overline{RST}$	V_{DD} = 5.0V	2.2			V
		V_{DD} = 3.0V	2.0			V
V_{IH1}	Input high voltage to XTAL1, $\overline{RST}$	For both 3.0V and 5.0V	0.7 V_{DD}			V
V_{OL}	Output low voltage, all ports, ALE, $\overline{PSEN}$[4]	I_{OL} = 3.2mA, V_{DD} = 5.0V			0.5	V
		I_{OL} = 1.0mA, V_{DD} = 3.0V			0.4	V
V_{OH1}	Output high voltage, all ports, ALE, $\overline{PSEN}$[2]	I_{OH} = –100µA, V_{DD} = 4.5V	2.4			V
		I_{OH} = –30µA, V_{DD} = 2.7V	2.0			V
V_{OH2}	Output high voltage, all ports ALE, $\overline{PSEN}$[3]	I_{OH} = –3.2mA, V_{DD} = 4.5V	2.4			V
		I_{OH} = –1.0mA, V_{DD} = 2.7V	2.2			V
C_{IO}	Input/Output pin capacitance[1]				15	pF
I_{IL}	Logical 0 input current, all ports[7]	V_{IN} = 0.45V			–50	µA
I_{LI}	Input leakage current, all ports[6]	V_{IN} = V_{IL} or V_{IH}			±10	µA
I_{TL}	Logical 1 to 0 transition current, all ports[5]	At V_{DD} = 5.5V			–650	µA
		At V_{DD} = 2.7V			–250	µA

NOTES:
1. Maximum 15pF for $\overline{EA}/V_{PP}$.
2. Ports in quasi-bidirectional mode with weak pullup (applies to ALE, $\overline{PSEN}$ only during $\overline{RESET}$).
3. Ports in PUSH-PULL mode, both pullup and pulldown assumed to be the same strength.
4. In all output modes.
5. Port pins source a transition current when used in quasi-bidirectional mode and externally driven from 1 to 0. This current is highest when V_{IN} is approximately 2V.
6. Measured with port in high impedance mode.
7. Measured with port in quasi-bidirectional mode.
8. Under steady state (non-transient) conditions, I_{OL} must be externally limited as follows:
   Maximum I_{OL} per port pin: 15mA
   Maximum I_{OL} per 8-bit port: 26mA
   Maximum total I_{OL} for all outputs: 71mA
   If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

A/D CONVERTER DC ELECTRICAL CHARACTERISTICS

T_{amb} = 0 to +70°C for commercial unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS		UNIT
			MIN	MAX	
AV_{DD}	Analog supply voltage		2.7	3.3	V
AI_{DD}	Analog supply current (operating)	Port 5 = 0 to AV_{DD}		tbd	mA
AI_{ID}	Analog supply current (Idle mode)			tbd	µA
AI_{PD}	Analog supply current (Power-Down mode)	2V < AV_{PD} < AV_{DD} max		tbd	µA
AV_{IN}	Analog input voltage		AV_{SS} –0.2	AV_{DD} +0.2	V
R_{REF}	Resistance between V_{REF+} and V_{REF-}		tbd	tbd	kΩ
C_{IA}	Analog input capacitance			15	pF
–	A/D input slew rate			tbd	mV/µs
DL_e	Differential non-linearity[1, 2, 3]			tbd	LSB
IL_e	Integral non-linearity[1, 4]			tbd	LSB
OS_e	Offset error[1, 5]			tbd	LSB
G_e	Gain error[1, 6]			tbd	LSB
A_e	Absolute voltage error[1, 7]			tbd	LSB
M_{CTC}	Channel-to-channel matching			tbd	LSB
C_t	Crosstalk between inputs of port[8]	0 – 100kHz		tbd	dB

NOTES:
1. Conditions: AV_{REF-} = 0V; AV_{REF+} = 3.0V.
2. The differential non-linearity (DL_e) is the difference between the actual step width and the ideal step width. See Figure 7.
3. The ADC is monotonic, there are no missing codes.
4. The integral non-linearity (IL_e) is the peak difference between the center of the steps of the actual and the ideal transfer curve after appropriate adjustment of gain and offset errors. See Figure 7.
5. The offset error (OS_e) is the absolute difference between the straight line which fits the actual transfer curve (after removing gain error), and the straight line which fits the ideal transfer curve. See Figure 7.
6. The gain error (G_e) is the relative difference in percent between the straight line fitting the actual transfer curve (after removing offset error), and the straight line which fits the ideal transfer curve. Gain error is constant at every point on the transfer curve. See Figure 7.
7. The absolute voltage error (A_e) is the maximum difference between the center of the steps of the actual transfer curve of the non-calibrated ADC and the ideal transfer curve.
8. This should be considered when both analog and digital signals are input simultaneously to Port 5. Parameter is guaranteed by design.

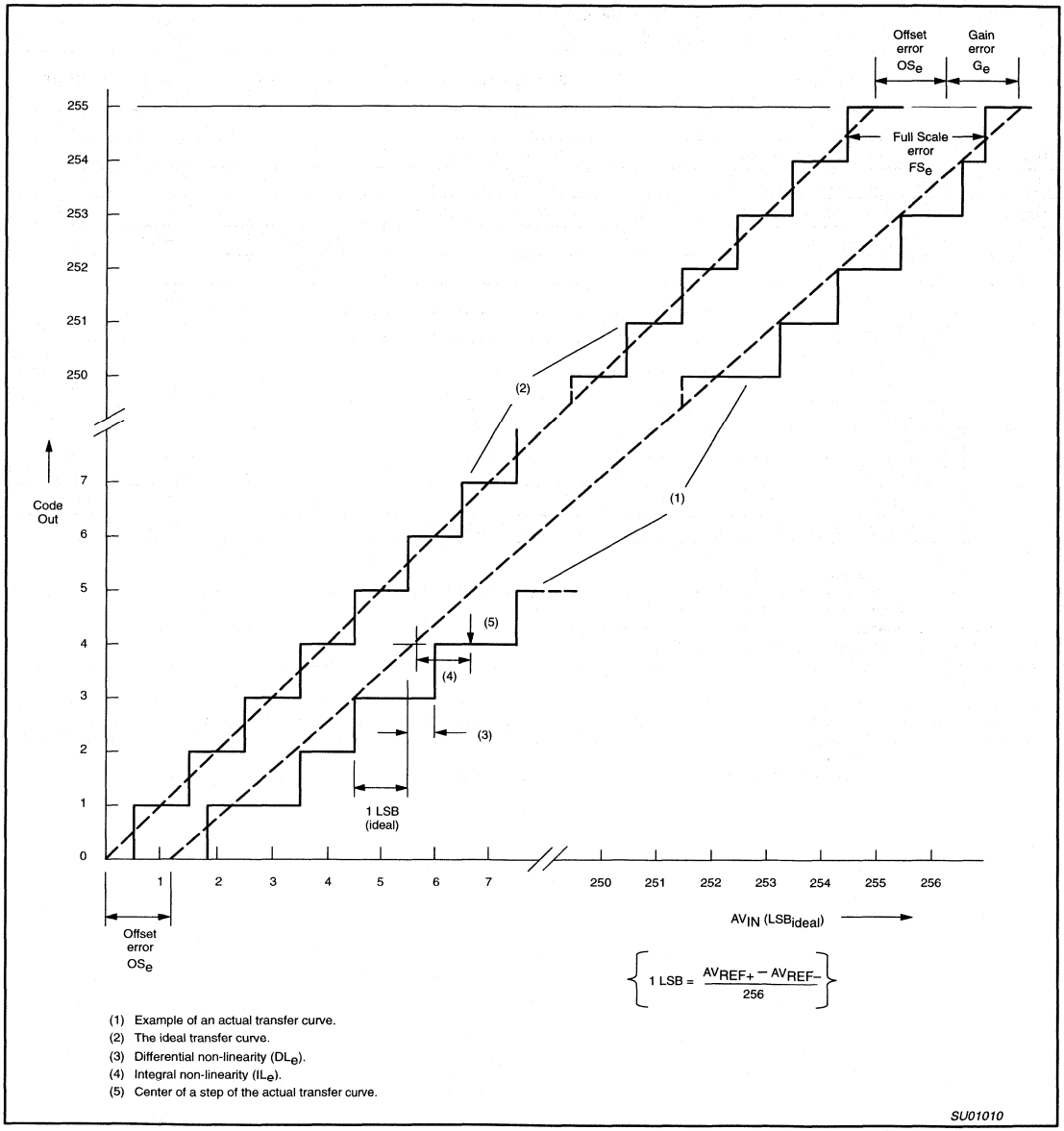


Figure 7. ADC Conversion Characteristic

AC ELECTRICAL CHARACTERISTICS (5V)

V_{DD} = 4.5V to 5.5V; T_{amb} = 0 to +70°C for commercial.

SYMBOL	FIGURE	PARAMETER	LIMITS		UNIT
			MIN	MAX	
External Clock					
f_C	14	Oscillator frequency	0	30	MHz
t_C	14	Clock period and CPU timing cycle	$1/f_C$		ns
t_{CHCX}	14	Clock high-time	$t_C * 0.5$		ns
t_{CLCX}	14	Clock low time	$t_C * 0.4$		ns
t_{CLCH}	14	Clock rise time		5	ns
t_{CHCL}	14	Clock fall time		5	ns
Address Cycle					
t_{LHLL}	8, 10, 12	ALE pulse width (programmable)	$(V1 * t_C) - 6$		ns
t_{AVLL}	8, 10, 12	Address valid to ALE de-asserted (set-up)	$(V1 * t_C) - 12$		ns
t_{LLAX}	8, 10, 12	Address hold after ALE de-asserted	$(t_C/2) - 10$		ns
Code Read Cycle					
t_{PLPH}	8	$\overline{PSEN}$ pulse width	$(V2 * t_C) - 10$		ns
t_{LLPL}	8	ALE de-asserted to $\overline{PSEN}$ asserted	$(t_C/2) - 7$		ns
t_{AVIVA}	8	Address valid to instruction valid, ALE cycle (access time)		$(V3 * t_C) - 36$	ns
t_{AVIVB}	9	Address valid to instruction valid, non-ALE cycle (access time)		$(V4 * t_C) - 29$	ns
t_{PLIV}	8	$\overline{PSEN}$ asserted to instruction valid (enable time)		$(V2 * t_C) - 29$	ns
t_{PHIX}	8	Instruction hold after $\overline{PSEN}$ de-asserted	0		ns
t_{PHIZ}	8	Bus 3-State after $\overline{PSEN}$ de-asserted		$t_C - 8$	ns
t_{IXUA}	8	Hold time of unlatched part of address after instruction latched	0		ns
Data Read Cycle					
t_{RLRH}	10	$\overline{RD}$ pulse width	$(V7 * t_C) - 10$		ns
t_{LLRL}	10	ALE de-asserted to $\overline{RD}$ asserted	$(t_C/2) - 7$		ns
t_{AVDVA}	10	Address valid to data input valid, ALE cycle (access time)		$(V6 * t_C) - 36$	ns
t_{AVDVB}	11	Address valid to data input valid, non-ALE cycle (access time)		$(V5 * t_C) - 29$	ns
t_{RLDV}	10	$\overline{RD}$ low to valid data in (enable time)		$(V7 * t_C) - 29$	ns
t_{RHDX}	10	Data hold time after $\overline{RD}$ de–asserted	0		ns
t_{RHDZ}	10	Bus 3-State after $\overline{RD}$ de-asserted (disable time)		$t_C - 8$	ns
t_{DXUA}	10	Hold time of unlatched part of address after data latched	0		ns
Data Write Cycle					
t_{WLWH}	12	$\overline{WR}$ pulse width	$(V8 * t_C) - 10$		ns
t_{LLWL}	12	ALE falling edge to $\overline{WR}$ asserted	$(V12 * t_C) - 10$		ns
t_{QVWX}	12	Data valid before $\overline{WR}$ asserted (data set-up time)	$(V13 * t_C) - 22$		ns
t_{WHQX}	12	Data hold time after $\overline{WR}$ de-asserted (Note 6)	$(V11 * t_C) - 5$		ns
t_{AVWL}	12	Address valid to $\overline{WR}$ asserted (address set-up time) (Note 5)	$(V9 * t_C) - 22$		ns
t_{UAWH}	12	Hold time of unlatched part of address after $\overline{WR}$ is de-asserted	$(V11 * t_C) - 7$		ns
Wait Input					
t_{WTH}	13	WAIT stable after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) asserted		$(V10 * t_C) - 30$	ns
t_{WTL}	13	WAIT hold after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) asserted	$(V10 * t_C) - 5$		ns

NOTES ON PAGE 376.

AC ELECTRICAL CHARACTERISTICS (5V) (continued)

This set of parameters is referenced to the XA-S3 clock output.

SYMBOL	FIGURE	PARAMETER	LIMITS		UNIT
			MIN	MAX	
Address Cycle					
t_{CHLH}	8	Delay from CLKOUT rising edge to ALE rising edge	10	40	ns
t_{CLLL}	8	Delay from CLKOUT falling edge to ALE falling edge			ns
t_{CHAV}	8	Delay from CLKOUT rising edge to address valid			ns
t_{CHAX}	8	Address hold after CLKOUT rising edge			ns
Code Read Cycle					
t_{CHPL}	8	Delay from CLKOUT rising edge to $\overline{PSEN}$ asserted			ns
t_{CHPH}	8	Delay from CLKOUT rising edge to $\overline{PSEN}$ de-asserted			ns
t_{IVCH}	8	Instruction valid to CLKOUT rising edge			ns
t_{CHIX}	8	Instruction hold from CLKOUT rising edge			ns
t_{CHIZ}	8	Bus 3-State after CLKOUT rising edge (code read)			ns
Data Read Cycle					
t_{CHRL}	10	Delay from CLKOUT rising edge to $\overline{RD}$ asserted			ns
t_{CHRH}	10	Delay from CLKOUT rising edge to $\overline{RD}$ de-asserted			ns
t_{DVCH}	10	Data valid to CLKOUT rising edge			ns
t_{CHDX}	10	Data hold after CLKOUT rising edge			ns
t_{CHDZ}	10	Bus 3-State after CLKOUT rising edge (data read)			ns
Data Write Cycle					
t_{CHWL}	12	Delay from CLKOUT rising edge to $\overline{WR}$ asserted			ns
t_{CHWH}	12	Delay from CLKOUT rising edge to $\overline{WR}$ de-asserted			ns
t_{QVCH}	12	Data valid to CLKOUT rising edge			ns
t_{CHQX}	12	Data hold after CLKOUT rising edge			ns
t_{CHQZ}	12	Bus 3-State after CLKOUT rising edge (data write)			ns
Wait Input					
t_{CHWTH}	13	WAIT stable before CLKOUT rising edge			ns
t_{CHWTL}	13	WAIT hold after CLKOUT rising edge			ns

NOTES ON PAGE 376.

AC ELECTRICAL CHARACTERISTICS (3V)

V_{DD} = 2.7V to 4.5V; T_{amb} = 0 to +70°C for commercial.

SYMBOL	FIGURE	PARAMETER	LIMITS		UNIT
			MIN	MAX	
Address Cycle					
t_{LHLL}	8, 10, 12	ALE pulse width (programmable)	$(V1 * t_C) - 10$		ns
t_{AVLL}	8, 10, 12	Address valid to ALE de-asserted (set-up)	$(V1 * t_C) - 18$		ns
t_{LLAX}	8, 10, 12	Address hold after ALE de-asserted	$(t_C/2) - 12$		ns
Code Read Cycle					
t_{PLPH}	8	$\overline{PSEN}$ pulse width	$(V2 * t_C) - 12$		ns
t_{LLPL}	8	ALE de-asserted to $\overline{PSEN}$ asserted	$(t_C/2) - 9$		ns
t_{AVIVA}	8	Address valid to instruction valid, ALE cycle (access time)		$(V3 * t_C) - 58$	ns
t_{AVIVB}	9	Address valid to instruction valid, non-ALE cycle (access time)		$(V4 * t_C) - 52$	ns
t_{PLIV}	8	$\overline{PSEN}$ asserted to instruction valid (enable time)		$(V2 * t_C) - 52$	ns
t_{PHIX}	8	Instruction hold after $\overline{PSEN}$ de-asserted	0		ns
t_{PHIZ}	8	Bus 3-State after $\overline{PSEN}$ de-asserted		$t_C - 8$	ns
t_{IXUA}	8	Hold time of unlatched part of address after instruction latched	0		ns
Data Read Cycle					
t_{RLRH}	10	$\overline{RD}$ pulse width	$(V7 * t_C) - 12$		ns
t_{LLRL}	10	ALE de-asserted to $\overline{RD}$ asserted	$(t_C/2) - 9$		ns
t_{AVDVA}	10	Address valid to data input valid, ALE cycle (access time)		$(V6 * t_C) - 58$	ns
t_{AVDVB}	11	Address valid to data input valid, non-ALE cycle (access time)		$(V5 * t_C) - 52$	ns
t_{RLDV}	10	$\overline{RD}$ low to valid data in (enable time)		$(V7 * t_C) - 52$	ns
t_{RHDX}	10	Data hold time after $\overline{RD}$ de–asserted	0		ns
t_{RHDZ}	10	Bus 3-State after $\overline{RD}$ de-asserted (disable time)		$t_C - 8$	ns
t_{DXUA}	10	Hold time of unlatched part of address after data latched	0		ns
Data Write Cycle					
t_{WLWH}	12	$\overline{WR}$ pulse width	$(V8 * t_C) - 12$		ns
t_{LLWL}	12	ALE falling edge to $\overline{WR}$ asserted	$(V12 * t_C) - 10$		ns
t_{QVWX}	12	Data valid before $\overline{WR}$ asserted (data set-up time)	$(V13 * t_C) - 28$		ns
t_{WHQX}	12	Data hold time after $\overline{WR}$ de-asserted (Note 6)	$(V11 * t_C) - 8$		ns
t_{AVWL}	12	Address valid to $\overline{WR}$ asserted (address set-up time) (Note 5)	$(V9 * t_C) - 28$		ns
t_{UAWH}	12	Hold time of unlatched part of address after $\overline{WR}$ is de-asserted	$(V11 * t_C) - 10$		ns
Wait Input					
t_{WTH}	13	WAIT stable after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) asserted		$(V10 * t_C) - 40$	ns
t_{WTL}	13	WAIT hold after bus strobe ($\overline{RD}$, $\overline{WR}$, or $\overline{PSEN}$) asserted	$(V10 * t_C) - 5$		ns

NOTES ON PAGE 376.

AC ELECTRICAL CHARACTERISTICS (3V) (continued)

This set of parameters is referenced to the XA-S3 clock output.

SYMBOL	FIGURE	PARAMETER	LIMITS		UNIT
			MIN	MAX	
Address Cycle					
t_{CHLH}	8	Delay from CLKOUT rising edge to ALE rising edge	15	60	ns
t_{CLLL}	8	Delay from CLKOUT falling edge to ALE falling edge			ns
t_{CHAV}	8	Delay from CLKOUT rising edge to address valid			ns
t_{CHAX}	8	Address hold after CLKOUT rising edge			ns
Code Read Cycle					
t_{CHPL}	8	Delay from CLKOUT rising edge to $\overline{PSEN}$ asserted			ns
t_{CHPH}	8	Delay from CLKOUT rising edge to $\overline{PSEN}$ de-asserted			ns
t_{IVCH}	8	Instruction valid to CLKOUT rising edge			ns
t_{CHIX}	8	Instruction hold from CLKOUT rising edge			ns
t_{CHIZ}	8	Bus 3-State after CLKOUT rising edge (code read)			ns
Data Read Cycle					
t_{CHRL}	10	Delay from CLKOUT rising edge to $\overline{RD}$ asserted			ns
t_{CHRH}	10	Delay from CLKOUT rising edge to $\overline{RD}$ de-asserted			ns
t_{DVCH}	10	Data valid to CLKOUT rising edge			ns
t_{CHDX}	10	Data hold after CLKOUT rising edge			ns
t_{CHDZ}	10	Bus 3-State after CLKOUT rising edge (data read)			ns
Data Write Cycle					
t_{CHWL}	12	Delay from CLKOUT rising edge to $\overline{WR}$ asserted			ns
t_{CHWH}	12	Delay from CLKOUT rising edge to $\overline{WR}$ de-asserted			ns
t_{QVCH}	12	Data valid to CLKOUT rising edge			ns
t_{CHQX}	12	Data hold after CLKOUT rising edge			ns
t_{CHQZ}	12	Bus 3-State after CLKOUT rising edge (data write)			ns
Wait Input					
t_{CHWTH}	13	WAIT stable before CLKOUT rising edge			ns
t_{CHWTL}	13	WAIT hold after CLKOUT rising edge			ns

NOTES:

1. Load capacitance for all outputs = 80pF.
2. Variables V1 through V13 reflect programmable bus timing, which is programmed via the Bus Timing registers (BTRH and BTRL). Refer to the *XA User Guide* for details of the bus timing settings.
   - V1) This variable represents the programmed width of the ALE pulse as determined by the ALEW bit in the BTRL register. V1 = 0.5 if the ALEW bit = 0, and 1.5 if the ALEW bit = 1.
   - V2) This variable represents the programmed width of the $\overline{PSEN}$ pulse as determined by the CR1 and CR0 bits or the CRA1, CRA0, and ALEW bits in the BTRL register.
     - For a bus cycle with **no** ALE, V2 = 1 if CR1/0 = 00, 2 if CR1/0 = 01, 3 if CR1/0 = 10, and 4 if CR1/0 = 11. Note that during burst mode code fetches, $\overline{PSEN}$ does not exhibit transitions at the boundaries of bus cycles. V2 still applies for the purpose of determining peripheral timing requirements.
     - For a bus cycle **with** an ALE, V2 = the total bus cycle duration (2 if CRA1/0 = 00, 3 if CRA1/0 = 01, 4 if CRA1/0 = 10, and 5 if CRA1/0 = 11) minus the number of clocks used by ALE (V1 + 0.5) = 2.
       Example: if CRA1/0 = 10 and ALEW = 1, the V2 = 4 – (1.5 + 0.5) = 2.
   - V3) This variable represents the programmed length of an entire code read cycle **with** ALE. This time is determined by the CRA1 and CRA0 bits in the BTRL register. V3 = the total bus cycle duration (2 if CRA1/0 = 00, 3 if CRA1/0 = 01, 4 if CRA1/0 = 10, and 5 if CRA1/0 = 11).
   - V4) This variable represents the programmed length of an entire code read cycle with **no** ALE. This time is determined by the CR1 and CR0 bits in the BTRL register. V4 = 1 if CR1/0 = 00, 2 if CR1/0 = 01, 3 if CR1/0 = 10, and 4 if CR1/0 = 11.
   - V5) This variable represents the programmed length of an entire data read cycle with **no** ALE. This time is determined by the DR1 and DR0 bits in the BTRH register. V5 = 1 if DR1/0 = 00, 2 if DR1/0 = 01, 3 if DR1/0 = 10, and 4 if DR1/0 = 11.
   - V6) This variable represents the programmed length of an entire data read cycle **with** ALE. The time is determined by the DRA1 and DRA0 bits in the BTRH register. V6 = the total bus cycle duration (2 if DRA1/0 = 00, 3 if DRA1/0 = 01, 4 if DRA1/0 = 10, and 5 if DRA1/0 = 11).

V7) This variable represents the programmed width of the $\overline{RD}$ pulse as determined by the DR1 and DR0 bits or the DRA1, DRA0 in the BTRH register, and the SLEW bit in the BTRL register. Note that during a 16-bit operation on an 8-bit external bus, $\overline{RD}$ remains low and does not exhibit a transition between the first and second byte bus cycles. V7 still applies for the purpose of determining peripheral timing requirements. The timing for the first byte is for a bus cycle with ALE, the timing for the second byte is for a bus cycle with no ALE.
- For a bus cycle with **no** ALE, V7 = 1 if DR1/0 = 00, 2 if DR1/0 = 01, 3 if DR1/0 = 10, and 4 if DR1/0 = 11.
- For a bus cycle **with** an ALE, V7 = the total bus cycle duration (2 if DRA1/0 = 00, 3 if DRA1/0 = 01, 4 if DRA1/0 = 10, and 5 if DRA1/0 = 11) minus the number of clocks used by ALE (V1 + 0.5).
  Example: if DRA1/0 = 00 and ALEW = 0, then V7 = 2 – (0.5 +0.5) = 1.

V8) This variable represents the programmed width of the $\overline{WRL}$ and/or $\overline{WRH}$ pulse as determined by the WM1 bit in the BTRL register. V8 = 1 if WM1 = 0, and 2 if WM1 = 1.

V9) This variable represents the programmed address setup time for a write as determined by the data write cycle duration (defined by DW1 and DW0 or the DWA1 and DWA0 bits in the BTRH register), the WM0 bit in the BTRL register, and the value of V8.
- For a bus cycle **with** an ALE, V9 = the total bus write cycle duration (2 if DWA1/0 = 00, 3 if DWA1/0 = 01, 4 if DWA1/0 = 10, and 5 if DWA1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8) minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1).
  Example: If DWA1/0 = 10, WM0 = 1, and WM1 = 1, then V9 = 4 – 1 – 2 = 1.
- For a bus cycle with **no** ALE, V9 = the total bus cycle duration (2 if DW1/0 = 00, 3 if DW1/0 = 01, 4 if DW1/0 = 10, and 5 if DW1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WMo = 0 and 1 if WM0 = 1).
  Example: If DW1/0 = 11, WM0 = 1, and WM1 = 0, then V9 = 5 – 1 – 1 = 3.

V10) This variable represents the length of a bus strobe for calculation of WAIT set-up and hold times. The strobe may be $\overline{RD}$ (for data read cycles), $\overline{WRL}$ and/or $\overline{WRH}$ (for data write cycles), or $\overline{PSEN}$ (for code read cycles), depending on the type of bus cycle being widened by WAIT. V10 = 2 for WAIT associated with a code read cycle using $\overline{PSEN}$. V10 = V8 for a data write cycle using $\overline{WRL}$ and/or $\overline{WRH}$. V10 = V7 – 1 for a data read cycle using $\overline{RD}$. This means that a single clock data read cycle cannot be stretched using WAIT. If WAIT is used to vary the duration of data read cycles, the $\overline{RD}$ strobe width must be set to be at least two clocks in duration. Also see Note 4.

V11) This variable represents the programmed write hold time as determined by the WM0 bit in the BTRL register. V11 0 if the WM0 bit = 0, and 1 if the WM0 bit = 1.

V12) this variable represents the programmed period between the end of the ALE pulse and the beginning of the $\overline{WRL}$ and/or $\overline{WRH}$ pulse as determined by the data write cycle duration (defined by the DWA1 and DWA0 bits in the BTRH register), the WM0 bit in the BTRL register, and the values of V1 and V8. V12 = the total bus cycle duration (2 if DWA1/0 = 00, 3 if DWA1/0 = 01, 4 if DWA1/0 = 10, and 5 if DWA1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1), minus the width of the ALE pulse (V1).
Example: If SWA1/0 = 11, WM0 = 1, WM1 = 0, and ALEW = 1, then V12 = 5 – 1 – 1 – 1.5 = 1.5.

V13) This variable represents the programmed data setup time for a write as determined by the data write cycle duration (defined by DW1 and DW0 or the DWA1 and DWA0 bits in the BTRH register), the WM0 bit in the BTRL register, and the values of V1 and V8.
- For a bus cycle **with** an ALE, V13 = the total bus cycle duration (2 if DWA1/0 = 00, 3 if DWA1/0 = 01, 4 if DWA1/0 = 10, and 5 if DWA1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1), minus the number of clocks used by ALE (V1 + 0.5).
  Example: If DWA1/0 = 11, WM0 = 1, WM1 = 1, and ALEW = 0, then V13 = 5 – 1 – 2 – 1 = 1.
- For a bus cycle with **no** ALE, V13 = the total bus cycle duration (2 if DW1/0 = 00, 3 if DW1/0 = 01, 4 if DW1/0 = 10, and 5 if DW1/0 = 11) minus the number of clocks used by the $\overline{WRL}$ and/or $\overline{WRH}$ pulse (V8), minus the number of clocks used by data hold time (0 if WM0 = 0 and 1 if WM0 = 1).
  Example: If DW1/0 = 01, WM0 = 1, and WM1 = 0, then V13 = 3 – 1 – 1 = 1.

3. Not all combinations of bus timing configuration values result in valid bus cycles. Please refer to the *XA User Guide* section on the External Bus for details.
4. When code is being fetched for execution on the external bus, a burst mode fetch is used that dows not have $\overline{PSEN}$ edges in every fetch cycle. This would be A3–A0 for an 8-bit bus, and A3–A1 for a 16-bit bus. Also, a 16-bit read operation conducted on an 8-bit wide bus similarly does not include two separate $\overline{RD}$ strobes. So, a rising edge on the low order address line (A0) must be used to trigger a WAIT in the second half of such a cycle.
5. This parameter is provided for peripherals that have the data clocked in on the falling edge of the $\overline{WR}$ strobe. This is not usually the case and in most applications this parameter is not used.
6. Please note that the XA-S3 requires that extended data bus hold time (WM0 = 1) to be used with external bus write cycles.

AC WAVEFORMS

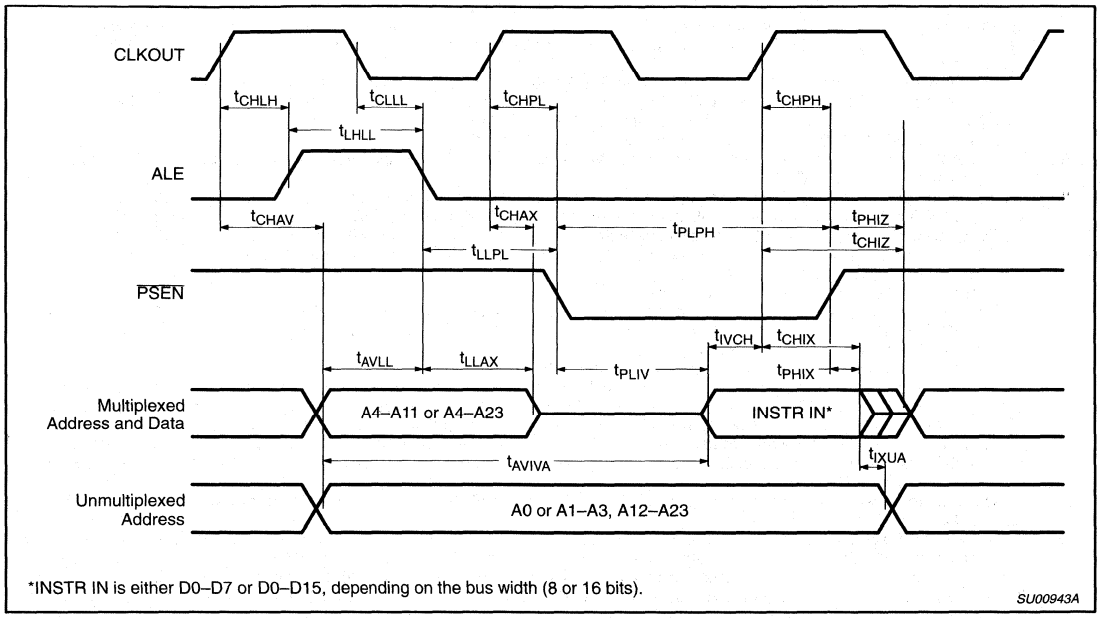


Figure 8. External Program Memory Read Cycle (ALE Cycle)

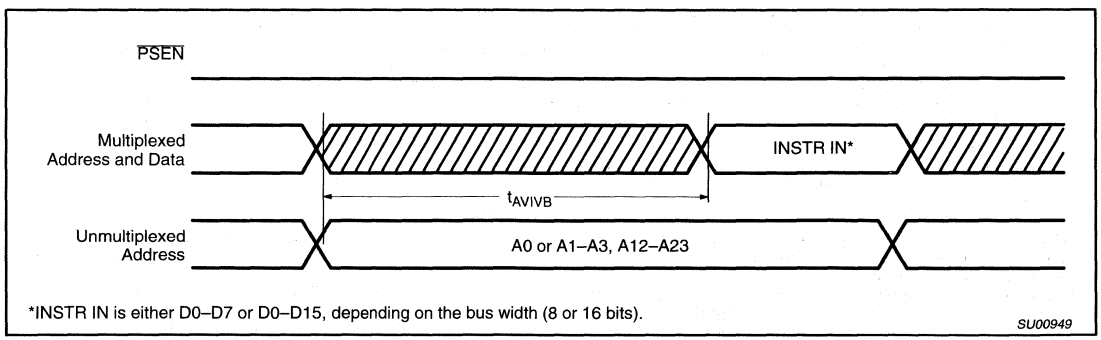


Figure 9. External Program Memory Read Cycle (Non-ALE Cycle)

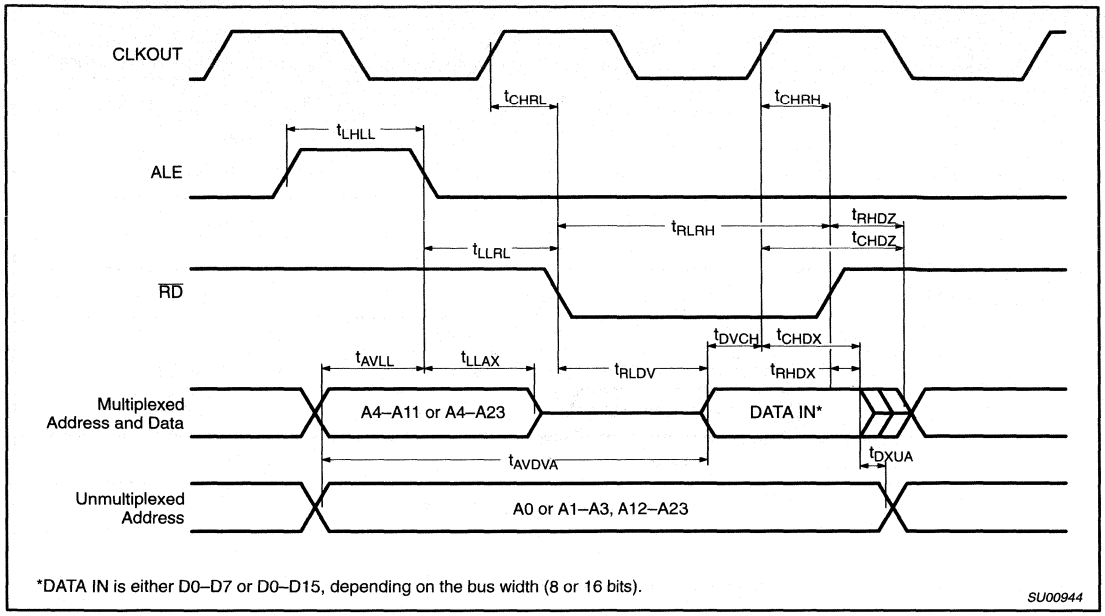


Figure 10. External Data Memory Read Cycle (ALE Cycle)

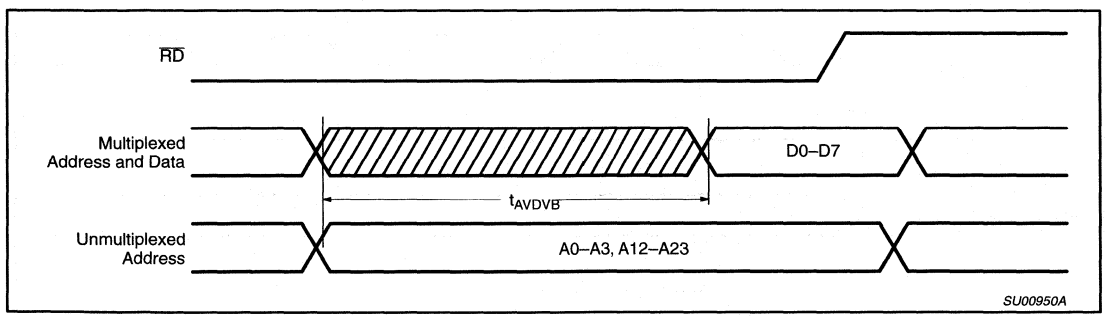


Figure 11. External Data Memory Read Cycle (Non-ALE Cycle)

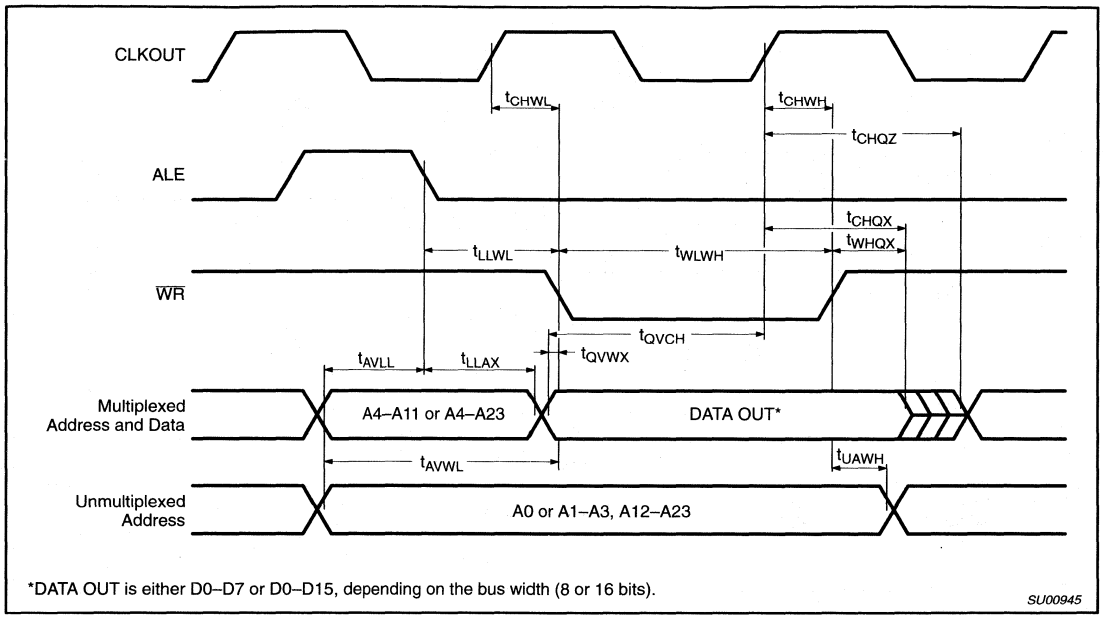


Figure 12. External Data Memory Write Cycle

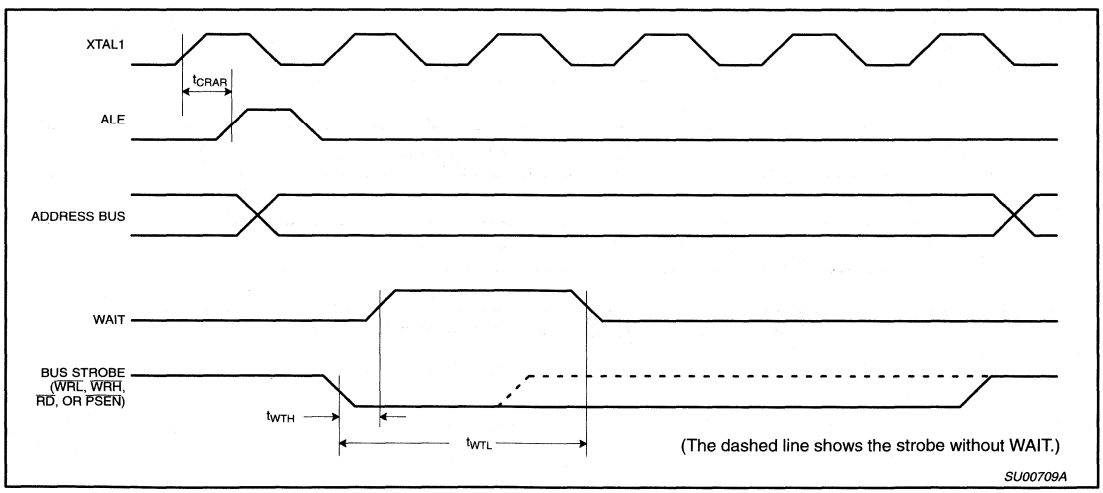


Figure 13. WAIT Signal Timing

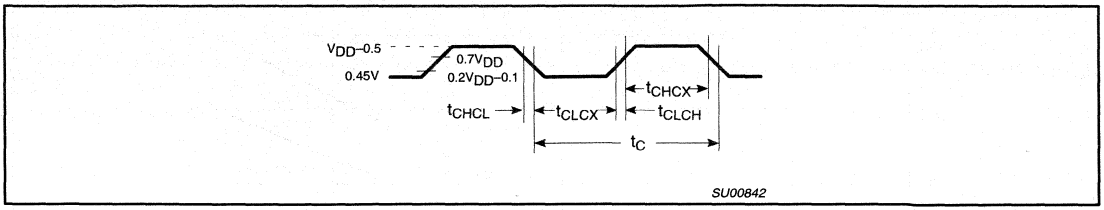


Figure 14. External Clock Drive

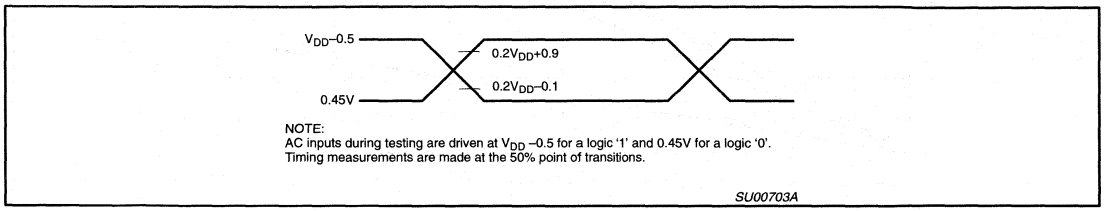


Figure 15. AC Testing Input/Output

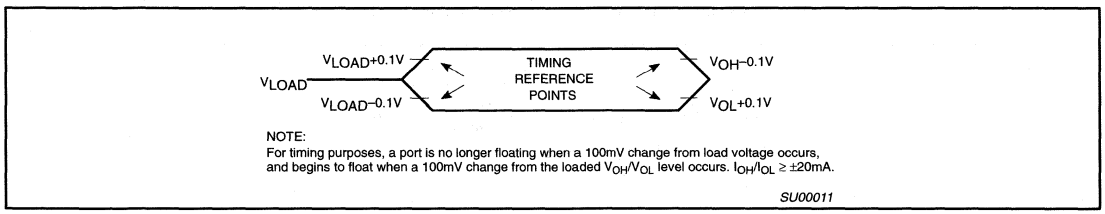


Figure 16. Float Waveform

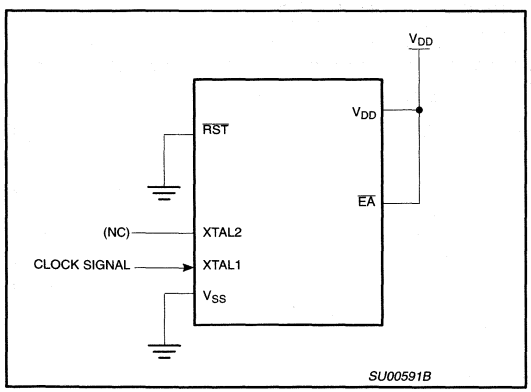


Figure 17. I_{DD} Test Condition, Active Mode
All other pins are disconnected

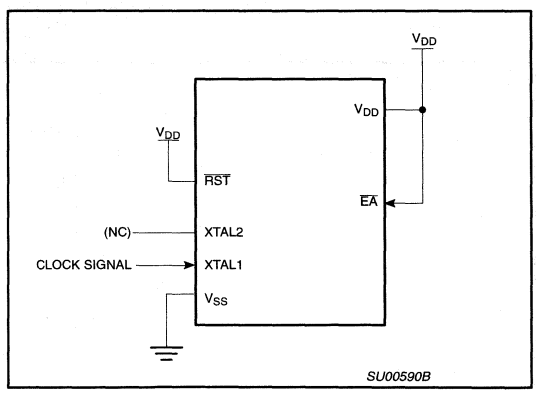


Figure 18. I_{DD} Test Condition, Idle Mode
All other pins are disconnected

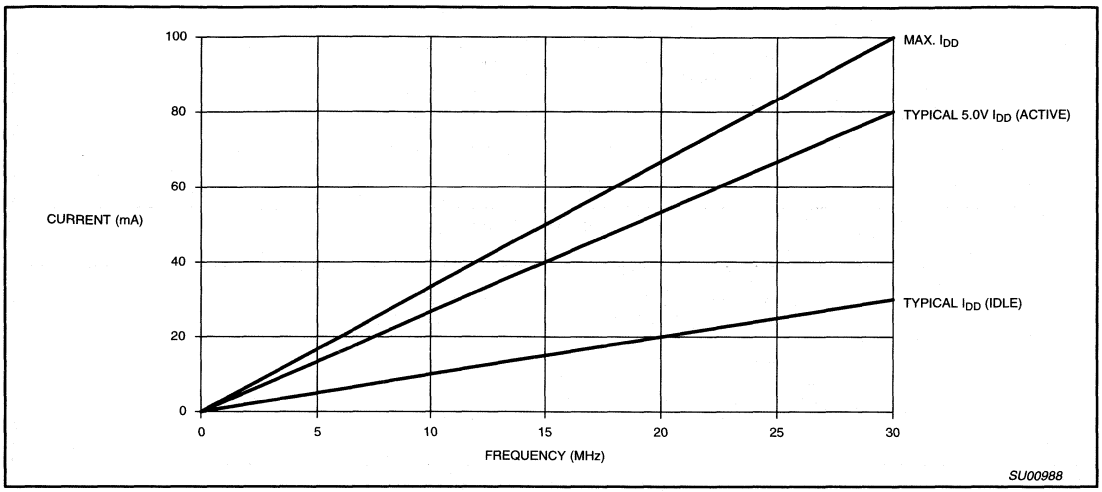


Figure 19. I_{DD} vs. Frequency
Valid only within frequency specification of the device under test.

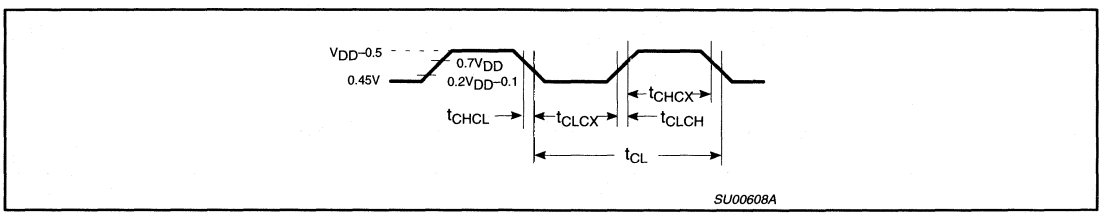


Figure 20. Clock Signal Waveform for I_{DD} Tests in Active and Idle Modes
$t_{CLCH} = t_{CHCL} = 5ns$

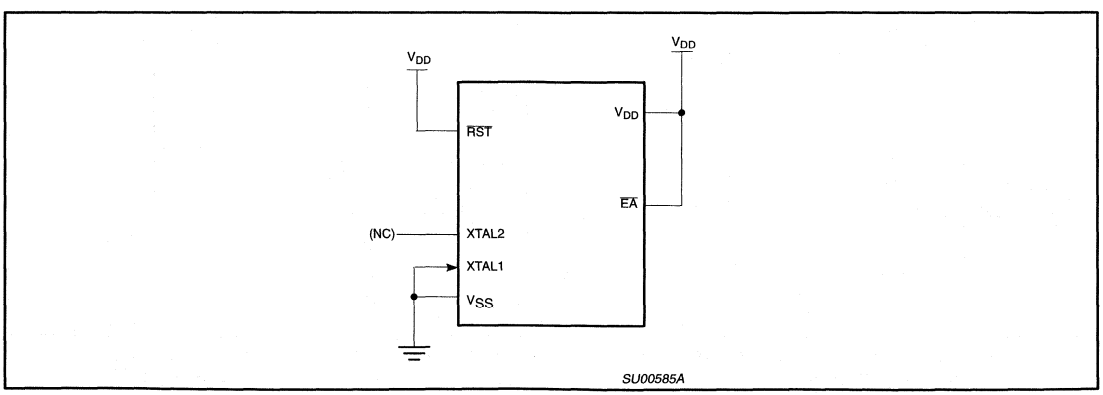


Figure 21. I_{DD} Test Condition, Power Down Mode
All other pins are disconnected. V_{DD}=2V to 5.5V

EPROM CHARACTERISTICS

The XA-S3 is programmed by using a modified Improved Quick-Pulse Programming™ algorithm. This algorithm is essentially the same as that used by 80C51 family EPROM parts. However different pins are used for many programming functions.

Detailed EPROM programming information may be obtained from the Internet at www.philipsmcu.com/ftp.html.

The XA-S3 contains three signature bytes that can be read and used by an EPROM programming system to identify the device. The signature bytes identify the device as an XA-S3 manufactured by Philips.

Security Bits

With none of the security bits programmed the code in the program memory can be verified. When only security bit 1 is programmed, MOVC instructions executed from external program memory are disabled from fetching code bytes from the internal memory. All further programming of the EPROM is disabled. When security bits 1 and 2 are programmed, in addition to the above, verify mode is disabled. When all three security bits are programmed, all of the conditions above apply and all external program memory execution is disabled. (See Table 4.)

Table 4. Program Security Bits

PROGRAM LOCK BITS				
	SB1	SB2	SB3	PROTECTION DESCRIPTION
1	U	U	U	No Program Security features enabled.
2	P	U	U	MOVC instructions executed from external program memory are disabled from fetching code bytes from internal memory and further programming of the EPROM is disabled.
3	P	P	U	Same as 2, also verify is disabled.
4	P	P	P	Same as 3, external execution is disabled. Internal data RAM is not accessible.

NOTES:
1. P – programmed. U – unprogrammed.
2. Any other combination of the security bits is not defined.

ROM CODE SUBMISSION

When submitting ROM code for the XA-S3, the following must be specified:
1. 32k byte user ROM data
2. ROM security bits.

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 7FFFH	DATA	7:0	User ROM Data
8020H	SEC	0	ROM Security Bit 1
8020H	SEC	1	ROM Security Bit 2 0 = enable security 1 = disable security
8020H	SEC	3	ROM Security Bit 3 0 = enable security 1 = disable security

™Trademark phrase of Intel Corporation.

GPS Baseband Processor SAA1575

INTRODUCTION

The SAA1575 is an integrated circuit which implements a complete baseband function for Global Positioning System (GPS) receivers. It combines a 16 bit Philips 80C51XA microcontroller, 8 GPS channel correlators and related peripherals in a single IC. Users can implement a complete GPS receiver using only the SAA1575, the UAA1570 front-end Philips IC (or similar), external memory and a few discrete components.

The IC is aimed at low cost applications. A low power solution was also used where possible, although this was of secondary importance to cost. The core of the SAA1575 runs at 3V. However, for compatibility with current automotive applications, the periphery is supplied from separate pins and can be run between 3V and 5V, as required.

The function of the SAA1575 is to read the one or two bit sampled if bitstream from a front-end IC and, under control of firmware on an external ROM, calculate the full GPS solution. The results are communicated to a host in NMEA format via a standard serial port. A second serial port can be used to provide differential GPS information to the processor for more advance applications. In addition, various other functions are integrated onto the IC such as a real time GPS clock, a power-down/reset controller, timer/counters and a watchdog timer.

In summary, the SAA1575 has the following functional units:

- 16 bit 80C51XA microcontroller core
- 2K words on chip SRAM (16 bit words)
- 8 GPS channel correlators
- 2 UARTs
- 8 general purpose I/O lines
- 3 timer/counters
- 1 real time clock
- 1 watchdog timer
- 1 power-down/reset controller

The structure is based on a 16 bit microcontroller core operating on all other units as memory mapped peripherals and registers. A 16 bit data bus and a 19 bit address bus are extended to external pins so that external data and program memory can be accessed. On-chip decoder circuits eliminate the need for external glue logic for external memory access.

Each of the 8 GPS channel correlators includes a carrier numerically controlled oscillator (NCO), PN code generator, phase rotator and low pass filter. They correlate the local PN sequence with the digitized input GPS signal and generate the filtered correlation result for the micro-controller. The firmware provided then generates a navigation solution and provides standard GPS data outputs to the user.

The GPS firmware is located in off-chip program memory. It processes the GPS signals from up to 8 satellites and generates GPS information that can be output to the host processor through one of the two serial ports. Much of hardware configuration of the SAA1575 can be controller by the firmware and so details such as the external bus timing may change between firmware revisions. For the purposed of this document, the standard Philips firmware has been assumed (release HP00).

FEATURES

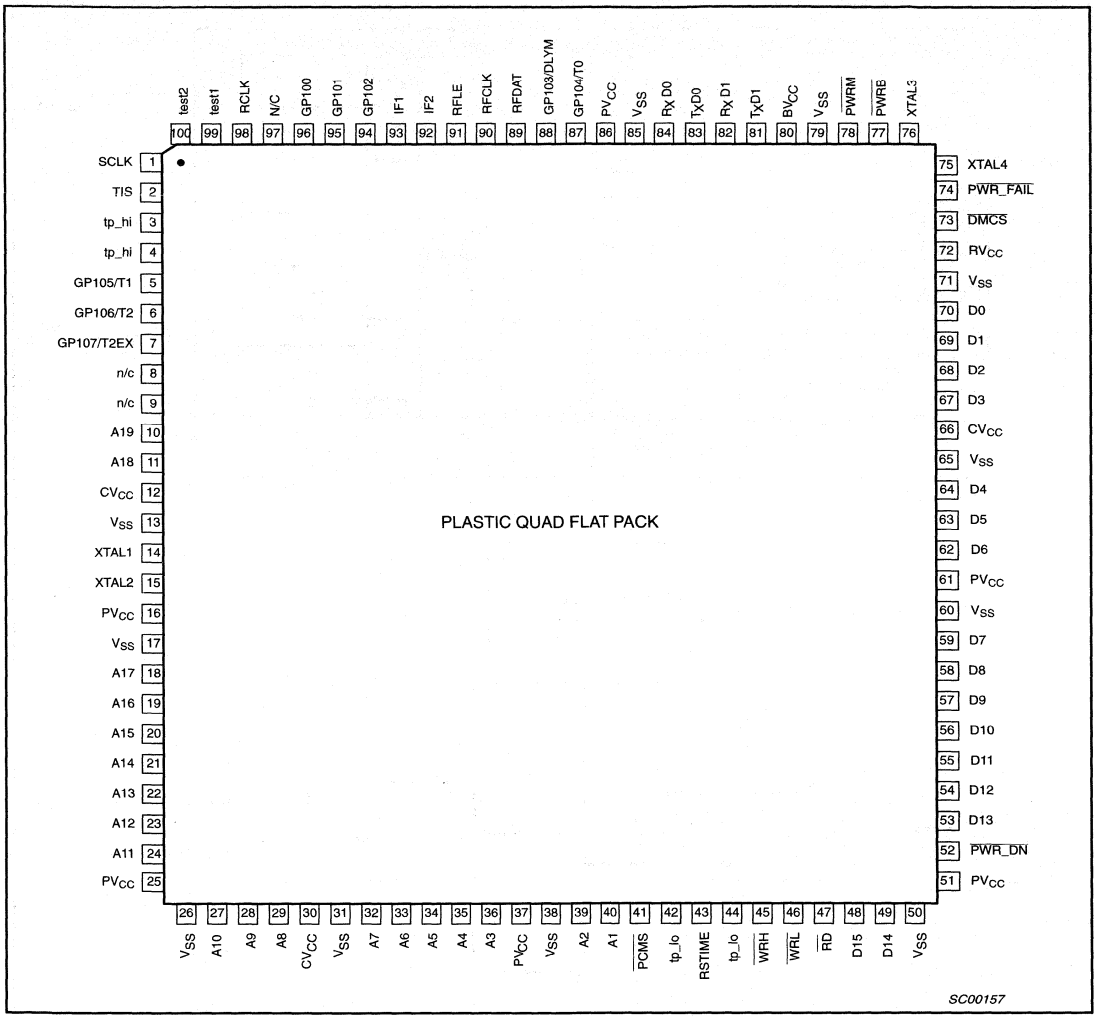
The following list summarizes the features of the SAA1575.

- Single chip GPS baseband solution with built in 16 bit micro-controller.
- All digital, 0.5 micron CMOS technology.
- Single power supply with fully 3V operation.
- Separate I/O power supply pins for operation with 3V or 5V external devices.
- Up to 30MHz system clock from on chip crystal oscillator or external clock input*
- 2K words internal data memory for fast execution.
- External bus for up to 512K words data memory and 512K words program memory.
- Programmable external bus timing to match external memory speed.
- Chip selection outputs to reduce glue logic requirements.
- Reset controller for power-down detection and servicing
- 8 GPS channel correlators driven by firmware for flexible GPS correlation algorithms.
- 1 second pulse output of GPS time
- 2 bit digital IF GPS signal input synchronized to external sample clock**
- 2 full duplex UARTs for communication with host system processor and other devices.
- Real time clock with 32.768kHz crystal and supply for low power timekeeping
- Watchdog timer.
- Power-down modes under firmware control
- 100 pin LQFP package
- 50mA I_{CC} typical when 8 GPS channels in track (approximate)

* 30MHz assumed for standard Philips firmware

** Only sign bit is used for standard Philips firmware

PIN CONFIGURATION



ORDERING INFORMATION

DESCRIPTION	TEMPERATURE RANGE	ORDER CODE	DRAWING NO.
100-Pin Plastic Low-Profile Quad Flat Pack (LQFP)			SOT407-1

BLOCK DIAGRAM

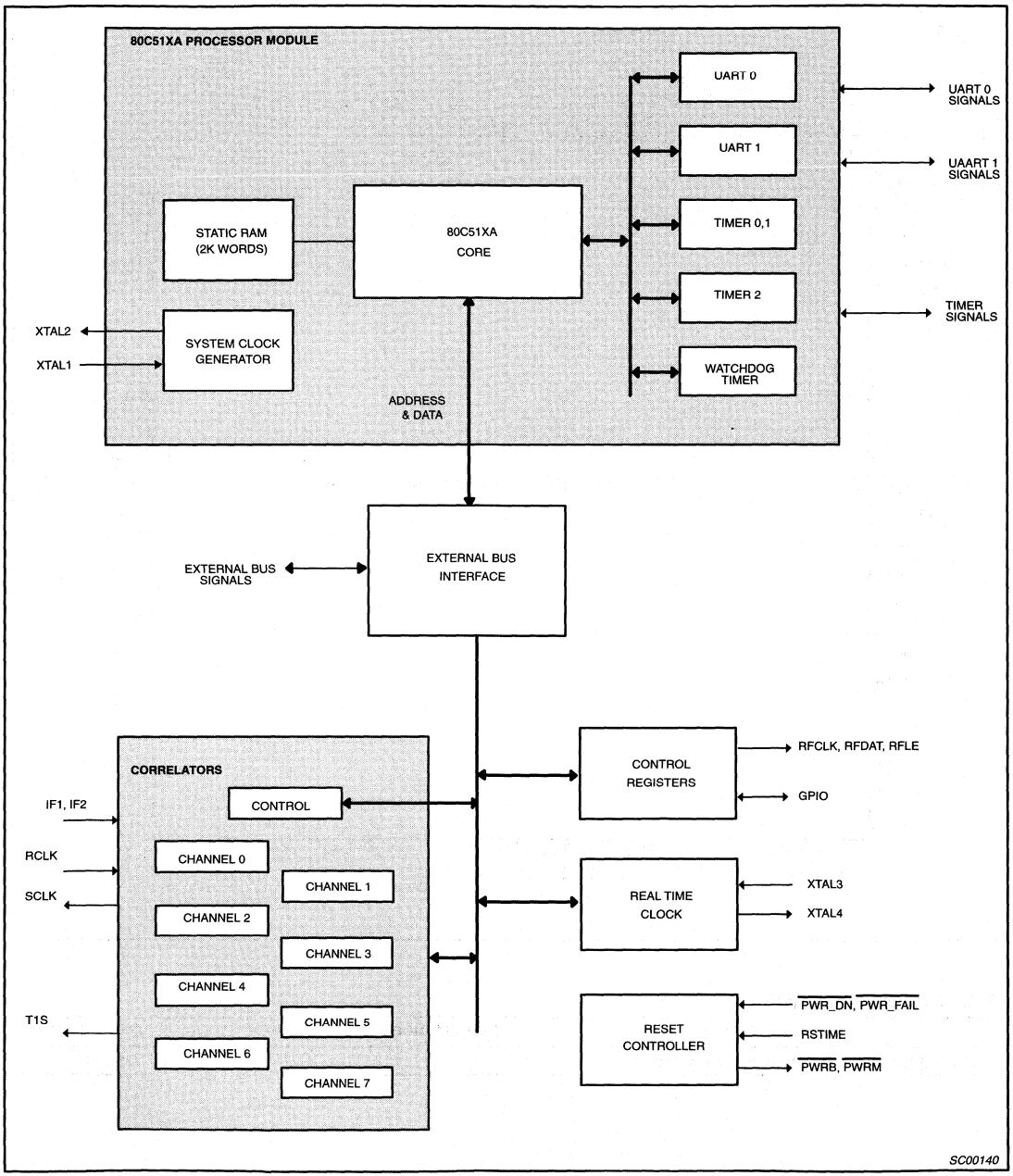
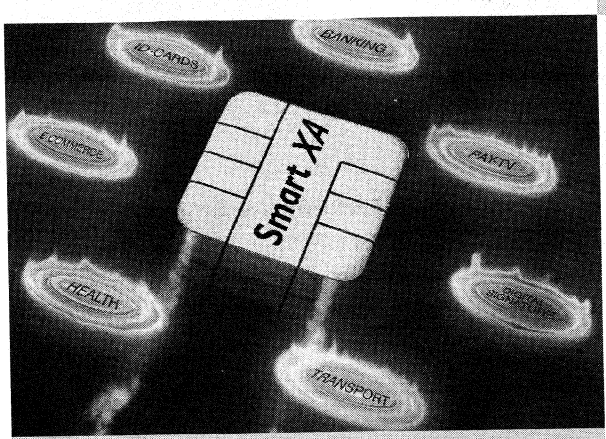


Figure 1. Block Diagram

SmartXA-Family
Card IC / Chip Module

SmartXA



Features

- **SmartXA 16-bit microcontroller**
  - 16 bit fully static CPU
  - 16 Mbyte address range for code
  - 1 Mbyte address range for data
  - Harvard architecture
- **Hardware Firewall concept providing system and user mode with memory protection**
- **Memory Management Unit (MMU)**
- **32 Kbyte User ROM**
- **1.5 / 2 Kbyte Data RAM**
- **8 / 16 / 32 Kbyte EEPROM**
- **80C51 compatible sub-mode**
- **21 CPU registers - each 16-bit wide**
- **FameX 32-bit co-processor for Public Key Crypto**
- **True Random Number Generator hardware**
- **Two 16-bit timers**
- **Multi-tasking and real-time executive support**
- **Multiple source vectorized interrupt system**
- **Multiple source reset system**
- **Low power / low voltage**
- **Exception sensors for**
  - high/low operating temperature
  - high/low clock frequency
  - high/low supply voltage
- **Three power-saving modes**
- **ISO/IEC 7816 contact interface**
- **Die-individual FabKey for customer defined security data, transport protection and quality tracing**

With the SmartXA Architecture Platform, Philips Semiconductors is the first to offer the computing power of a personal computer in a single smart card IC. A true **16-bit microcontroller core** with system and user mode capabilities and the integrated **Memory Management Unit (MMU)** for dynamic code allocation allow multi-application beyond today's constraints. In addition to this sophisticated architecture, the built-in **80C51 sub-mode** provides an easy upgrade path for already existing code.

The hardware firewall, built upon the system/user mode and MMU, guarantees the integrity of all applications in operation. Such a trusted hardware firewall is a prerequisite for future-proofed multi-application / multi-provider smart cards. A **True Random Number Generator** (TRND) together with the **FameX** crypto co-processor complete this leading edge product family.

SmartXA increases the performance of applications by 30 times, thus providing ideal preconditions to run complex multi-application smart card operating systems and languages, such as **MULTOS** and **JAVA™**. Even 80C51 code is executed four times faster than with today's 8-bit microcontrollers.

The complete instruction set is tailored and optimized to support high level languages. All operations, registers and address modes can be combined to achieve an unmatched **high code density**. Less memory consumption and a faster time to market are the major benefits of this innovation. Simulators, Emulators and C-Compilers for easy and fast development of card operating systems are available.

SmartXA - The Quantum Leap in Smart Card Technology

Java™ is a trademark of Sun Microsystems, Inc.
MULTOS is a trademark of MAOSCO Ltd.

Philips Semiconductors

PHILIPS

Let's make things better.

Specification Smart*XA* Family

Operating frequency	1 to 8 MHz
Operating supply voltage	2.7 to 5.5 V
Memory EEPROM size	8 / 16 / 32 Kbyte
Write endurance	min. 100.000 cycles
Data retention	min. 10 years
EEPROM page mode programming time	4.5 ms
16 Kbyte EEPROM personalization time	1.5 s typ.
FabKey area	32 byte, customized
Fame*X* Crypto Co-processor for Standard Public Key Algorithms: RSA, DSS, Eliptic Curve, Diffie-Hellmann, Guillou-Quisquater, ...	
2048 bits max. key length for RSA	
ESD protection on ISO pads (acc. to MIL Standard 883-C method 3015)	5 KV
Operating temperature	- 25 to + 85°C
Wafer thickness	160 µm
Wafer diameter	6 inches / 8 inches
Test data	mapped and inked / mapped

Product Information

SXA08W01AEW	Wafer on FFC / 160 mm
SXA08W01AEU	Wafer / 160 mm
SXA08W01AEV	Module 8 contacts

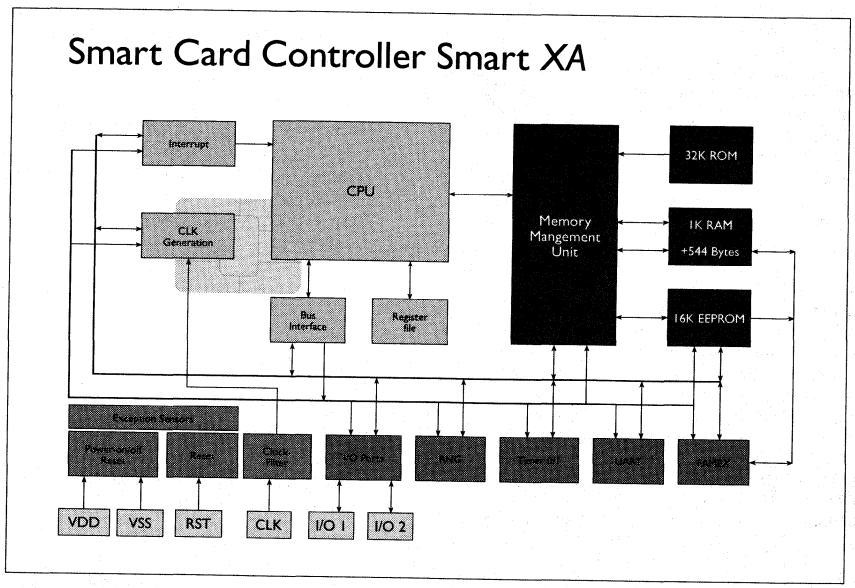
x = 8; 8 K EEPROM
x = 16; 16K EEPROM
x = 32; 32 K EEPROM

Fame*X* Performance Benchmark

RSA Key Length	Signature Generation	Signature Verification
512 bit	≤ 45 ms (CRT)	140 ms
1024 bit	≤ 250 ms (CRT)	800 ms

Note: Performance data is independent of external clock, and thus valid for 1-8 MHz; CRT = Chinese Remainder Theorem

For further information please contact your local Philips Semiconductors Sales Office.



Specification subject to change without notice.

PHILIPS

Philips Semiconductors

Let's make things better.

Section 5
Future Derivatives

CONTENTS

CMOS single-chip 16-bit microcontroller with 64K embedded FLASH

XA-G49

GENERAL DESCRIPTION

The XA-G49 is a member of Philips' 80C51 XA (eXtended Architecture) family of high performance 16-bit single-chip microcontrollers.

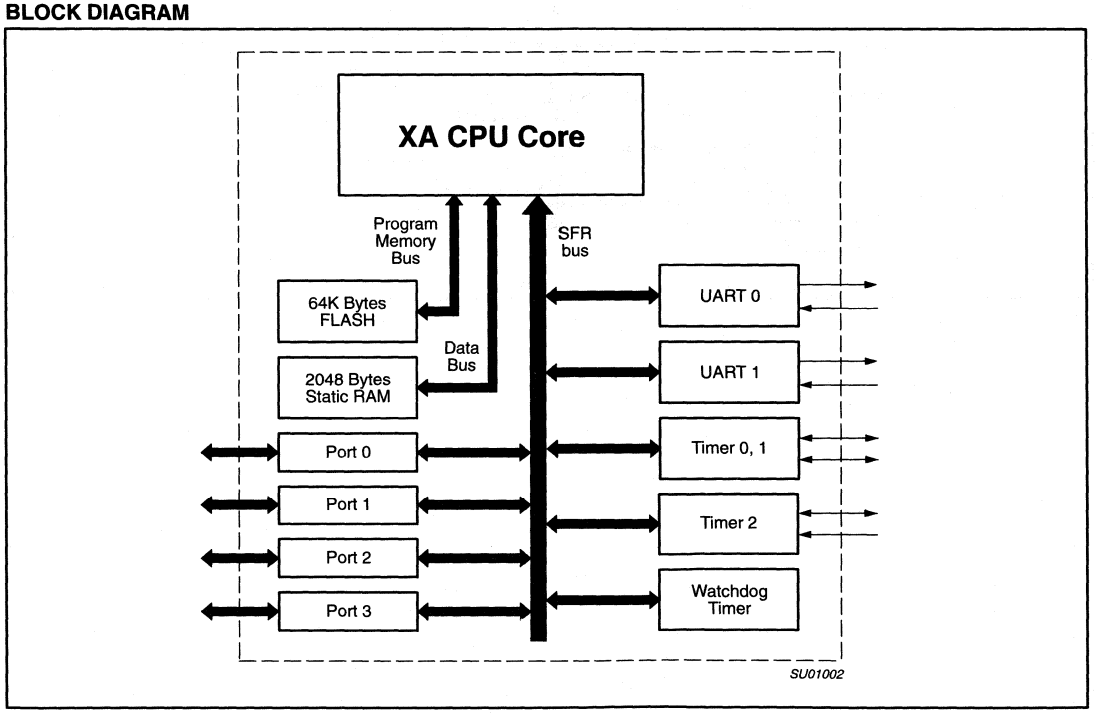
The XA-G49 contains 64k bytes of Flash program memory, and provides three general purpose timers/counters, a watchdog timer, dual UARTs, and four general purpose I/O ports with programmable output configurations.

A default serial loader program in the Boot ROM allows In System Programming (ISP) of the Flash memory without the need for a loader in the Flash code. User programs may erase and reprogram the Flash memory at will through the use of standard routines contained in the Boot ROM.

FEATURES

- 2.7V to 5.5V operation
- 64K bytes of on-chip Flash program memory with In System Programming capability
- Single voltage In System Programming of the Flash memory
- Boot ROM contains low level Flash programming routines and a default serial loader using the UART
- 2048 bytes of on-chip data RAM
- Supports off-chip program and data addressing up to 1 megabyte (20 address lines)
- Three standard counter/timers with enhanced features (same as XA-G3 T0, T1, and T2). All timers have a toggle output capability
- Watchdog timer
- Two enhanced UARTs with independent baud rates
- Seven software interrupts
- Four 8-bit I/O ports, with 4 programmable output configurations for each pin
- 30 MHz operating frequency at 5V, 16 MHz at 2.7V
- Power saving operating modes: Idle and Power-Down. Wake-Up from power-down via an external interrupt is supported.
- 44-pin PLCC and 44-pin LQFP packages

BLOCK DIAGRAM



CMOS single-chip 16-bit CAN 2.0B microcontroller XA-C3

GENERAL DESCRIPTION

The XA-C3 is a CAN 2.0B derivative in the Philips high performance 80C51**XA** (e**X**tended **A**rchitecture) 16-bit microcontroller family.

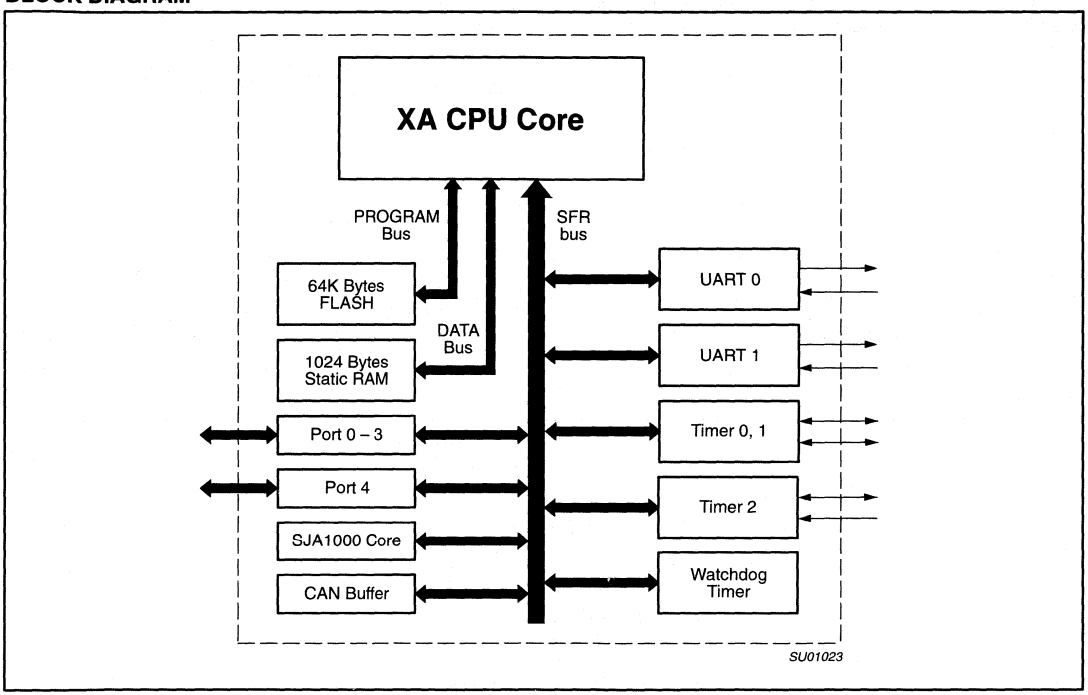
The XA-C3 contains an XA-G3 microcontroller **AND** the core of today's SJA1000 Stand-alone CAN 2.0B controller.

The XA-C3 is software compatible with XA-G3 and SJA1000 while adding FullCAN Objects and Buffer Management.

FEATURES

- 30MHz clock frequency at 5V, 16MHz at 2.7V
- 1024 bytes of on-chip DATA RAM
- Separate CAN 2.0B message buffer
- New Port 4 with two pins – CAN Rx and CAN Tx

BLOCK DIAGRAM



XA-Serial Communications Controller XA-SCC

INTRODUCTION

The new XA-Serial Communications Controller (SCC) offers an integrated, "single-chip" solution for networking and communications applications. It is the ideal system-level answer for ISDN Terminal Adapters, multiplexers, inverse multiplexers, and other remote-access products. With its high-performance 16-bit XA core, flexible communications architecture, and programmability, the XA-SCC supports concurrent operation of different protocols. It also can be configured to support a variety of industry-standard interfaces.

The XA-SCC features four full-duplex serial channels that handle both synchronous and asynchronous protocols, and an integrated, on-board DMA controller for efficient data movement with minimal CPU overhead. With the XA-SCC, designers have a "glueless" interface with few external components. The XA-SCC includes an onboard DRAM controller, complete memory interface for SRAM, DRAM, Flash, or EPROM, six fully configurable chip selects, and a de-multiplexed address/data bus. A rich complement of integrated peripherals and 32 general-purpose I/O pins add to the versatility of the XA-SCC.

KEY FEATURES OF THE XA-SCC CONTROLLER

- XA core allowing 16-bit and 8-bit external transfers
- On-chip data RAM (256 bytes)
- Onboard memory controller
  - Supports SRAM, DRAM, EPROM, and Flash
  - Address space: 2 × 16 MB
- Two enhanced XA counter/timers
- Watchdog timer with output via Reset pin
- Operating frequency of 29.4912 MHz at 5.0 V V_{CC} (commercial temperature)
- De-multiplexed address/data bus
- Four Serial Communications Channels
  - High speed, full duplex
  - Support DMA
  - No CPU bottleneck
- NMSI[1] and MSI[2] interface support for IDL[3]
- High-performance DMA engine with eight DMA channels
- Four 8-bit I/O ports with four programmable pin configurations
- Programmable chip selects for up to six external devices
- Operation at 3.0 V and 5.0 V
- Onboard DRAM Controller
  - Full refresh
  - Supports up to 32 MB of FPM or EDO RAM
- Three-wire Serial Communications Port (SCP)
- Auto Baud Rate generators
  - Up to 921.6 K at f_{OSC} = 29.4912 MHz
- Prescaler (7/8) for 56–Kbps or 64–Kbps support
- Test pattern generator/checkers for V.54 and 2047
- Standard IDL Interface: supports 10-Bit Frame or 8-Bit Frame (Short Frame)
- Integrated 100-pin LQFP package

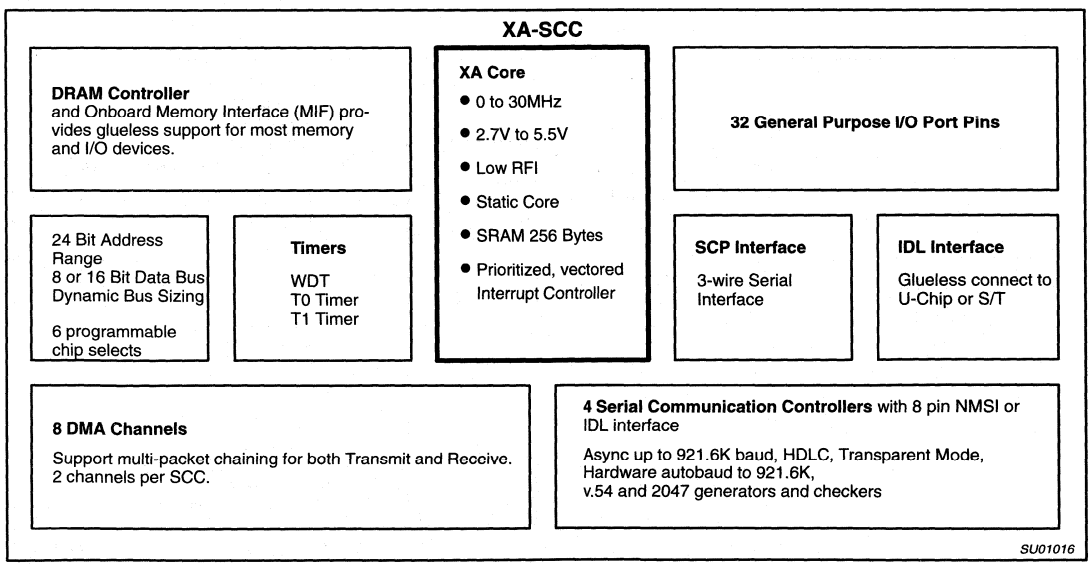


Figure 1. Major features of the XA-SCC Serial Communications Controller

1. Non-multiplexed serial interface
2. Multiplexed serial interface
3. Interchip digital link

DEVELOPMENT TOOLS FOR THE LATEST APPLICATIONS

Philips Semiconductors offers a suite of hardware and software development tools for the XA-SCC. These tools, which support design and debugging activities, are available from several third-party vendors of development tools. These vendors include manufacturers of emulators, programmers, demo boards, real-time operating systems, C compilers, adapters and sockets, and a variety of software tools.

The V2.0 r0 release of the Tasking XA C compiler, which is available now, supports the XA-SCC. Also, the Crossview Debugger/ROM Monitor from Tasking now supports the XA-SCC. An XTEND-SCC development kit will be available late in 2Q98. Tasking and FDI have worked together to port the Crossview Debugger to the XTEND development kit, so that a complete hardware and software development environment will now be available for the XA-SCC. Nohau plans to release an emulator for the XA-SCC, in 3Q98.

For the latest listing of development tools available for the XA family, please contact Philips Semiconductors and request the *XA Development Tools Linecard.*

MEMORY INTERFACE

Eight DMA channels are individually dedicated to either receive or transmit. Each channel is assigned to one of the four Serial Rx channels and four Serial Dx channels. Internal buffers in the DMA channels each hold 4 bytes (2 words deep). The DMA register primarily supports circular buffers in memory large enough to handle multiple packets. This can relax the stringency of the interrupt response times required of the processor.

Memory Interface Features

- Operation at 3.0 V and 5.0 V
- Direct, "glueless" interface to Fast–Page–Mode DRAM, EDO-Page-Mode DRAM, SRAM, flash, EPROM/ROM, and generic memory interface
- DRAM controller supporting DRAMs between 256 KB and 4 M words
- Six memory banks
  - Unified code and data address spaces
  - One boot bank (bank 0, which can be swapped with bank 1)
  - Five general–purpose banks (banks 1–5)
  - Dynamic 8- or 16-bit bus sizing so that each bank can be either 8 or 16 bits wide
- Programmable bus timing generator, also supporting wait states
- Intelligent arbiter for assigning priorities between refresh, CPU access, and DMA access
- Relocatable Memory Mapped Register (MMR) locations for SCC–related configurations and data

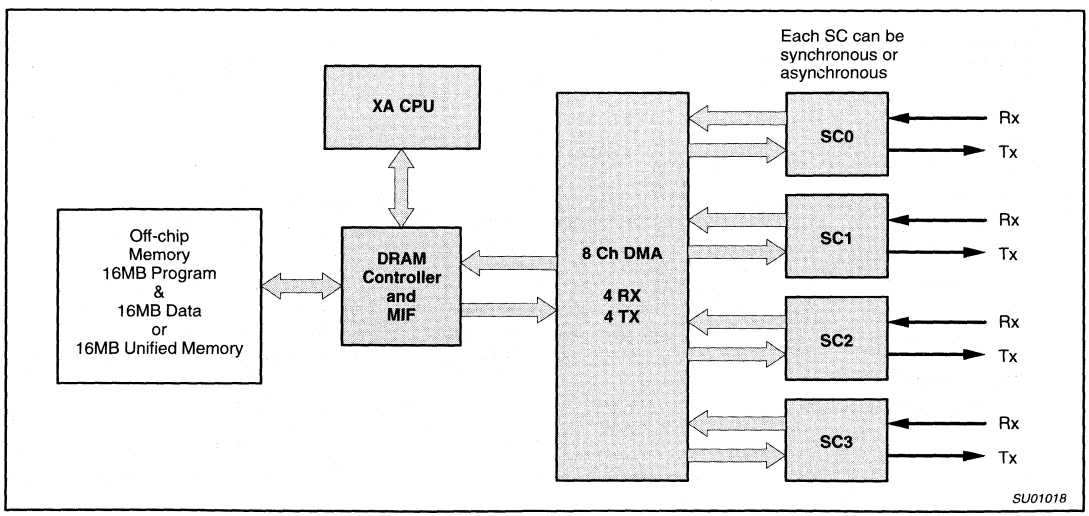


Figure 2. XA-SCC NMSI data path

XA-Serial Communications Controller XA-SCC

SERIAL COMMUNICATION CHANNELS

The XA-SCC can be configured to support a variety of communications protocols. It has four independent, full-duplex serial communications channels. Based on industry-standard 85c30 cells, the channels offer functional enhancements and full DMA capability. Protocols supported include:

- High-level Data Link Control/Synchronous Data Link Control (HDLC/SDLC)
- Universal Asynchronous Receiver/Transmitter (UART)
- Transparent mode

Each protocol can be implemented with an IDL or NMSI physical layer interface, and can be configured to operate in Echo or Loop-Back mode. (Use Echo mode to provide a return signal from the serial channel by retransmitting the received signal. Use Loop-Back mode for a local feedback connection that allows the serial channel to receive the signal that it is transmitting.) Each serial channel has two dedicated DMA channels: one for receive and one for transmit operations. These DMA channels transfer data between each serial channel and external RAM.

Each serial channel can be clocked by an external source (clock pins ComClk, RTxC, or TRxC), or by an internal clock through a baud rate generator. (Each serial channel has its own baud rate generator.) The serial channel's transmitter and receiver sections operate independently and may be clocked by different clocks. Serial channels also enable CRC calculation and protocol generation.

All seven standard NMSI interface signals (RXD, TXD, RCLK, TCLK, RTS, CTS, and CD) are available on the serial channels. I/O pins on the XA-SCC support other modem signals, including DSR and DTR. Three of the serial channels can be internally connected to the MSI for support of the ISDN B1, B2, and D channels. The MSI provides IDL "glueless" compatibility with ISDN Layer One devices from several different vendors.

General Serial Port Features

- Bits per character: 5, 6, 7, or 8
- Stop bits: 1, 1-1/2, or 2
- Parity: odd, even, or none
- Error detection, including parity, overrun, and framing
- NRZ, NRZI, FM0, and FM1 data encoding support

Asynchronous Protocol Features

- Efficient use of DRAM bandwidth
  ≅ 10% of bandwidth
  Maximum data rate of 921.6 Kbps x 8 paths
- Autobaud capability to 921.6 Kbps
- Closely coupled DMA, including "dead air character time out"
- Baud rate generator clock can be derived internally or brought in on pins

Synchronous Protocol Features

- Data rates to 4 Mbps sustained on all eight paths simultaneously
- HDLC/SDLC, Transparent Mode, BiSync, MonoSync
- CRC generation and checking (can be disabled for "end to end CRC")
- Closely coupled IDL interface (provides 2B + D)
- Clock Prescaler (7/8) supports 56-Kbps or 64-Kbps speeds

ISDN Terminal Adapters or "modems" allow access to the Internet at high speed. These "modems" also provide support for telecommuting and SOHO applications. The ISDN TA is the "modem" of choice for Internet power users. With the XA-SCC, a new generation of faster, smaller, and lower cost ISDN TAs can be produced. This is just one of the many applications for which designers can use the XA-SCC to change the nature of communications.

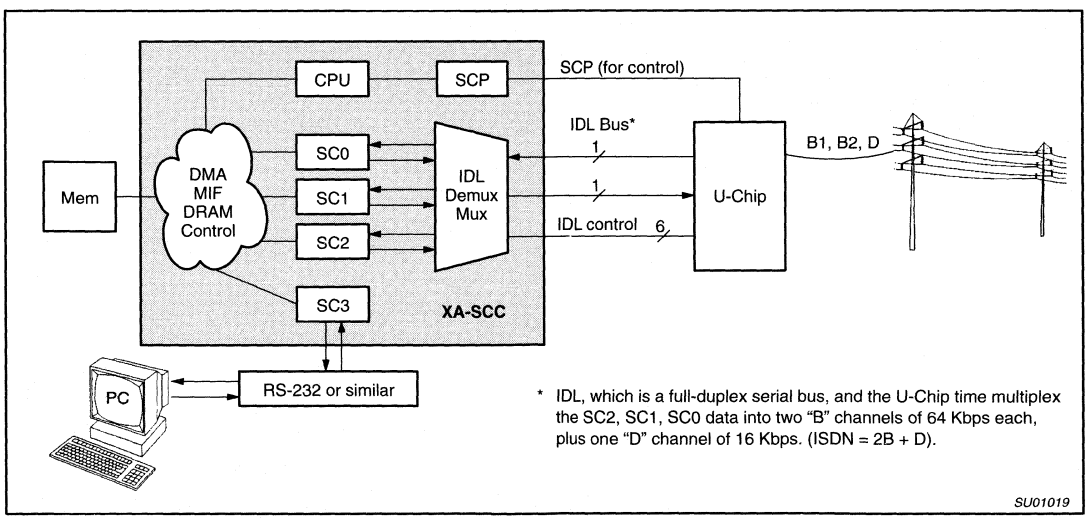


Figure 3. ISDN "Modem" or IDSL

Section 6
Control Area Network (CAN) bus

CONTENTS

Philips CAN solutions

PHILIPS CAN SOLUTIONS

With the increasing number of distributed microcontrollers and intelligent peripherals used in today's electronic systems, such as vehicle controls, networking protocols between the units have become extremely important. A wide range of these applications are using Controller Area Network (CAN) for network communication.

The Controller Area Network (CAN) is a serial bus system which complies to the 7-layer ISO/OSI communication reference model. Developed originally for passenger cars, CAN is already in use in over six million industrial control devices, sensors, and actuators. Hallmarks of the internationally standardized bus system (ISO 11898) are its simplicity, high transmission reliability and extremely short reaction times. CAN provides two communication functions: sending a message and asking for a message.

The CAN protocol describes the method by which information is passed between devices. It conforms to the OSI model and defines the lowest two layers: the data link and physical layer. The application layers are linked to the physical layer by various emerging protocols or a proprietary user-defined scheme. The CAN protocol allows for two identifier field lengths: part A specifies 11 bits, which allows 2032 different IDs out of 2048, while extended CAN (part B) additionally offers 29 bits, giving over 536 million unique identifiers.

Philips CAN devices

For connecting to different transmission media in the CAN physical layer Philips offers its CAN Transceivers: P82C250, P82C251, and P82C252. The SJA1000 is a stand-alone CAN Controller (CAN 2.0B compliant) which can be easily interfaced to a Philips 80C51 or XA microcontroller. Philips also offers several microcontrollers with on-chip CAN, the 8XC592 with 16 Kbytes of ROM or OTP memory and the 8xCE598 with 32 Kbytes. For high-performance, 16-bit solutions, Philips offers its 16-bit CAN chipset, consisting of the SJA1000 and P51XA-G3 microcontroller.

CAN outlook

For high-end CAN solutions, there's a 16-bit microcontroller with on-chip CAN under development, expected to be available early next year. And, for low-end applications, Philips has an 8xC51 CAN derivative in preparation for release in 1999. Plus, the 8xC592/8xCE598 will be continued with package compatible successor derivatives beyond the year 2000.

CAN applications

Today a wide range of applications use CAN for distributed networks such as automobiles, industrial control, agricultural machinery, marine engineering equipment, medical equipment, textile machines, and elevator monitoring systems.

CMOS single-chip 16-bit CAN 2.0B microcontroller XA-C3

GENERAL DESCRIPTION

The XA-C3 is a CAN 2.0B derivative in the Philips high performance 80C51**XA** (e**X**tended **A**rchitecture) 16-bit microcontroller family.

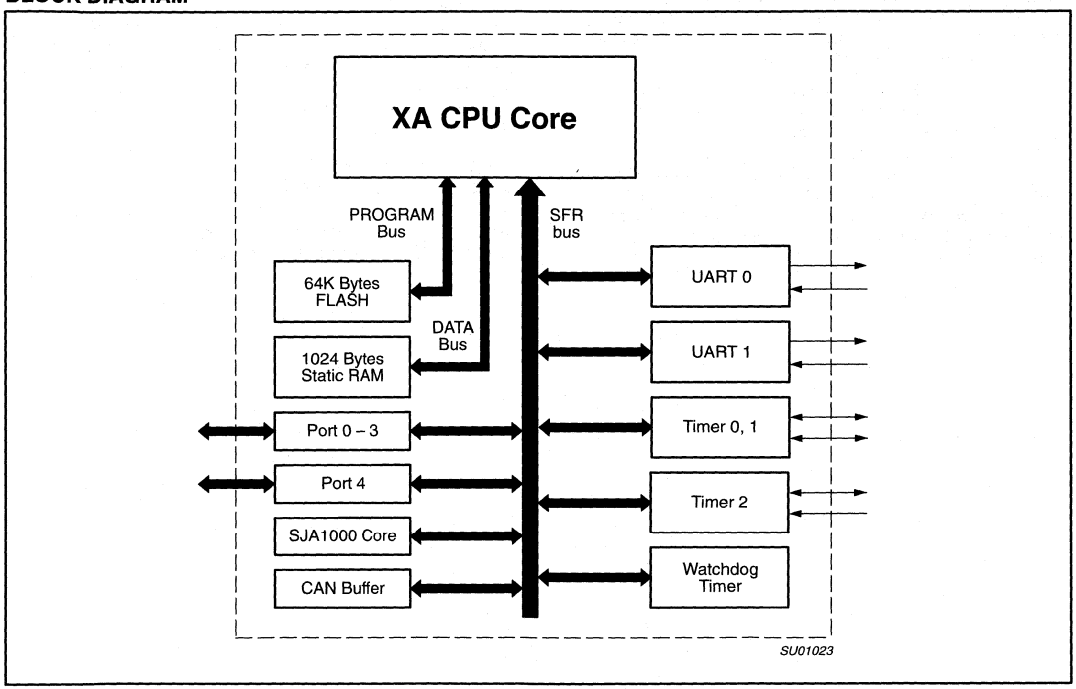
The XA-C3 contains an XA-G3 microcontroller **AND** the core of today's SJA1000 Stand-alone CAN 2.0B controller.

The XA-C3 is software compatible with XA-G3 and SJA1000 while adding FullCAN Objects and Buffer Management.

FEATURES

- 30MHz clock frequency at 5V, 16MHz at 2.7V
- 1024 bytes of on-chip DATA RAM
- Separate CAN 2.0B message buffer
- New Port 4 with two pins – CAN Rx and CAN Tx

BLOCK DIAGRAM



CONTENTS

1 FEATURES

- Pin compatibility to the PCA82C200 stand-alone CAN controller
- Electrical compatibility to the PCA82C200 stand-alone CAN controller
- Software-compatibility mode to the PCA82C200 (BasicCAN mode is default)
- Extended receive buffer (64-byte FIFO)
- CAN 2.0B protocol compatibility (extended frame passive in PCA82C200 compatibility mode)
- Supports 11-bit identifier as well as 29-bit identifier
- Bit rates up to 1 Mbits/s
- PeliCAN mode extensions:
  - Error counters with read/write access
  - Programmable error warning limit
  - Last error code register
  - Error interrupt for each CAN-bus error
  - Arbitration lost interrupt with detailed bit position
  - Single-shot transmission (no re-transmission)
  - Listen only mode (no acknowledge, no active error flags)
  - Hot plugging support (software driven bit rate detection)
  - Acceptance filter extension (4-byte code, 4-byte mask)
  - Reception of 'own' messages (self reception request)
- 24 MHz clock frequency
- Interfaces to a variety of microprocessors
- Programmable CAN output driver configuration
- Extended ambient temperature range (–40 to +125 °C).

2 GENERAL DESCRIPTION

The SJA1000 is a stand-alone controller for the Controller Area Network (CAN) used within automotive and general industrial environments. It is designed to be hardware and software compatible to the PCA82C200 CAN controller (BasicCAN) from Philips Semiconductors. Additionally, a new mode of operation is implemented (PeliCAN) which supports the CAN 2.0B protocol specification with several new features.

3 ORDERING INFORMATION

TYPE NUMBER	PACKAGE		
	NAME	DESCRIPTION	VERSION
SJA1000	DIP28	plastic dual in-line package; 28 leads (600 mil)	SOT117-1
SJA1000T	SO28	plastic small outline package; 28 leads; body width 7.5 mm	SOT136-1

4 BLOCK DIAGRAM

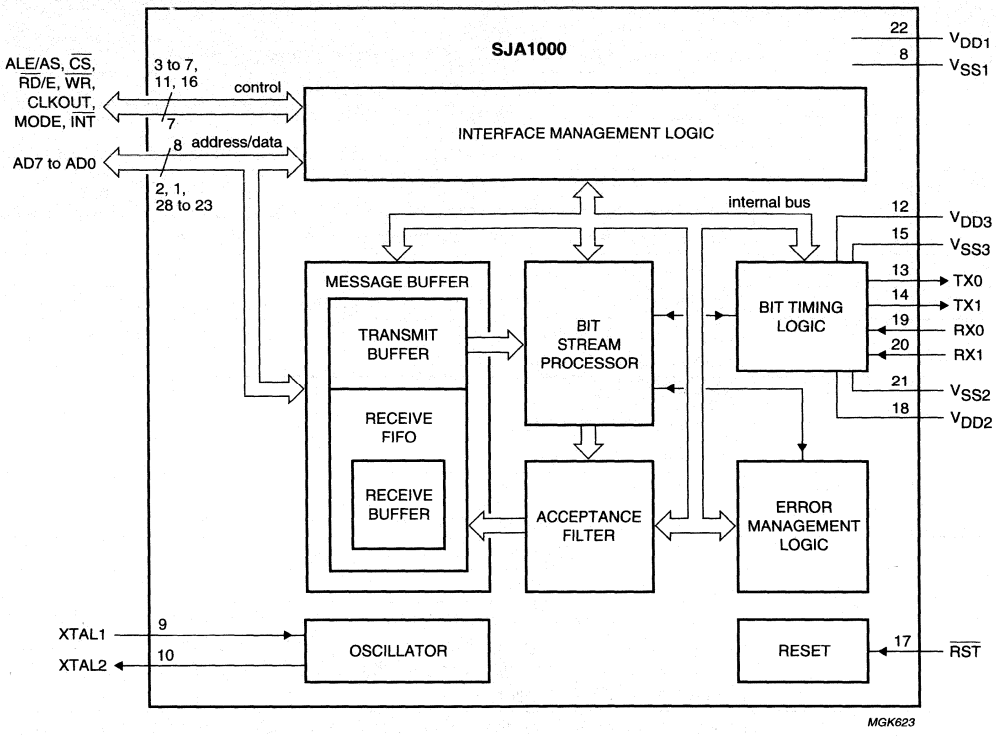


Fig.1 Block diagram.

5 PINNING

SYMBOL	PIN	DESCRIPTION
AD7 to AD0	2, 1, 28 to 23	multiplexed address/data bus
ALE/AS	3	ALE input signal (Intel mode), AS input signal (Motorola mode)
$\overline{CS}$	4	chip select input, LOW level allows access to the SJA1000
$\overline{RD}$/E	5	$\overline{RD}$ signal (Intel mode) or E enable signal (Motorola mode) from the microcontroller
$\overline{WR}$	6	$\overline{WR}$ signal (Intel mode) or RD/$\overline{WR}$ signal (Motorola mode) from the microcontroller
CLKOUT	7	clock output signal produced by the SJA1000 for the microcontroller; the clock signal is derived from the built-in oscillator via the programmable divider; the clock off bit within the clock divider register allows this pin to disable
V_{SS1}	8	ground for logic circuits
XTAL1	9	input to the oscillator amplifier; external oscillator signal is input via this pin; note 1
XTAL2	10	output from the oscillator amplifier; the output must be left open-circuit when an external oscillator signal is used; note 1
MODE	11	mode select input 1 = selects Intel mode 0 = selects Motorola mode
V_{DD3}	12	5 V supply for output driver
TX0	13	output from the CAN output driver 0 to the physical bus line
TX1	14	output from the CAN output driver 1 to the physical bus line
V_{SS3}	15	ground for output driver
$\overline{INT}$	16	interrupt output, used to interrupt the microcontroller; $\overline{INT}$ is active LOW if any bit of the internal interrupt register is set; $\overline{INT}$ is an open-drain output and is designed to be a wired-OR with other $\overline{INT}$ outputs within the system; a LOW level on this pin will reactivate the IC from sleep mode
$\overline{RST}$	17	reset input, used to reset the CAN interface (active LOW); automatic power-on reset can be obtained by connecting $\overline{RST}$ via a capacitor to V_{SS} and a resistor to V_{DD} (e.g. C = 1 μF; R = 50 kΩ)
V_{DD2}	18	5 V supply for input comparator
RX0, RX1	19, 20	input from the physical CAN-bus line to the input comparator of the SJA1000; a dominant level will wake up the SJA1000 if sleeping; a dominant level is read, if RX1 is higher than RX0 and vice versa for the recessive level; if the CBP bit (see Table 49) is set in the clock divider register, the CAN input comparator is bypassed to achieve lower internal delays if an external transceiver circuitry is connected to the SJA1000; in this case only RX0 is active; HIGH is interpreted as recessive level and LOW is interpreted as dominant level
V_{SS2}	21	ground for input comparator
V_{DD1}	22	5 V supply for logic circuits

Note

1. XTAL1 and XTAL2 pins should be connected to V_{SS1} via 15 pF capacitors.

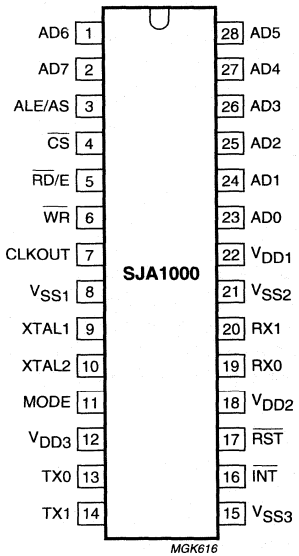


Fig.2 Pin configuration (DIP28).

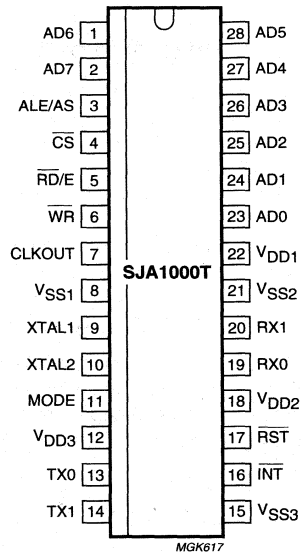


Fig.3 Pin configuration (SO28).

6 FUNCTIONAL DESCRIPTION

6.1 Description of the CAN controller blocks

6.1.1 Interface Management Logic (IML)

The interface management logic interprets commands from the CPU, controls addressing of the CAN registers and provides interrupts and status information to the host microcontroller.

6.1.2 Transmit Buffer (TXB)

The transmit buffer is an interface between the CPU and the Bit Stream Processor (BSP) that is able to store a complete message for transmission over the CAN network. The buffer is 13 bytes long, written to by the CPU and read out by the BSP.

6.1.3 Receive Buffer (RXB, RXFIFO)

The receive buffer is an interface between the acceptance filter and the CPU that stores the received and accepted messages from the CAN-bus line. The Receive Buffer (RXB) represents a CPU-accessible 13-byte window of the Receive FIFO (RXFIFO), which has a total length of 64 bytes.
With the help of this FIFO the CPU is able to process one message while other messages are being received.

6.1.4 Acceptance Filter (ACF)

The acceptance filter compares the received identifier with the acceptance filter register contents and decides whether this message should be accepted or not. In the event of a positive acceptance test, the complete message is stored in the RXFIFO.

6.1.5 Bit Stream Processor (BSP)

The bit stream processor is a sequencer which controls the data stream between the transmit buffer, RXFIFO and the CAN-bus. It also performs the error detection, arbitration, stuffing and error handling on the CAN-bus.

6.1.6 Bit Timing Logic (BTL)

The bit timing logic monitors the serial CAN-bus line and handles the bus line-related bit timing. It is synchronized to the bit stream on the CAN-bus on a 'recessive-to-dominant' bus line transition at the beginning of a message (hard synchronization) and re-synchronized on further transitions during the reception of a message (soft synchronization). The BTL also provides programmable time segments to compensate for the propagation delay times and phase shifts (e.g. due to oscillator drifts) and to define the sample point and the number of samples to be taken within a bit time.

6.1.7 Error Management Logic (EML)

The EML is responsible for the error confinement of the transfer-layer modules. It receives error announcements from the BSP and then informs the BSP and IML about error statistics.

6.2 Detailed description of the CAN controller

The SJA1000 is designed to be software and pin-compatible to its predecessor, the PCA82C200 stand-alone CAN controller. Additionally, a lot of new functions are implemented. To achieve the software compatibility, two different modes of operation are implemented:

- BasicCAN mode; PCA82C200 compatible
- PeliCAN mode; extended features.

The mode of operation is selected with the CAN-mode bit located within the clock divider register. Default mode upon reset is the BasicCAN mode.

6.2.1 PCA82C200 compatibility

In BasicCAN mode the SJA1000 emulates all known registers from the PCA82C200 stand-alone CAN controller. The characteristics, as described in Sections 6.2.1.1 to 6.2.1.4 are different from the PCA82C200 design with respect to software compatibility.

6.2.1.1 Synchronization mode

The SYNC bit in the control register is removed (CR.6 in the PCA82C200). Synchronization is only possible by a recessive-to-dominant transition on the CAN-bus. Writing to this bit has no effect. To achieve compatibility to existing application software, a read access to this bit will reflect the previously written value (flip-flop without effect).

6.2.1.2 Clock divider register

The clock divider register is used to select the CAN mode of operation (BasicCAN/PeliCAN). Therefore one of the reserved bits within the PCA82C200 is used. Writing a value between 0 and 7, as allowed for the PCA82C200, will enter the BasicCAN mode. The default state is divide by 12 for Motorola mode and divide by 2 for Intel mode. An additional function is implemented within another of the reserved bits. Setting of bit CBP (see Table 49) enables the internal RX input comparator to be bypassed thereby reducing the internal delays if an external transceiver circuit is used.

6.2.1.3 Receive buffer

The dual receive buffer concept of the PCA82C200 is replaced by the receive FIFO from the PeliCAN controller. This has no effect to the application software except for the data overrun probability. Now more than two messages may be received (up to 64 bytes) until a data overrun occurs.

6.2.1.4 CAN 2.0B

The SJA1000 is designed to support the full CAN 2.0B protocol specification, which means that the extended oscillator tolerance is implemented as well as the processing of extended frame messages. In BasicCAN mode it is possible to transmit and receive standard frame messages only (11-bit identifier). If extended frame messages (29-bit identifier) are detected on the CAN-bus, they are tolerated and an acknowledge is given if the message was correct, but there is no receive interrupt generated.

6.2.2 Differences between BasicCAN and PeliCAN mode

In the PeliCAN mode the SJA1000 appears with a re-organized register mapping with a lot of new features. All known bits from the PCA82C200 design are available as well as several new ones. In the PeliCAN mode the complete CAN 2.0B functionality is supported (29-bit identifier).

Main new features of the SJA1000 are:

- Reception and transmission of standard and extended frame format messages
- Receive FIFO (64-byte)
- Single/dual acceptance filter with mask and code register for standard and extended frame
- Error counters with read/write access
- Programmable error warning limit
- Last error code register
- Error interrupt for each CAN-bus error
- Arbitration lost interrupt with detailed bit position
- Single-shot transmission (no re-transmission on error or arbitration lost)
- Listen only mode (monitoring of the CAN-bus, no acknowledge, no error flags)
- Hot plugging supported (disturbance-free software driven bit rate detection)
- Disable CLKOUT by hardware.

6.3 BasicCAN mode

6.3.1 BasicCAN address layout

The SJA1000 appears to a microcontroller as a memory-mapped I/O device. An independent operation of both devices is guaranteed by a RAM-like implementation of the on-chip registers.

The address area of the SJA1000 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 microcontroller. During initialization the CLKOUT signal may be programmed to a value determined by the microcontroller.

A message, which should be transmitted, has to be written to the transmit buffer. After a successful reception the microcontroller may read the received message from the receive buffer and then release it for further use.

The exchange of status, control and command signals between the microcontroller and the SJA1000 is performed in the control segment. The layout of this segment is shown in Table 3. After an initial download, the contents of the registers acceptance code, acceptance mask, bus timing registers 0 and 1 and output control should not be changed. Therefore these registers may only be accessed when the reset request bit in the control register is set HIGH.

For register access, two different modes have to be distinguished:

- Reset mode
- Operating mode.

The reset mode (see Table 3, control register, bit Reset Request) is entered automatically after a hardware reset or when the controller enters the bus-off state (see Table 5, status register, bit Bus Status). The operating mode is activated by resetting of the reset request bit in the control register.

Table 1 BasicCAN address allocation; note 1

CAN ADDRESS	SEGMENT	OPERATING MODE		RESET MODE	
		READ	WRITE	READ	WRITE
0	control	control	control	control	control
1		(FFH)[2]	command	(FFH)[2]	command
2		status	–	status	–
3		interrupt	–	interrupt	–
4		(FFH)[2]	–	acceptance code	acceptance code
5		(FFH)[2]	–	acceptance mask	acceptance mask
6		(FFH)[2]	–	bus timing 0	bus timing 0
7		(FFH)[2]	–	bus timing 1	bus timing 1
8		(FFH)[2]	–	output control	output control
9		test	test	test	test
10	transmit buffer	identifier (10 to 3)	identifier (10 to 3)	(FFH)[2]	–
11		identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC	(FFH)[2]	–
12		data byte 1	data byte 1	(FFH)[2]	–
13		data byte 2	data byte 2	(FFH)[2]	–
14		data byte 3	data byte 3	(FFH)[2]	–
15		data byte 4	data byte 4	(FFH)[2]	–
16		data byte 5	data byte 5	(FFH)[2]	–
17		data byte 6	data byte 6	(FFH)[2]	–
18		data byte 7	data byte 7	(FFH)[2]	–
19		data byte 8	data byte 8	(FFH)[2]	–
20	receive buffer	identifier (10 to 3)	identifier (10 to 3)	identifier (10 to 3)	identifier (10 to 3)
21		identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC	identifier (2 to 0), RTR and DLC
22		data byte 1	data byte 1	data byte 1	data byte 1
23		data byte 2	data byte 2	data byte 2	data byte 2
24		data byte 3	data byte 3	data byte 3	data byte 3
25		data byte 4	data byte 4	data byte 4	data byte 4
26		data byte 5	data byte 5	data byte 5	data byte 5
27		data byte 6	data byte 6	data byte 6	data byte 6
28		data byte 7	data byte 7	data byte 7	data byte 7
29		data byte 8	data byte 8	data byte 8	data byte 8
30		(FFH)[2]	–	(FFH)[2]	–
31		clock divider	clock divider; note 3	clock divider	clock divider

Notes

1. It should be noted that the registers are repeated within higher CAN address areas (the most significant bits of the 8-bit CPU address are not decoded: CAN address 32 continues with CAN address 0 and so on).
2. During read-out of this register a zero is always given.
3. Some bits are writeable in reset mode only (CAN mode and CBP).

6.3.2 RESET VALUES

Detection of a 'reset request' results in aborting the current transmission/reception of a message and entering the reset mode. On the '1-to-0' transition of the reset request bit, the CAN controller returns to the operating mode.

Table 2 Reset mode configuration; notes 1 and 2

REGISTER	BIT	SYMBOL	NAME	VALUE	
				RESET BY HARDWARE	SETTING BIT CR.0 BY SOFTWARE OR DUE TO BUS-OFF
Control	CR.7	–	reserved	0	0
	CR.6	–	reserved	X	X
	CR.5	–	reserved	1	1
	CR.4	OIE	Overrun Interrupt Enable	X	X
	CR.3	EIE	Error Interrupt Enable	X	X
	CR.2	TIE	Transmit Interrupt Enable	X	X
	CR.1	RIE	Receive Interrupt Enable	X	X
	CR.0	RR	Reset Request	1 (reset mode)	1 (reset mode)
Command	CMR.7	–	reserved	note 3	note 3
	CMR.6	–	reserved		
	CMR.5	–	reserved		
	CMR.4	GTS	Go To Sleep		
	CMR.3	CDO	Clear Data Overrun		
	CMR.2	RRB	Release Receive Buffer		
	CMR.1	AT	Abort Transmission		
	CMR.0	TR	Transmission Request		
Status	SR.7	BS	Bus Status	0 (bus-on)	X
	SR.6	ES	Error Status	0 (ok)	X
	SR.5	TS	Transmit Status	0 (idle)	0 (idle)
	SR.4	RS	Receive Status	0 (idle)	0 (idle)
	SR.3	TCS	Transmission Complete Status	1 (complete)	X
	SR.2	TBS	Transmit Buffer Status	1 (released)	1 (released)
	SR.1	DOS	Data Overrun Status	0 (absent)	0 (absent)
	SR.0	RBS	Receive Buffer Status	0 (empty)	0 (empty)
Interrupt	IR.7	–	reserved	1	1
	IR.6	–	reserved	1	1
	IR.5	–	reserved	1	1
	IR.4	WUI	Wake-Up Interrupt	0 (reset)	0 (reset)
	IR.3	DOI	Data Overrun Interrupt	0 (reset)	0 (reset)
	IR.2	EI	Error Interrupt	0 (reset)	X; note 4
	IR.1	TI	Transmit Interrupt	0 (reset)	0 (reset)
	IR.0	RI	Receive Interrupt	0 (reset)	0 (reset)

REGISTER	BIT	SYMBOL	NAME	VALUE	
				RESET BY HARDWARE	SETTING BIT CR.0 BY SOFTWARE OR DUE TO BUS-OFF
Acceptance code	AC.7 to 0	AC	Acceptance Code	X	X
Acceptance mask	AM.7 to 0	AM	Acceptance Mask	X	X
Bus timing 0	BTR0.7	SJW.1	Synchronization Jump Width 1	X	X
	BTR0.6	SJW.0	Synchronization Jump Width 0	X	X
	BTR0.5	BRP.5	Baud Rate Prescaler 5	X	X
	BTR0.4	BRP.4	Baud Rate Prescaler 4	X	X
	BTR0.3	BRP.3	Baud Rate Prescaler 3	X	X
	BTR0.2	BRP.2	Baud Rate Prescaler 2	X	X
	BTR0.1	BRP.1	Baud Rate Prescaler 1	X	X
	BTR0.0	BRP.0	Baud Rate Prescaler 0	X	X
Bus timing 1	BTR1.7	SAM	Sampling	X	X
	BTR1.6	TSEG2.2	Time Segment 2.2	X	X
	BTR1.5	TSEG2.1	Time Segment 2.1	X	X
	BTR1.4	TSEG2.0	Time Segment 2.0	X	X
	BTR1.3	TSEG1.3	Time Segment 1.3	X	X
	BTR1.2	TSEG1.2	Time Segment 1.2	X	X
	BTR1.1	TSEG1.1	Time Segment 1.1	X	X
	BTR1.0	TSEG1.0	Time Segment 1.0	X	X
Output control	OC.7	OCTP1	Output Control Transistor P1	X	X
	OC.6	OCTN1	Output Control Transistor N1	X	X
	OC.5	OCPOL1	Output Control Polarity 1	X	X
	OC.4	OCTP0	Output Control Transistor P0	X	X
	OC.3	OCTN0	Output Control Transistor N0	X	X
	OC.2	OCPOL0	Output Control Polarity 0	X	X
	OC.1	OCMODE1	Output Control Mode 1	X	X
	OC.0	OCMODE0	Output Control Mode 0	X	X
Transmit buffer	–	TXB	Transmit Buffer	X	X
Receive buffer	–	RXB	Receive Buffer	X; note 5	X; note 5
Clock divider	–	CDR	Clock Divider Register	00000000 (Intel); 00000101 (Motorola)	X

Notes

1. X means that the value of these registers or bits is not influenced.
2. Remarks in brackets explain functional meaning.
3. Reading the command register will always reflect a binary '11111111'.
4. On bus-off the error interrupt is set, if enabled.
5. Internal read/write pointers of the RXFIFO are reset to their initial values. A subsequent read access to the RXB would show undefined data values (parts of old messages). If a message is transmitted, this message is written in parallel to the receive buffer but no receive interrupt is generated and the receive buffer area is not locked. So, even if the receive buffer is empty, the last transmitted message may be read from the receive buffer until it is overridden by the next received or transmitted message.
   Upon a hardware reset, the RXFIFO pointers are reset to the physical RAM address '0'. Setting CR.0 by software or due to the bus-off event will reset the RXFIFO pointers to the currently valid FIFO start address which is different from the RAM address '0' after the first release receive buffer command.

6.3.3 Control Register (CR)

The contents of the control register are used to change the behaviour of the CAN controller. Bits may be set or reset by the attached microcontroller which uses the control register as a read/write memory.

Table 3 Bit interpretation of the control register (CR); CAN address 0

BIT	SYMBOL	NAME	VALUE	FUNCTION
CR.7	–	–	–	reserved; note 1
CR.6	–	–	–	reserved; note 2
CR.5	–	–	–	reserved; note 3
CR.4	OIE	Overrun Interrupt Enable	1	enabled; if the data overrun bit is set, the microcontroller receives an overrun interrupt signal (see also status register; Table 5)
			0	disabled; the microcontroller receives no overrun interrupt signal from the SJA1000
CR.3	EIE	Error Interrupt Enable	1	enabled; if the error or bus status change, the microcontroller receives an error interrupt signal (see also status register; Table 5)
			0	disabled; the microcontroller receives no error interrupt signal from the SJA1000
CR.2	TIE	Transmit Interrupt Enable	1	enabled; when a message has been successfully transmitted or the transmit buffer is accessible again, (e.g. after an abort transmission command) the SJA1000 transmits a transmit interrupt signal to the microcontroller
			0	disabled; the microcontroller receives no transmit interrupt signal from the SJA1000
CR.1	RIE	Receive Interrupt Enable	1	enabled; when a message has been received without errors, the SJA1000 transmits a receive interrupt signal to the microcontroller
			0	disabled; the microcontroller receives no transmit interrupt signal from the SJA1000

BIT	SYMBOL	NAME	VALUE	FUNCTION
CR.0	RR	Reset Request; note 4	1	present; detection of a reset request results in aborting the current transmission/reception of a message and entering the reset mode
			0	absent; on the '1-to-0' transition of the reset request bit, the SJA1000 returns to the operating mode

Notes

1. Any write access to the control register has to set this bit to logic 0 (reset value is logic 0).
2. In the PCA82C200 this bit was used to select the synchronization mode. Because this mode is not longer implemented, setting this bit has no influence on the microcontroller. Due to software compatibility setting this bit is allowed. This bit will not change after hardware or software reset. In addition the value written by users software is reflected.
3. Reading this bit will always reflect a logic 1.
4. During a hardware reset or when the bus status bit is set to logic 1 (bus-off), the reset request bit is set to logic 1 (present). If this bit is accessed by software, a value change will become visible and takes effect first with the next positive edge of the internal clock which operates with ½ of the external oscillator frequency. During an external reset the microcontroller cannot set the reset request bit to logic 0 (absent). Therefore, after having set the reset request bit to logic 0, the microcontroller must check this bit to ensure that the external reset pin is not being held HIGH. Changes of the reset request bit are synchronized with the internal divided clock. Reading the reset request bit reflects the synchronized status.
   After the reset request bit is set to logic 0 the SJA1000 will wait for:
   a) One occurrence of bus-free signal (11 recessive bits), if the preceding reset request has been caused by a hardware reset or a CPU-initiated reset
   b) 128 occurrences of bus-free, if the preceding reset request has been caused by a CAN controller initiated bus-off, before re-entering the bus-on mode; it should be noted that several registers are modified if the reset request bit was set (see also Table 2).

6.3.4 Command Register (CMR)

A command bit initiates an action within the transfer layer of the SJA1000. The command register appears to the microcontroller as a write only memory. If a read access is performed to this address the byte '11111111' is returned. Between two commands at least one internal clock cycle is needed to process. The internal clock is divided by two from the external oscillator frequency.

Table 4 Bit interpretation of the command register (CMR); CAN address 1

BIT	SYMBOL	NAME	VALUE	FUNCTION
CMR.7	–	–	–	reserved
CMR.6	–	–	–	reserved
CMR.5	–	–	–	reserved
CMR.4	GTS	Go To Sleep; note 1	1	sleep; the SJA1000 enters sleep mode if no CAN interrupt is pending and there is no bus activity
			0	wake up; SJA1000 operates normal
CMR.3	CDO	Clear Data Overrun; note 2	1	clear; data overrun status bit is cleared
			0	no action
CMR.2	RRB	Release Receive Buffer; note 3	1	released; the receive buffer, representing the message memory space in the RXFIFO is released
			0	no action
CMR.1	AT	Abort Transmission; note 4	1	present; if not already in progress, a pending transmission request is cancelled
			0	absent; no action
CMR.0	TR	Transmission Request; note 5	1	present; a message will be transmitted
			0	absent; no action

Notes

1. The SJA1000 will enter sleep mode if the sleep bit is set to logic 1 (sleep); there is no bus activity and no interrupt is pending. Setting of GTS with at least one of the previously mentioned exceptions valid will result in a wake-up interrupt. After sleep mode is set, the CLKOUT signal continues until at least 15 bit times have passed, to allow a host microcontroller clocked via this signal to enter its own standby mode before the CLKOUT goes LOW. The SJA1000 will wake up when one of the three previously mentioned conditions is negated: after 'Go To Sleep' is set LOW (wake-up), there is bus activity or $\overline{INT}$ is driven LOW (active). On wake-up, the oscillator is started and a wake-up interrupt is generated. A sleeping SJA1000 which wakes up due to bus activity will not be able to receive this message until it detects 11 consecutive recessive bits (bus-free sequence). It should be noted that setting of GTS is not possible in reset mode. After clearing of reset request, setting of GTS is possible first, when bus-free is detected again.
2. This command bit is used to clear the data overrun condition indicated by the data overrun status bit. As long as the data overrun status bit is set no further data overrun interrupt is generated. It is allowed to give the clear data overrun command at the same time as a release receive buffer command.
3. After reading the contents of the receive buffer, the microcontroller can release this memory space of the RXFIFO by setting the release receive buffer bit to logic 1. This may result in another message becoming immediately available within the receive buffer. This event will force another receive interrupt, if enabled. If there is no other message available no further receive interrupt is generated and the receive buffer status bit is cleared.
4. The abort transmission bit is used when the CPU requires the suspension of the previously requested transmission, e.g. to transmit a more urgent message before. A transmission already in progress is not stopped. In order to see if the original message had been either transmitted successfully or aborted, the transmission complete status bit should be checked. This should be done after the transmit buffer status bit has been set to logic 1 (released) or a transmit interrupt has been generated.
5. If the transmission request was set to logic 1 in a previous command, it cannot be cancelled by setting the transmission request bit to logic 0. The requested transmission may be cancelled by setting the abort transmission bit to logic 1.

6.3.5 STATUS REGISTER (SR)

The content of the status register reflects the status of the SJA1000. The status register appears to the microcontroller as a read only memory.

Table 5 Bit interpretation of the status register (SR); CAN address 2

BIT	SYMBOL	NAME	VALUE	FUNCTION
SR.7	BS	Bus Status; note 1	1	bus-off; the SJA1000 is not involved in bus activities
			0	bus-on; the SJA1000 is involved in bus activities
SR.6	ES	Error Status; note 2	1	error; at least one of the error counters has reached or exceeded the CPU warning limit
			0	ok; both error counters are below the warning limit
SR.5	TS	Transmit Status; note 3	1	transmit; the SJA1000 is transmitting a message
			0	idle; no transmit message is in progress
SR.4	RS	Receive Status; note 3	1	receive; the SJA1000 is receiving a message
			0	idle; no receive message is in progress
SR.3	TCS	Transmission Complete Status; note 4	1	complete; the last requested transmission has been successfully completed
			0	incomplete; the previously requested transmission is not yet completed
SR.2	TBS	Transmit Buffer Status; note 5	1	released; the CPU may write a message into the transmit buffer
			0	locked; the CPU cannot access the transmit buffer; a message is waiting for transmission or is already in process
SR.1	DOS	Data Overrun Status; note 6	1	overrun; a message was lost because there was not enough space for that message in the RXFIFO
			0	absent; no data overrun has occurred since the last clear data overrun command was given
SR.0	RBS	Receive Buffer Status; note 7	1	full; one or more messages are available in the RXFIFO
			0	empty; no message is available

Notes

1. When the transmit error counter exceeds the limit of 255 [the bus status bit is set to logic 1 (bus-off)] the CAN controller will set the reset request bit to logic 1 (present) and an error interrupt is generated, if enabled. It will stay in this mode until the CPU clears the reset request bit. Once this is completed the CAN controller will wait the minimum protocol-defined time (128 occurrences of the bus-free signal). After that the bus status bit is cleared (bus-on), the error status bit is set to logic 0 (ok), the error counters are reset and an error interrupt is generated, if enabled.
2. Errors detected during reception or transmission will affect the error counters according to the CAN 2.0B protocol specification. The error status bit is set when at least one of the error counters has reached or exceeded the CPU warning limit of 96. An error interrupt is generated, if enabled.
3. If both the receive status and the transmit status bits are logic 0 (idle) the CAN-bus is idle.
4. The transmission complete status bit is set to logic 0 (incomplete) whenever the transmission request bit is set to logic 1. The transmission complete status bit will remain at logic 0 (incomplete) until a message is transmitted successfully.
5. If the CPU tries to write to the transmit buffer when the transmit buffer status bit is at logic 0 (locked), the written byte will not be accepted and will be lost without being indicated.
6. When a message that shall be received has passed the acceptance filter successfully (i.e. earliest after arbitration field), the CAN controller needs space in the RXFIFO to store the message descriptor. Accordingly there must be enough space for each data byte which has been received. If there is not enough space to store the message, that message will be dropped and the data overrun condition will be indicated to the CPU only, if this received message has no errors until the last but one bit of end of frame (message becomes valid).
7. After reading a message stored in the RXFIFO and releasing this memory space with the command release receive buffer, this bit is cleared. If there is another message available within the FIFO this bit is set again with the next bit quantum (t_{scl}).

6.3.6 Interrupt Register (IR)

The interrupt register allows the identification of an interrupt source. When one or more bits of this register are set, the $\overline{INT}$ pin is activated (LOW). After this register is read by the microcontroller, all bits are reset what results in a floating level at $\overline{INT}$. The interrupt register appears to the microcontroller as a read only memory.

Table 6 Bit interpretation of the interrupt register (IR); CAN address 3

BIT	SYMBOL	NAME	VALUE	FUNCTION
IR.7	–	–	–	reserved
IR.6	–	–	–	reserved
IR.5	–	–	–	reserved
IR.4	WUI	Wake-Up Interrupt; note 1	1	set; this bit is set when the sleep mode is left
			0	reset; this bit is cleared by any read access of the microcontroller
IR.3	DOI	Data Overrun Interrupt; note 2	1	set; this bit is set on a '0-to-1' transition of the data overrun status bit, when the data overrun interrupt enable is set to logic 1 (enabled)
			0	reset; this bit is cleared by any read access of the microcontroller
IR.2	EI	Error Interrupt	1	set; this bit is set on a change of either the error status or bus status bits if the error interrupt enable is set to logic 1 (enabled)
			0	reset; this bit is cleared by any read access of the microcontroller
IR.1	TI	Transmit Interrupt	1	set; this bit is set whenever the transmit buffer status changes from logic 0 to logic 1 (released) and transmit interrupt enable is set to logic 1 (enabled)
			0	reset; this bit is cleared by any read access of the microcontroller
IR.0	RI	Receive Interrupt; note 3	1	set; this bit is set while the receive FIFO is not empty and the receive interrupt enable bit is set to logic 1 (enabled)
			0	reset; this bit is cleared by any read access of the microcontroller

Notes

1. A wake-up interrupt is also generated if the CPU tries to set go to sleep while the CAN controller is involved in bus activities or a CAN interrupt is pending.
2. The overrun interrupt bit (if enabled) and the data overrun status bit are set at the same time.
3. The receive interrupt bit (if enabled) and the receive buffer status bit are set at the same time.
   It should be noted that the receive interrupt bit is cleared upon a read access, even if there is another message available within the FIFO. The moment the release receive buffer command is given and there is another message valid within the receive buffer, the receive interrupt is set again (if enabled) with the next t_{scl}.

6.3.7 TRANSMIT BUFFER LAYOUT

The global layout of the transmit buffer is shown in Table 7. The buffer serves to store a message from the microcontroller to be transmitted by the SJA1000. It is subdivided into a descriptor and data field. The transmit buffer can be written to and read out by the microcontroller in operating mode only. In reset mode a 'FFH' is reflected for all bytes.

Table 7 Layout of transmit buffer

CAN ADDRESS	FIELD	NAME	BITS							
			7	6	5	4	3	2	1	0
10	descriptor	identifier byte 1	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
11		identifier byte 2	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
12	data	TX data 1	transmit data byte 1							
13		TX data 2	transmit data byte 2							
14		TX data 3	transmit data byte 3							
15		TX data 4	transmit data byte 4							
16		TX data 5	transmit data byte 5							
17		TX data 6	transmit data byte 6							
18		TX data 7	transmit data byte 7							
19		TX data 8	transmit data byte 8							

6.3.7.1 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. It is used in a receiver for acceptance filtering and also determining the bus access priority during the arbitration process. The lower the binary value of the identifier the higher the priority. This is due to a larger number of leading dominant bits during arbitration.

6.3.7.2 Remote Transmission Request (RTR)

If this bit is set, a remote frame will be transmitted via the bus. This means that no data bytes are included within this frame. Nevertheless, it is necessary to specify the correct data length code which depends on the corresponding data frame with the same identifier coding.

If the RTR bit is not set, a data frame will be sent including the number of data bytes as specified by the data length code.

6.3.7.3 Data Length Code (DLC)

The number of bytes 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 at logic 1 (remote). This forces the number of transmitted/received data bytes to be logic 0. Nevertheless, the data length code must be specified correctly to avoid bus errors if two CAN controllers start a remote frame transmission with the same identifier simultaneously.

The range of the data byte count is 0 to 8 bytes and is coded as follows:

DataByteCount = 8 × DLC.3 + 4 × DLC.2 + 2 × DLC.1 + DLC.0

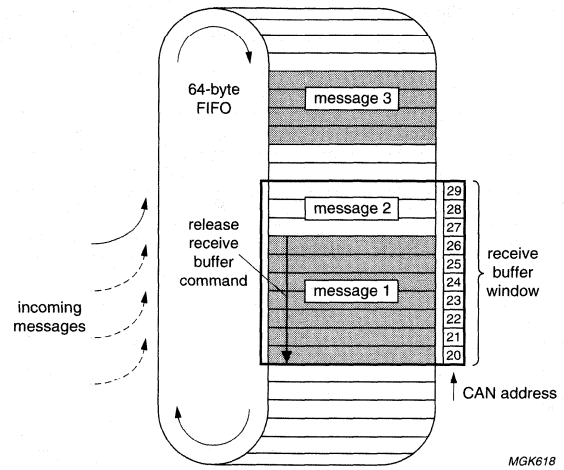
For reasons of compatibility no data length code >8 should be used. If a value >8 is selected, 8 bytes are transmitted in the data frame with the data length code specified in DLC.

6.3.7.4 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.

6.3.8 RECEIVE BUFFER

The global layout of the receive buffer is very similar to the transmit buffer described in Section 6.3.7. The receive buffer is the accessible part of the RXFIFO and is located in the range between CAN address 20 and 29.



Message 1 is now available in the receive buffer.

Fig.4 Example of the message storage within the RXFIFO.

Identifier, remote transmission request bit and data length code have the same meaning and location as described in the transmit buffer but within the address range 20 to 29.

As illustrated in Fig.4 the RXFIFO has space for 64 message bytes in total. The number of messages that can be stored in the FIFO at any particular moment depends on the length of the individual messages. If there is not enough space for a new message within the RXFIFO, the CAN controller generates a data overrun condition. A message which is partly written into the RXFIFO, when the data overrun condition occurs, is deleted. This situation is indicated to the microcontroller via the status register and the data overrun interrupt, if enabled and the frame was received without any errors until the last but one bit of end of frame (RX message becomes valid).

6.3.9 ACCEPTANCE FILTER

With the help of the acceptance filter the CAN controller is able to allow passing of received messages to the RXFIFO only when the identifier bits of the received message are equal to the predefined ones within the acceptance filter registers. The acceptance filter is defined by the acceptance code register (ACR; see Section 6.3.9.1) and the acceptance mask register (AMR; see Section 6.3.9.2).

6.3.9.1 Acceptance Code Register (ACR)

Table 8 ACR bit allocation; can address 4

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
AC.7	AC.6	AC.5	AC.4	AC.3	AC.2	AC.1	AC.0

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 there is receive buffer space left, then the respective descriptor and data field are sequentially stored in the RXFIFO. When the complete message has been correctly received the following occurs:

- The receive 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 to AC.0) and the eight most significant bits of the message's identifier (ID.10 to ID.3) must be equal to those bit positions which are marked relevant by the acceptance mask bits (AM.7 to AM.0). If the conditions as described in the following equation are fulfilled, acceptance is given:

(ID.10 to ID.3) ≡ (AC.7 to AC.0)] ∨ (AM.7 to AM.0) ≡ 11111111

6.3.9.2 Acceptance Mask Register (AMR)

Table 9 AMR bit allocation; CAN address 5

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
AM.7	AM.6	AM.5	AM.4	AM.3	AM.2	AM.1	AM.0

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' (AM.X = 0) or 'don't care' (AM.X = 1) for acceptance filtering.

6.3.9.3 Other registers

The other registers are described in Section 6.5.

6.4 PeliCAN mode

6.4.1 PeliCAN address layout

The CAN controller's internal registers appear to the CPU as on-chip memory mapped peripheral registers. Because the CAN controller can operate in different modes (operating/reset; see also Section 6.4.3), one has to distinguish between different internal address definitions.

Starting from CAN address 32 the complete internal RAM (80-byte) is mapped to the CPU interface.

Table 10 PeliCAN address allocation; note 1

CAN ADDRESS	OPERATING MODE				RESET MODE	
	READ		WRITE		READ	WRITE
0	mode		mode		mode	mode
1	(00H)		command		(00H)	command
2	status		–		status	–
3	interrupt		–		interrupt	–
4	interrupt enable		interrupt enable		interrupt enable	interrupt enable
5	reserved (00H)		–		reserved (00H)	–
6	bus timing 0		–		bus timing 0	bus timing 0
7	bus timing 1		–		bus timing 1	bus timing 1
8	output control		–		output control	output control
9	test		test		test	test
10	reserved (00H)		–		reserved (00H)	–
11	arbitration lost capture		–		arbitration lost capture	–
12	error code capture		–		error code capture	–
13	error warning limit		–		error warning limit	error warning limit
14	RX error counter		–		RX error counter	RX error counter
15	TX error counter		–		TX error counter	TX error counter
16	RX frame information SFF; note 2	RX frame information EFF; note 3	TX frame information SFF; note 2	TX frame information EFF; note 3	acceptance code 0	acceptance code 0
17	RX identifier 1	RX identifier 1	TX identifier 1	TX identifier 1	acceptance code 1	acceptance code 1
18	RX identifier 2	RX identifier 2	TX identifier 2	TX identifier 2	acceptance code 2	acceptance code 2
19	RX data 1	RX identifier 3	TX data 1	TX identifier 3	acceptance code 3	acceptance code 3
20	RX data 2	RX identifier 4	TX data 2	TX identifier 4	acceptance mask 0	acceptance mask 0
21	RX data 3	RX data 1	TX data 3	TX data 1	acceptance mask 1	acceptance mask 1
22	RX data 4	RX data 2	TX data 4	TX data 2	acceptance mask 2	acceptance mask 2
23	RX data 5	RX data 3	TX data 5	TX data 3	acceptance mask 3	acceptance mask 3
24	RX data 6	RX data 4	TX data 6	TX data 4	reserved (00H)	–
25	RX data 7	RX data 5	TX data 7	TX data 5	reserved (00H)	–
26	RX data 8	RX data 6	TX data 8	TX data 6	reserved (00H)	–

CAN ADDRESS	OPERATING MODE				RESET MODE	
	READ		WRITE		READ	WRITE
27	(FIFO RAM); note 4	RX data 7	–	TX data 7	reserved (00H)	–
28	(FIFO RAM); note 4	RX data 8	–	TX data 8	reserved (00H)	–
29	RX message counter		–		RX message counter	–
30	RX buffer start address		–		RX buffer start address	RX buffer start address
31	clock divider		clock divider; note 5		clock divider	clock divider
32	internal RAM address 0 (FIFO)		–		internal RAM address 0	internal RAM address 0
33	internal RAM address 1 (FIFO)		–		internal RAM address 1	internal RAM address 1
↓	↓		↓		↓	↓
95	internal RAM address 63 (FIFO)		–		internal RAM address 63	internal RAM address 63
96	internal RAM address 64 (TX buffer)		–		internal RAM address 64	internal RAM address 64
↓	↓		↓		↓	↓
108	internal RAM address 76 (TX buffer)		–		internal RAM address 76	internal RAM address 76
109	internal RAM address 77 (free)		–		internal RAM address 77	internal RAM address 77
110	internal RAM address 78 (free)		–		internal RAM address 78	internal RAM address 78
111	internal RAM address 79 (free)		–		internal RAM address 79	internal RAM address 79
112	(00H)		–		(00H)	–
↓	↓		↓		↓	↓
127	(00H)		–		(00H)	–

Notes

1. It should be noted that the registers are repeated within higher CAN address areas (the most significant bit of the 8-bit CPU address is not decoded: CAN address 128 continues with CAN address 0 and so on).
2. SFF = Standard Frame Format.
3. EFF = Extended Frame Format.
4. These address allocations reflect the FIFO RAM space behind the current message. The contents are random after power-up and contain the beginning of the next message which is received after the current one. If no further message is received, parts of old messages may occur here.
5. Some bits are writeable in reset mode only (CAN mode, CBP, RXINTEN and clock off).

6.4.2 RESET VALUES

Detection of a set reset mode bit results in aborting the current transmission/reception of a message and entering the reset mode. On the '1-to-0' transition of the reset mode bit, the CAN controller returns to the mode defined within the mode register.

Table 11 Reset mode configuration; notes 1 and 2

REGISTER	BIT	SYMBOL	NAME	VALUE	
				RESET BY HARDWARE	SETTING MOD.0 BY SOFTWARE OR DUE TO BUS-OFF
Mode	MOD.7 to 5	–	reserved	0 (reserved)	0 (reserved)
	MOD.4	SM	Sleep Mode	0 (wake-up)	0 (wake-up)
	MOD.3	AFM	Acceptance Filter Mode	0 (dual)	X
	MOD.2	STM	Self Test Mode	0 (normal)	X
	MOD.1	LOM	Listen Only Mode	0 (normal)	X
	MOD.0	RM	Reset Mode	1 (present)	1 (present)
Command	CMR.7 to 5	–	reserved	0 (reserved)	0 (reserved)
	CMR.4	SRR	Self Reception Request	0 (absent)	0 (absent)
	CMR.3	CDO	Clear Data Overrun	0 (no action)	0 (no action)
	CMR.2	RRB	Release Receive Buffer	0 (no action)	0 (no action)
	CMR.1	AT	Abort Transmission	0 (absent)	0 (absent)
	CMR.0	TR	Transmission Request	0 (absent)	0 (absent)
Status	SR.7	BS	Bus Status	0 (bus-on)	X
	SR.6	ES	Error Status	0 (ok)	X
	SR.5	TS	Transmit Status	1 (wait idle)	1 (wait idle)
	SR.4	RS	Receive Status	1 (wait idle)	1 (wait idle)
	SR.3	TCS	Transmission Complete Status	1 (complete)	X
	SR.2	TBS	Transmit Buffer Status	1 (released)	1 (released)
	SR.1	DOS	Data Overrun Status	0 (absent)	0 (absent)
	SR.0	RBS	Receive Buffer Status	0 (empty)	0 (empty)
Interrupt	IR.7	BEI	Bus Error Interrupt	0 (reset)	0 (reset)
	IR.6	ALI	Arbitration Lost Interrupt	0 (reset)	0 (reset)
	IR.5	EPI	Error Passive Interrupt	0 (reset)	0 (reset)
	IR.4	WUI	Wake-Up Interrupt	0 (reset)	0 (reset)
	IR.3	DOI	Data Overrun Interrupt	0 (reset)	0 (reset)
	IR.2	EI	Error Warning Interrupt	0 (reset)	X; note 3
	IR.1	TI	Transmit Interrupt	0 (reset)	0 (reset)
	IR.0	RI	Receive Interrupt	0 (reset)	0 (reset)

REGISTER	BIT	SYMBOL	NAME	VALUE	
				RESET BY HARDWARE	SETTING MOD.0 BY SOFTWARE OR DUE TO BUS-OFF
Interrupt enable	IER.7	BEIE	Bus Error Interrupt Enable	X	X
	IER.6	ALIE	Arbitration Lost Interrupt Enable	X	X
	IER.5	EPIE	Error Passive Interrupt Enable	X	X
	IER.4	WUIE	Wake-Up Interrupt Enable	X	X
	IER.3	DOIE	Data Overrun Interrupt Enable	X	X
	IER.2	EIE	Error Warning Interrupt Enable	X	X
	IER.1	TIE	Transmit Interrupt Enable	X	X
	IER.0	RIE	Receive Interrupt Enable	X	X
Bus timing 0	BTR0.7	SJW.1	Synchronization Jump Width 1	X	X
	BTR0.6	SJW.0	Synchronization Jump Width 0	X	X
	BTR0.5	BRP.5	Baud Rate Prescaler 5	X	X
	BTR0.4	BRP.4	Baud Rate Prescaler 4	X	X
	BTR0.3	BRP.3	Baud Rate Prescaler 3	X	X
	BTR0.2	BRP.2	Baud Rate Prescaler 2	X	X
	BTR0.1	BRP.1	Baud Rate Prescaler 1	X	X
	BTR0.0	BRP.0	Baud Rate Prescaler 0	X	X
Bus timing 1	BTR1.7	SAM	Sampling	X	X
	BTR1.6	TSEG2.2	Time Segment 2.2	X	X
	BTR1.5	TSEG2.1	Time Segment 2.1	X	X
	BTR1.4	TSEG2.0	Time Segment 2.0	X	X
	BTR1.3	TSEG1.3	Time Segment 1.3	X	X
	BTR1.2	TSEG1.2	Time Segment 1.2	X	X
	BTR1.1	TSEG1.1	Time Segment 1.1	X	X
	BTR1.0	TSEG1.0	Time Segment 1.0	X	X

REGISTER	BIT	SYMBOL	NAME	VALUE	
				RESET BY HARDWARE	SETTING MOD.0 BY SOFTWARE OR DUE TO BUS-OFF
Output control	OCR.7	OCTP1	Output Control Transistor P1	X	X
	OCR.6	OCTN1	Output Control Transistor N1	X	X
	OCR.5	OCPOL1	Output Control Polarity 1	X	X
	OCR.4	OCTP0	Output Control Transistor P0	X	X
	OCR.3	OCTN0	Output Control Transistor N0	X	X
	OCR.2	OCPOL0	Output Control Polarity 0	X	X
	OCR.1	OCMODE1	Output Control Mode 1	X	X
	OCR.0	OCMODE0	Output Control Mode 0	X	X
Arbitration lost capture	–	ALC	Arbitration Lost Capture	0	X
Error code capture	–	ECC	Error Code Capture	0	X
Error warning limit	–	EWLR	Error Warning Limit Register	96	X
RX error counter	–	RXERR	Receive Error Counter	0 (reset)	X; note 4
TX error counter	–	TXERR	Transmit Error Counter	0 (reset)	X; note 4
TX buffer	–	TXB	Transmit Buffer	X	X
RX buffer	–	RXB	Receive Buffer	X; note 5	X; note 5
ACR 0 to 3	–	ACR0 to ACR3	Acceptance Code Registers	X	X
AMR 0 to 3	–	AMR0 to AMR3	Acceptance Mask Registers	X	X
RX message counter	–	RMC	RX Message Counter	0	0
RX buffer start address	–	RBSA	RX Buffer Start Address	00000000	X
Clock divider	–	CDR	Clock Divider Register	00000000 Intel; 00000101 Motorola	X

Notes

1. X means that the value of these registers or bits is not influenced.
2. Remarks in brackets explain functional meaning.
3. On bus-off the error warning interrupt is set, if enabled.
4. If the reset mode was entered due to a bus-off condition, the receive error counter is cleared and the transmit error counter is initialized to 127 to count-down the CAN-defined bus-off recovery time consisting of 128 occurrences of 11 consecutive recessive bits.
5. Internal read/write pointers of the RXFIFO are reset to their initial values. A subsequent read access to the RXB would show undefined data values (parts of old messages).
   If a message is transmitted, this message is written in parallel to the receive buffer. A receive interrupt is generated only if this transmission was forced by the self reception request. So, even if the receive buffer is empty, the last transmitted message may be read from the receive buffer until it is overwritten by the next received or transmitted message.
   Upon a hardware reset, the RXFIFO pointers are reset to the physical RAM address '0'. Setting CR.0 by software or due to the bus-off event will reset the RXFIFO pointers to the currently valid FIFO start address (RBSA register) which is different from the RAM address '0' after the first release receive buffer command.

6.4.3 Mode Register (MOD)

The contents of the mode register are used to change the behaviour of the CAN controller. Bits may be set or reset by the CPU which uses the control register as a read/write memory. Reserved bits are read as logic 0.

Table 12 Bit interpretation of the mode register (MOD); CAN address '0'

BIT	SYMBOL	NAME	VALUE	FUNCTION
MOD.7	–	–	–	reserved
MOD.6	–	–	–	reserved
MOD.5	–	–	–	reserved
MOD.4	SM	Sleep Mode; note 1	1	sleep; the CAN controller enters sleep mode if no CAN interrupt is pending and if there is no bus activity
			0	wake-up; the CAN controller wakes up if sleeping
MOD.3	AFM	Acceptance Filter Mode; note 2	1	single; the single acceptance filter option is enabled (one filter with the length of 32 bit is active)
			0	dual; the dual acceptance filter option is enabled (two filters, each with the length of 16 bit are active)
MOD.2	STM	Self Test Mode; note 2	1	self test; in this mode a full node test is possible without any other active node on the bus using the self reception request command; the CAN controller will perform a successful transmission, even if there is no acknowledge received
			0	normal; an acknowledge is required for successful transmission

BIT	SYMBOL	NAME	VALUE	FUNCTION
MOD.1	LOM	Listen Only Mode; notes 2 and 3	1	listen only; in this mode the CAN would give no acknowledge to the CAN-bus, even if a message is received successfully
			0	normal; the error counters are stopped at the current value
MOD.0	RM	Reset Mode; note 4	1	reset; detection of a set reset mode bit results in aborting the current transmission/reception of a message and entering the reset mode
			0	normal; on the '1-to-0' transition of the reset mode bit, the CAN controller returns to the operating mode

Notes

1. The SJA1000 will enter sleep mode if the sleep mode bit is set to logic 1 (sleep); then there is no bus activity and no interrupt is pending. Setting of SM with at least one of the previously mentioned exceptions valid will result in a wake-up interrupt. After sleep mode is set, the CLKOUT signal continues until at least 15 bit times have passed, to allow a host microcontroller clocked via this signal to enter its own standby mode before the CLKOUT goes LOW. The SJA1000 will wake up when one of the three previously mentioned conditions is negated: after SM is set LOW (wake-up), there is bus activity or $\overline{INT}$ is driven LOW (active). On wake-up, the oscillator is started and a wake-up interrupt is generated. A sleeping SJA1000 which wakes up due to bus activity will not be able to receive this message until it detects 11 consecutive recessive bits (bus-free sequence). It should be noted that setting of SM is not possible in reset mode. After clearing of reset mode, setting of SM is possible first, when bus-free is detected again.
2. A write access to the bits MOD.1 to MOD.3 is only possible, if the reset mode is entered previously.
3. This mode of operation forces the CAN controller to be error passive. Message transmission is not possible. The listen only mode can be used e.g. for software driven bit rate detection and 'hot plugging'.
4. During a hardware reset or when the bus status bit is set to logic 1 (bus-off), the reset mode bit is also set to logic 1 (present). If this bit is accessed by software, a value change will become visible and takes effect first with the next positive edge of the internal clock which operates at half of the external oscillator frequency. During an external reset the microcontroller cannot set the reset mode bit to logic 0 (absent). Therefore, after having set the reset mode bit to logic 1, the microcontroller must check this bit to ensure that the external reset pin is not being held HIGH. Changes of the reset request bit are synchronized with the internal divided clock. Reading the reset request bit reflects the synchronized status. After the reset mode bit is set to logic 0 the CAN controller will wait for:
   a) One occurrence of bus-free signal (11 recessive bits), if the preceding reset has been caused by a hardware reset or a CPU-initiated reset.
   b) 128 occurrences of bus-free, if the preceding reset has been caused by a CAN controller initiated bus-off, before re-entering the bus-on mode.

6.4.4 COMMAND REGISTER (CMR)

A command bit initiates an action within the transfer layer of the CAN controller. This register is write only, all bits will return a logic 0 when being read. Between two commands at least one internal clock cycle is needed in order to proceed. The internal clock is half of the external oscillator frequency.

Table 13 Bit interpretation of the command register (CMR); CAN address 1

BIT	SYMBOL	NAME	VALUE	FUNCTION
CMR.7	–	reserved	–	–
CMR.6	–	reserved	–	–
CMR.5	–	reserved	–	–
CMR.4	SRR	Self Reception Request; notes 1 and 2	1	present; a message shall be transmitted and received simultaneously
			0	– (absent)
CMR.3	CDO	Clear Data Overrun; note 3	1	clear; the data overrun status bit is cleared
			0	– (no action)
CMR.2	RRB	Release Receive Buffer; note 4	1	released; the receive buffer, representing the message memory space in the RXFIFO is released
			0	– (no action)
CMR.1	AT	Abort Transmission; notes 5 and 2	1	present; if not already in progress, a pending transmission request is cancelled
			0	– (absent)
CMR.0	TR	Transmission Request; notes 6 and 2	1	present; a message shall be transmitted
			0	– (absent)

Notes

1. Upon self reception request a message is transmitted and simultaneously received if the acceptance filter is set to the corresponding identifier. A receive and a transmit interrupt will indicate correct self reception (see also self test mode in mode register).
2. Setting the command bits CMR.0 and CMR.1 simultaneously results in sending the transmit message once. No re-transmission will be performed in the event of an error or arbitration lost (single-shot transmission).
   Setting the command bits CMR.4 and CMR.1 simultaneously results in sending the transmit message once using the self reception feature. No re-transmission will be performed in the event of an error or arbitration lost.
   Setting the command bits CMR.0, CMR.1 and CMR.4 simultaneously results in sending the transmit message once as described for CMR.0 and CMR.1.
   The moment the transmit status bit is set within the status register, the internal transmission request bit is cleared automatically.
   Setting CMR.0 and CMR.4 simultaneously will ignore the set CMR.4 bit.
3. This command bit is used to clear the data overrun condition indicated by the data overrun status bit. As long as the data overrun status bit is set no further data overrun interrupt is generated.
4. After reading the contents of the receive buffer, the CPU can release this memory space in the RXFIFO by setting the release receive buffer bit to logic 1. This may result in another message becoming immediately available within the receive buffer. If there is no other message available, the receive interrupt bit is reset.

5. The abort transmission bit is used when the CPU requires the suspension of the previously requested transmission, e.g. to transmit a more urgent message before. A transmission already in progress is not stopped. In order to see if the original message has been either transmitted successfully or aborted, the transmission complete status bit should be checked. This should be done after the transmit buffer status bit has been set to logic 1 or a transmit interrupt has been generated.
   It should be noted that a transmit interrupt is generated even if the message was aborted because the transmit buffer status bit changes to 'released'.
6. If the transmission request was set to logic 1 in a previous command, it cannot be cancelled by setting the transmission request bit to logic 0. The requested transmission may be cancelled by setting the abort transmission bit to logic 1.

6.4.5 Status Register (SR)

The content of the status register reflects the status of the CAN controller. The status register appears to the CPU as a read only memory.

Table 14 Bit interpretation of the status register (SR); CAN address 2

BIT	SYMBOL	NAME	VALUE	FUNCTION
SR.7	BS	Bus Status; note 1	1	bus-off; the CAN controller is not involved in bus activities
			0	bus-on; the CAN controller is involved in bus activities
SR.6	ES	Error Status; note 2	1	error; at least one of the error counters has reached or exceeded the CPU warning limit defined by the Error Warning Limit Register (EWLR)
			0	ok; both error counters are below the warning limit
SR.5	TS	Transmit Status; note 3	1	transmit; the CAN controller is transmitting a message
			0	idle
SR.4	RS	Receive Status; note 3	1	receive; the CAN controller is receiving a message
			0	idle
SR.3	TCS	Transmission Complete Status; note 4	1	complete; last requested transmission has been successfully completed
			0	incomplete; previously requested transmission is not yet completed
SR.2	TBS	Transmit Buffer Status; note 5	1	released; the CPU may write a message into the transmit buffer
			0	locked; the CPU cannot access the transmit buffer; a message is either waiting for transmission or is in the process of being transmitted
SR.1	DOS	Data Overrun Status; note 6	1	overrun; a message was lost because there was not enough space for that message in the RXFIFO
			0	absent; no data overrun has occurred since the last clear data overrun command was given

BIT	SYMBOL	NAME	VALUE	FUNCTION
SR.0	RBS	Receive Buffer Status; note 7	1	full; one or more complete messages are available in the RXFIFO
			0	empty; no message is available

Notes

1. When the transmit error counter exceeds the limit of 255, the bus status bit is set to logic 1 (bus-off), the CAN controller will set the reset mode bit to logic 1 (present) and an error warning interrupt is generated, if enabled. The transmit error counter is set to 127 and the receive error counter is cleared. It will stay in this mode until the CPU clears the reset mode bit. Once this is completed the CAN controller will wait the minimum protocol-defined time (128 occurrences of the bus-free signal) counting down the transmit error counter. After that the bus status bit is cleared (bus-on), the error status bit is set to logic 0 (ok), the error counters are reset and an error warning interrupt is generated, if enabled. Reading the TX error counter during this time gives information about the status of the bus-off recovery.
2. Errors detected during reception or transmission will effect the error counters according to the CAN 2.0B protocol specification. The error status bit is set when at least one of the error counters has reached or exceeded the CPU warning limit (EWLR). An error warning interrupt is generated, if enabled. The default value of EWLR after hardware reset is 96.
3. If both the receive status and the transmit status bits are logic 0 (idle) the CAN-bus is idle. If both bits are set the controller is waiting to become idle again. After a hardware reset 11 consecutive recessive bits have to be detected until the idle status is reached. After bus-off this will take 128 of 11 consecutive recessive bits.
4. The transmission complete status bit is set to logic 0 (incomplete) whenever the transmission request bit or the self reception request bit is set to logic 1. The transmission complete status bit will remain at logic 0 until a message is transmitted successfully.
5. If the CPU tries to write to the transmit buffer when the transmit buffer status bit is logic 0 (locked), the written byte will not be accepted and will be lost without this being indicated.
6. When a message that is to be received has passed the acceptance filter successfully, the CAN controller needs space in the RXFIFO to store the message descriptor and for each data byte which has been received. If there is not enough space to store the message, that message is dropped and the data overrun condition is indicated to the CPU at the moment this message becomes valid. If this message is not completed successfully (e.g. due to an error), no overrun condition is indicated.
7. After reading all messages within the RXFIFO and releasing their memory space with the command release receive buffer this bit is cleared.

6.4.6 Interrupt Register (IR)

The interrupt register allows the identification of an interrupt source. When one or more bits of this register are set, a CAN interrupt will be indicated to the CPU. After this register is read by the CPU all bits are reset except for the receive interrupt bit.

The interrupt register appears to the CPU as a read only memory.

Table 15 Bit interpretation of the interrupt register (IR); CAN address 3

BIT	SYMBOL	NAME	VALUE	FUNCTION
IR.7	BEI	Bus Error Interrupt	1	set; this bit is set when the CAN controller detects an error on the CAN-bus and the BEIE bit is set within the interrupt enable register
			0	reset
IR.6	ALI	Arbitration Lost Interrupt	1	set; this bit is set when the CAN controller lost the arbitration and becomes a receiver and the ALIE bit is set within the interrupt enable register
			0	reset
IR.5	EPI	Error Passive Interrupt	1	set; this bit is set whenever the CAN controller has reached the error passive status (at least one error counter exceeds the protocol-defined level of 127) or if the CAN controller is in the error passive status and enters the error active status again and the EPIE bit is set within the interrupt enable register
			0	reset
IR.4	WUI	Wake-Up Interrupt; note 1	1	set; this bit is set when the CAN controller is sleeping and bus activity is detected and the WUIE bit is set within the interrupt enable register
			0	reset
IR.3	DOI	Data Overrun Interrupt	1	set; this bit is set on a '0-to-1' transition of the data overrun status bit and the DOIE bit is set within the interrupt enable register
			0	reset
IR.2	EI	Error Warning Interrupt	1	set; this bit is set on every change (set and clear) of either the error status or bus status bits and the EIE bit is set within the interrupt enable register
			0	reset
IR.1	TI	Transmit Interrupt	1	set; this bit is set whenever the transmit buffer status changes from '0-to-1' (released) and the TIE bit is set within the interrupt enable register
			0	reset
IR.0	RI	Receive Interrupt; note 2	1	set; this bit is set while the receive FIFO is not empty and the RIE bit is set within the interrupt enable register
			0	reset; no more message is available within the RXFIFO

Notes

1. A wake-up interrupt is also generated, if the CPU tries to set the sleep bit while the CAN controller is involved in bus activities or a CAN interrupt is pending.
2. The behaviour of this bit is equivalent to that of the receive buffer status bit with the exception, that RI depends on the corresponding interrupt enable bit (RIE). So the receive interrupt bit is not cleared upon a read access to the interrupt register. Giving the command 'release receive buffer' will clear RI temporarily. If there is another message available within the FIFO after the release command, RI is set again. Otherwise RI remains cleared.

6.4.7 Interrupt Enable Register (IER)

The register allows to enable different types of interrupt sources which are indicated to the CPU.

The interrupt enable register appears to the CPU as a read/write memory.

Table 16 Bit interpretation of the interrupt enable register (IER); CAN address 4

BIT	SYMBOL	NAME	VALUE	FUNCTION
IER.7	BEIE	Bus Error Interrupt Enable	1	enabled; if an bus error has been detected, the CAN controller requests the respective interrupt
			0	disabled
IER.6	ALIE	Arbitration Lost Interrupt Enable	1	enabled; if the CAN controller has lost arbitration, the respective interrupt is requested
			0	disabled
IER.5	EPIE	Error Passive Interrupt Enable	1	enabled; if the error status of the CAN controller changes from error active to error passive or vice versa, the respective interrupt is requested
			0	disabled
IER.4	WUIE	Wake-Up Interrupt Enable	1	enabled; if the sleeping CAN controller wakes up, the respective interrupt is requested
			0	disabled
IER.3	DOIE	Data Overrun Interrupt Enable	1	enabled; if the data overrun status bit is set (see status register; Table 14), the CAN controller requests the respective interrupt
			0	disabled
IER.2	EIE	Error Warning Interrupt Enable	1	enabled; if the error or bus status change (see status register; Table 14), the CAN controller requests the respective interrupt
			0	disabled
IER.1	TIE	Transmit Interrupt Enable	1	enabled; when a message has been successfully transmitted or the transmit buffer is accessible again (e.g. after an abort transmission command), the CAN controller requests the respective interrupt
			0	disabled
IER.0	RIE	Receive Interrupt Enable; note 1	1	enabled; when the receive buffer status is 'full' the CAN controller requests the respective interrupt
			0	disabled

Note

1. The receive interrupt enable bit has direct influence to the receive interrupt bit and the external interrupt output $\overline{INT}$. If RIE is cleared, the external $\overline{INT}$ pin will become HIGH immediately, if there is no other interrupt pending.

6.4.8 Arbitration Lost Capture register (ALC)

This register contains information about the bit position of losing arbitration. The arbitration lost capture register appears to the CPU as a read only memory. Reserved bits are read as logic 0.

Table 17 Bit interpretation of the arbitration lost capture register (ALC); CAN address 11

BIT	SYMBOL	NAME	VALUE	FUNCTION
ALC.7 to ALC.5	–	reserved	For value and function see Table 18	
ALC.4	BITNO4	bit number 4		
ALC.3	BITNO3	bit number 3		
ALC.2	BITNO2	bit number 2		
ALC.1	BITNO1	bit number 1		
ALC.0	BITNO0	bit number 0		

On arbitration lost, the corresponding arbitration lost interrupt is forced, if enabled. At the same time, the current bit position of the bit stream processor is captured into the arbitration lost capture register. The content within this register is fixed until the users software has read out its contents once. The capture mechanism is then activated again.

The corresponding interrupt flag located in the interrupt register is cleared during the read access to the interrupt register. A new arbitration lost interrupt is not possible until the arbitration lost capture register is read out once.

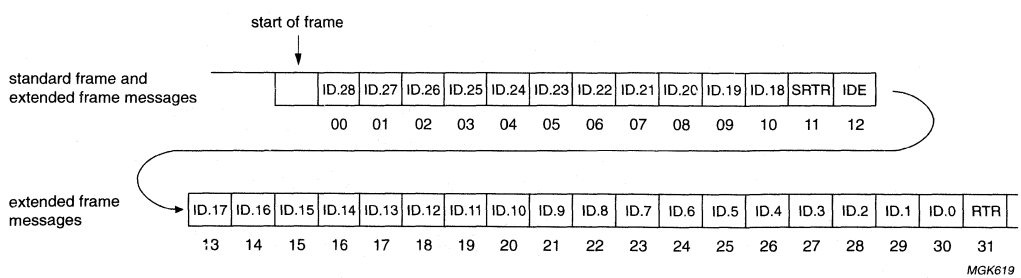


Fig.5 Arbitration lost bit number interpretation.

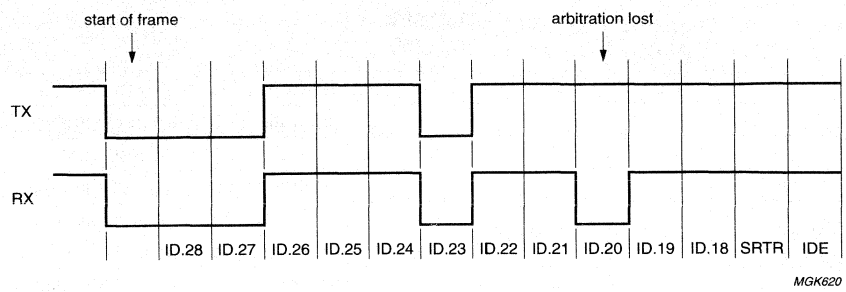


Fig.6 Example of arbitration lost bit number interpretation; result: ALC = 08.

Table 18 Function of bits 4 to 0 of the arbitration lost capture register

BITS[1]					DECIMAL VALUE	FUNCTION
ALC.4	ALC.3	ALC.2	ALC.1	ALC.0		
0	0	0	0	0	00	arbitration lost in bit 1 of identifier
0	0	0	0	1	01	arbitration lost in bit 2 of identifier
0	0	0	1	0	02	arbitration lost in bit 3 of identifier
0	0	0	1	1	03	arbitration lost in bit 4 of identifier
0	0	1	0	0	04	arbitration lost in bit 5 of identifier
0	0	1	0	1	05	arbitration lost in bit 6 of identifier
0	0	1	1	0	06	arbitration lost in bit 7 of identifier
0	0	1	1	1	07	arbitration lost in bit 8 of identifier
0	1	0	0	0	08	arbitration lost in bit 9 of identifier
0	1	0	0	1	09	arbitration lost in bit 10 of identifier
0	1	0	1	0	10	arbitration lost in bit 11 of identifier
0	1	0	1	1	11	arbitration lost in bit SRTR; note 2
0	1	1	0	0	12	arbitration lost in bit IDE
0	1	1	0	1	13	arbitration lost in bit 12 of identifier; note 3
0	1	1	1	0	14	arbitration lost in bit 13 of identifier; note 3
0	1	1	1	1	15	arbitration lost in bit 14 of identifier; note 3
1	0	0	0	0	16	arbitration lost in bit 15 of identifier; note 3
1	0	0	0	1	17	arbitration lost in bit 16 of identifier; note 3
1	0	0	1	0	18	arbitration lost in bit 17 of identifier; note 3
1	0	0	1	1	19	arbitration lost in bit 18 of identifier; note 3
1	0	1	0	0	20	arbitration lost in bit 19 of identifier; note 3
1	0	1	0	1	21	arbitration lost in bit 20 of identifier; note 3
1	0	1	1	0	22	arbitration lost in bit 21 of identifier; note 3
1	0	1	1	1	23	arbitration lost in bit 22 of identifier; note 3
1	1	0	0	0	24	arbitration lost in bit 23 of identifier; note 3
1	1	0	0	1	25	arbitration lost in bit 24 of identifier; note 3
1	1	0	1	0	26	arbitration lost in bit 25 of identifier; note 3
1	1	0	1	1	27	arbitration lost in bit 26 of identifier; note 3
1	1	1	0	0	28	arbitration lost in bit 27 of identifier; note 3
1	1	1	0	1	29	arbitration lost in bit 28 of identifier; note 3
1	1	1	1	0	30	arbitration lost in bit 29 of identifier; note 3
1	1	1	1	1	31	arbitration lost in bit RTR; note 3

Notes

1. Binary coded frame bit number where arbitration was lost.
2. Bit RTR for standard frame messages.
3. Extended frame messages only.

6.4.9 Error Code Capture register (ECC)

This register contains information about the type and location of errors on the bus. The error code capture register appears to the CPU as a read only memory.

Table 19 Bit interpretation of the error code capture register (ECC); CAN address 12

BIT	SYMBOL	NAME	VALUE	FUNCTION
ECC.7[1]	ERRC1	Error Code 1	–	–
ECC.6[1]	ERRC0	Error Code 0	–	–
ECC.5	DIR	Direction	1	RX; error occurred during reception
			0	TX; error occurred during transmission
ECC.4[2]	SEG4	Segment 4	–	–
ECC.3[2]	SEG3	Segment 3	–	–
ECC.2[2]	SEG2	Segment 2	–	–
ECC.1[2]	SEG1	Segment 1	–	–
ECC.0[2]	SEG0	Segment 0	–	–

Notes

1. For bit interpretation of bits ECC.7 and ECC.6 see Table 20.
2. For bit interpretation of bits ECC.4 to ECC.0 see Table 21.

Table 20 Bit interpretation of bits ECC.7 and ECC.6

BIT ECC.7	BIT ECC.6	FUNCTION
0	0	bit error
0	1	form error
1	0	stuff error
1	1	other type of error

Table 21 Bit interpretation of bits ECC.4 to ECC.0; note 1

BIT ECC.4	BIT ECC.3	BIT ECC.2	BIT ECC.1	BIT ECC.0	FUNCTION
0	0	0	1	1	start of frame
0	0	0	1	0	ID.28 to ID.21
0	0	1	1	0	ID.20 to ID.18
0	0	1	0	0	bit SRTR
0	0	1	0	1	bit IDE
0	0	1	1	1	ID.17 to ID.13
0	1	1	1	1	ID.12 to ID.5
0	1	1	1	0	ID.4 to ID.0
0	1	1	0	0	bit RTR
0	1	1	0	1	reserved bit 1
0	1	0	0	1	reserved bit 0
0	1	0	1	1	data length code
0	1	0	1	0	data field
0	1	0	0	0	CRC sequence
1	1	0	0	0	CRC delimiter
1	1	0	0	1	acknowledge slot
1	1	0	1	1	acknowledge delimiter
1	1	0	1	0	end of frame
1	0	0	1	0	intermission
1	0	0	0	1	active error flag
1	0	1	1	0	passive error flag
1	0	0	1	1	tolerate dominant bits
1	0	1	1	1	error delimiter
1	1	1	0	0	overload flag

Note

1. Bit settings reflect the current frame segment to distinguish between different error events.

If a bus error occurs, the corresponding bus error interrupt is always forced, if enabled. At the same time, the current position of the bit stream processor is captured into the error code capture register. The content within this register is fixed until the users software has read out its content once. The capture mechanism is then activated again.

The corresponding interrupt flag located in the interrupt register is cleared during the read access to the interrupt register. A new bus error interrupt is not possible until the capture register is read out once.

6.4.10 Error Warning Limit Register (EWLR)

The error warning limit can be defined within this register. The default value (after hardware reset) is 96. In reset mode this register appears to the CPU as a read/write memory. In operating mode it is read only.

Note, that a content change of the EWLR is only possible, if the reset mode was entered previously. An error status change (see status register; Table 14) and an error warning interrupt forced by the new register content will not occur until the reset mode is cancelled again.

Table 22 Bit interpretation of the error warning limit register (EWLR); CAN address 13

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
EWL.7	EWL.6	EWL.5	EWL.4	EWL.3	EWL.2	EWL.1	EWL.0

6.4.11 RX Error Counter Register (RXERR)

The RX error counter register reflects the current value of the receive error counter. After a hardware reset this register is initialized to logic 0. In operating mode this register appears to the CPU as a read only memory. A write access to this register is possible only in reset mode.

If a bus-off event occurs, the RX error counter is initialized to logic 0. The time bus-off is valid, writing to this register has no effect.

Note, that a CPU-forced content change of the RX error counter is only possible, if the reset mode was entered previously. An error status change (see status register; Table 14), an error warning or an error passive interrupt forced by the new register content will not occur, until the reset mode is cancelled again.

Table 23 Bit interpretation of the RX error counter register (RXERR); CAN address 14

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
RXERR.7	RXERR.6	RXERR.5	RXERR.4	RXERR.3	RXERR.2	RXERR.1	RXERR.0

6.4.12 TX Error Counter Register (TXERR)

The TX error counter register reflects the current value of the transmit error counter.

In operating mode this register appears to the CPU as a read only memory. A write access to this register is possible only in reset mode. After a hardware reset this register is initialized to logic 0. If a bus-off event occurs, the TX error counter is initialized to 127 to count the minimum protocol-defined time (128 occurrences of the bus-free signal). Reading the TX error counter during this time gives information about the status of the bus-off recovery.

If bus-off is active, a write access to TXERR in the range from 0 to 254 clears the bus-off flag and the controller will wait for one occurrence of 11 consecutive recessive bits (bus-free) after the reset mode has been cleared.

Table 24 Bit interpretation of the TX error counter register (TXERR); CAN address 15

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
TXERR.7	TXERR.6	TXERR.5	TXERR.4	TXERR.3	TXERR.2	TXERR.1	TXERR.0

Writing 255 to TXERR allows to initiate a CPU-driven bus-off event. It should be noted that a CPU-forced content change of the TX error counter is only possible, if the reset mode was entered previously. An error or bus status change (see status register; Table 14), an error warning or an error passive interrupt forced by the new register content will not occur until the reset mode is cancelled again. After leaving the reset mode, the new TX counter content is interpreted and the bus-off event is performed in the same way, as if it was forced by a bus error event. That means, that the reset mode is entered again, the TX error counter is initialized to 127, the RX counter is cleared and all concerned status and interrupt register bits are set.

Clearing of reset mode now will perform the protocol-defined bus-off recovery sequence (waiting for 128 occurrences of the bus-free signal).

If the reset mode is entered again before the end of bus-off recovery (TXERR > 0), bus-off keeps active and TXERR is frozen.

6.4.13 TRANSMIT BUFFER

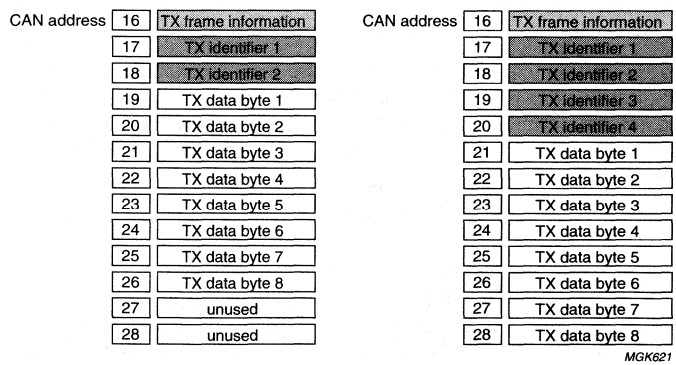
The global layout of the transmit buffer is shown in Fig.7. One has to distinguish between the Standard Frame Format (SFF) and the Extended Frame Format (EFF) configuration. The transmit buffer allows the definition of one transmit message with up to eight data bytes.

6.4.13.1 Transmit buffer layout

The transmit buffer layout is subdivided into descriptor and data fields where the first byte of the descriptor field is the frame information byte (frame information). It describes the frame format (SFF or EFF), remote or data frame and the data length. Two identifier bytes for SFF or four bytes for EFF messages follow. The data field contains up to eight data bytes.

The transmit buffer has a length of 13 bytes and is located in the CAN address range from 16 to 28.

Note, that a direct access to the transmit buffer RAM is possible using the CAN address space from 96 to 108. This RAM area is reserved for the transmit buffer. The three following bytes may be used for general purposes (CAN address 109, 110 and 111).



a. Standard frame format. b. Extended frame format.

Fig.7 Transmit buffer layout for standard and extended frame format configurations.

6.4.13.2 Descriptor field of the transmit buffer

The bit layout of the transmit buffer is represented in Tables 25 to 27 for SFF and Tables 28 to 32 for EFF. The given configuration is chosen to be compatible with the receive buffer layout (see Section 6.4.14.1).

Table 25 TX frame information (SFF); CAN address 16

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
FF(1)	RTR(2)	X(3)	X(3)	DLC.3(4)	DLC.2(4)	DLC.1(4)	DLC.0(4)

Notes

1. Frame format.
2. Remote transmission request.
3. Don't care; recommended to be compatible to receive buffer (0) in case of using the self reception facility (self test).
4. Data length code bit.

Table 26 TX identifier 1 (SFF); CAN address 17; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

Note

1. ID.X means identifier bit X.

Table 27 TX identifier 2 (SFF); CAN address 18; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.20	ID.19	ID.18	X(2)	X(3)	X(3)	X(3)	X(3)

Notes

1. ID.X means identifier bit X.
2. Don't care; recommended to be compatible to receive buffer (RTR) in case of using the self reception facility (self test).
3. Don't care; recommended to be compatible to receive buffer (0) in case of using the self reception facility (self test).

Table 28 TX frame information (EFF); CAN address 16

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
FF(1)	RTR(2)	X(3)	X(3)	DLC.3(4)	DLC.2(4)	DLC.1(4)	DLC.0(4)

Notes

1. Frame format.
2. Remote transmission request.
3. Don't care; recommended to be compatible to receive buffer (0) in case of using the self reception facility (self test).
4. Data length code bit.

Table 29 TX identifier 1 (EFF); CAN address 17; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

Note

1. ID.X means identifier bit X.

Table 30 TX identifier 2 (EFF); CAN address 18; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.20	ID.19	ID.18	ID.17	ID.16	ID.15	ID.14	ID.13

Note

1. ID.X means identifier bit X.

Table 31 TX identifier 3 (EFF); CAN address 19; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.12	ID.11	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5

Note

1. ID.X means identifier bit X.

Table 32 TX identifier 4 (EFF); CAN address 20; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.4	ID.3	ID.2	ID.1	ID.0	X[2]	X[3]	X[3]

Notes

1. ID.X means identifier bit X.
2. Don't care; recommended to be compatible to receive buffer (RTR) in case of using the self reception facility (self test).
3. Don't care; recommended to be compatible to receive buffer (0) in case of using the self reception facility (self test).

Table 33 Frame Format (FF) and Remote Transmission Request (RTR) bits

BIT	VALUE	FUNCTION
FF	1	EFF; extended frame format will be transmitted by the CAN controller
	0	SFF; standard frame format will be transmitted by the CAN controller
RTR	1	remote; remote frame will be transmitted by the CAN controller
	0	data; data frame will be transmitted by the CAN controller

6.4.13.3 Data Length Code (DLC)

The number of bytes 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 logic 1 (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 with the same identifier simultaneously.

The range of the data byte count is 0 to 8 bytes and is coded as follows:

DataByteCount = 8 × DLC.3 + 4 × DLC.2 + 2 × DLC.1 + DLC.0

For reasons of compatibility no data length code >8 should be used. If a value >8 is selected, 8 bytes are transmitted in the data frame with the Data Length Code specified in DLC.

6.4.13.4 Identifier (ID)

In Standard Frame Format (SFF) the identifier consists of 11 bits (ID.28 to ID.18) and in Extended Frame Format (EFF) messages the identifier consists of 29 bits (ID.28 to ID.0). ID.28 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.

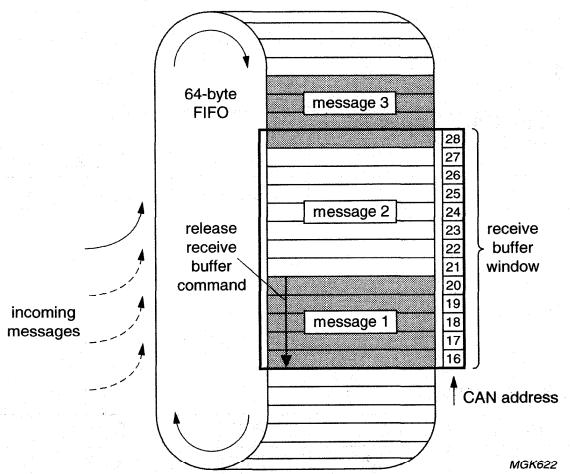
The lower the binary value of the identifier the higher the priority. This is due to the larger number of leading dominant bits during arbitration.

6.4.13.5 Data field

The number of transferred data bytes is defined by the data length code. The first bit transmitted is the most significant bit of data byte 1 at CAN address 19 (SFF) or CAN address 21 (EFF).

6.4.14 RECEIVE BUFFER

The global layout of the receive buffer is very similar to the transmit buffer described in the previous section. The receive buffer is the accessible part of the RXFIFO and is located in the range between CAN address 16 and 28. Each message is subdivided into a descriptor and a data field.



Message 1 is now available in the receive buffer.

Fig.8 Example of the message storage within the RXFIFO.

6.4.14.1 Descriptor field of the receive buffer

The bit layout of the receive buffer is represented in Tables 34 to 36 for SFF and Tables 37 to 41 for EFF. The given configuration is chosen to be compatible with the transmit buffer layout (see Section 6.4.13.2).

Table 34 RX frame information (SFF); CAN address 16

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
FF(1)	RTR(2)	0	0	DLC.3(3)	DLC.2(3)	DLC.1(3)	DLC.0(3)

Notes

1. Frame format.
2. Remote transmission request.
3. Data length code bit.

Table 35 RX identifier 1 (SFF); CAN address 17; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

Note

1. ID.X means identifier bit X.

Table 36 RX identifier 2 (SFF); CAN address 18; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.20	ID.19	ID.18	RTR[2]	0	0	0	0

Notes

1. ID.X means identifier bit X.
2. Remote transmission request.

Table 37 RX frame information (EFF); CAN address 16

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
FF[1]	RTR[2]	0	0	DLC.3[3]	DLC.2[3]	DLC.1[3]	DLC.0[3]

Notes

1. Frame format.
2. Remote transmission request.
3. Data length code bit.

Table 38 RX identifier 1 (EFF); CAN address 17; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.28	ID.27	ID.26	ID.25	ID.24	ID.23	ID.22	ID.21

Note

1. ID.X means identifier bit X.

Table 39 RX identifier 2 (EFF); CAN address 18; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.20	ID.19	ID.18	ID.17	ID.16	ID.15	ID.14	ID.13

Note

1. ID.X means identifier bit X.

Table 40 RX identifier 3 (EFF); CAN address 19; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.12	ID.11	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5

Note

1. ID.X means identifier bit X.

Table 41 RX identifier 4 (EFF); can address 20; note 1

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ID.4	ID.3	ID.2	ID.1	ID.0	RTR(2)	0	0

Notes

1. ID.X means identifier bit X.
2. Remote transmission request.

Remark: the received data length code located in the frame information byte represents the real sent data length code, which may be greater than 8 (depends on sender). Nevertheless the maximum number of received data bytes is 8. This should be taken into account by reading a message from the receive buffer.

As described in Fig.8 the RXFIFO has space for 64 message bytes in total. It depends on the data length how many messages can fit in it at one time. If there is not enough space for a new message within the RXFIFO, the CAN controller generates a data overrun condition the moment this message becomes valid and the acceptance test was positive. A message which is partly written into the RXFIFO, when the data overrun situation occurs, is deleted. This situation is indicated to the CPU via the status register and the data overrun interrupt, if enabled.

6.4.15 Acceptance filter

With the help of the acceptance filter the CAN controller is able to allow passing of received messages to the RXFIFO only when the identifier bits of the received message are equal to the predefined ones within the acceptance filter registers.

The acceptance filter is defined by the Acceptance Code Registers (ACRn) and the Acceptance Mask Registers (AMRn). The bit patterns of messages to be received are defined within the acceptance code registers.
The corresponding acceptance mask registers allow to define certain bit positions to be 'don't care'.

Two different filter modes are selectable within the mode register (MOD.3, AFM; see Section 6.4.3):

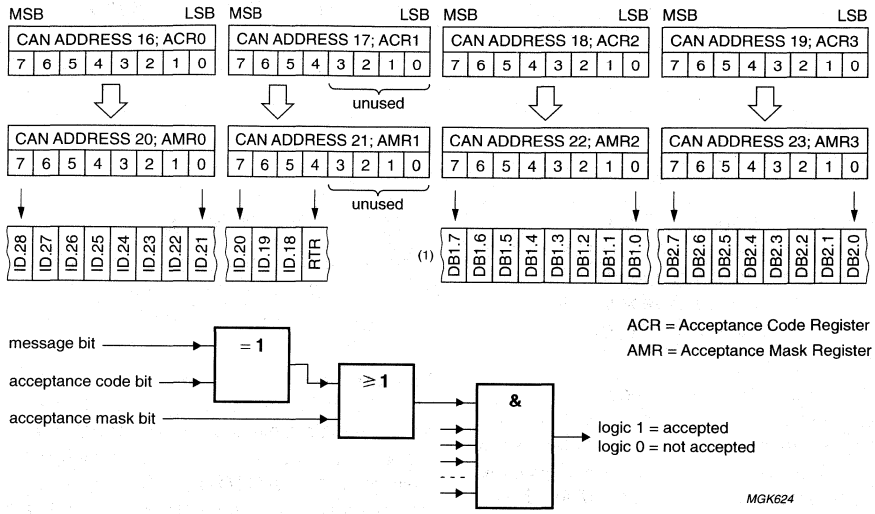
- Single filter mode (bit AFM is logic 1)
- Dual filter mode (bit AFM is logic 0).

6.4.15.1 Single filter configuration

In this filter configuration one long filter (4-bytes) could be defined. The bit correspondences between the filter bytes and the message bytes depend on the currently received frame format.

Standard frame: if a standard frame format message is received, the complete identifier including the RTR bit and the first two data bytes are used for acceptance filtering. Messages may also be accepted if there are no data bytes existing due to a set RTR bit or if there is none or only one data byte because of the corresponding data length code.

For a successful reception of a message, all single bit comparisons have to signal acceptance.
Note, that the 4 least significant bits of AMR1 and ACR1 are not used. In order to be compatible with future products these bits should be programmed to be 'don't care' by setting AMR1.3, AMR1.2, AMR1.1 and AMR1.0 to logic 1.



DBX.Y means data byte X, bit Y.

Fig.9 Single filter configuration, receiving standard frame messages.

Extended frame: if an extended frame format message is received, the complete identifier including the RTR bit is used for acceptance filtering.

For a successful reception of a message, all single bit comparisons have to signal acceptance.
It should be noted that the 2 least significant bits of AMR3 and ACR3 are not used. In order to be compatible with future products these bits should be programmed to be 'don't care' by setting AMR3.1 and AMR3.0 to logic 1.

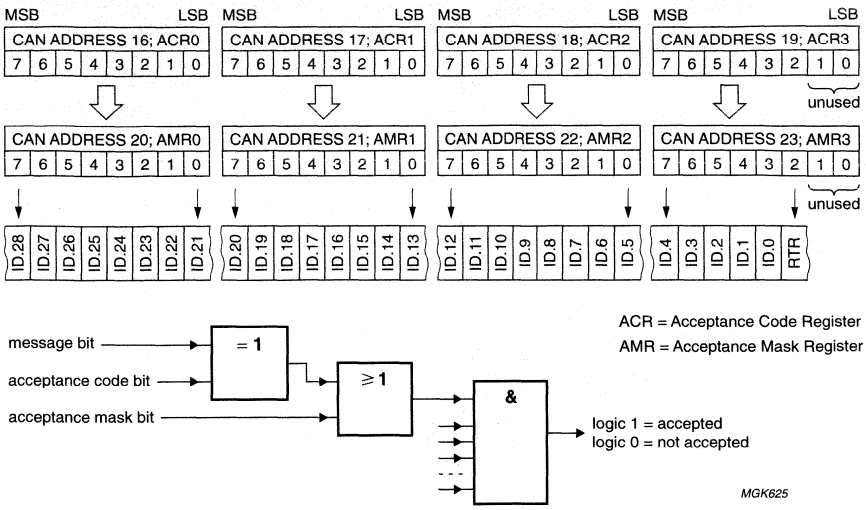


Fig.10 Single filter configuration, receiving extended frame messages.

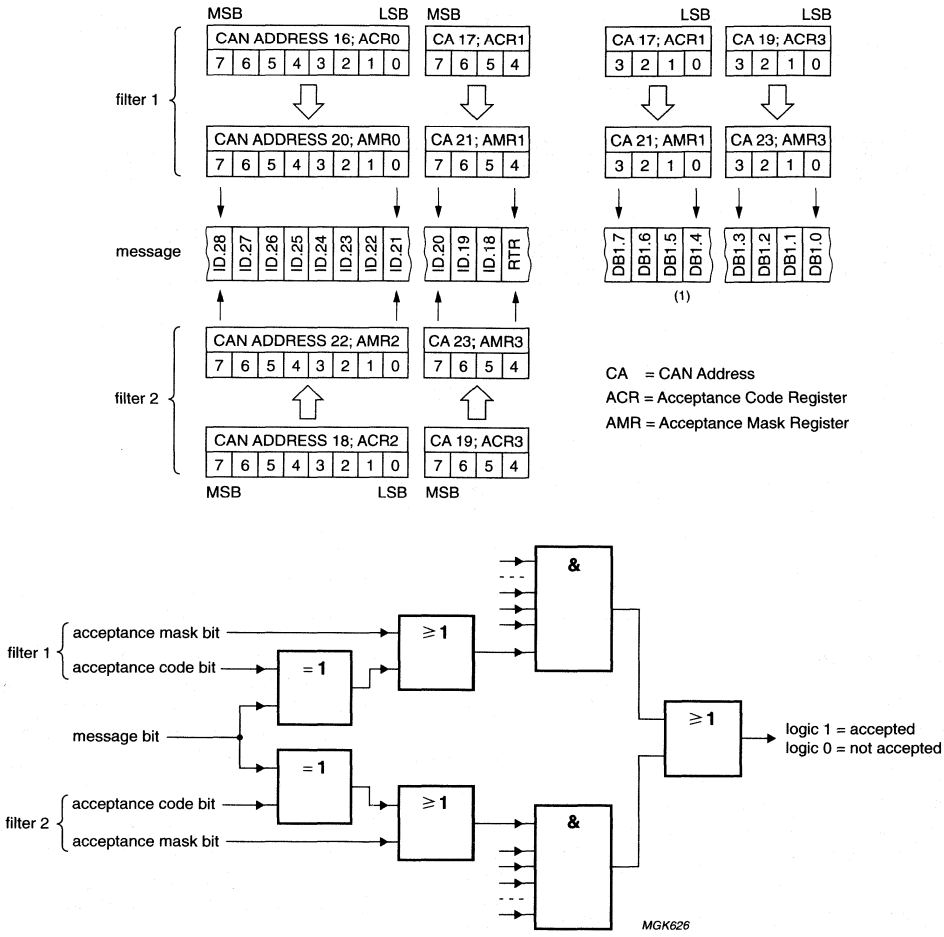
6.4.15.2 Dual filter configuration

In this filter configuration two short filters can be defined. A received message is compared with both filters to decide, whether this message should be copied into the receive buffer or not. If at least one of the filters signals an acceptance, the received message becomes valid. The bit correspondences between the filter bytes and the message bytes depends on the currently received frame format.

Standard frame: if a standard frame message is received, the two defined filters are looking different. The first filter compares the complete standard identifier including the RTR bit and the first data byte of the message. The second filter just compares the complete standard identifier including the RTR bit.

For a successful reception of a message, all single bit comparisons of at least one complete filter have to signal acceptance. In case of a set RTR bit or a data length code of logic 0 no data byte is existing. Nevertheless a message may pass filter 1, if the first part up to the RTR bit signals acceptance.

If no data byte filtering is required for filter 1, the four least significant bits of AMR1 and AMR3 have to be set to logic 1 (don't care). Then both filters are working identically using the standard identifier range including the RTR bit.



DBX.Y = data byte X, bit Y.

Fig.11 Dual filter configuration, receiving standard frame messages.

Extended frame: if an extended frame message is received, the two defined filters are looking identically. Both filters are comparing the first two bytes of the extended identifier range only.

For a successful reception of a message, all single bit comparisons of at least one complete filter have to indicate acceptance.

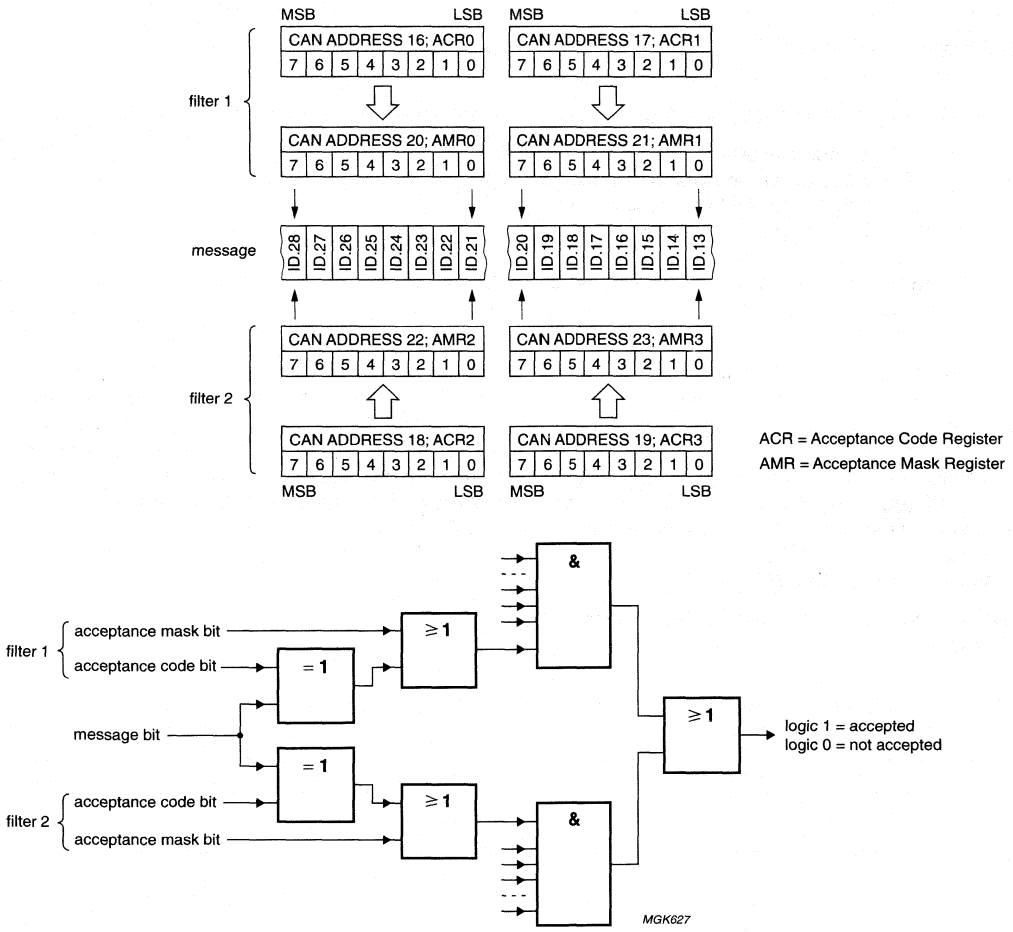


Fig.12 Dual filter configuration, receiving extended frame messages.

6.4.16 RX Message Counter (RMC)

The RMC register (CAN address 29) reflects the number of messages available within the RXFIFO. The value is incremented with each receive event and decremented by the release receive buffer command. After any reset event, this register is cleared.

Table 42 Bit interpretation of the RX message counter (RMC); CAN address 29

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
(0)[1]	(0)[1]	(0)[1]	RMC.4	RMC.3	RMC.2	RMC.1	RMC.0

Note

1. This bit cannot be written. During read-out of this register always a zero is given.

6.4.17 RX Buffer Start Address register (RBSA)

The RBSA register (CAN address 30) reflects the currently valid internal RAM address, where the first byte of the received message, which is mapped to the receive buffer window, is stored. With the help of this information it is possible to interpret the internal RAM contents.
The internal RAM address area begins at CAN address 32 and may be accessed by the CPU for reading and writing (writing in reset mode only).

Example: if RBSA is set to 24 (decimal), the current message visible in the receive buffer window (CAN address 16 to 28) is stored within the internal RAM beginning at RAM address 24. Because the RAM is also mapped directly to the CAN address space beginning at CAN address 32 (equal to RAM address 0) this message may also be accessed using CAN address 56 and the following bytes
(CAN address = RBSA + 32 > 24 + 32 = 56).

If a message exceeds RAM address 63, it continues at RAM address 0.

The release receive buffer command is always given while there is at least one more message available within the FIFO. RBSA is updated to the beginning of the next message.

On hardware reset, this pointer is initialized to '00H'. Upon a software reset (setting of reset mode) this pointer keeps its old value, but the FIFO is cleared; this means that the RAM contents are not changed, but the next received (or transmitted) message will override the currently visible message within the receive buffer window.

The RX buffer start address register appears to the CPU as a read only memory in operating mode and as read/write memory in reset mode. It should be noted that a write access to RBSA takes effect first after the next positive edge of the internal clock frequency, which is half of the external oscillator frequency.

Table 43 Bit interpretation of the RX buffer start address register (RBSA); CAN address 30

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
(0)[1]	(0)[1]	RBSA.5	RBSA.4	RBSA.3	RBSA.2	RBSA.1	RBSA.0

Note

1. This bit cannot be written. During read-out of this register always a zero is given.

6.5 Common registers

6.5.1 Bus Timing Register 0 (BTR0)

The contents of the bus timing register 0 defines the values of the Baud Rate Prescaler (BRP) and the Synchronization Jump Width (SJW). This register can be accessed (read/write) if the reset mode is active.

In operating mode this register is read only, if the PeliCAN mode is selected. In BasicCAN mode a 'FFH' is reflected.

Table 44 Bit interpretation of bus timing register 0 (BTR0); CAN address 6

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SJW.1	SJW.0	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0

6.5.1.1 Baud Rate Prescaler (BRP)

The period of the CAN system clock t_{scl} is programmable and determines the individual bit timing. The CAN system clock is calculated using the following equation:

$$t_{scl} = 2 \times t_{CLK} \times (32 \times BRP.5 + 16 \times BRP.4 + 8 \times BRP.3 + 4 \times BRP.2 + 2 \times BRP.1 + BRP.0 + 1)$$

where t_{CLK} = time period of the XTAL frequency = $\frac{1}{f_{XTAL}}$

6.5.1.2 Synchronization Jump Width (SJW)

To compensate for phase shifts between clock oscillators of different bus controllers, any bus controller must re-synchronize 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 re-synchronization:

$$t_{SJW} = t_{scl} \times (2 \times SJW.1 + SJW.0 + 1)$$

6.5.2 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 can be accessed (read/write) if the reset mode is active.

In operating mode, this register is read only, if the PeliCAN mode is selected. In BasicCAN mode a 'FFH' is reflected.

Table 45 Bit interpretation of bus timing register 1 (BTR1); CAN address 7

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

6.5.2.1 Sampling (SAM)

BIT	VALUE	FUNCTION
SAM	1	triple; the bus is sampled three times; recommended for low/medium speed buses (class A and B) where filtering spikes on the bus line is beneficial
	0	single; the bus is sampled once; recommended for high speed buses (SAE class C)

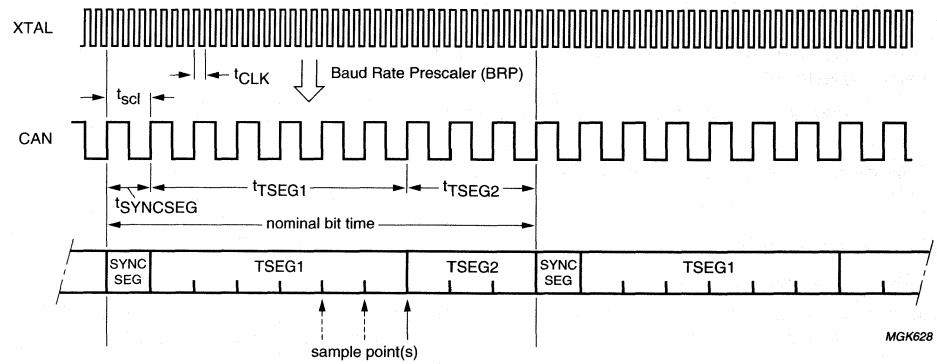
6.5.2.2 Time Segment 1 (TSEG1) and Time Segment 2 (TSEG2)

TSEG1 and TSEG2 determine the number of clock cycles per bit period and the location of the sample point, where:

$t_{SYNCSEG} = 1 \times t_{scl}$

$t_{TSEG1} = t_{scl} \times (8 \times TSEG1.3 + 4 \times TSEG1.2 + 2 \times TSEG1.1 + TSEG1.0 + 1)$

$t_{TSEG2} = t_{scl} \times (4 \times TSEG2.2 + 2 \times TSEG2.1 + TSEG2.0 + 1)$



Possible values are BRP = 000001, TSEG1 = 0101 and TSEG2 = 010.

Fig.13 General structure of a bit period.

6.5.3 OUTPUT CONTROL REGISTER (OCR)

The output control register allows the set-up of different output driver configurations under software control.

This register may be accessed (read/write) if the reset mode is active. In operating mode, this register is read only, if the PeliCAN mode is selected. In BasicCAN mode a 'FFH' is reflected.

Table 46 Bit interpretation of the output control register (OCR); CAN address 8

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
OCTP1	OCTN1	OCPOL1	OCTP0	OCTN0	OCPOL0	OCMODE1	OCMODE0

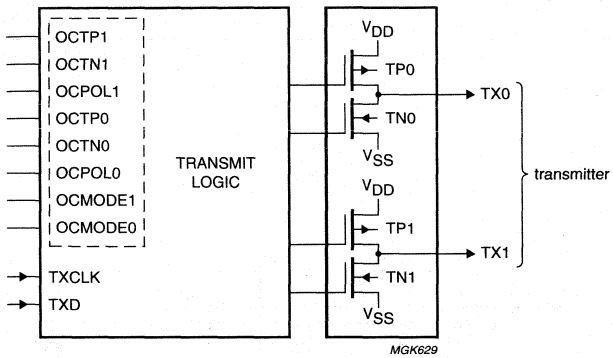


Fig.14 Transceiver input/output control logic.

If the SJA1000 is in the sleep mode a recessive level is output on the TX0 and TX1 pins with respect to the contents within the output control register. If the SJA1000 is in the reset state (reset request = HIGH) or the external reset pin $\overline{RST}$ is pulled LOW the outputs TX0 and TX1 are floating.

The transmit output stage is able to operate in different modes. Table 47 shows the output control register settings.

Table 47 Interpretation of OCMODE bits

OCMODE1	OCMODE0	DESCRIPTION
0	0	bi-phase output mode
0	1	test output mode; note 1
1	0	normal output mode
1	1	clock output mode

Note

1. In test output mode TXn will reflect the bit, detected on RX pins, with the next positive edge of the system clock. TN1, TN0, TP1 and TP0 are configured in accordance with the setting of OCR.

6.5.3.1 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 TX0 and TX1 depend on both the driver characteristic programmed by OCTPx, OCTNx (float, pull-up, pull-down, push-pull) and the output polarity programmed by OCPOLx.

6.5.3.2 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 $1 \times t_{scl}$.

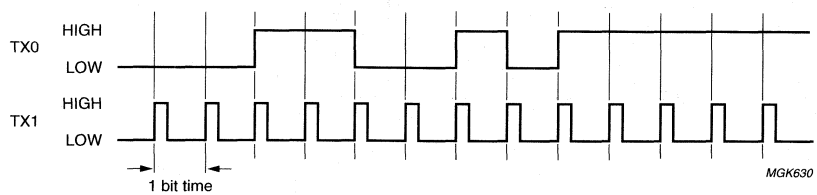


Fig.15 Example of clock output mode.

6.5.3.3 Bi-phase output mode

In contrast to the 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 (floating). Dominant bits are sent with alternating levels 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, and so on. One possible configuration example of the bi-phase output mode timing is shown in Fig.16.

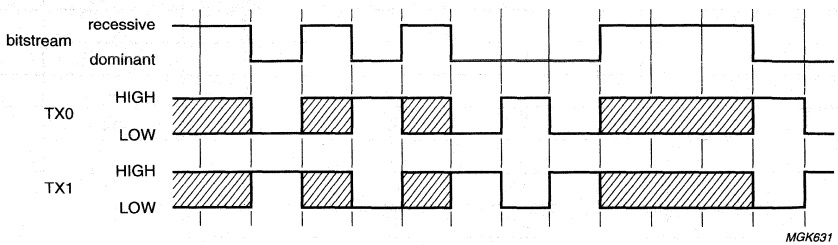


Fig.16 Bi-phase output mode example (output control register = F8H).

6.5.3.4 Test output mode

In test output mode the level connected to RX is reflected at TXn with the next positive edge of the system clock $\frac{f_{osc}}{2}$ corresponding to the programmed polarity in the output control register.

Table 48 shows the relationship between the bits of the output control register and the output pins TX0 and TX1.

Table 48 Output pin configuration; note 1

DRIVE	TXD	OCTPX	OCTNX	OCPOLX	TPX[2]	TNX[3]	TXX[4]
Float	X	0	0	X	off	off	float
Pull-down	0	0	1	0	off	on	LOW
	1	0	1	0	off	off	float
	0	0	1	1	off	off	float
	1	0	1	1	off	on	LOW
Pull-up	0	1	0	0	off	off	float
	1	1	0	0	on	off	HIGH
	0	1	0	1	on	off	HIGH
	1	1	0	1	off	off	float
Push-pull	0	1	1	0	off	on	LOW
	1	1	1	0	on	off	HIGH
	0	1	1	1	on	off	HIGH
	1	1	1	1	off	on	LOW

Notes

1. X = don't care.
2. TPX is the on-chip output transistor X, connected to V_{DD}.
3. TNX is the on-chip output transistor X, connected to V_{SS}.
4. TXX is the serial output level on pin TX0 or TX1. It is required that the output level on the CAN-bus line is dominant when TXD = 0 and recessive when TXD = 1.

The bit sequence (TXD) is sent via TX0 and TX1. The voltage levels on the output driver pins depends on both the driver characteristics programmed by OCTP, OCTN (float, pull-up, pull-down, push-pull) and the output polarity programmed by OCPOL.

6.5.4 Clock Divider Register (CDR)

The clock divider register controls the CLKOUT frequency for the microcontroller and allows to deactivate the CLKOUT pin. Additionally a dedicated receive interrupt pulse on TX1, a receive comparator bypass and the selection between BasicCAN mode and PeliCAN mode is made here. The default state of the register after hardware reset is divide-by-12 for Motorola mode (00000101) and divide-by-2 for Intel mode (00000000).

On software reset (reset request/reset mode) this register is not influenced.

The reserved bit (CDR.4) will always reflect a logic 0. The application software should always write a logic 0 to this bit in order to be compatible with future features, which may be 1-active using this bit.

Table 49 Bit interpretation of the clock divider register (CDR); CAN address 31

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
CAN mode	CBP	RXINTEN	(0)[1]	clock off	CD.2	CD.1	CD.0

Note

1. This bit cannot be written. During read-out of this register always a zero is given.

6.5.4.1 CD.2 to CD.0

The bits CD.2 to CD.0 are accessible without restrictions in reset mode as well as in operating mode. These bits are used to define the frequency at the external CLKOUT pin. For an overview of selectable frequencies see Table 50.

Table 50 CLKOUT frequency selection; note 1

CD.2	CD.1	CD.0	CLKOUT FREQUENCY
0	0	0	$\frac{f_{osc}}{2}$
0	0	1	$\frac{f_{osc}}{4}$
0	1	0	$\frac{f_{osc}}{6}$
0	1	1	$\frac{f_{osc}}{8}$
1	0	0	$\frac{f_{osc}}{10}$
1	0	1	$\frac{f_{osc}}{12}$
1	1	0	$\frac{f_{osc}}{14}$
1	1	1	f_{osc}

Note

1. f_{osc} is the frequency of the external oscillator (XTAL).

6.5.4.2 Clock off

Setting of this bit allows to disable the external CLKOUT pin of the SJA1000. A write access is possible only in reset mode (reset request bit is set in BasicCAN mode).

6.5.4.3 RXINTEN

This bit allows to use the TX1 output as a dedicated receive interrupt output. When a received message has passed the acceptance filter successfully, a receive interrupt pulse with the length of one bit time is always output at the TX1 pin (during the last bit of end of frame). The polarity and output drive are programmable via the output control register (see also Section 6.5.3). A write access is only possible in reset mode (the reset request bit is set in BasicCAN mode).

6.5.4.4 CBP

Setting of CDR.6 allows to bypass the CAN input comparator and is only possible in reset mode. This is useful in the event that the SJA1000 is connected to an external transceiver circuit. The internal delay of the SJA1000 is reduced, which will result in a longer maximum possible bus length. If CBP is set, only RX0 is active. The unused RX1 input should be connected to a defined level (e.g. V_{SS}).

6.5.4.5 CAN mode

CDR.7 defines the CAN mode. If CDR.7 is at logic 0 the CAN controller operates in BasicCAN mode. If set to logic 1 the CAN controller operates in PeliCAN mode. Write access is only possible in reset mode.

7 LIMITING VALUES

In accordance with the Absolute Maximum Rating System (IEC 134); all voltages referenced to V_{SS}.

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
V_{DD}	supply voltage		4.5	5.5	V
I_I, I_O	input/output current on all pins except TX0 and TX1		–	±4	mA
$I_{OT(sink)}$	sink current of TX0 and TX1 together	note 1	–	30	mA
$I_{OT(source)}$	source current of TX0 and TX1 together	note 1	–	–20	mA
T_{amb}	operating ambient temperature		–40	+125	°C
T_{stg}	storage temperature		–65	+150	°C
P_{tot}	total power dissipation	note 2	–	1.0	W
V_{esd}	electrostatic discharge on all pins	note 3	–1000	+1000	V
		note 4	–200	+200	V

Notes

1. 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 12 mA for each TX output.
2. This value is based on the maximum allowable die temperature and the thermal resistance of the package, not on device power consumption.
3. Human body model: equivalent to discharging a 100 pF capacitor through a 1.5 kΩ resistor.
4. Machine model: equivalent to discharging a 200 pF capacitor through a 25 Ω plus 2.5 H circuit.

8 THERMAL CHARACTERISTICS

SYMBOL	PARAMETER	CONDITION	VALUE	UNIT
$R_{th(j-a)}$	thermal resistance from junction to ambient	in free air	67	K/W

9 DC CHARACTERISTICS

V_{DD} = 5 V (±10%); V_{SS} = 0 V; T_{amb} = –40 to +125 °C; all voltages referenced to V_{SS}; unless otherwise specified.

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
Supplies					
V_{DD}	supply voltage		4.5	5.5	V
I_{DD}	operating supply current	$\overline{RST} = V_{SS}$; f_{osc} = 24 MHz (note 1); oscillator inactive (note 2)	–	15	mA
I_{sm}	sleep mode supply current		–	40	µA

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
Inputs					
V_{IL1}	LOW-level input voltage on pins ALE/AS, $\overline{CS}$, $\overline{RD}$/E, $\overline{WR}$ and MODE		−0.5	+0.8	V
V_{IL2}	LOW-level input voltage on pins XTAL1 and $\overline{INT}$		−	$0.3V_{DD}$	V
V_{IL3}	LOW-level input voltage on pins $\overline{RST}$, AD0 to AD7 and RX0[5]		−0.5	+0.6	V
V_{IH1}	HIGH-level input voltage on pins ALE/AS, $\overline{CS}$, $\overline{RD}$/E, $\overline{WR}$ and MODE		2.0	V_{DD} + 0.5	V
V_{IH2}	HIGH-level input voltage on pins XTAL1 and $\overline{INT}$		$0.7V_{DD}$	−	V
V_{IH3}	HIGH-level input voltage on pins $\overline{RST}$, AD0 to AD7 and RX0[5]		2.4	V_{DD} + 0.5	V
hys_{RST}	input hysteresis at pins $\overline{RST}$, AD0 to AD7 and RX0[5]		500	−	mV
I_{LI}	input leakage current on all pins except XTAL1, RX0 and RX1	0.45 V < $V_{I(D)}$ < V_{DD}; note 3	−	±2	μA
Outputs					
V_{OL}	LOW-level output voltage for pins AD0 to AD7, CLKOUT and $\overline{INT}$	I_{OL} = 4 mA	−	0.4	V
V_{OH}	HIGH-level output voltage for pins AD0 to AD7 and CLKOUT	I_{OH} = −4 mA	V_{DD} − 0.4	−	V
CAN input comparator (see also Fig.22)					
$V_{th(i)(diff)}$	differential input threshold voltage	V_{DD} = 5 V ±10%; 1.4 V < $V_{I(RX)}$ < V_{DD} − 1.4 V; note 4	−	±32	mV
V_{hys}	hysteresis voltage		8	30	mV
I_I	input current		−	±400	nA
CAN output driver					
$V_{OL(TX)}$	LOW-level output voltage at pins TX0 and TX1	V_{DD} = 5 V ±10%			
		I_O = 1.2 mA; note 4	−	0.05	V
		I_O = 12 mA	−	0.4	V
$V_{OH(TX)}$	HIGH-level output voltage at pins TX0 and TX1	V_{DD} = 5 V ±10%			
		I_O = 1.2 mA; note 4	V_{DD} − 0.05	−	V
		I_O = 12 mA	V_{DD} − 0.4	−	V

Notes

1. AD0 to AD7 = ALE = $\overline{RD}$ = $\overline{WR}$ = $\overline{CS}$ = V_{DD}; MODE = V_{SS}; RX0 = 2.7 V; RX1 = 2.3 V; XTAL1 = $\frac{0.5}{V_{DD} - 0.5}$ V; all outputs unloaded.
2. AD0 to AD7 = ALE = $\overline{RD}$ = $\overline{WR}$ = INT = RST = $\overline{CS}$ = MODE = RX0 = V_{DD}; RX1 = XTAL1 = V_{SS}; all outputs unloaded.
3. $V_{I(D)}$ = input voltage on all digital input pins.
4. Not tested during production; $V_{I(RX)}$ = input voltage on pins RX0 and RX1.
5. Only during bypass mode.

10 AC CHARACTERISTICS

V_{DD} = 5 V ±10%; V_{SS} = 0 V; C_L = 50 pF (output pins); T_{amb} = −40 to +125 °C; unless otherwise specified; note 1.

SYMBOL	PARAMETER	CONDITIONS	MIN.	MAX.	UNIT
f_{osc}	oscillator frequency		–	24	MHz
$t_{su(A-AL)}$	address set-up to ALE/AS LOW		8	–	ns
$t_{h(AL-A)}$	address hold after ALE LOW		2	–	ns
$t_{W(AL)}$	ALE/AS pulse width		8	–	ns
t_{RLQV}	$\overline{RD}$ LOW to valid data output	Intel mode	–	45	ns
t_{EHQV}	E HIGH to valid data output	Motorola mode	–	45	ns
t_{RHDZ}	data float after $\overline{RD}$ HIGH	Intel mode	–	15	ns
t_{ELDZ}	data float after E LOW	Motorola mode	–	15	ns
t_{DVWH}	input data valid to $\overline{WR}$ HIGH	Intel mode	5	–	ns
t_{WHDX}	input data hold after $\overline{WR}$ HIGH	Intel mode	5	–	ns
t_{WHLH}	$\overline{WR}$ HIGH to next ALE HIGH		15	–	ns
t_{ELAH}	E LOW to next AS HIGH	Motorola mode	15	–	ns
$t_{su(i)(D-EL)}$	input data set-up to E LOW	Motorola mode	5	–	ns
$t_{h(i)(EL-D)}$	input data hold after E LOW	Motorola mode	5	–	ns
t_{LLWL}	ALE LOW to $\overline{WR}$ LOW	Intel mode	10	–	ns
t_{LLRL}	ALE LOW to $\overline{RD}$ LOW	Intel mode	10	–	ns
t_{LLEH}	AS LOW to E HIGH	Motorola mode	10	–	ns
$t_{su(R-EH)}$	set-up time of RD/$\overline{WR}$ to E HIGH	Motorola mode	5	–	ns
$t_{W(W)}$	$\overline{WR}$ pulse width	Intel mode	20	–	ns
$t_{W(R)}$	$\overline{RD}$ pulse width	Intel mode	60	–	ns
$t_{W(E)}$	E pulse width	Motorola mode	60	–	ns
t_{CLWL}	$\overline{CS}$ LOW to $\overline{WR}$ LOW	Intel mode	0	–	ns
t_{CLRL}	$\overline{CS}$ LOW to $\overline{RD}$ LOW	Intel mode	0	–	ns
t_{CLEH}	$\overline{CS}$ LOW to E HIGH	Motorola mode	0	–	ns
Input comparator/output driver					
t_{SD}	sum of input and output delays	V_{DD} = 5 V ±10%; V_{DIF} = ±42 mV; 1.4 V < $V_{I(RX)}$ < V_{DD} − 1.4 V; note 2	–	36	ns

Notes

1. AC characteristics are not tested.
2. The analog input comparator may be bypassed internally using the COMP bit in the clock divider register, if external transceiver circuitry is used. This results in reduced delays (<23 ns). $V_{I(RX)}$ = input voltage on pins RX0 and RX1.

10.1 AC timing diagrams

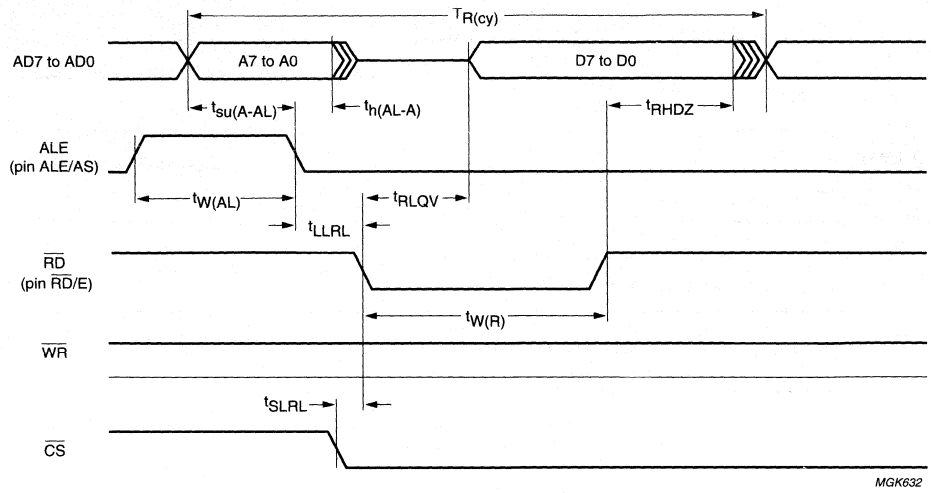


Fig.17 Read cycle timing diagram; Intel mode.

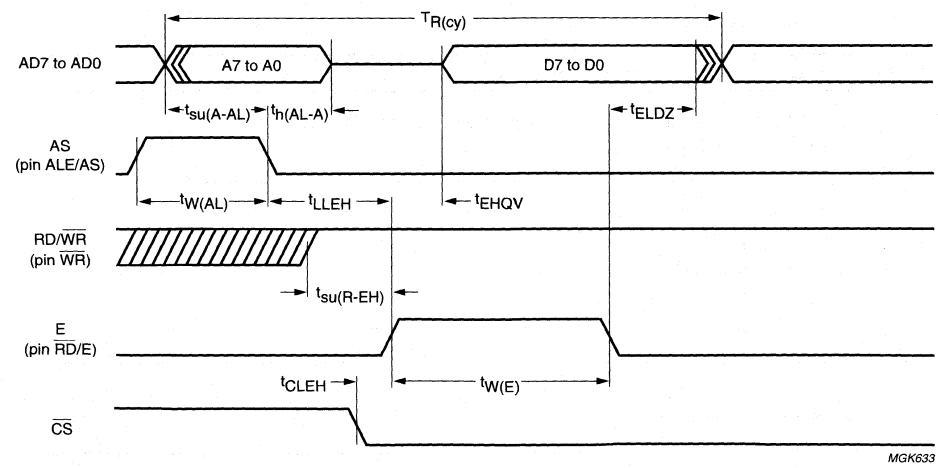


Fig.18 Read cycle timing diagram; Motorola mode.

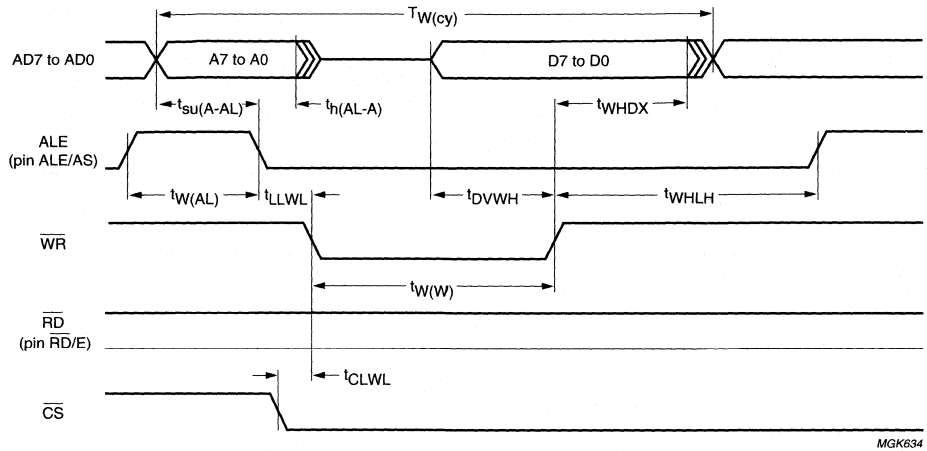


Fig.19 Write cycle timing diagram; Intel mode.

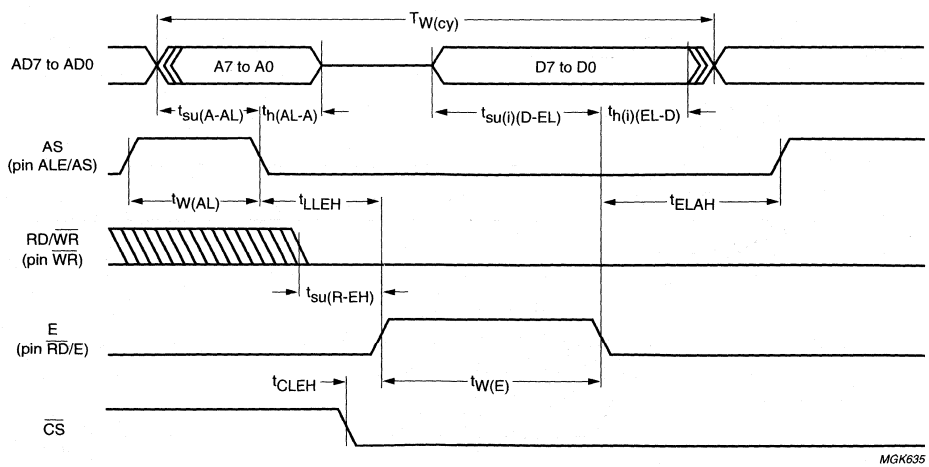


Fig.20 Write cycle timing diagram; Motorola mode.

10.2 Additional AC information

To provide optimum noise immunity under worst case conditions, the chip is powered by three separate pins and grounded by three separate pins.

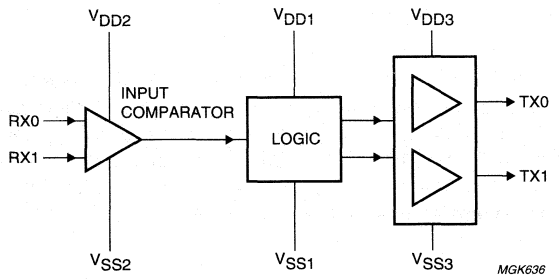
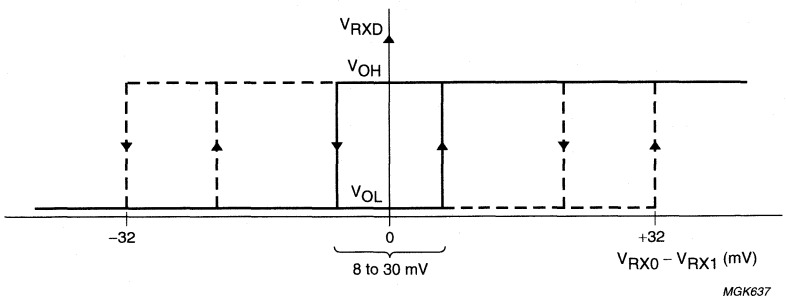


Fig.21 Optimized noise immunity block diagram.



Absolute input voltage at RX pins: 1.4 V < V_{RX} < V_{DD} − 1.4 V.
The minimum differential input voltage at the RX pins has to be greater than ±32 mV under all conditions to obtain a defined RXD output level.

Fig.22 Input comparator definitions.

SJA1000 Stand-alone CAN controller

Application Note AN97076

Authors: Peter Hank, Egon Jöhnk; Systems Laboratory, Hamburg, Germany

Abstract

The Controller Area Network (CAN) is a serial, asynchronous, multi-master communication protocol for connecting electronic control modules, sensors and actuators in automotive and industrial applications.

With the SJA1000, Philips Semiconductors provides a stand-alone CAN controller which is more than a simple replacement of the PCA82C200.

Attractive features are implemented for a wide range of applications, supporting system optimization, diagnosis and maintenance.

Summary

This application note focuses on the description of the SJA1000 as a part of a system. Diagrams illustrate the interface capability of the SJA1000 for the connection to a variety of microcontrollers and CAN transceiver circuits. General flow diagrams for programming the device in different modes are shown in detail. Configuration, transmission and Reception program examples are attached. Special emphasis has been placed on the description of the SJA1000 PeliCAN features, including useful examples, e.g., for automatic bit-rate detection, global clock synchronization and system self test.

CONTENTS

1. INTRODUCTION

The SJA1000 is a stand-alone CAN Controller product with advanced features for use in automotive and general industrial applications. It is intended to replace the PCA82C200 because it is hardware and software compatible. Due to an enhanced set of functions this device is well suited for many applications especially when system optimization, diagnosis and maintenance are important.

This report is intended to guide the user in designing complete CAN nodes based on the SJA1000. The report provides typical application circuit diagrams and flow charts for programming.

2. OVERVIEW

The stand-alone CAN controller SJA1000 [1] has two different Modes of Operation:

- BasicCAN Mode (PCA82C200 compatible)
- PeliCAN Mode

Upon Power-up the BasicCAN Mode is the default mode of operation. Consequently, existing hardware and software developed for the PCA82C200 can be used without any change. In addition to the functions known from the PCA82C200 [7], some extra features have been implemented in this mode which make the device more attractive. However, they do not influence the compatibility to the PCA82C200.

The PeliCAN Mode is a new mode of operation which is able to handle all frame types according to CAN specification 2.0B [8]. Furthermore it provides a couple of enhanced features which makes the SJA1000 suitable for a wide range of applications.

2.1 SJA1000 Features

The features of the SJA1000 can be clustered into three main groups:

Well-established PCA82C200 Functions

Features of this group have already been implemented in the PCA82C200.

Improved PCA82C200 Functions

Partly these functions have already been implemented in the PCA82C200. However, in the SJA1000 they have been improved in terms of speed, size or performance.

Enhanced Functions in PeliCAN Mode

In PeliCAN Mode the SJA1000 offers a couple of Error Analysis Functions supporting diagnosis, system maintenance and optimization. Furthermore functions for general *CPU* support and System Self Test have been added in this mode.

In the following table all SJA1000 features are listed including their main benefits for the application.

Table 1: SJA1000 Features with benefits for the application

Well-established PCA82C200 Functions

Flexible microprocessor interface	Allows interfacing most microprocessors or microcontrollers.
Programmable CAN output driver	Interface to all kind of physical layers.
CAN bit-rates up to 1Mbit/s	The SJA1000 covers the whole range of bit-rates, including high speed applications.

Improved PCA82C200 Functions

CAN 2.0B (passive)	The CAN 2.0B passive characteristics of the SJA1000 allows the CAN controller to tolerate CAN messages with 29-bit identifiers.
64 byte Receive FIFO	Up to 21 messages can be stored in the Receive FIFO, this lengthens the max. interrupt service time and avoids data overrun conditions.
24 MHz Clock frequency	Faster microprocessor access and more CAN bit-timing options.
Receive Comparator Bypass	Shortens the internal delays, resulting in a much higher CAN bus length due to an improved bit-timing programming.

Enhanced Functions in PeliCAN Mode

CAN 2.0B (active)	CAN 2.0B active support extends application field to networks with 29-bit identifiers.
Transmit Buffer	Single message transmit buffer for messages with 11-bit or 29-bit identifiers.
Enhanced Acceptance Filter	Two acceptance filter modes supporting both 11-bit and 29-bit identifier filtering.
Readable Error Counters	Supports error analysis which can be used for:
Programmable Error Warning Limit	- diagnostics, system maintenance and system optimization
Error Code Capture Register	during the prototype phase and during normal operation.
Error Interrupts	
Arbitration Lost Capture Interrupt	Supports system optimization including message latency time analysis.
Single Shot Transmission	Minimizes software commands and allows fast reloading of transmit buffer.
Listen Only Mode	SJA1000 can operate as a passive CAN monitor which can be used for analyzing the CAN bus traffic or for automatic bit-rate detection.
Self Test Mode	Supports functional self tests of complete CAN nodes or self reception in a system.

2.2 CAN Node Architecture

Generally each CAN module can be divided into different functional blocks. The connection to the CAN bus lines is usually built with a **CAN Transceiver** optimized for the applications [3], [4], [5]. The transceiver controls the logic level signals from the CAN controller into the physical levels on the bus and vice versa.

The next upper level is a **CAN Controller** which implements the complete CAN protocol defined in the CAN Specification [8]. Often it also covers message buffering and acceptance filtering.

All these CAN functions are controlled by a **Module Controller** which performs the functionality of the application. For example, it controls actuators, reads sensors and handles the man-machine interface (MMI).

As shown in Figure 1 the SJA1000 stand-alone CAN controller is always located between a microcontroller and the transceiver, which is an integrated circuit in most cases.

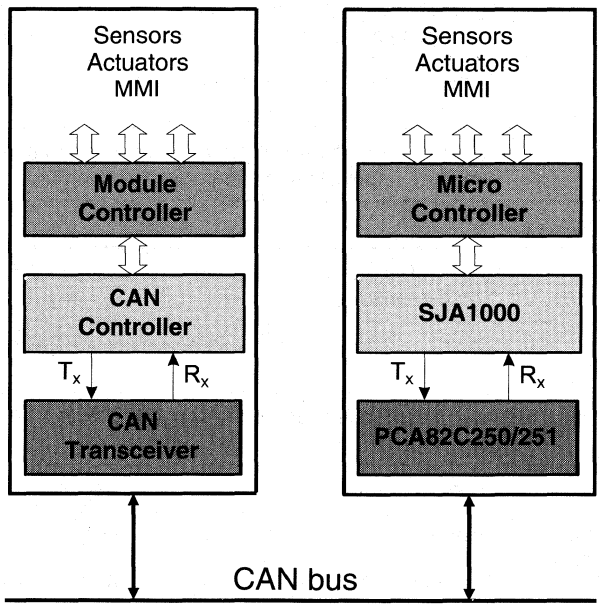


Figure 1: CAN Module Set-up

2.3 Block Diagram

The following figure shows the block diagram of the SJA1000.

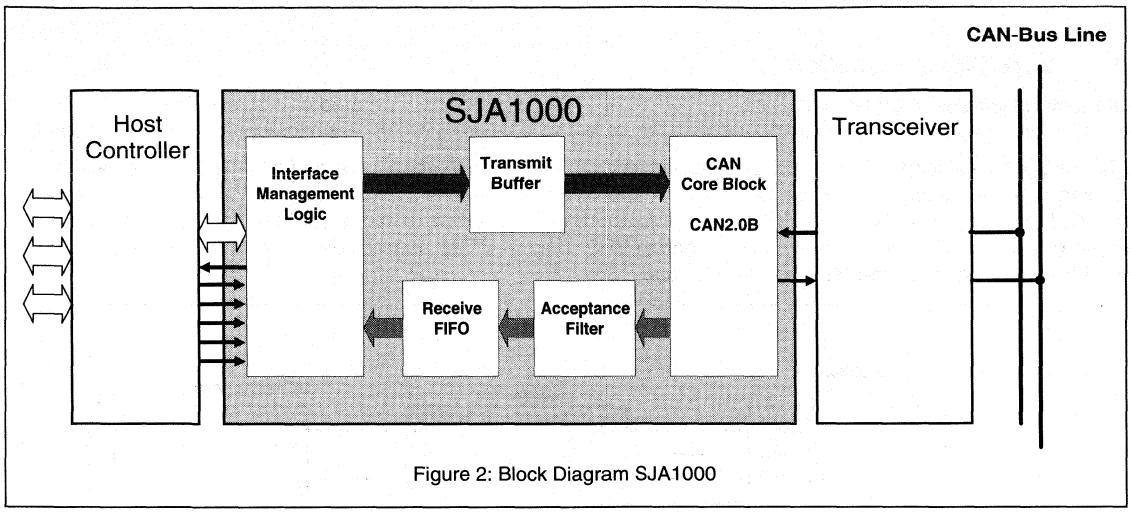


Figure 2: Block Diagram SJA1000

The **CAN Core Block** controls the transmission and reception of CAN frames according to the CAN specification.

The **Interface Management Logic** block performs a link to the external host controller which can be a microcontroller or any other device. Every register access via the SJA1000 multiplexed address/data bus and controlling of the read/write strobes is handled in this unit. Additionally to the BasicCAN functions known from the PCA82C200, new PeliCAN features have been added. As a consequence of this, additional registers and logic have been implemented mainly in this block.

The **Transmit Buffer** of the SJA1000 is able to store one complete message (Extended or Standard). Whenever a transmission is initiated by the host controller the Interface Management Logic forces the CAN Core Block to read the CAN message from the Transmit Buffer.

When receiving a message, the CAN Core Block converts the serial bit stream into parallel data for the **Acceptance Filter**. With this programmable filter the SJA1000 decides which messages actually are received by the host controller.

All received messages accepted by the acceptance filter are stored within a **Receive FIFO**. Depending on the mode of operation and the data length up to 32 messages can be stored. This enables the user to be more flexible when specifying interrupt services and interrupt priorities for the system because the probability of data overrun conditions is reduced extremely.

3. SYSTEM

For connection to the host controller, the SJA1000 provides a multiplexed address/data bus and additional read/write control signals. The SJA1000 could be seen as a peripheral memory mapped I/O device for the host controller.

3.1 SJA1000 Application

Configuration Registers and pins of the SJA1000 allow to use all kinds of integrated or discrete CAN transceivers. Due to the flexible microcontroller interface applications with different microcontrollers are possible.

In Figure 3 a typical SJA1000 application diagram including 80C51 microcontroller and PCA82C251 transceiver is shown. The CAN controller functions as a clock source and the reset signal is generated by an external reset circuitry. In this example the chip select of the SJA1000 is controlled by the microcontroller port function P2.7. Instead of this, the chip select input could be tied to VSS. Control via an address decoder is possible, e.g., when the address/data bus is used for other peripherals.

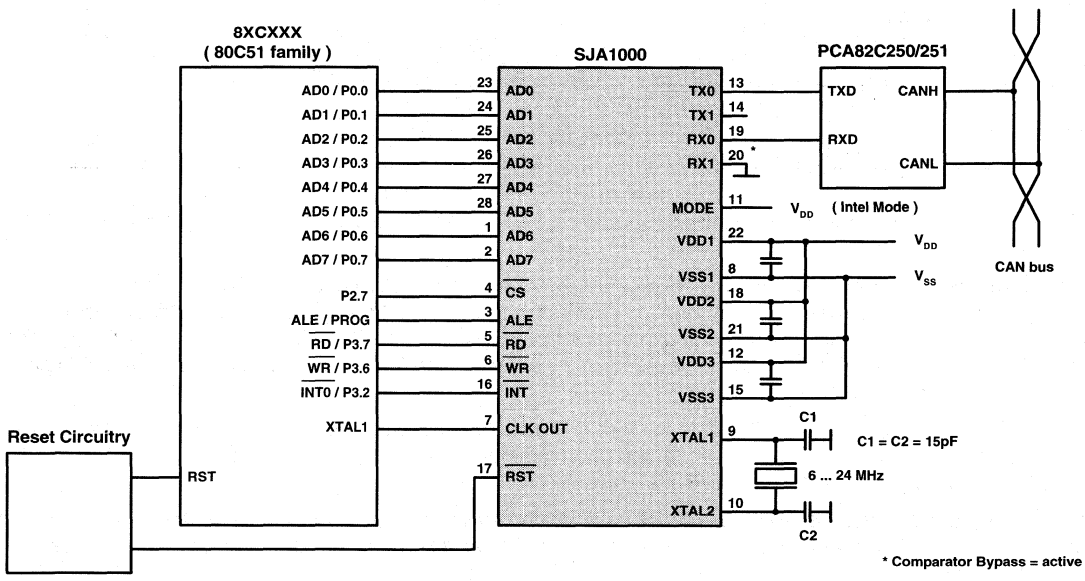


Figure 3: Typical SJA1000 Application

3.2 Power Supply

The SJA1000 has three pairs of voltage supply pins which are used for different digital and analog internal blocks of the CAN controller.

VDD1 / VSS1: internal logic (digital)
VDD2 / VSS2: input comparator (analog)
VDD3 / VSS3: output driver (analog)

The supply has been separated for better EME behaviour. For instance the VDD2 can be de-coupled via an RC filter for noise suppression of the comparator.

3.3 Reset

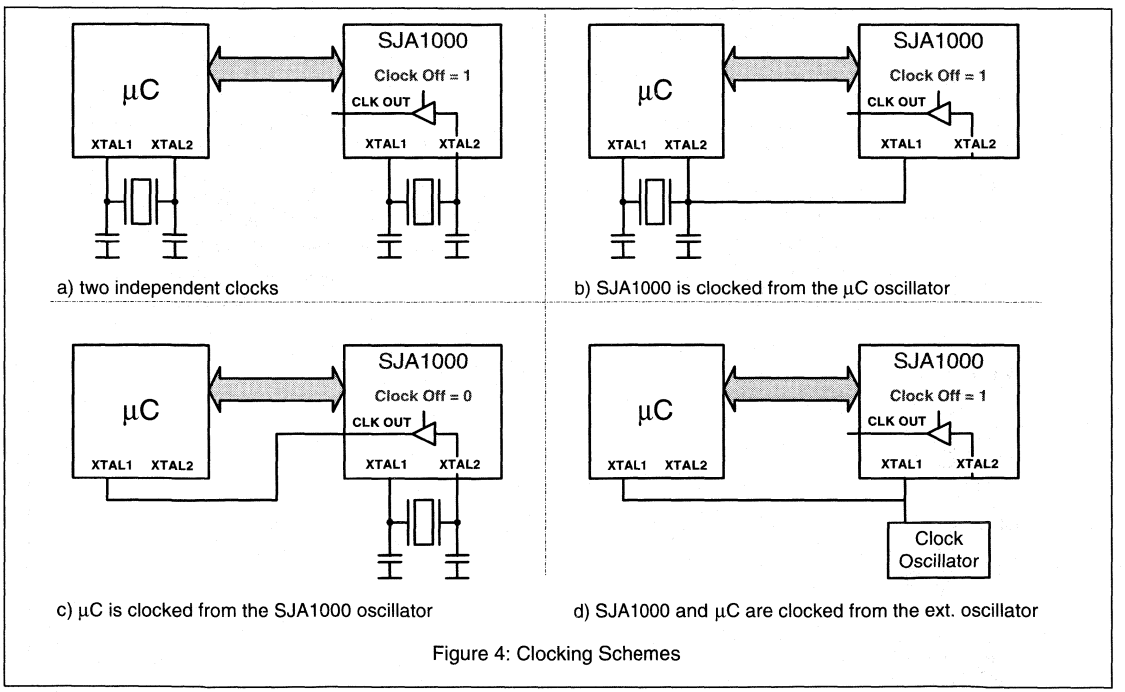
For a proper reset of the SJA1000 a stable oscillator clock has to be provided at XTAL1 of the CAN controller, see also chapter 3.4. An external reset on pin 17 is synchronized and internally lengthened to 15 t_{XTAL}. This guarantees a correct reset of all SJA1000 registers (see [1]). Note that an oscillator start-up time has to be taken into account upon power-up.

3.4 Oscillator and Clocking Strategy

The SJA1000 can operate with the on-chip oscillator or with external clock sources. Additionally the CLK OUT pin can be enabled to output the clock frequency for the host controller. Figure 4 shows four different clocking principles for applications with the SJA1000. If the CLK OUT signal is not needed, it can be switched off with the Clock Divider register (Clock Off = 1). This will improve the EME performance of the CAN node.
The frequency of the CLK OUT signal can be changed with the Clock Divider Register:

$f_{CLK\ OUT} = f_{XTAL}$ / Clock Divider factor (1,2,4,6,8,10,12,14).

Upon power up or hardware reset the default value for the Clock Divider factor depends on the selected interface mode (pin 11). If a 16 MHz crystal is used in Intel mode, the frequency at CLK OUT is 8 MHz. In Motorola mode a Clock Divider factor of 12 is used upon reset which results in 1,33 MHz in this case.



a) two independent clocks

b) SJA1000 is clocked from the µC oscillator

c) µC is clocked from the SJA1000 oscillator

d) SJA1000 and µC are clocked from the ext. oscillator

Figure 4: Clocking Schemes

3.4.1 Sleep and Wake-up

Upon setting the Go To Sleep bit in the Command Register (BasicCAN mode) or the Sleep Mode bit in the Mode Register (PeliCAN mode) the SJA1000 will enter Sleep Mode if there is no bus activity and no interrupt is pending. The oscillator keeps on running until 15 CAN bit times have been passed. This allows a microcontroller clocked with the CLK OUT frequency to enter its own low power consumption mode.
If one of three possible wake-up conditions [1] occurs the oscillator is started again and a Wake-up interrupt is generated. As soon as the oscillator is stable the CLK OUT frequency is active.

3.5 CPU Interface

The SJA1000 supports the direct connection to two famous microcontroller families: 80C51 and 68xx. With the MODE pin of the SJA1000 the interface mode is selected.

Intel Mode: MODE = high
Motorola Mode: MODE = low

The connection for the address/data bus and the read/write control signals in both Intel and Motorola mode is shown in Figure 5. For Philips 8-bit microcontrollers based on the 80C51 family and the 16-bit microcontrollers with XA architecture the Intel Mode is used.

For other controllers additional glue logic is necessary for adaptation of the address/data bus and the control signals. However, it has to be made sure that no write pulses are generated during power-up. Another possibility is to disable the CAN controller with a high-level on the chip select input in this time.

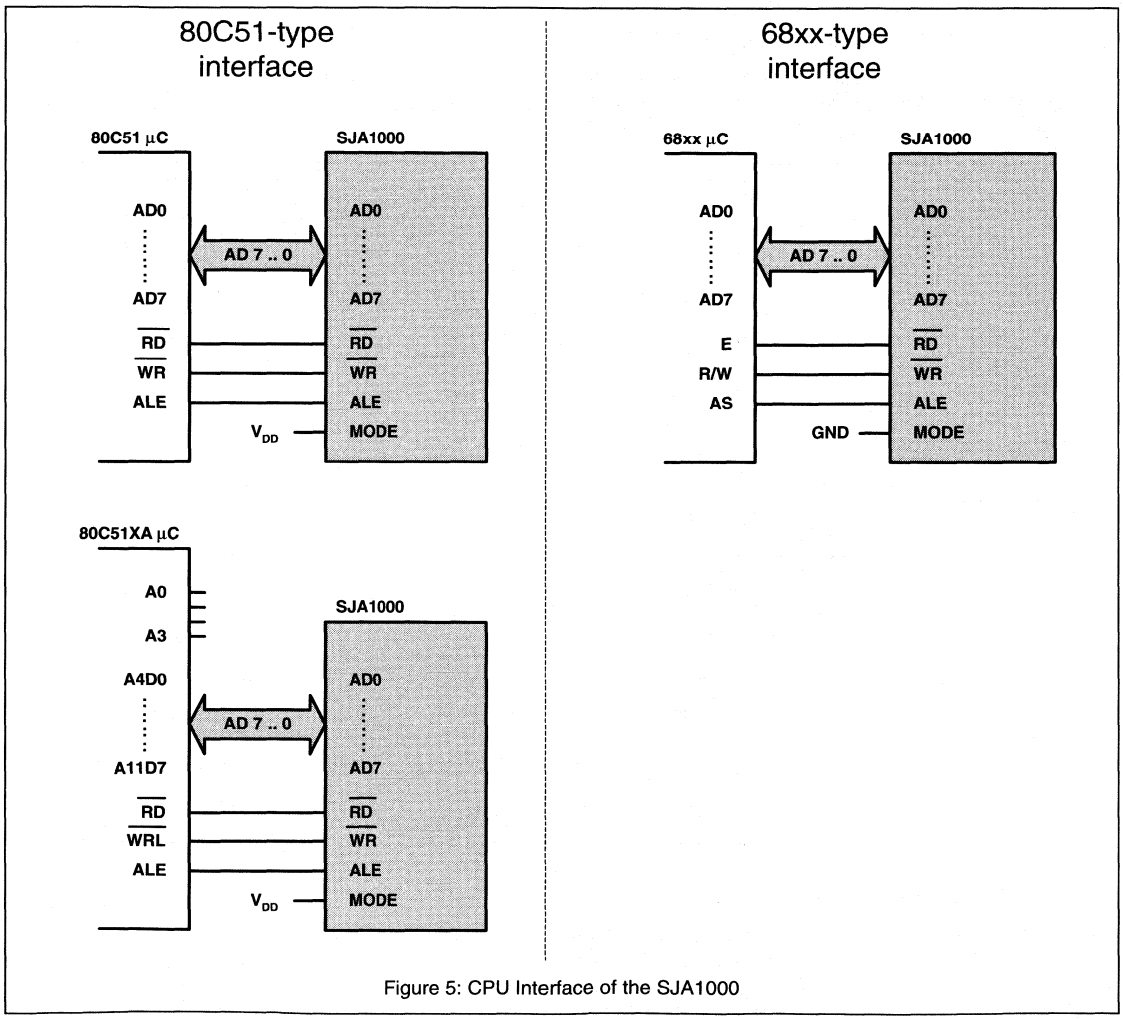


Figure 5: CPU Interface of the SJA1000

3.6 Physical Layer Interface

For compatibility purposes with the PCA82C200, the SJA1000 includes an analog receive input comparator circuit. This integrated comparator can be used if the transceiver function is realized with discrete components.

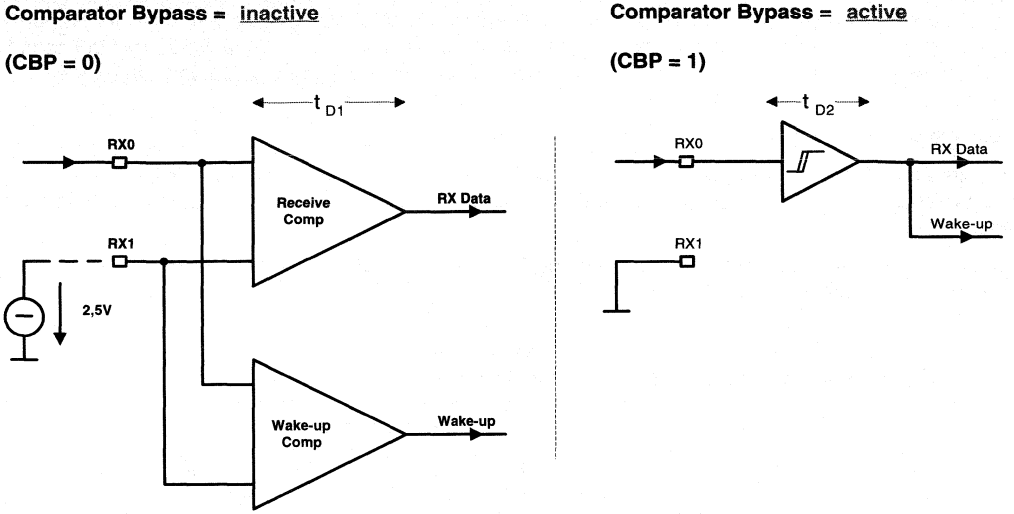


Figure 6: SJA1000 Receive Input Comparator

If an external integrated transceiver circuit is used and the comparator bypass function is not enabled in the Clock Divider Register, the RX1 input has to be connected to a reference voltage of 2.5V (reference voltage output of existing transceiver circuits). Figure 6 shows the equivalent circuits for both configurations: CBP = active and CBP = inactive. Additionally the path for the wake-up signal is drawn.
For all new applications where an integrated transceiver circuit is used, it is recommended to activate the comparator bypass function of the SJA1000 (Figure 7). If this function is enabled, a schmitt-trigger input is used and the internal propagation delay t_{D2} is much shorter as the delay t_{D1}. of the receive comparator. This has a positive impact on the maximum bus length [6]. Additionally, it will reduce the supply current in sleep mode significantly.

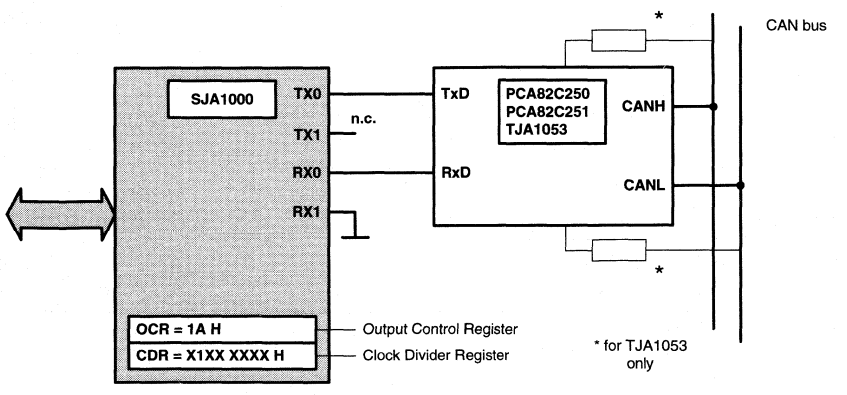


Figure 7: Standard application with integrated transceiver circuit

4. CONTROL OF CAN COMMUNICATION

4.1 Basic Functions and Registers for Controlling the SJA1000

The functionality with respect to configuration and activities of the SJA1000 is given by the program of the host controller. Thus the SJA1000 is tailored to meet the requirements of CAN-bus systems with different properties. The data exchange between the host controller and the SJA1000 is done via a set of registers (control segment) and a RAM (message buffer). The registers and an address window to a part of the RAM, making up the Transmit and Receive Buffers, appear to the host controller as peripheral registers.

Table 2 lists these registers grouped according to their usage in a system.

Note, that some registers are available in PeliCAN mode only and that the Control Register is available in BasicCAN mode only. Furthermore some registers are read only or write only and some can be accessed during Reset Mode only.

More information about the registers with respect to read and/or write access, bit definition and reset values, can be found in the data sheet [1].

Table 2: Classification of the internal registers of the SJA1000

Type of Usage	Register Name	(Symbol)	Register Address: PeliCAN mode	Register Address: BasicCAN mode	Functionality
elements for selecting different operation modes	Mode	(MOD)	0	—	Sleep-. Acceptance Filter-, Self Test-, Listen Only- and Reset-Mode selection
	Control	(CR)	—	0	Reset Mode selection in BasicCAN mode
	Command	(CMR)	—	1	Sleep mode command in BasicCAN mode
	Clock Divider	(CDR)	31	31	set-up of clock signal at CLKOUT (pin 7) selection of PeliCAN Mode, Comparator Bypass Mode, TX1 (pin 14) Output Mode
elements for setting up the CAN communication	Acceptance Code, Mask	(ACR) (AMR)	16-19 20-23	4, 5	selection of bit patterns for Acceptance Filtering
	Bus Timing 0 1	(BTR0) (BTR1)	6 7	6 7	set-up of Bit Timing Parameters
	Output Control	(OCR)	8	8	selection of Output Driver properties

Table 2: Classification of the internal registers of the SJA1000 **(continued)**

Type of Usage	Register Name (Symbol)	Register Address: PeliCAN mode	Register Address: BasicCAN mode	Functionality
basic elements for the CAN communication	Command (CMR)	1	1	commands for Self Reception, Clear Data Overrun, Release Receive Buffer, Abort Transmission and Transmission Request
	Status (SR)	2	2	status of message buffers, status of CAN Core Block
	Interrupt (IR)	3	3	CAN Interrupt flags
	Interrupt Enable (IER)	4	—	enable/disable of interrupt events in PeliCAN mode
	Control (CR)	—	0	enable/disable of interrupt events in BasicCAN mode
elements for a comprehensive error detection and analysing	Arbitration Lost Capture (ALC)	11	—	shows bit position, where arbitration was lost
	Error Code Capture (ECC)	12	—	shows last error type and location
	Error Warning Limit (EWLR)	13	—	selection of threshold for generating an Error Warning Interrupt
	RX Error Counter (RXERR)	14	—	reflects the current value of the Receive Error Counter
	TX Error Counter (TXERR)	14, 15	—	reflects the current value of the Transmit Error Counter
	Rx Message Counter (RMC)	29	—	number of messages in the Receive FIFO
	Rx Buffer Start Addr. (RBSA)	30	—	shows the current internal RAM address of the message available in the Receive Buffer
message buffers	Transmit Buffer (TXBUF)	16-28	10-19	
	Receive Buffer (RXBUF)	16-28	20-29	

4.1.1 Transmit Buffer / Receive Buffer

The data to be transmitted on the CAN bus is loaded into the memory area of the SJA1000, called "Transmit Buffer". The data received from the CAN bus is stored in the memory area of the SJA1000, called "Receive Buffer". These buffers contains 2, 3 or 5 bytes for the identifier and frame information (dependent on mode and frame type) and up to 8 data bytes. For further information about the definition and composition of the bits in the message buffers see the data sheet [1].

- BasicCAN mode: The buffers are 10-bytes long (see Table 3).
  - 2 identifier bytes
  - up to 8 data bytes.
- PeliCAN mode: The buffers are 13 bytes long (see Table 4).
  - 1 byte for Frame Information
  - 2 or 4 identifier bytes (Standard Frame or Extended Frame)
  - up to 8 data bytes.

Table 3: Layout of Rx- and Tx-Buffer in BasicCAN mode

CAN Addr. (dec.)	Name	Composition and Remarks
Tx-Buffer: 10 Rx-Buffer: 20	Identifier Byte 1	8 Identifier bits
Tx-Buffer: 11 Rx-Buffer: 21	Identifier Byte 2	3 Identifier bits, 1 Remote Transmission Request bit, 4 bits for the Data Length Code, indicating the amount of data bytes
Tx-Buffer: 12-19 Rx-Buffer: 22-29	Data Byte 1 - 8	up to 8 data bytes as indicated by the Data Length Code

Table 4: Layout of Rx-[1] (read access) and Tx-Buffer (write access[2]) in PeliCAN mode

CAN Addr. (dec.)	Name	Composition and Remarks
16	Frame Information	1 bit indicating, if the message contains a Standard or Extended frame 1 Remote Transmission Request bit 4 bits for the Data Length Code, indicating the amount of data bytes
17, 18	Identifier Byte 1, 2	Standard Frame: 11 Identifier bits Extended Frame: 16 Identifier bits
19, 20	Identifier Byte 3, 4	Extended Frame only: 13 Identifier bits
Frame type Standard: 19 - 26 Extended: 21 - 28	Data Byte 1 - 8	up to 8 data bytes as indicated by the Data Length Code

1. The whole Receive FIFO (64 bytes) can be accessed using the CAN addresses 32 to 95 (see also chapter 5.1).
2. A read access of the Tx-Buffer can be done using the CAN addresses 96 to 108 (see also chapter 5.1)

4.1.2 Acceptance Filter

The stand-alone CAN controller SJA1000 is equipped with a versatile acceptance filter, which allows an automatic check of the identifier and data bytes. Using these effective filtering methods, messages or a group of messages not valid for a certain node can be prevented from being stored in the Receive Buffer. Thus it is possible to reduce the processing load of the host controller.

The filter is controlled by the acceptance code and mask registers according to the algorithms given in the data sheet [1]. The received data is compared bitwise with the value contained in the Acceptance Code register. The Acceptance Mask Register defines the bit positions, which are relevant for the comparison (0 = relevant, 1 = not relevant). For accepting a message all **relevant** received bits have to match the respective bits in the Acceptance Code Register.

Acceptance Filtering in BasicCAN Mode

This mode is implemented in the SJA1000 as a plug-and-play replacement (hardware and software) for the PCA82C200. Thus the acceptance filtering corresponds to the possibilities, which were found in the PCA82C200 [7]. The filter is controlled by two 8-bit wide registers – Acceptance Code Register (ACR) and Acceptance Mask Register (AMR). The 8 most significant bits of the identifier of the CAN message are compared to the values contained in these registers, see also Figure 8. Thus always groups of eight identifiers can be defined to be accepted for any node.

Example:

The Acceptance Code register (ACR) contains:

The Acceptance Mask register (AMR) contains:

Messages with the following 11-bit identifiers are accepted

(x = don't care)

	MSB							LSB		
ACR	**0**	**1**	1	1	0	**0**	**1**	**0**		
AMR	0	0	***1***	***1***	***1***	0	0	0		
	0	**1**	x	x	x	**0**	**1**	**0**	x	x	x
	ID.10										ID.0

At the bit positions containing a "1" in the Acceptance Mask register, any value is allowed in the composition of the identifier. The same is valid for the three least significant bits. Thus 64 different identifiers are accepted in this example. The other bit positions must be equal to the values in the Acceptance Code register.

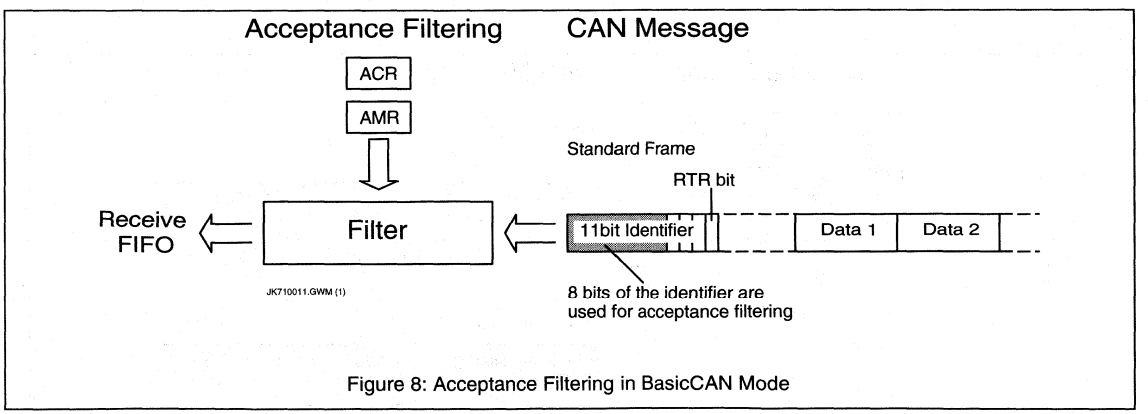


Figure 8: Acceptance Filtering in BasicCAN Mode

Acceptance Filtering in PeliCAN Mode

The acceptance filtering has been expanded for the PeliCAN mode: Four 8-bit wide Acceptance Code registers (ACR0, ACR1, ACR2 and ACR3) and Acceptance Mask registers (AMR0, AMR1, AMR2 and AMR3) are available for a versatile filtering of messages. These registers can be used for controlling a single long filter or two shorter filters, as shown in Figure 9 and Figure 10. Which bits of the message are used for the acceptance filtering, depend on the received frame (Standard or Extended) and on the selected filter mode (single or dual filter). Table 5 gives more information about which bits of the message are compared with the Acceptance Code and Mask bits. As it is seen from the figures and the table, it is possible to include the RTR bit and even data bytes in the acceptance filtering for Standard Frames. In any case for all message bits, which shall **not** be included in the acceptance filtering (e.g. if groups of messages are defined for acceptance), the Acceptance Mask Register must contain a "1" at the corresponding bit position.

If a message doesn't contain data bytes (e.g. in a Remote Frame or if the Data Length Code is zero) but data bytes are included in the acceptance filtering, such messages are accepted, if the identifier up to the RTR bit is valid.

Example 1:

Let us assume, that the same 64 Standard Frame messages as described in the example on page 18 have to be filtered in PeliCAN mode.

This can be done using one long filter (Single Filter Mode).

The Acceptance Code Registers (ACRn) and Acceptance Mask Registers (AMRn) contain:

n	0	1 (upper 4 bits)	2	3
ACRn	0 1 X X X 0 1 0	X X X X	X X X X X X X X	X X X X X X X X
AMRn	0 0 1 1 1 0 0 0	1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
accepted messages (ID.28..ID.18, RTR)	0 1 x x x 0 1 0 x x x x			

("X" = irrelevant, "x" = don't care, only the upper 4 bits of ACR1 and AMR1 are used)

At the bit positions containing a "1" in the Acceptance Mask registers, any value is allowed in the composition of the identifier, for the Remote Transmission Request bit and for the bits of data byte 1 and 2.

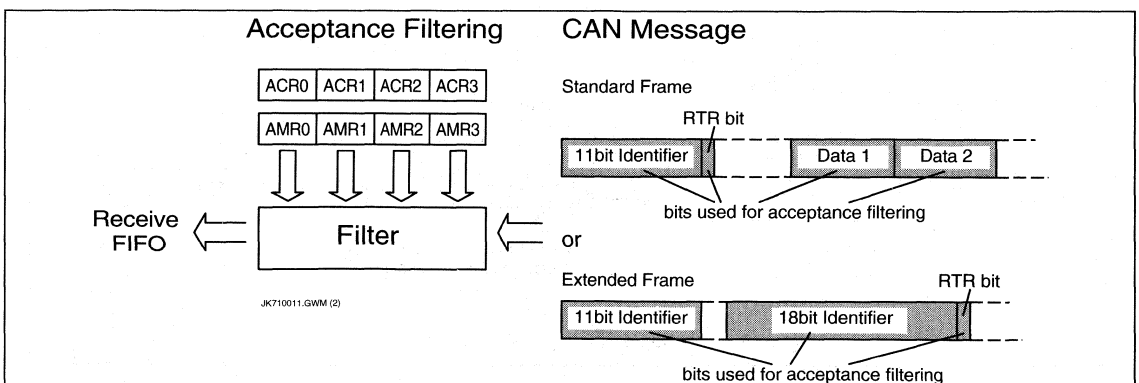


Figure 9: Acceptance Filtering in PeliCAN Mode (Single Filter Mode)

Example 2:

Suppose the following 2 messages with a Standard Frame Identifier have to be accepted without any further decoding of the identifier bits. Data and Remote Frames have to be received correctly. Data bytes are not involved in the acceptance filtering.

message 1: (ID.28) 1***0***11 ***1***100 101 (ID.18)
message 2: (ID.28) 1***1***11 ***0***100 101 (ID.18)

Using the Single Filter Mode results in accepting four messages and not only the requested two:

n	0	1 (upper 4 bits)	2	3
ACRn	1X11 X100	101X	XXXX XXXX	XXXX XXXX
AMRn	0100 1000	0001	1111 1111	1111 1111
accepted messages (ID.28..ID.18, RTR)	1011 0100 1011 0100 1011 1100 1111 1100	101x 101x 101x 101x	(message 2) (message 1)	

("X" = irrelevant, "x" = don't care, only the upper 4 bits of ACR1 and AMR1 are used)

This result does **not** meet the request for receiving 2 messages without any further decoding.

Using the Dual Filter mode gives the correct result:

	Filter 1			Filter 2	
n	0	1	3 lower 4 bits	2	3 upper 4 bits
ACRn	1011 1100	101X XXXX	... XXXX	1111 0100	101X ...
AMRn	0000 0000	0001 1111	... 1111	0000 0000	0001 ...
accepted messages (ID.28..ID.18, RTR)	1011 1100 101x (message 1)			1111 0100 101x (message 2)	

("X" = irrelevant, "x" = don't care)

Message 1 is accepted by Filter 1 and message 2 by Filter 2. As messages are accepted and stored into the Receive FIFO if they are accepted at least by one of the two filters, this solution meets the request.

Example 3:

In this example a group of messages with an Extended Frame Identifier are filtered using a long single acceptance filter.

n	0	1	2	3 (upper 6 bits)
ACRn	1011 0100	1011 000X	1100 XXXX	0011 0XXX
AMRn	0000 0000	0000 0001	0000 1111	0000 0111
accepted messages (ID.28..ID.0, RTR)	1011 0100 1011 000x 1100 xxxx 0011 0x			

("X" = irrelevant, "x" = don't care, only the upper 6 bits of ACR3 and AMR3 are used)

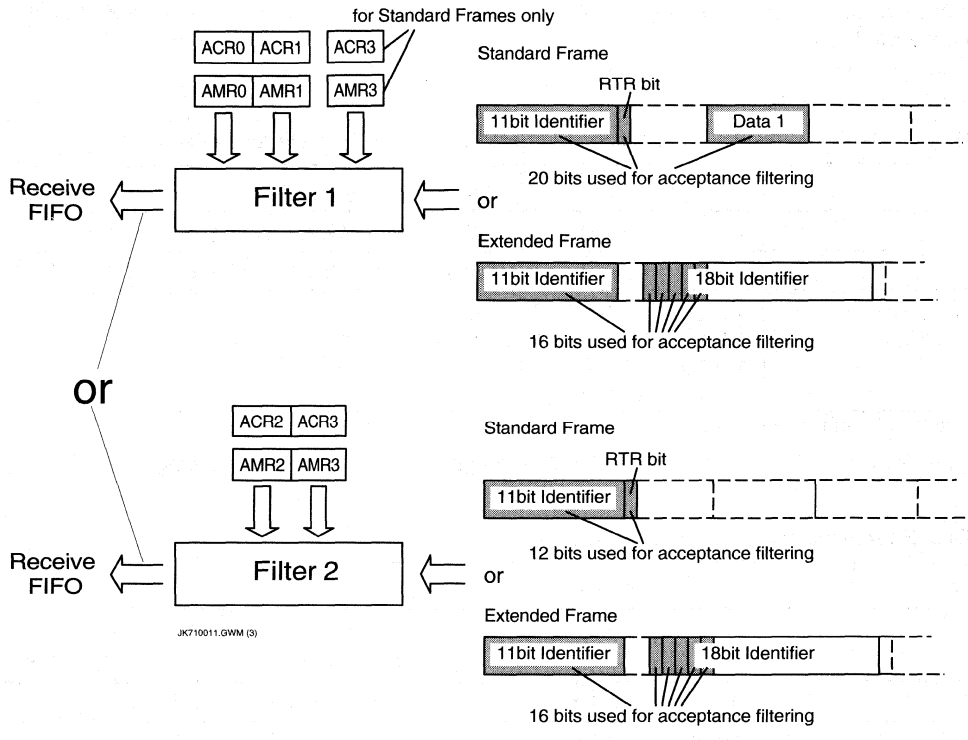


Figure 10: Acceptance Filtering in PeliCAN Mode (Dual Filter Mode)

Example 4:

There are systems, which use Standard Frames only and identify messages by the 11-bit identifier and the first two data bytes. Such a protocol is used, e.g., in the DeviceNet, where the first two data bytes define a message header and the fragmentation protocol, if messages contain more than 8 data bytes. For this system type the SJA1000 can filter two data bytes in single filter mode and one data byte in dual filter mode in addition to the 11-bit identifier and the RTR-bit.

Using the Dual Filter mode, the following example shows effective filtering of messages in such a system:

	Filter 1			Filter 2	
n	0	1	3 lower 4bits	2	3 upper 4 bits
ACRn	1 1 1 0 1 0 1 1	0 0 1 0 1 1 1 1	... 1 0 0 1	1 1 1 1 0 1 0 0	X X X 0 ...
AMRn	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	... 0 0 0 0	0 0 0 0 0 0 0 0	1 1 1 0 ...
accepted messages	1 1 1 0 1 0 1 1 ID + RTR	0 0 1 0 1 1 1 1	... 1 0 0 1 first data byte	1 1 1 1 0 1 0 0 ID	x x x 0 RTR

("X" = irrelevant, "x" = don't care)

Filter 1 is used for filtering messages with
- the identifier "1 1 1 0 1 0 1 1 0 0 1"
- RTR = "0" i.e. Data Frames only and
- the data byte "1 1 1 1 1 0 0 1" (this means e.g. for the DeviceNet: all fragments for one message are filtered).

Filter 2 is used for filtering a group of 8 messages with

- the identifiers "1 1 1 1 0 1 0 0 0 0 0" through "1 1 1 1 0 1 0 0 1 1 1" and
- RTR = "0", i.e. Data Frames only.

Table 5: Summary of Acceptance Filter in PeliCAN mode

Frame Type	Single Filter mode (Figure 9)	Dual Filter mode (Figure 10)
Standard	message bits used for acceptance: - 11 bit identifier - RTR Bit - 1st data byte (8 bit) - 2nd data byte (8 bit) Acceptance Code & Mask registers used: - ACR0/upper 4 bits of ACR1/ACR2/ACR3 - AMR0/upper 4 bits of AMR1/AMR2/AMR3 (unused bits of the Acceptance Mask Register should be set to "1")	Filter 1 message bits used for acceptance: - 11 bit identifier - RTR Bit - 1st data byte (8 bit) Acceptance Code & Mask registers used: - ACR0/ACR1/lower 4 bits of ACR3 - AMR0/AMR1/lower 4 bits of AMR3 Filter 2 message bits tested for acceptance: - 11 bit identifier - RTR Bit Acceptance Code & Mask registers used: - ACR2/upper 4 bits of ACR3 - AMR2/upper 4 bits of AMR3
Extended	message bits used for acceptance: - 11 bit basic identifier - 18 bit extended identifier - RTR Bit Acceptance Code & Mask registers used: - ACR0/ACR1/ACR2/upper 6 bits of ACR3 - AMR0/ AMR1/ AMR2/ upper 6 bits of AMR3 (unused bits of the Acceptance Mask Register should be set to "1")	Filter 1 message bits used for acceptance: - 11 bit basic identifier - 5 most significant bits of extended identifier Acceptance Code & Mask registers used: - ACR0/ACR1 and AMR0/AMR1 Filter 2 message bits tested for acceptance: - 11 bit basic identifier - 5 most significant bits of extended identifier Acceptance Code & Mask registers used: - ACR2/ACR3 and AMR2/AMR3

4.2 Functions for CAN Communications

The steps to be taken for establishing communication via the CAN bus are:

- after power-on of the system
  - setting up the host controller with respect to hardware and software links to the SJA1000
  - setting up the CAN controller for the communication with respect to the selection of mode, acceptance filtering, bit timing etc. – to be done also after a hardware reset of the SJA1000
- during the main process of the application
  - prepare messages to be transmitted and activate the SJA1000 to transmit them
  - react on messages received by the CAN controller
  - react on errors occurred during communication

Figure 11 shows the general flow of a program. In the following paragraphs the flows, which refer directly to controlling the SJA1000, are described in more detail.

4.2.1 Initialization

As mentioned before, the stand-alone CAN controller SJA1000 has to be set up for CAN communication after power-on or after a hardware reset. Furthermore the SJA1000 may be re-configured (re-initialized) during operation by the host controller, which may send a (software) reset request. The flow is given in Figure 12. A programming example using an 80C51 microcontroller derivative is given in this chapter.

After power-on the host controller runs through its own special reset routine and then it enters the set-up routine for the SJA1000. As the part "configure control lines..." of Figure 11 is specific to the used microcontroller, it can not be discussed in general in this place. However, the example in this chapter shows, how to configure an 80C51 derivative.

For the following description of the initialization processing see Figure 12. It is assumed, that after power-on also the stand-alone CAN controller gets a reset pulse (LOW level) at the pin 17, enabling it to enter the reset mode. Before setting up registers of the SJA1000, the host controller should check by reading the reset mode/request flag, if the SJA1000 has reached the reset mode, because the registers, which get the configuration information, can be written only during reset mode.

The host controller has to configure the following registers of the control segment of the SJA1000 in reset mode:

- Mode Register (in PeliCAN mode only), selecting the following modes of operation for this application
  - Acceptance Filter mode
  - Self Test mode
  - Listen Only mode
- Clock Divider Register, defining
  - if the BasicCAN or the PeliCAN mode is used
  - if the CLKOUT pin is enabled
  - if the CAN input comparator is bypassed
  - if the TX1 output is used as a dedicated receive interrupt output
- Acceptance Code and Acceptance Mask Registers
  - defining the acceptance code for messages to be received
  - defining the acceptance mask for relevant bits of the message to be compared with corresponding bits of the acceptance code

- Bus Timing Registers, see also [6]
  - defining the bit-rate on the bus
  - defining the sample point in a bit period (bit sample point)
  - defining the number of samples taken in a bit period
- Output Control Register
  - defining the used output mode of the CAN bus output pins TX0 and TX1
    Normal Output Mode, Clock Output Mode, Bi-Phase Output Mode or Test Output Mode
  - defining the output pin configuration for TX0 and TX1
    Float, Pull-down, Pull-up or Push/Pull and polarity

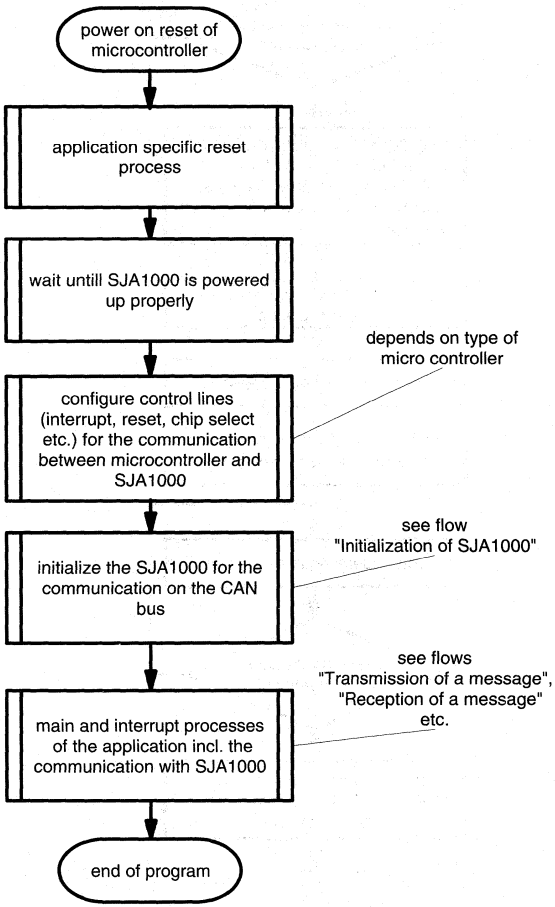


Figure 11: General program flow

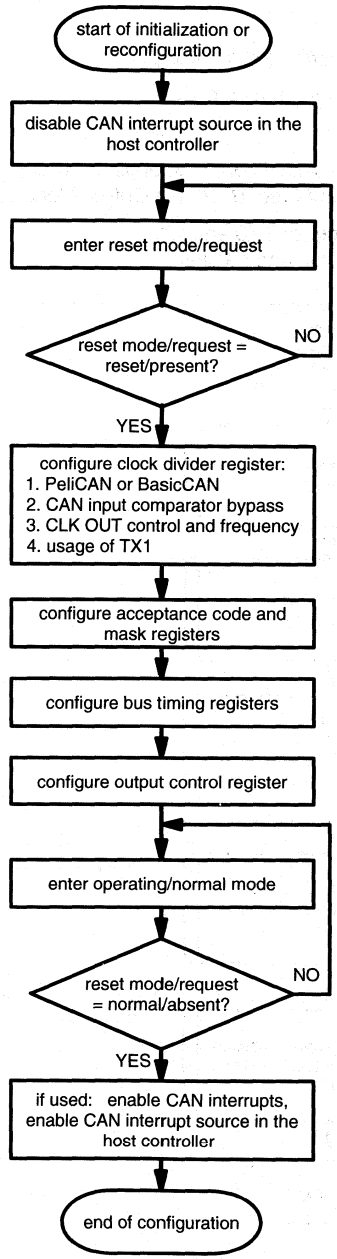


Figure 12: Flow Diagram "Initialization of SJA1000"

After having transferred this information to the control segment of the SJA1000, it is switched into operation mode by clearing the reset mode/request flag. It has to be checked, if the flag is really cleared and the operation mode is entered before going on further. This is done by reading the flag in a loop.

The reset mode/request flag cannot be cleared as long as a hardware reset still is pending (LOW-level at pin 17), because this will force the reset mode/request flag to "reset/present" (see the data sheet for further information [1]). Thus this loop is used to continuously trying to clear the flag **and** checking if the reset mode was left successfully.

After having entered the operation mode, the interrupts from the CAN controller may be enabled, if appropriate.

Example: Configuration and Initialization of SJA1000

This example is based on the application example given in Figure 3 on page 11. In the following programming examples a micro controller S87C654 is assumed as host controller. It is clocked by the clock output from the SJA1000. During power-on a reset circuit delivers the hardware reset for both the micro controller and the CAN controller. The Clock Divider Register of the SJA1000 is cleared during reset [1]. Thus the CAN controller comes up in BasicCAN mode with the clock output enabled, being able to deliver the clock for the S87C654 as soon as the crystal oscillator is running. The frequency of this clock is $f_{CLK}/2$ as pin 11 is connected to support controllers of the 80C51-family. Upon receiving the clock the micro controller starts its own reset process as shown in Figure 11.

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., "InterruptEnReg" points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language "C" is used for programming.

In this example it is assumed, that the CAN controller has to be initialized for being used in PeliCAN mode. It should be easy to derive the corresponding initialization for the BasicCAN mode.

The first step must be to set up a communication link (chip select, interrupts, etc.) between the host controller and the SJA1000 ("configure Control lines..." in Figure 11).

```
/* define interrupt priority & control (level-activated, see chapter 4.2.5)   */
 PX0 = PRIORITY_HIGH;   /* CAN HAS A HIGH PRIORITY INTERRUPT                   */
 IT0 = INTLEVELACT;     /* set interrupt0 to level activated                   */

/* enable the communication interface of the SJA1000                          */
 CS  = ENABLE_N;        /* Enable the SJA1000 interface                        */

/*- end of the definition of the communication link -------------------------*/
```

The second step is to initialize all internal registers of the SJA1000. As some registers can be written to during reset mode only, this has to be checked before writing. After power-on the SJA1000 is set into reset mode, but in a loop it can be checked, if the reset mode has been set.

```
/* disable interrupts, if used (not necessary after power-on)                 */
 EA       = DISABLE;    /* disable all interrupts                            */
 SJAIntEn = DISABLE;    /* disable external interrupt from SJA1000           */

/* set reset mode/request (Note: after power-on SJA1000 is in BasicCAN mode)
   leave loop after a time out and signal an error                           */
 while((ModeControlReg & RM_RR_Bit ) == ClrByte)
 {
   /* other bits than the reset mode/request bit are unchanged               */
   ModeControlReg = ModeControlReg | RM_RR_Bit ;
 }
/* set the Clock Divider Register according to the given hardware of Figure 3
     select PeliCAN mode
     bypass CAN input comparator as external transceiver is used
     select the clock for the controller S87C654                             */
 ClockDivideReg = CANMode_Bit  | CBP_Bit  | DivBy2;
```

```
/* disable CAN interrupts, if required (always necessary after power-on)
   (write to SJA1000 Interrupt Enable / Control Register)                    */
InterruptEnReg = ClrIntEnSJA;

/* define acceptance code and mask                                           */
 AcceptCode0Reg = ClrByte;
 AcceptCode1Reg = ClrByte;
 AcceptCode2Reg = ClrByte;
 AcceptCode3Reg = ClrByte;
 AcceptMask0Reg = DontCare;    /* every identifier is accepted               */
 AcceptMask1Reg = DontCare;    /* every identifier is accepted               */
 AcceptMask2Reg = DontCare;    /* every identifier is accepted               */
 AcceptMask3Reg = DontCare;    /* every identifier is accepted               */

/* configure bus timing                                                      */
/* bit-rate = 1 Mbit/s @ 24 MHz, the bus is sampled once                     */
 BusTiming0Reg = SJW_MB_24 | Presc_MB_24;
 BusTiming1Reg = TSEG2_MB_24 | TSEG1_MB_24;

/* configure CAN outputs: float on TX1, Push/Pull on TX0,
                          normal output mode                               */
 OutControlReg = Tx1Float | Tx0PshPull | NormalMode;

/* leave the reset mode/request i.e. switch to operating mode,
   the interrupts of the S87C654 are enabled
   but not the CAN interrupts of the SJA1000, which can be done separately
   for the different tasks in a system                                      */

/* clear Reset Mode bit, select dual Acceptance Filter Mode,
   switch off Self Test Mode and Listen Only Mode,
   clear Sleep Mode (wake up)                                               */
 do                                 /* wait until RM_RR_Bit  is cleared     */
/* break loop after a time out and signal an error                          */
 {
   ModeControlReg  = ClrByte;
 } while((ModeControlReg & RM_RR_Bit ) != ClrByte);

 SJAIntEn = ENABLE;    /* enable external interrupt from SJA1000             */
 EA       = ENABLE;    /* enable all interrupts                              */

/*----- end of Initialization Example of the SJA1000 ------------------------*/
```

4.2.2 Transmission

A transmission of a message is done autonomously by the CAN controller SJA1000 according to the CAN protocol specification [8]. The host controller has to transfer the message to be transmitted into the Transmit Buffer of the SJA1000 and set the flag "Transmit Request" in the command register. The transmission process can be controlled either by an interrupt request from the SJA1000 or by polling status flags in the control segment of the SJA1000.

Interrupt Controlled Transmission

According to the main processing of the controller as given in Figure 13, the transmit interrupt of the CAN controller and the external interrupt used by the host controller for the communication with the SJA1000 are enabled prior to the start of a transmission, which is controlled by interrupt. The interrupt enable flags are located in the Control Register for the BasicCAN mode and in the Interrupt Enable Register for the PeliCAN mode (see Table 2 and [1]).

As long as the SJA1000 is transmitting a message, the Transmit Buffer is locked for writing. Thus the host controller has to check the "Transmit Buffer Status" flag (TBS) of the Status Register (see [1]), if a new message can be placed into the Transmit Buffer.

- The Transmit Buffer is locked:

  The host controller stores the new message temporarily in its own memory and sets a flag, indicating that a message is waiting for being transmitted. It is up to the software designer how to handle this temporary storage, which may be designed to store several messages to be transmitted. The start of a transmission of the message will then be handled during the interrupt service routine, which is initiated at the end of the current running transmission.

  Upon reception of an interrupt from the CAN controller (see the interrupt processing of Figure 13), the host controller checks the type of interrupt. If it was a Transmit Interrupt, it checks, whether further messages have to be transmitted or not. A waiting message is copied from the temporary store into the Transmit Buffer and the flag indicating further messages to be transmitted is cleared. The flag "Transmission Request" (TR) of the Command Register (see [1]) is set, which will cause the SJA1000 to start the transmission.

- The Transmit Buffer is released:

  The host controller writes the new message into the Transmit Buffer and sets the flag "Transmission Request" (TR) of the Command Register (see [1]), which will cause the SJA1000 to start the transmission. At the end of a successful transmission, a Transmit Interrupt is generated by the CAN controller.

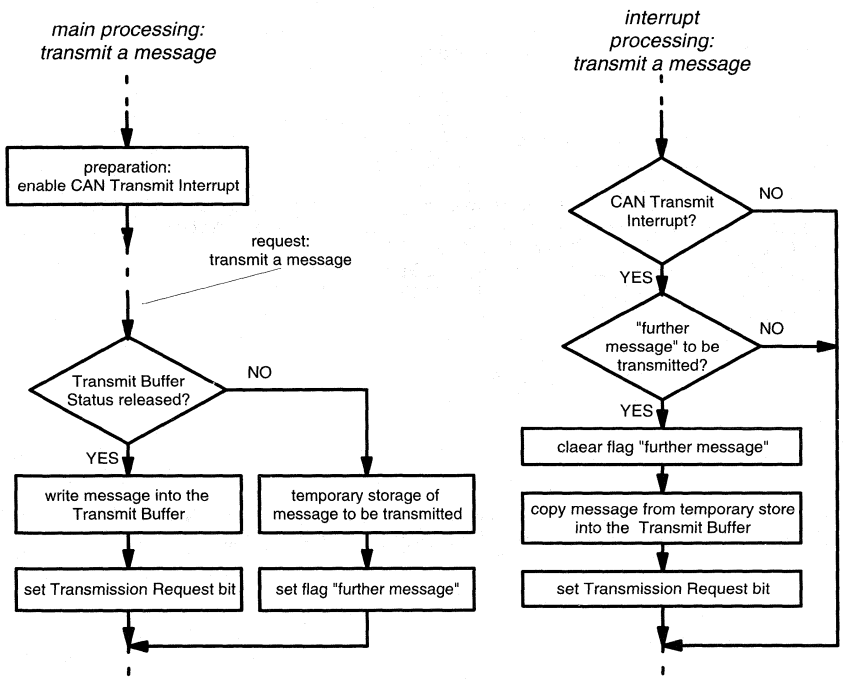


Figure 13: Flow Diagram "Transmission of a message" (interrupt controlled)

Polling Controlled Transmission

The flow is shown in Figure 14. The transmission interrupt of the CAN controller is disabled for this type of transmission control.

As long as the SJA1000 is transmitting a message, the Transmit Buffer is locked for writing. Thus the host controller has to check the "Transmit Buffer Status" flag (TBS) of the Status Register (see [1]), if a new message can be placed into the Transmit Buffer.

- The Transmit Buffer is locked:

  Polling the Status Register periodically, the host controller waits, until the Transmit Buffer is released.

- The Transmit Buffer is released:

  The host controller writes the new message into the Transmit Buffer and sets the flag "Transmission Request" (TR) of the Command Register (see [1]), which will cause the SJA1000 to start the transmission.

Example for the PeliCAN mode:

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., "InterruptEnReg" points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language "C" is used for programming.

After having initialized the CAN controller according to the example given in chapter 4.2.1, normal communication can be started.

```
.
.
/* wait until the Transmit Buffer is released                              */
 do
 {
   /* start a polling timer and run some tasks while waiting
      break the loop and signal an error if time too long                  */
 } while((StatusReg & TBS_Bit  ) != TBS_Bit  );

/* Transmit Buffer is released, a message may be written into the buffer   */
/* in this example a Standard Frame message shall be transmitted           */
 TxFrameInfo = 0x08;       /* SFF (data), DLC=8                            */
 TxBuffer1   = 0xA5;       /* ID1   = A5, (1010 0101)                      */
 TxBuffer2   = 0x20;       /* ID2   = 20, (0010 0000)                      */
 TxBuffer3   = 0x51;       /* data1 = 51                                   */
 .
 .
 TxBuffer10  = 0x58;       /* data8 = 58                                   */

/* Start the transmission                                                  */
 CommandReg = TR_Bit  ;       /* Set Transmission Request bit               */
.
.
```

The TS and RS flags in the Status Register can be used for detecting, that the CAN controller has reached the idle-state. The TBS- and TCS-flags can be checked for a successful transmission.

Example for the BasicCAN mode:

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., "InterruptEnReg" points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language "C" is used for programming.

After having initialized the CAN controller according to the example given in chapter 4.2.1, normal communication can be started.

```
.
/* wait until the Transmit Buffer is released                              */
 do
 {
   /* start a polling timer and run some tasks while waiting
      break the loop and signal an error if time too long                  */
 } while((StatusReg & TBS_Bit  ) != TBS_Bit  );

/* Transmit Buffer is released, a message may be written into the buffer   */
/* only Standard Frame messages are possible in BasicCAN mode              */
  TxBuffer1    = 0xA5;       /* ID1 = A5, (1010 0101)                     */
  TxBuffer2    = 0x28;       /* ID2 = 28, (0010 1000)  (DLC=8)            */
  TxBuffer3    = 0x51;       /* data1 = 51                                */
.
  TxBuffer10   = 0x58;       /* data8 = 58                                */

/* Start the transmission                                                  */
 CommandReg = TR_Bit  ;       /* Set Transmission Request bit               */
.
```

The TBS- and TCS-flags can be checked for a successful transmission.

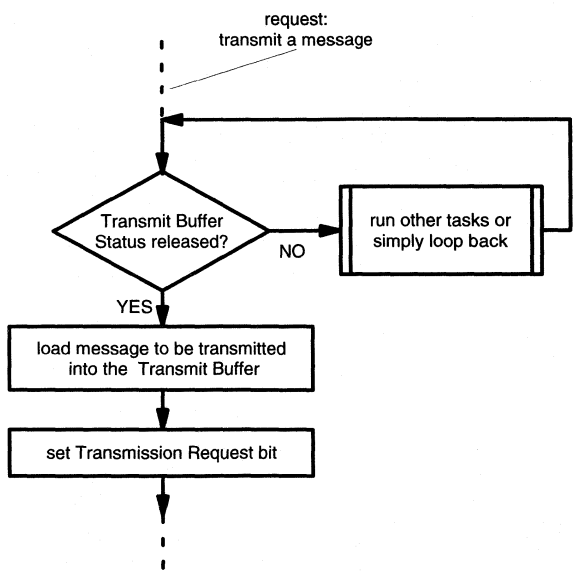


Figure 14: Flow Diagram "Transmission of a message" (polling controlled)

4.2.3 Abort Transmission

The transmission of a message, which was requested, may be aborted using the "Abort Transmission" command by setting the corresponding bit in the Command Register [1]. This feature may be used e.g. for transmitting an urgent message prior to the message, which has been written into the transmit buffer previously, but which was not transmitted successfully until now.

Figure 15 shows a flow using the transmit interrupt. The flow illustrates the situation, where a message has to be aborted in order to transmit a message with a higher priority. Other reasons for aborting a message may require a different interrupt flow.

A corresponding flow can be derived for the polling controlled transmission handling.

In case a message is still waiting for being served due to different reasons, the Transmit Buffer is locked (see the main flow part in Figure 15). If a transmission of an urgent message is requested, the Abort Transmission bit is set in the Command Register. When the message waiting to be served has either been transmitted successfully or aborted, the Transmit Buffer is released and a Transmit Interrupt is generated. During the interrupt flow the Transmission Complete flag of the Status Register has to be checked, if the previous transmission has been successful or not. The status "incomplete" indicates, that the transmission was aborted. In this case the host controller can run through a special routine dealing with a strategy for aborted transmissions, e.g., repeat the transmission of the aborted message after having checked, if it is still valid.

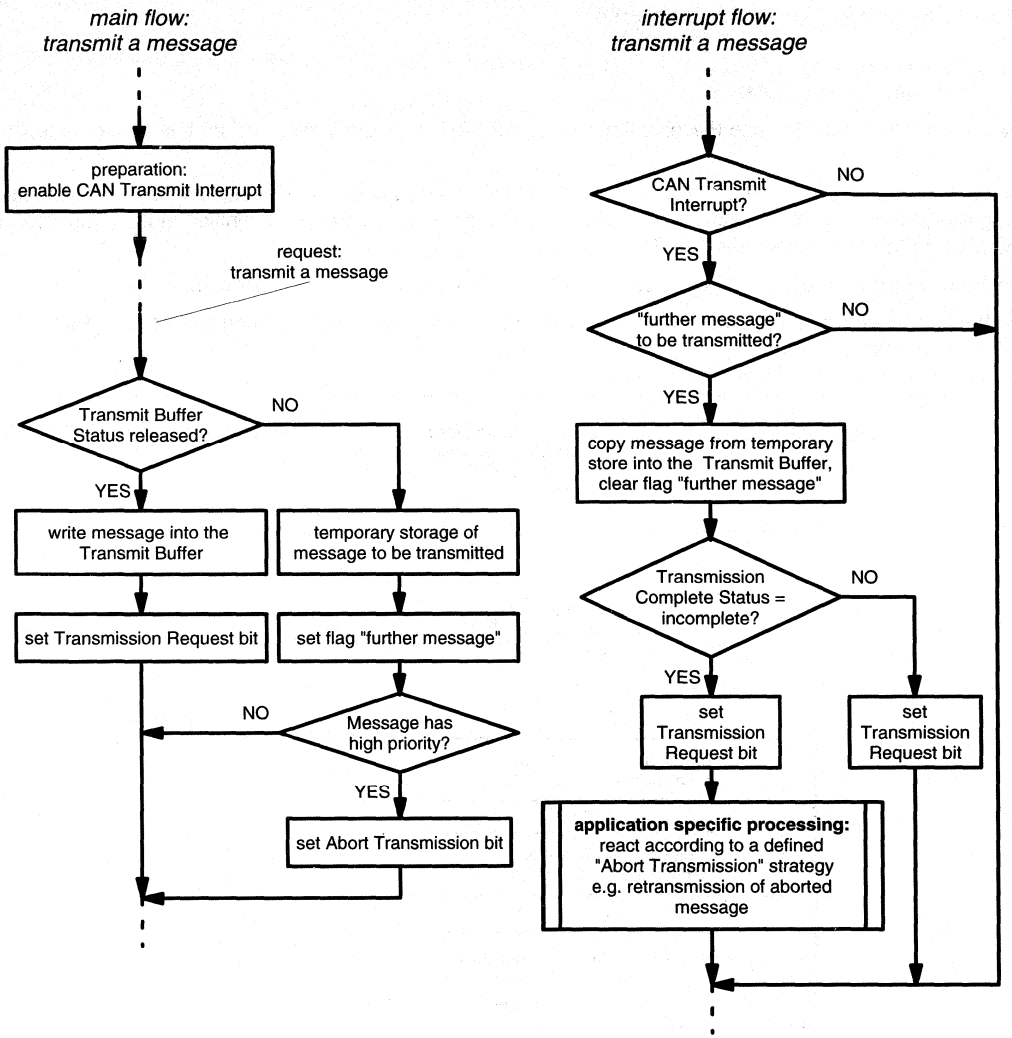


Figure 15: Flow Diagram "Abort Transmission of a message" (interrupt controlled)

4.2.4 Reception

The reception of messages is done autonomously by the CAN controller SJA1000 according to the CAN protocol specification [8]. Received messages are placed into the Receive Buffer (see chapter 4.1.1 and 5.1). A message, ready to be transferred to the host controller, is signalled by the Receive Buffer Status flag "RBS" (see [1]) of the Status Register and by a Receive Interrupt flag "RI" (see [1]), if enabled. The host controller has to

transfer the message to its local message memory, release the Receive Buffer and react on the content of the message. The transfer process can be controlled either by an interrupt request from the SJA1000 or by polling status flags in the control segment of the SJA1000.

Polling Controlled Reception

The flow is shown in Figure 16. The Receive Interrupt of the CAN controller is disabled for this type of reception control.

The host controller reads the Status Register of the SJA1000 on a regular basis, checking if the Receive Buffer Status flag (RBS) indicates, that at least one message has been received. For the definition of the flags located in the registers of the control segment see [1].

- The Receive Buffer Status flag indicates "empty", i.e., no message has been received:

  The host controller continues with the current task until a new request for checking the Receive Buffer Status is generated.

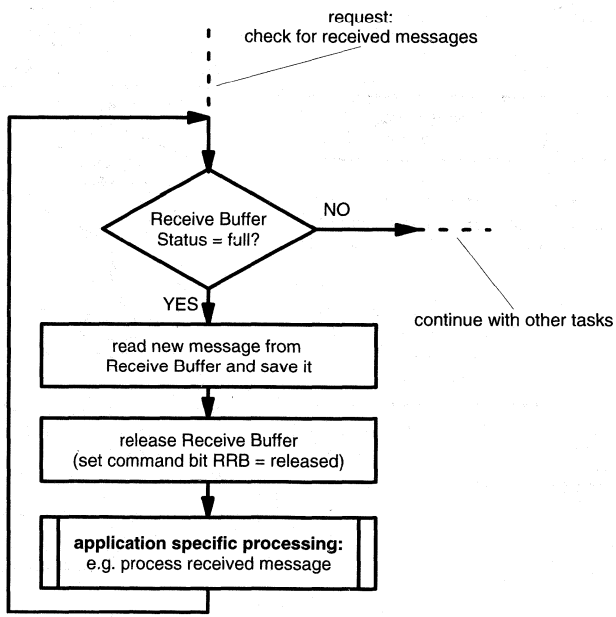


Figure 16: Flow Diagram "Reception of a message" (polling controlled)

- The Receive Buffer Status flag indicates "full", i.e., one or more messages have been received:

  The host controller gets the first message from the SJA1000 and sends a Release Receive Buffer command afterwards by setting the corresponding flag in the Command Register. The host controller can process each received message before checking for further messages, as indicated in Figure 16. But it is also possible to check at once for further messages by polling the Receive Buffer Status bit again and process the received messages all together later. In this case the local message memory of the host controller has to be large enough to store more than one message before they are processed. After having transferred and processed one or all messages, the host controller can continue with other tasks.

Interrupt Controlled Reception

According to the main processing of the controller as given in Figure 17, the receive interrupt of the CAN controller and the external interrupt used by the host controller for the communication with the SJA1000 are enabled prior to an interrupt controlled reception of messages. The interrupt enable flags are located in the Control Register (for the BasicCAN mode) or in the Interrupt Enable Register (for the PeliCAN mode) - see Table 2 and [1].

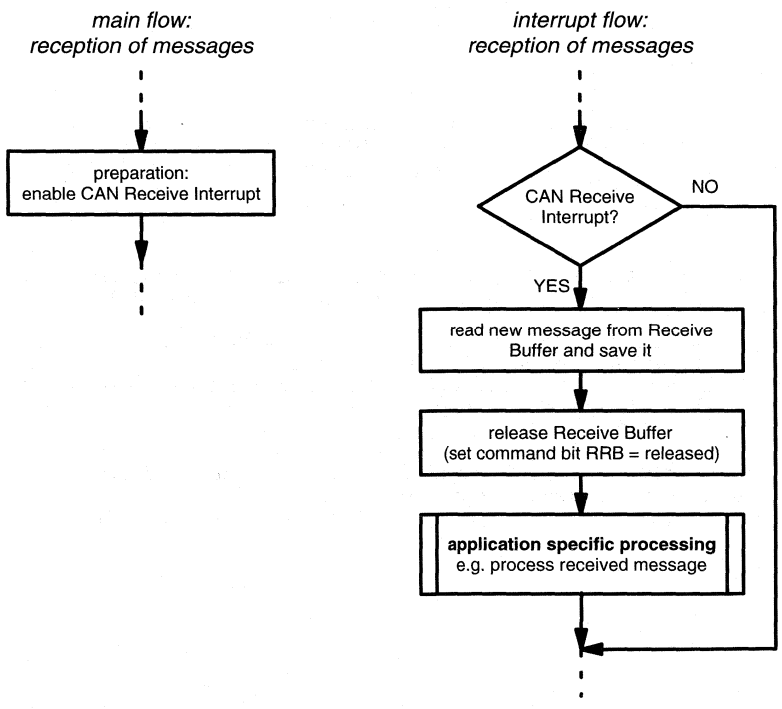


Figure 17: Flow Diagram "Reception of a message" (interrupt controlled)

If the SJA1000 has received a message, which has passed the acceptance filter and has been placed into the Receive FIFO, a receive interrupt is generated. Thus the host controller can react immediately, transferring the received message into its message memory and send a Release Receive Buffer command afterwards by setting the corresponding flag "RRB" (see [1]) in the Command Register. Further messages in the Receive FIFO will

generate a new receive interrupt, so it is not necessary to read all messages available in the Receive FIFO during one interrupt. Contrary to this solution the procedure for reading all available messages at once is used in Figure 18. After having released the Receive Buffer, the Receive Buffer Status (RBS) in the Status Register is checked for further messages and all available are read in a loop.

As given in Figure 17, the whole reception process may be done during the interrupt routine, without interaction with the main program. If feasible, even the reaction on messages can be done in the interrupt too.

Example:

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., "InterruptEnReg" points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language "C" is used for programming.

After having initialized the CAN controller according to the example given in chapter 4.2.1, normal communication can be started.

1. part of the main processing

```
.
.
/* enable the receive interrupt                                          */
 InterruptEnReg = RIE_Bit;
.
.
```

2. part of the interrupt 0 service routine

```
.
/* read the Interrupt Register content from SJA1000 and save temporarily
   all interrupt flags are cleared (in PeliCAN mode the Receive
   Interrupt (RI) is cleared first, when giving the Release Buffer command)
*/
CANInterrupt = InterruptReg;
.
.
/* check for the Receive Interrupt and read one or all received messages  */
 if (RI_VarBit == YES)                  /* Receive Interrupt detected     */
 {
    /* get the content of the Receive Buffer from SJA1000 and store the
       message into internal memory of the controller,
       it is possible at once to decode the FrameInfo and Data Length Code
       and adapt the fetch appropriately                                  */
    .
    .
    /* release the Receive Buffer, now the Receive Interrupt flag is cleared,
       further messages will generate a new interrupt                     */
    CommandReg = RRB_Bit;               /* Release Receive Buffer         */
 }
.
.
```

Data Overrun Handling

In case the Receive FIFO is full but another message is being received, a Data Overrun is signalled to the host controller by setting the Data Overrun Status in the Status Register and, if enabled, a Data Overrun Interrupt is generated by the SJA1000.

Running into a Data Overrun situation states, that the host controller is extremely overloaded, as it did not have enough time to fetch received messages from the Receive Buffer in time. A Data Overrun signals, that data are lost, possibly causing inconsistencies in the system. Normally a system should be designed in such a way, that the received messages are transferred and processed fast enough to avoid a Data Overrun condition. An exception handler dealing with an application specific processing should be implemented in the host controller, if Data Overrun situations cannot be avoided.

Figure 18 illustrates the program flow, in case a Data Overrun Interrupt has to be handled.

After having transferred the message, which caused the receive interrupt, and released the Receive Buffer, it is checked, if further messages are available in the Receive FIFO by reading the Receive Buffer Status. Thus all messages can be fetched from the Receive FIFO before going on further. Of course reading a message and perhaps processing it already during the interrupt, should be done faster, than it takes the SJA1000 to receive a new message. Otherwise it could happen, that the host controller stays in the interrupt forever reading messages.

Detecting a Data Overrun starts an exception handling according to a "Data Overrun" strategy. This strategy can decide between two situations:

- A Data Overrun occurred together with a Receive Interrupt:
  Messages may have been lost.
- A Data Overrun occurred, but no Receive Interrupt was detected:
  Messages may have been lost. The Receive Interrupt may have been disabled.

It is up to the system designer how the host controller should react on these situations.

An equivalent handling can also be done during a polling controlled reception of messages.

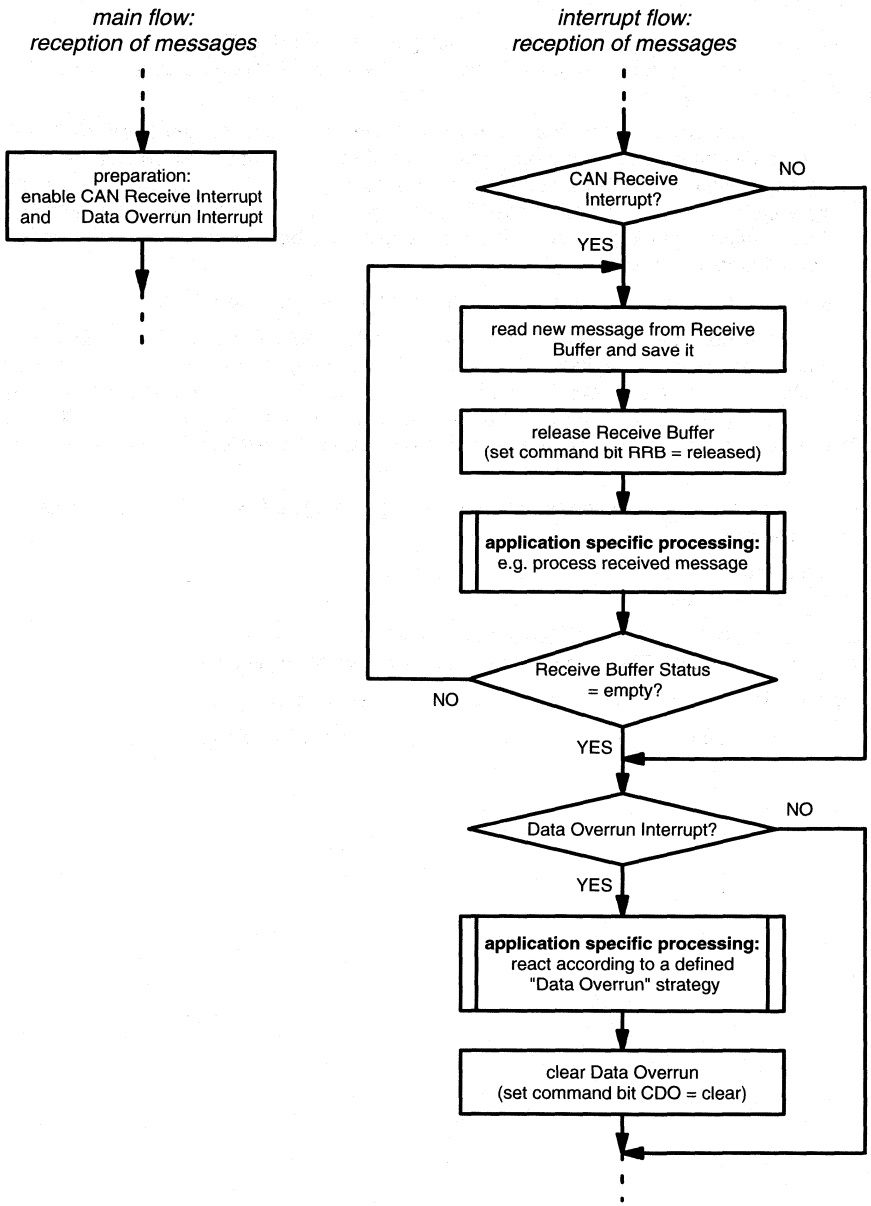


Figure 18: Flow Diagram "Data Overrun and reception of messages" (interrupt controlled)

4.2.5 Interrupts

In PeliCAN mode the SJA1000 has 8 different interrupts (in BasicCAN mode there are only 5), which may be used for causing immediate actions by the host controller on certain states of the CAN controller.

In case a CAN interrupt is present, the SJA1000 sets the interrupt output (pin 16) to LOW-level. The output stays at LOW-level, until the host controller reacts on the interrupt by reading the Interrupt Register of the SJA1000; – in case of a receive interrupt in PeliCAN mode upon releasing the Receive Buffer. After this reaction from the host controller the SJA1000 switches the interrupt output back to HIGH-level. In case further interrupts did arrive in the meantime, or further messages are available in the Receive FIFO, the SJA1000 at once sets the interrupt output to LOW-level again. Thus the output may stay HIGH for a very short time only. Both the handshaking during serving the interrupt request and the possible short HIGH-level pulse during two interrupts require, that the interrupt of the host controller must be level-activated.

The flow in Figure 19 gives an overview of all possible interrupts and references to more detailed descriptions in this Application Note. The order, in which the different interrupts are handled in this flow, is one possible solution only. It depends very much on the system and the requested behaviour of it, in which order the interrupts have to be served. This has to be decided by the designer of the overall system.

The reactions on the Transmission, Receive and Data Overrun Interrupts are already discussed in the previous paragraphs.

The flows after a Wake Up Interrupt, Arbitration Lost Interrupt and three different error interrupts are given in more detail in Figure 20, Figure 21 and Figure 22. All error interrupts may be used for implementing a versatile error strategy in the system. This strategy should deal with system optimization in the development phase and automatic system optimization and system maintenance in the operational phase. Also the Arbitration Lost Interrupt may be used for system optimization and maintenance. See also the following chapters and the data sheet [1] for more details on the different error signals, arbitration lost handling and related information.

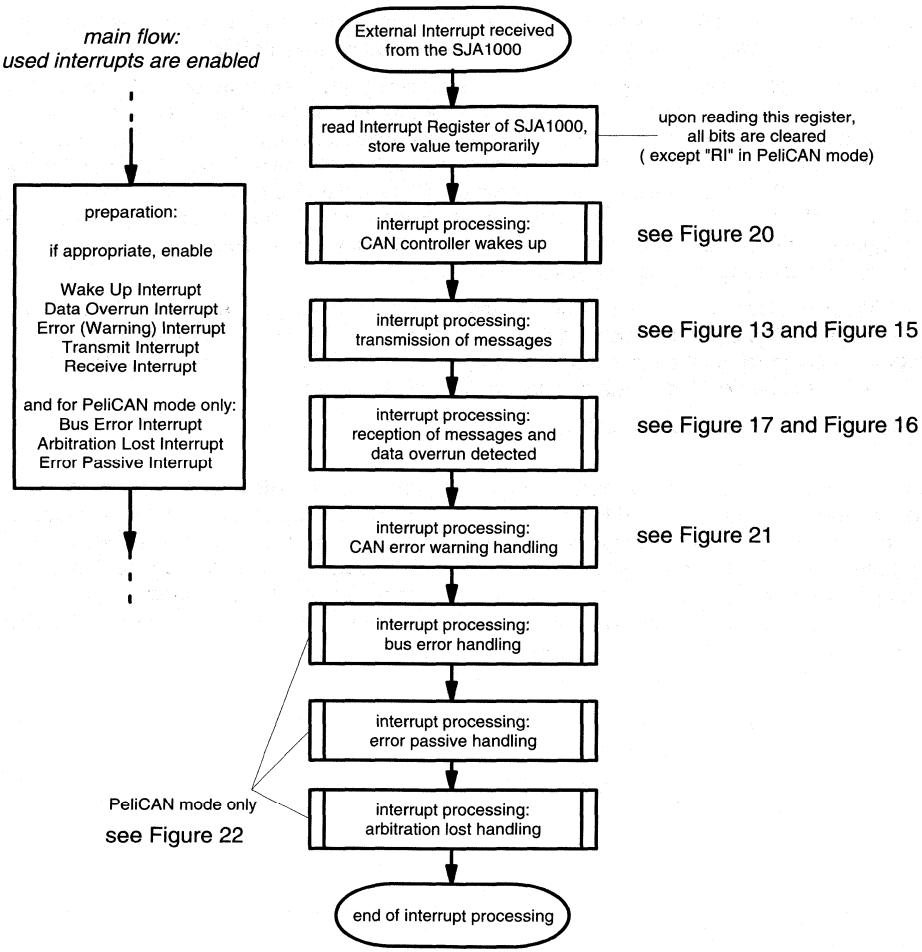


Figure 19: General interrupt flow

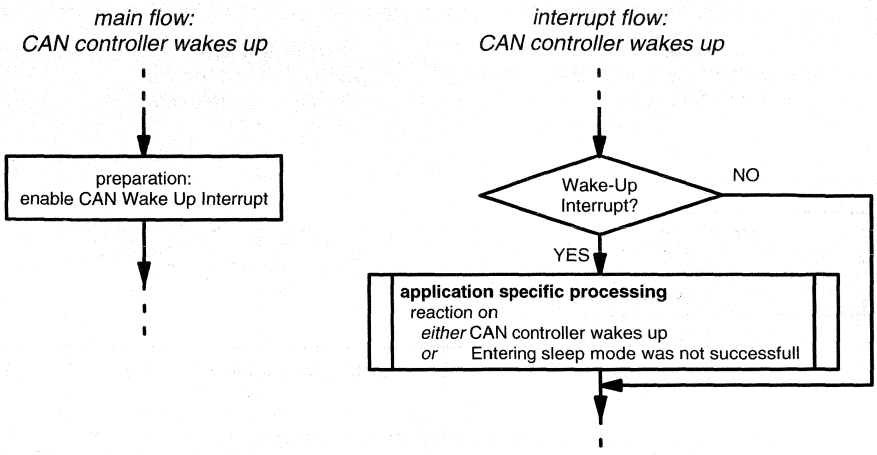


Figure 20: Flow Diagram "CAN controller wakes up"

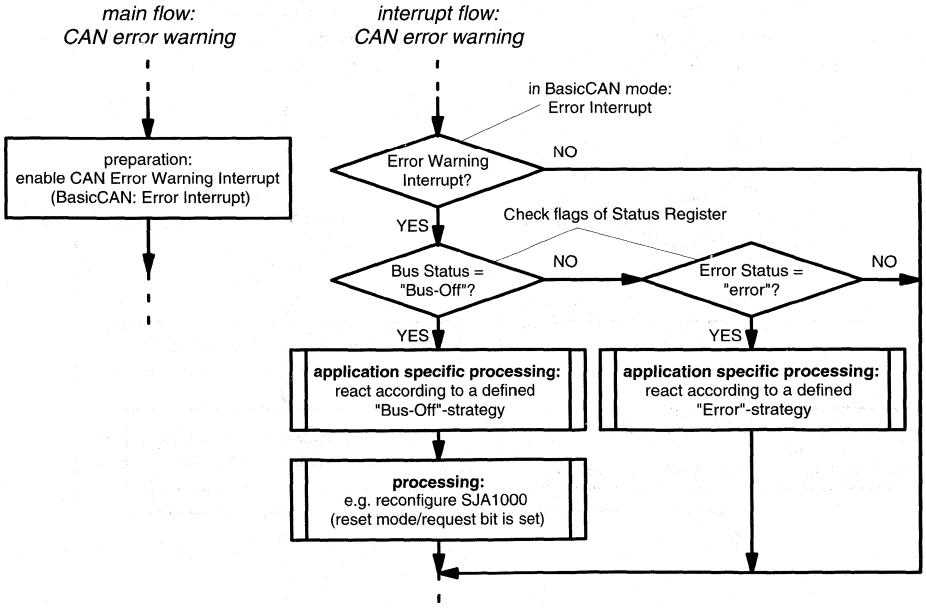


Figure 21: Flow Diagram "Error Warning Interrupt"

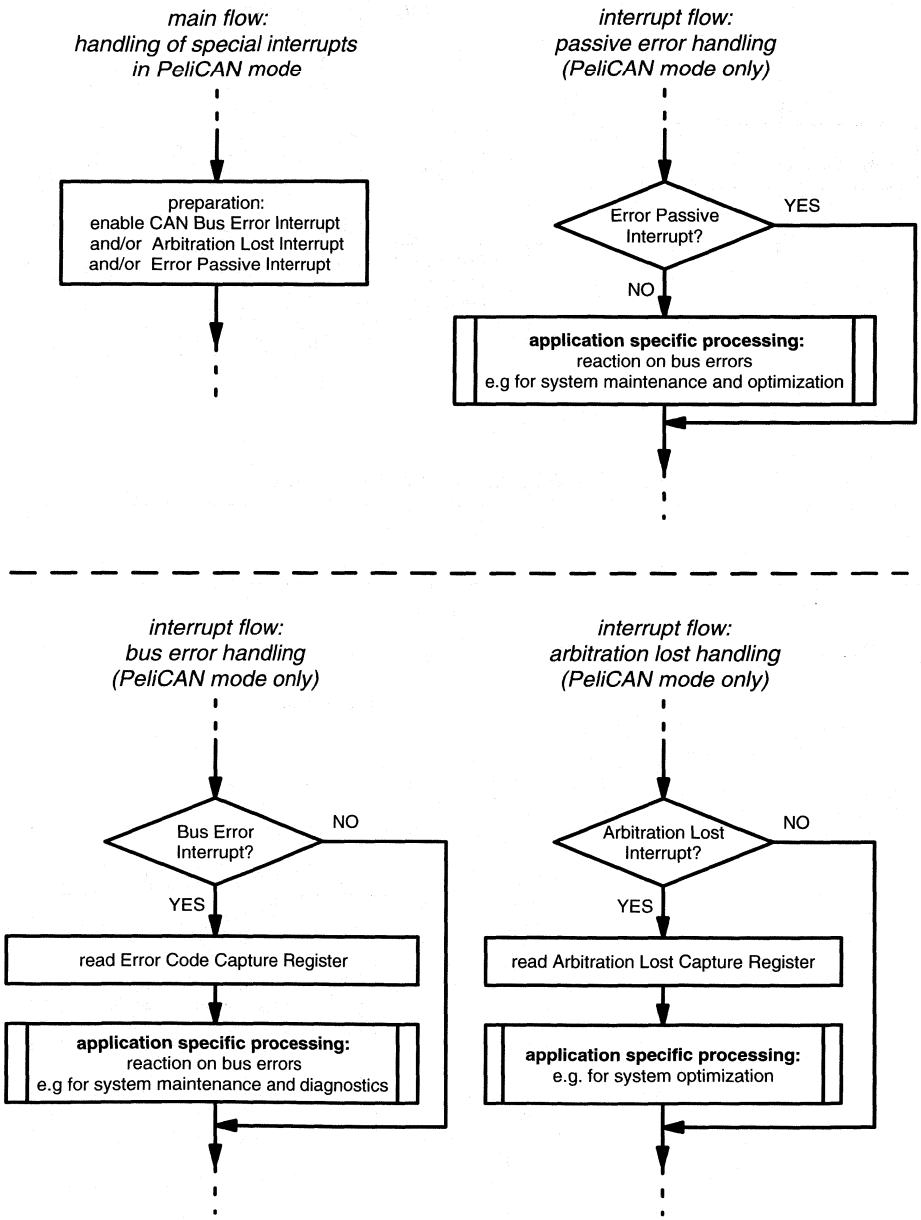


Figure 22: Flow Diagram "Processing of special PeliCAN interrupts"

5. PELICAN MODE FUNCTIONS

5.1 Receive FIFO / Message Counter / Direct RAM Access

The SJA1000 registers and message buffers appear to the host controller as peripheral registers which can be addressed via the multiplexed address/data bus. Depending on the selected mode (Operating or Reset) different registers are accessible. The address range for normal operation is: Address 0 .. 31. It contains registers for initialization, status and control purposes. Furthermore the CAN message buffers are allocated between address 16 and 28. With a host controller write access the user can address the CAN controller's Transmit Buffer and with a read access the Receive Buffer contents is read.

Additionally to the range described above the whole Receive FIFO is mapped between CAN address 32 and 95, see also Figure 23. Furthermore the Transmit Buffer of the SJA1000 which is also part of the internal 80 byte RAM is available between CAN address 96 and 108.

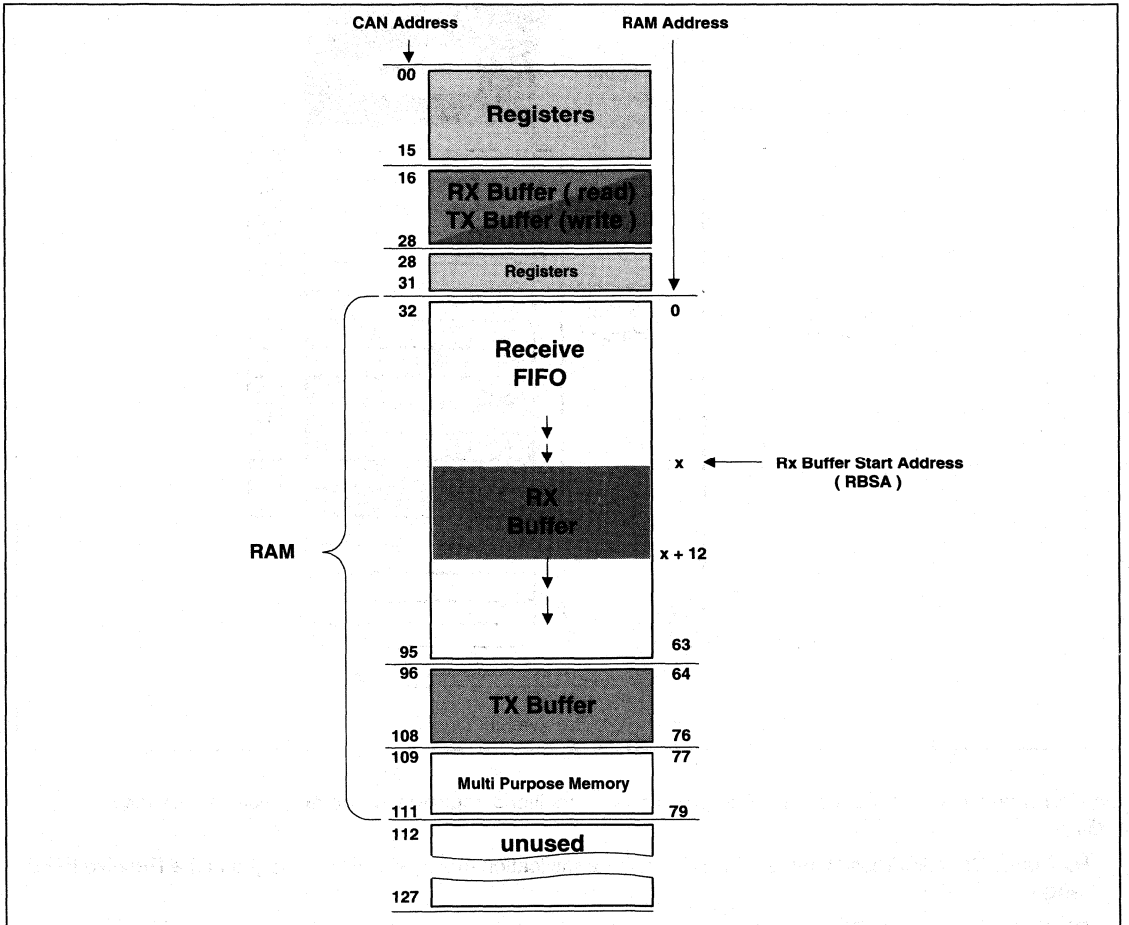


Figure 23: Register and RAM Address Allocations

With the described direct RAM access it is possible to read the Transmit Buffer and the complete Receive FIFO.

In PeliCAN mode the Receive FIFO is able to store up to $n = 21$ messages. With the help of the following equation it is possible to calculate the maximum number of messages:

$$n = \frac{64}{3 + data\_length\_code}$$

The Receive Buffer is defined as a 13 byte window always containing the current receive message of the Receive FIFO. As shown in Figure 24 it could happen that parts or the complete following message is already available in the Receive Buffer window.

However, upon command 'Release Receive Buffer' the next receive message in the Receive FIFO will become completely visible in the Receive Buffer window starting at CAN address 16.

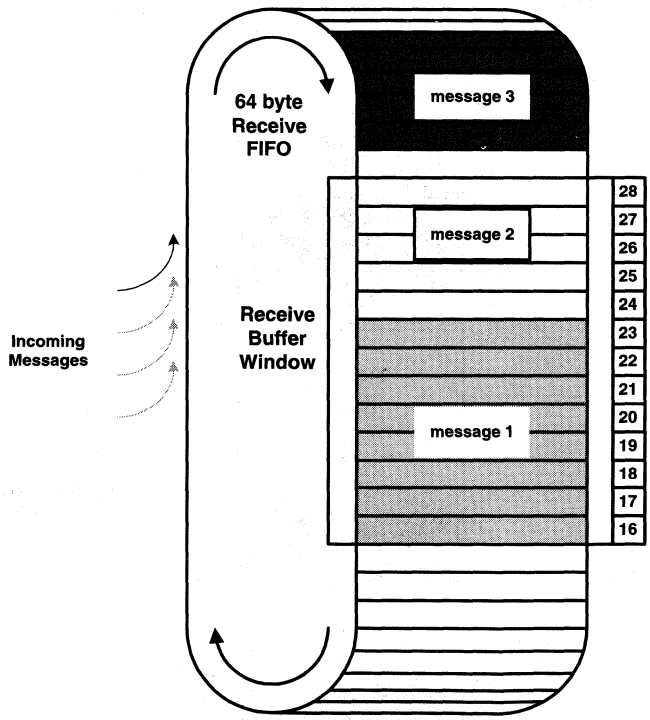


Figure 24: Receive FIFO

Mainly for analysis purposes the SJA1000 provides two additional registers supporting receive message handling:

- Rx Buffer Start Address Register (RBSA) allows identification of single CAN messages in the Receive FIFO range.
- RX Message Counter Register which contains the current number of stored messages in the Receive FIFO.

Figure 23 shows the relation between the physical RAM address and the CAN address.

5.2 Error Analysis Functions

Depending on the value of the error counters each CAN controller can operate in one of three possible error states: error active, error passive or bus-off. The CAN controller is error active if both error counters are between 0... 127 . In case of an error condition an active error flag (6 dominant bits) is generated. The SJA1000 is error passive if one of the error counters is between 128 and 255. A passive error flag (6 recessive bits) is generated upon detection of an error condition in this case. If the Transmit Error Counter is greater than 255 the bus-off status is reached. In this state the reset request bit is set automatically and the SJA1000 can not influence the bus. As shown in Figure 25 bus-off can only be terminated with the host controller command ´Reset Request = 0´. This will start the bus-off recovery time where the Transmit Error Counter is used to count 128 occurrences of a bus free signal. At the end of this time both error counters are 0 and the device is error active again.

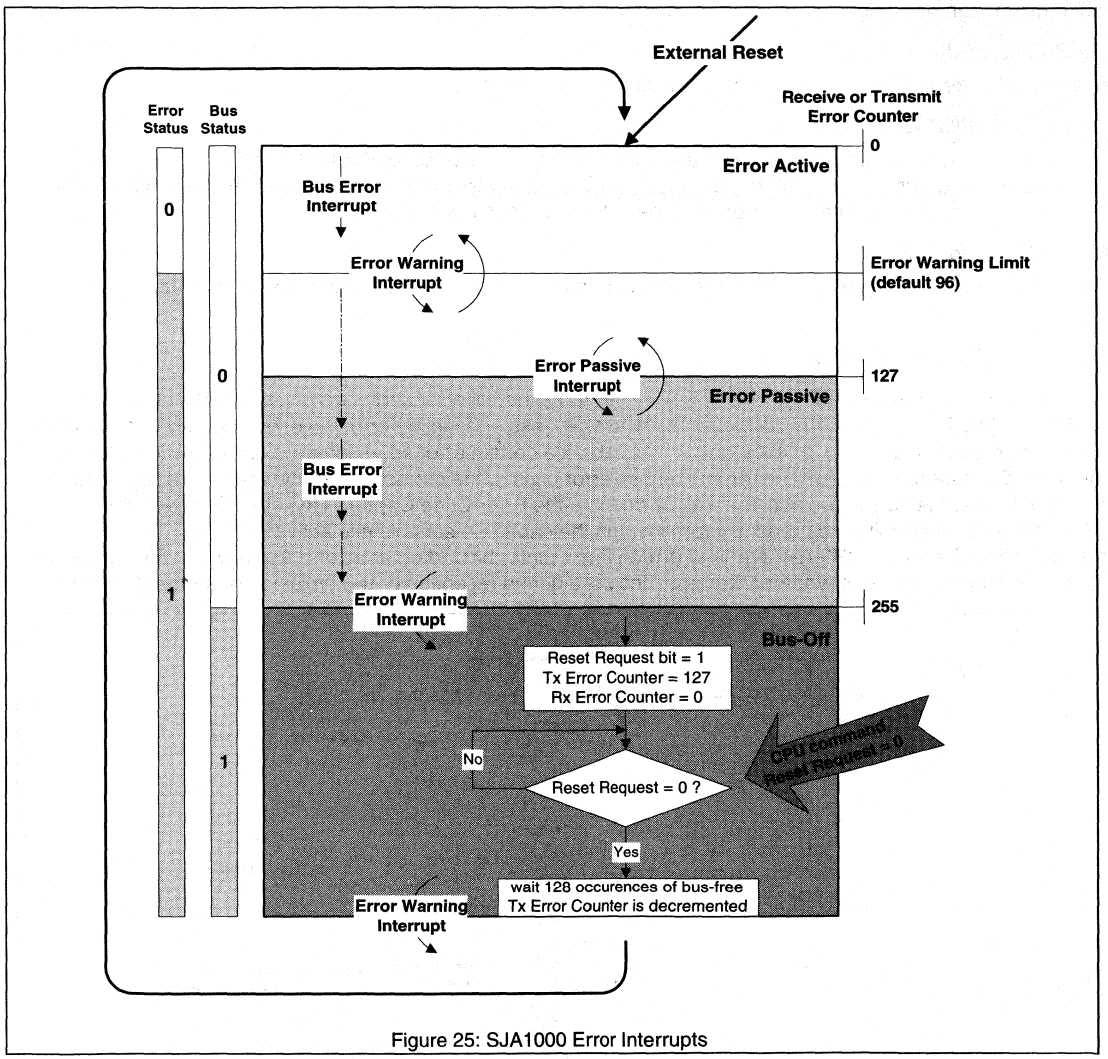


Figure 25: SJA1000 Error Interrupts

Furthermore the figure shows the value for both Error and Bus status at different error states.

5.2.1 Error Counters

As described above the error states of the CAN controller are directly related to the values of the Transmit and Receive Error Counters.
To allow a deep look inside into the error confinement and to support an enhanced error analysis with the SJA1000 the CAN controller provides readable error counters. Additionally, in Reset Mode a write access to both error counters is allowed.

5.2.2 Error Interrupts

Three interrupt sources have been implemented to signal error conditions to the host controller, see Figure 25. Each interrupt can be enabled separately in the Interrupt Enable Register.

Bus Error Interrupt:

This interrupt is generated upon any error condition on the CAN bus.

Error Warning Interrupt:

The Error Warning Interrupt is generated if the error warning limit is passed. Furthermore it is generated if the CAN controller enters the bus-off state and upon re-entry into error active state. The error warning limit of the SJA1000 is programmable in reset mode. The default value upon reset Is 96.

Error Passive Interrupt:

If the error status changes from error active to error passive or vice versa an error passive interrupt is signalled.

5.2.3 Error Code Capture

As described in the previous section the SJA1000 performs the full error confinement specified in the CAN2.0B specification [8]. As in every CAN controller the whole process of handling errors is executed fully automatically. However, to provide the user with additional details about a certain error condition the SJA1000 contains the Error Code Capture function. Whenever a CAN bus error occurs, the corresponding bus error interrupt is forced. At the same time, the current bit position is captured into the Error Code Capture Register. The captured data is fixed until the host controller has read it. From now on the capture mechanism is activated again. The register contents distinguishes four different types of errors: form,-stuff,- bit and other errors. As shown in Figure 26 the register additionally indicates whether the error occurred during reception or transmission of a message. Five

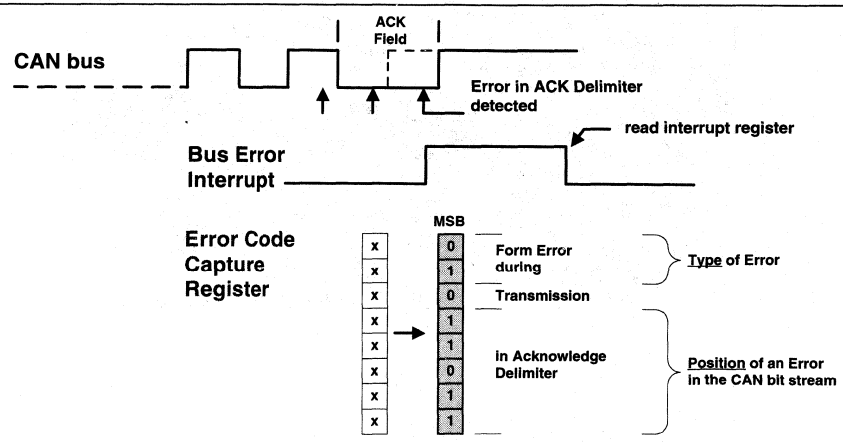


Figure 26: Example for the Error Code Capture Function

bits in this register indicate the erroneous bit position in the CAN frame, see also the following tables and the data sheet for more details.
As defined in the CAN specification, every single bit on the CAN bus can only have special types of errors. The next two tables show all possible errors during transmission and reception of CAN messages. The left part contains the position and the type of an error, captured by the Error Code Capture Register. The right part of each table is a translation into an upper level error description and can be derived directly from the register contents. With the help of these tables further information concerning error counter change and the erroneous state at the transmit and receive pins of the device can be derived. While using this table, e.g., in the error analysis software it is possible to analyze every single error situation in detail. The information about type and position of CAN errors can be used for error statistics and system maintenance or for corrective actions during system optimization.

Table 6: Possible errors during reception

Error Code Capture		RX Error Count	Description	
Position of an Error in the CAN bit stream	**Type of Error**			
Identifier SRR, IDE and RTR bit Reserved Bits Data Length Code Data Field CRC Sequence	Stuff	+ 1	more than 5 consecutive bits with same level received	- -
CRC Delimiter	Form Stuff	+ 1 + 1	Rx = dominant more than 5 consecutive bits with same level received	bit has to be recessive
Acknowledge Slot	Bit	+ 1	Tx = dominant but Rx = recessive	can't write dominant bit
Acknowledge Delimiter[1]	Form	+ 1	Rx = dominant or CRC error detected[1]	critical bus timing or bus length CRC sequence not correct
End of Frame	Form Other	+ 1 ± 0	Rx = dominant in first 6 bits Rx = dominant in last bit	- - reaction: overload flag will be sent, data duplication is possible if transmitter starts re-transmission
Intermission	Other	± 0	Rx = dominant	reaction: overload flag will be sent by receiver
Active Error Flag	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
Tolerate Dominant Bits	Other	+ 8	Rx = dominant in first bit upon error flag Rx = dominant for more than 7 bits upon error or overload flag	
Error Delimiter	Form Other	+ 1 ± 0	Rx = dominant within first 7 bits Rx = dominant in last bit of delimiter	- - overload flag will be sent
Overload Flag	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit

[1] if the CRC is not o.k., then the error is processed in the Acknowledge Delimiter resulting in a 'Form Error'.

Table 7: Possible errors during transmission

Error Code Capture		TX Error Count	Description	
Position of an Error in the CAN bit stream	**Type of Error**			
Start Of Frame	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
Identifier	Bit Stuff	+ 8 ± 0	Tx = dominant but Rx = recessive Tx = recessive but Rx = dominant	can't write dominant bit - -
SRR Bit	Bit Stuff	+ 8 ± 0	Tx = dominant but Rx = recessive Tx = recessive but Rx = dominant	can't write dominant bit - -
IDE and RTR Bit	Bit Stuff	+ 8 + 8	Tx = dominant but Rx = recessive Tx = recessive but Rx = dominant	can't write dominant bit - -
Reserved Bits, Data Length Code, Data Field, CRC Sequence,	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
CRC Delimiter	Form	+ 8	Rx = dominant	bit has to be recessive
Acknowledge Slot Acknowledge Delimiter	Other Other Form	+ 8 ± 0 + 8	Rx = recessive (error active) Rx = recessive (error passive) Rx = dominant	no acknowledge no acknowledge, node is probably alone on the bus critical bus timing or bus length
End of Frame	Form Other	+ 8 + 8	Rx = dominant within first 6 bits Rx = dominant in last bit	- - frame has already been received by some nodes, re-transmission may result in data duplication in receivers
Intermission	Other	± 0	Rx = dominant	overload flag from 'old' CAN controllers
Active Error Flag Overload Flag	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
Tolerate Dominant Bits	Form	+ 8	Rx = dominant for more than 7 bit times after active error flag or overload flag	- -
Error Delimiter	Form Other	+ 8 ± 0	Rx = dominant within first 7 bits Rx = dominant in last bit of delimiter	- - overload flag from 'old' CAN controller
Passive Error Flag	Other	+ 8	Rx = dominant (error passive)	no acknowledge received, node is not alone on the bus

5.3 Arbitration Lost Capture

The SJA1000 is able to identify the exact CAN bit stream position where the arbitration has been lost. Immediately upon this an ´Arbitration Lost Interrupt´ is generated. Furthermore the bit number is captured in the Arbitration Lost Capture Register. As soon as the host controller has read the contents of this register, the capture function is activated for the next arbitration lost situation.

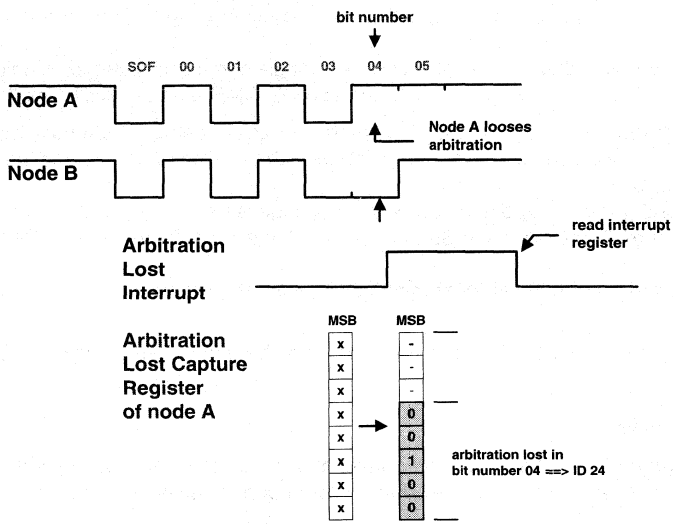


Figure 27: Arbitration Lost Capture Function

With the help of this feature the SJA1000 is able to monitor each CAN bus access. For diagnostics or during system configuration it is possible to identify every situation where the arbitration was not successful.

The next example shows how the arbitration lost function can be used.

First the Arbitration Lost Interrupt is enabled in the Interrupt Enable Register. Upon interrupt the contents of the Interrupt Register is saved. If the arbitration lost interrupt flag is set, the contents of the Arbitration Lost Capture Register is analyzed.

Example: **Arbitration Lost**

```
....
InterruptEnReg = ALIE_Bit;          /* Enable Arbitration Lost Interrupt       */
....

/* ----- Interrupt Service Routine ------------------------------------------ */
....
int_reg_copy = InterruptReg;        /* save interrupt register contents        */
....
if (int_reg_copy & ALIE_Bit)
   candat = ArbLostCapReg;          /* read arbitration lost capture register  */
...
...
```

5.4 Single Shot Transmission

In some applications the automatic re-transmission of CAN messages does not make sense: it could happen that a CAN node looses arbitration several times and the data have become obsolete.

In order to request a ´Single Shot Transmission´ previous CAN controllers have to perform the following steps:

1. Transmission Request
2. Wait for transmission status
3. Abort Transmission

The software necessary to process this can be minimized to a single command with the ´Single Shot Transmission´ option, which is initiated by setting the command bits CMR.0 and CMR.1 simultaneously.

In this case no status bit polling is needed and the host controller can concentrate on other tasks. The described `Single Shot Transmission´ function can be combined perfectly with the arbitration lost and the error code capture functions of the SJA1000.
In case of arbitration lost or if an error condition occurs the message is not re-transmitted by the SJA1000. As soon as the Transmit Status bit is set within the Status Register, the internal Transmission Request Bit is cleared automatically.
With the additional information from both capture registers it is under the control of the user whether a message is re-transmitted or not.

As described in chapter 5.7 the Single Shot Transmission can also be used together with the Self Test Mode.

5.5 Listen Only Mode

In Listen Only Mode the SJA1000 is not able to write dominant bits onto the CAN bus. Neither active error flags or overload flags are written nor a positive acknowledge is given upon successful reception.

Errors are treated like in error passive mode. The error analysis functions, e.g., error code capture and error interrupts are working as known from normal operating mode.

However, the status of the error counters is frozen.

Reception of messages is possible, transmission is not possible. Therefore, this mode can be used for automatic bit-rate detection, see also chapter 5.6, and other applications with monitor characteristics.

Note, before entering the Listen Only Mode the Reset Mode has to be entered.

Example: **Listen Only Mode**

```
....
ModeControlReg = RM_RR_Bit;       /* Enter Reset Mode       */
ClockDivideReg = CANMode_Bit;     /* PeliCAN Mode           */
....
ModeControlReg = LOM_Bit;         /* Enter Listen Only Mode */
                                  /* and leave Reset Mode   */
...
```

5.6 Automatic Bit-Rate Detection

The major drawback of existing trial and error concepts for automatic bit-rate detection is the generation of CAN error frames which is not acceptable. The SJA1000 supports the requirements for an automatic bit-rate detection with new features of the PeliCAN mode. This section briefly describes an application example without influencing running operations on the network.
In Listen Only Mode, the SJA1000 is neither able to transmit messages nor to generate error frames. Only message reception is feasible in this mode. A pre-defined table within the software contains all possible bit-rates including their bit-timing parameters. Before starting message reception with the highest possible bit-rate, the SJA1000 enables both receive and error interrupts.
In case of one or more errors on the CAN bus, the software switches to the next lower bit-rate.
Upon successful reception of a message, the SJA1000 has detected the right bit-rate and can switch to normal operating mode. From now on this node is able to operate as any other active CAN node in the system.

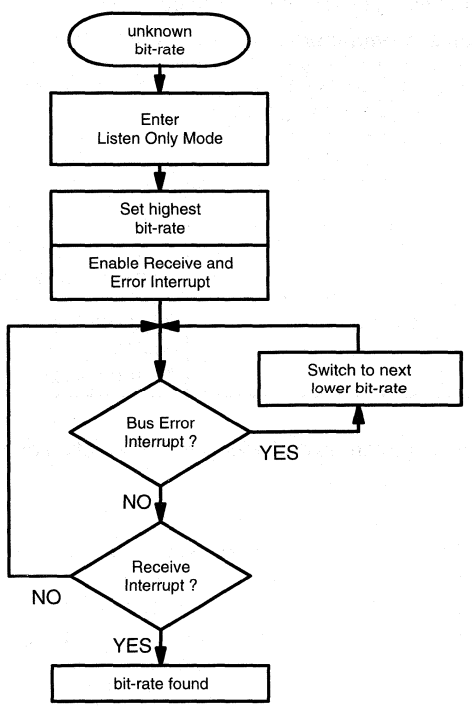


Figure 28: Algorithm of Bit-Rate Detection

5.7 CAN Self Tests

The SJA1000 supports two different options for self tests:

- Local Self Test
- Global Self Test

A *Local Self Test*, e.g., can be used perfectly for single node tests because an acknowledge from other nodes is not needed. In this case the SJA1000 has to be put into 'Self Test Mode' (Mode register) and the command 'Self Reception Request' is given.

For a *Global Self Test* the SJA1000 performs the same command in Operating Mode. However, for a Global Self Test in a running system a CAN acknowledge is needed.

Note that in both cases a physical layer interface must be available including CAN bus lines with a termination. A transmission or self reception is initiated by setting the appropriate bits in the Command Register.

The SJA1000 provides three command bits for the initiation of CAN transmissions and self receptions . Table 8 shows all possible combinations depending on the selected mode of operation.

Table 8: CAN Transmission Request Commands

Command	CMR =	Interrupt(s) upon successful operation	Self Test Mode	Operating Mode
Self Reception Request	0x10	RX and TX	local self test	global self test
Transmission Request	0x01	TX	normal transmission[1]	normal transmission
Single Shot	0x03	TX	transmission without re-transmission[1]	transmission without re-transmission
Single Shot and Self Reception Request	0x12	RX and TX	local self test without re-transmission	global self test without re-transmission

The following example presents basic programming elements for the initiation of a local self test.

Example: **Local Self Test**

```
....
ModeControlReg = RM_RR_Bit;        /* Enter Reset Mode        */
ClockDivideReg = CANMode_Bit;      /* PeliCAN Mode            */
ModeControlReg = STM_Bit;          /* Enter Self Test Mode    */
                                   /* and leave Reset Mode    */
TxFrameInfo = 0x03;                /* Fill Transmit Buffer    */
TxBuffer1   = 0x53;                /*                         */
...
TxBuffer5   = 0xAA;                /* Last Transmit Byte      */

CommandReg = SRR_Bit;              /* Self Reception Request */
.........................
if (RxBuffer1 != TxBufferRd1) comparison = false;
if (RxBuffer2 != TxBufferRd2) comparison = false;
```

[1] A normal transmission with or without re-transmission is usually performed in Operating Mode

5.8 Receive Sync Pulse Generation

The SJA1000 allows the generation of a pulse on the TX1 pin upon successful reception of a message. It is generated if the message is completely stored within the Receive FIFO. The pulse can be enabled in the Clock Divider Register and is active for the duration of the 6th bit in ´End Of Frame´.
Therefore, it can be used for versatile event triggered tasks, e.g., as a dedicated receive interrupt source or for a global clock synchronization in a distributed system which is briefly described in the following section.

In distributed systems it is difficult to implement a system wide clock without having an extra synchronization line [9]. All nodes connected onto the bus have local clocks with clock drifts. Lets assume that one CAN node in the network is assumed to operate as a ´master´ clock and the remaining clocks in the network are synchronized to the value of the master clock.

The Self Reception Request feature together with the fact that each SJA1000 is able to generate a pulse at a definite time upon message reception, can be used to support clock synchronization in distributed systems.

In Figure 29 a system master transmits a ´Self Reception Message´ onto the CAN bus. After message reception, each node, including the master, generates a Receive Sync Pulse. In every slave node it is used, i.e., to reset the timers. Simultaneously the master node uses this pulse to capture the master clock value $\mathbf{t_M}$.

In a next step the $\mathbf{t_M}$ value is sent as a ´Reference Time Message´ to all slaves by the master. A simple adder function in every slave, followed by reloading all timers with $\mathbf{t_S}$ synchronizes the system wide clock.

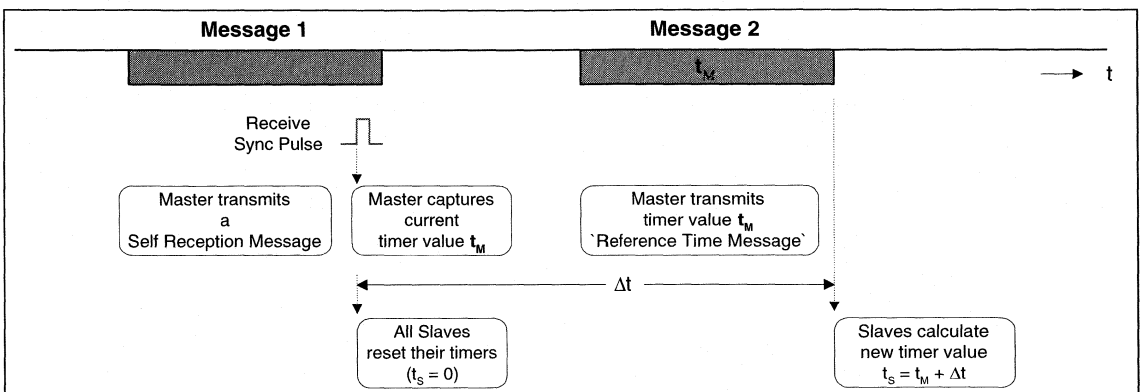


Figure 29: Timing Diagram during System Synchronization

The major advantage of this concept is the simplicity of implementation without complicated time stamp handling. No software cycle count is necessary because critical paths are hardware controlled and therefore deterministic. Furthermore it is independent of network parameters. Interrupt events may happen during the complete period without influencing the synchronization process.

6. REFERENCES

[1] Data Sheet SJA1000, Philips Semiconductors

[2] Eisele, H. and Jöhnk, E.: PCA82C250/251 CAN Transceiver, Application Note AN96116, Philips Semiconductors, 1996

[3] Data Sheet PCA82C250, Philips Semiconductors, September 1994

[4] Data Sheet PCA82C251, Philips Semiconductors, October 1996

[5] Data Sheet TJA1053, Philips Semiconductors,

[6] Jöhnk, E. and Dietmayer, K.: Determination of Bit Timing Parameters for the CAN Controller SJA1000, Application Note AN97046, Philips Semiconductors, 1997

[7] Data Sheet PCx82C200, Philips Semiconductors, November 1992

[8] CAN Specification Version 2.0, Parts A and B, Philips Semiconductors, 1992

[9] Hank, P.: PeliCAN: A New CAN Controller Supporting Diagnosis and System Optimization, 4th International CAN Conference, Berlin, Germany, October 1997

7. APPENDIX

For the examples in the Application Note the C-language (C-compiler from Keil) is used to describe possible flows for the programming of the SJA1000. The target controller in these examples is the S87C654 from Philips Semiconductors, but any other derivative of the 80C51 family can be used. Be sure to include the correct register declaration for the targeted derivative in the main program.

Register and bit definitions for the SJA1000

```
/* definition for direct access to 8051 memory areas                */

#define XBYTE ((unsigned char volatile xdata *) 0)


/* address and bit definitions for the Mode & Control Register      */

#define ModeControlReg  XBYTE[0]

#define RM_RR_Bit     0x01    /* reset mode (request) bit           */
#if defined (PeliCANMode)
#define LOM_Bit       0x02    /* listen only mode bit               */
#define STM_Bit       0x04    /* self test mode bit                 */
#define AFM_Bit       0x08    /* acceptance filter mode bit         */
#define SM_Bit        0x10    /* enter sleep mode bit               */
#endif


/* address and bit definitions for the
                          Interrupt Enable & Control Register       */

#if defined (PeliCANMode)
#define InterruptEnReg  XBYTE[4] /* PeliCAN mode                    */

#define RIE_Bit       0x01    /* receive interrupt enable bit       */
#define TIE_Bit       0x02    /* transmit interrupt enable bit      */
#define EIE_Bit       0x04    /* error warning interrupt enable bit */
#define DOIE_Bit      0x08    /* data overrun interrupt enable bit  */
#define WUIE_Bit      0x10    /* wake-up interrupt enable bit       */
#define EPIE_Bit      0x20    /* error passive interrupt enable bit */
#define ALIE_Bit      0x40    /* arbitration lost interr. enable bit*/
#define BEIE_Bit      0x80    /* bus error interrupt enable bit     */
#else  /* BasicCAN mode */
#define InterruptEnReg  XBYTE[0] /* Control Register                */

#define RIE_Bit       0x02    /* Receive Interrupt enable bit       */
#define TIE_Bit       0x04    /* Transmit Interrupt enable bit      */
#define EIE_Bit       0x08    /* Error Interrupt enable bit         */
#define DOIE_Bit      0x10    /* Overrun Interrupt enable bit       */
#endif


/* address and bit definitions for the Command Register             */

#define CommandReg      XBYTE[1]

#define TR_Bit        0x01    /* transmission request bit           */
#define AT_Bit        0x02    /* abort transmission bit             */
#define RRB_Bit       0x04    /* release receive buffer bit         */
#define CDO_Bit       0x08    /* clear data overrun bit             */
#if defined (PeliCANMode)
#define SRR_Bit       0x10    /* self reception request bit         */
```

```
#else  /* BasicCAN mode */
#define GTS_Bit       0x10   /* goto sleep bit (BasicCAN mode)     */
#endif


/* address and bit definitions for the Status Register            */

#define StatusReg         XBYTE[2]

#define RBS_Bit       0x01   /* receive buffer status bit          */
#define DOS_Bit       0x02   /* data overrun status bit            */
#define TBS_Bit       0x04   /* transmit buffer status bit         */
#define TCS_Bit       0x08   /* transmission complete status bit   */
#define RS_Bit        0x10   /* receive status bit                 */
#define TS_Bit        0x20   /* transmit status bit                */
#define ES_Bit        0x40   /* error status bit                   */
#define BS_Bit        0x80   /* bus status bit                     */


/* address and bit definitions for the Interrupt Register         */

#define InterruptReg    XBYTE[3]

#define RI_Bit        0x01   /* receive interrupt bit              */
#define TI_Bit        0x02   /* transmit interrupt bit             */
#define EI_Bit        0x04   /* error warning interrupt bit        */
#define DOI_Bit       0x08   /* data overrun interrupt bit         */
#define WUI_Bit       0x10   /* wake-up interrupt bit              */
#if defined (PeliCANMode)
#define EPI_Bit       0x20   /* error passive interrupt bit        */
#define ALI_Bit       0x40   /* arbitration lost interrupt bit     */
#define BEI_Bit       0x80   /* bus error interrupt bit            */
#endif


/* address and bit definitions for the Bus Timing Registers       */

#define BusTiming0Reg    XBYTE[6]
#define BusTiming1Reg    XBYTE[7]

#define SAM_Bit       0x80   /* sample mode bit
                                1 == the bus is sampled 3 times
                                0 == the bus is sampled once       */


/* address and bit definitions for the Output Control Register    */

#define OutControlReg    XBYTE[8]

                             /* OCMODE1, OCMODE0                   */
#define BiPhaseMode   0x00   /* bi-phase output mode               */
#define NormalMode    0x02   /* normal output mode                 */
#define ClkOutMode    0x03   /* clock output mode                  */

                             /* output pin configuration for TX1   */
#define OCPOL1_Bit    0x20   /* output polarity control bit        */
#define Tx1Float      0x00   /* configured as float                */
#define Tx1PullDn     0x40   /* configured as pull-down            */
#define Tx1PullUp     0x80   /* configured as pull-up              */
#define Tx1PshPull    0xC0   /* configured as push/pull            */

                             /* output pin configuration for TX0   */
#define OCPOL0_Bit    0x04   /* output polarity control bit        */
#define Tx0Float      0x00   /* configured as float                */
#define Tx0PullDn     0x08   /* configured as pull-down            */
```

```
#define Tx0PullUp     0x10    /* configured as pull-up                */
#define Tx0PshPull    0x18    /* configured as push/pull              */


/* address definitions of Acceptance Code & Mask Registers           */

#if defined (PeliCANMode)
#define AcceptCode0Reg    XBYTE[16]
#define AcceptCode1Reg    XBYTE[17]
#define AcceptCode2Reg    XBYTE[18]
#define AcceptCode3Reg    XBYTE[19]
#define AcceptMask0Reg    XBYTE[20]
#define AcceptMask1Reg    XBYTE[21]
#define AcceptMask2Reg    XBYTE[22]
#define AcceptMask3Reg    XBYTE[23]
#else  /* BasicCAN mode */
#define AcceptCodeReg     XBYTE[4]
#define AcceptMaskReg     XBYTE[5]
#endif


/* address definitions of the Rx-Buffer                              */

#if defined (PeliCANMode)
#define RxFrameInfo       XBYTE[16]
#define RxBuffer1         XBYTE[17]
#define RxBuffer2         XBYTE[18]
#define RxBuffer3         XBYTE[19]
#define RxBuffer4         XBYTE[20]
#define RxBuffer5         XBYTE[21]
#define RxBuffer6         XBYTE[22]
#define RxBuffer7         XBYTE[23]
#define RxBuffer8         XBYTE[24]
#define RxBuffer9         XBYTE[25]
#define RxBuffer10        XBYTE[26]
#define RxBuffer11        XBYTE[27]
#define RxBuffer12        XBYTE[28]
#else  /* BasicCAN mode */
#define RxBuffer1         XBYTE[20]
#define RxBuffer2         XBYTE[21]
#define RxBuffer3         XBYTE[22]
#define RxBuffer4         XBYTE[23]
#define RxBuffer5         XBYTE[24]
#define RxBuffer6         XBYTE[25]
#define RxBuffer7         XBYTE[26]
#define RxBuffer8         XBYTE[27]
#define RxBuffer9         XBYTE[28]
#define RxBuffer10        XBYTE[29]
#endif


/* address definitions of the Tx-Buffer                              */

#if defined (PeliCANMode)
/* write only addresses                                              */
#define TxFrameInfo       XBYTE[16]
#define TxBuffer1         XBYTE[17]
#define TxBuffer2         XBYTE[18]
#define TxBuffer3         XBYTE[19]
#define TxBuffer4         XBYTE[20]
#define TxBuffer5         XBYTE[21]
#define TxBuffer6         XBYTE[22]
#define TxBuffer7         XBYTE[23]
#define TxBuffer8         XBYTE[24]
#define TxBuffer9         XBYTE[25]
```

```
#define TxBuffer10        XBYTE[26]
#define TxBuffer11        XBYTE[27]
#define TxBuffer12        XBYTE[28]
/* read only addresses                                          */
#define TxFrameInfoRd     XBYTE[96]
#define TxBufferRd1       XBYTE[97]
#define TxBufferRd2       XBYTE[98]
#define TxBufferRd3       XBYTE[99]
#define TxBufferRd4       XBYTE[100]
#define TxBufferRd5       XBYTE[101]
#define TxBufferRd6       XBYTE[102]
#define TxBufferRd7       XBYTE[103]
#define TxBufferRd8       XBYTE[104]
#define TxBufferRd9       XBYTE[105]
#define TxBufferRd10      XBYTE[106]
#define TxBufferRd11      XBYTE[107]
#define TxBufferRd12      XBYTE[108]
#else  /* BasicCAN mode */
#define TxBuffer1         XBYTE[10]
#define TxBuffer2         XBYTE[11]
#define TxBuffer3         XBYTE[12]
#define TxBuffer4         XBYTE[13]
#define TxBuffer5         XBYTE[14]
#define TxBuffer6         XBYTE[15]
#define TxBuffer7         XBYTE[16]
#define TxBuffer8         XBYTE[17]
#define TxBuffer9         XBYTE[18]
#define TxBuffer10        XBYTE[19]
#endif


/* address definitions of Other Registers                       */

#if defined (PeliCANMode)
#define ArbLostCapReg     XBYTE[11]
#define ErrCodeCapReg     XBYTE[12]
#define ErrWarnLimitReg   XBYTE[13]
#define RxErrCountReg     XBYTE[14]
#define TxErrCountReg     XBYTE[15]
#define RxMsgCountReg     XBYTE[29]
#define RxBufStartAdr     XBYTE[30]
#endif


/* address and bit definitions for the Clock Divider Register   */

#define ClockDivideReg    XBYTE[31]

#define DivBy1        0x07    /* CLKOUT = oscillator frequency      */
#define DivBy2        0x00    /* CLKOUT = 1/2 oscillator frequency  */

#define ClkOff_Bit    0x08    /* clock off bit,
                                 control of the CLK OUT pin  */
#define RXINTEN_Bit   0x20    /* pin TX1 used for receive interrupt */
#define CBP_Bit       0x40    /* CAN comparator bypass control bit  */
#define CANMode_Bit   0x80    /* CAN mode definition bit            */
```

Register and bit definitions for the Micro Controller S87C654

```
/*  Port 2 Register  "P2"                                              */
sfr P2      = 0xA0;

sbit P2_7   = 0xA7;   /* MSB of port 2, used for chip select for SJA1000 */
.

/* alternate functions of port 3 "P3"                                  */
sfr P3      = 0xB0;
.
sbit int0   = 0xB2;
.

/*  Timer Control Register   "TCON"  */
sfr TCON    = 0x88;

.
sbit IE0    = 0x89;   /* external interrupt 0 edge flag                */
sbit IT0    = 0x88;   /* interrupt 0 type control bit
                         (edge or low-level triggered                  */
.

/*  Interrupt Enable Register   "IE" */
sfr IE      = 0xA8;

sbit EA     = 0xAF;   /* overall interrupt enable/disable flag         */
.
sbit EX0    = 0xA8;   /* enable or disable external interrupt 0        */
.

/*  Interrupt Priority Register   "IP" */
sfr IP      = 0xB8;

.
sbit PX0    = 0xB8;   /* external interrupt 0 priority level control   */
.
```

Definitions of Variables and Constants for the Examples

```
/*- definition of hardware / software connections ----------------------*/

/* controller: S87C654; CAN controller: SJA1000(see Figure 3 on page 11)*/

#define CS        P2_7    /* ChipSelect for the SJA1000                  */

#define SJAIntInp int0    /* external interrupt 0 (from SJA1000)         */

#define SJAIntEn  EX0     /* external interrupt 0 enable flag            */


/*- definition of used constants ---------------------------------------*/

#define YES           1
#define NO            0

#define ENABLE        1
#define DISABLE       0
#define ENABLE_N      0
#define DISABLE_N     1

#define INTLEVELACT   0
#define INTEDGEACT    1

#define PRIORITY_LOW  0
#define PRIORITY_HIGH 1

/* default (reset) value for register content, clear register     */
#define ClrByte      0x00

/* constant: clear Interrupt Enable Register                      */
#if defined (PeliCANMode)
#define ClrIntEnSJA ClrByte
#else
#define ClrIntEnSJA ClrByte | RM_RR_Bit /* preserve reset request */
#endif

/* definitions for the acceptance code and mask register          */
#define DontCare     0xFF


/*- definition of bus timing values for different examples -------*/

/* bus timing values for the example given in (AN97046)
   - bit-rate                          : 250 kBit/s
   - oscillator frequency              : 24 MHz, 1,0%
   - maximum propagation delay         : 1630 ns
   - minimum requested propagation delay :  120 ns                */
#define PrescExample     0x02 /* baud rate prescaler : 3          */
#define SJWExample       0xC0 /* SJW                 : 4          */
#define TSEG1Example     0x0A /* TSEG1               : 11         */
#define TSEG2Example     0x30 /* TSEG2               : 4          */

/* bus timing values for
   - bit-rate                          : 1 MBit/s
   - oscillator frequency              : 24 MHz, 0,1%
   - maximum tolerated propagation delay : 747 ns
   - minimum requested propagation delay :  45 ns                 */
#define Presc_MB_24      0x00 /* baud rate prescaler : 1          */
#define SJW_MB_24        0x00 /* SJW                 : 1          */
```

```
#define TSEG1_MB_24       0x08 /* TSEG1                 : 9          */
#define TSEG2_MB_24       0x10 /* TSEG2                 : 2          */

/* bus timing values for
   - bit-rate                          : 100 kBit/s
   - oscillator frequency              : 24 MHz, 1,0%
   - maximum tolerated propagation delay : 4250 ns
   - minimum requested propagation delay :  100 ns                   */
#define Presc_kB_24       0x07 /* baud rate prescaler : 8          */
#define SJW_kB_24         0xC0 /* SJW                 : 4          */
#define TSEG1_kB_24       0x09 /* TSEG1               : 10         */
#define TSEG2_kB_24       0x30 /* TSEG2               : 4          */

/* bus timing values for
   - bit-rate                          : 1 MBit/s
   - oscillator frequency              : 16 MHz, 0,1%
   - maximum tolerated propagation delay : 623 ns
   - minimum requested propagation delay :  23 ns                    */
#define Presc_MB_16       0x00 /* baud rate prescaler : 1          */
#define SJW_MB_16         0x00 /* SJW                 : 1          */
#define TSEG1_MB_16       0x04 /* TSEG1               : 5          */
#define TSEG2_MB_16       0x10 /* TSEG2               : 2          */

/* bus timing values for
   - bit-rate                          : 100 kBit/s
   - oscillator frequency              : 16 MHz, 1,0%
   - maximum tolerated propagation delay : 4450 ns
   - minimum requested propagation delay :  500 ns                   */
#define Presc_kB_16       0x04 /* baud rate prescaler : 5          */
#define SJW_kB_16         0xC0 /* SJW                 : 4          */
#define TSEG1_kB_16       0x0A /* TSEG1               : 11         */
#define TSEG2_kB_16       0x30 /* TSEG2               : 4          */

/*- end of definitions of bus timing values ----------------------*/


/*- definition of used variables ---------------------------------*/

/* intermediate storage of the content of the Interrupt Register  */
BYTE bdata CANInterrupt;                /* bit addressable byte   */
sbit RI_BitVar     = CANInterrupt ^ 0;
sbit TI_BitVar     = CANInterrupt ^ 1;
sbit EI_BitVar     = CANInterrupt ^ 2;
sbit DOI_BitVar    = CANInterrupt ^ 3;
sbit WUI_BitVar    = CANInterrupt ^ 4;
sbit EPI_BitVar    = CANInterrupt ^ 5;
sbit ALI_BitVar    = CANInterrupt ^ 6;
sbit BEI_BitVar    = CANInterrupt ^ 7;
```

Section 7
Application Notes

CONTENTS

Digital filtering using XA

AN700

Author: Santanu Roy, MCO Applications Group, Sunnyvale, California

SUMMARY

This report describes a method of implementation of FIR filters using Philips XA microcontroller. Appended with this application note is a generic routine that could be used to implement a N-point FIR filter.

INTRODUCTION

The term "digital filter" refers to the computational process or algorithm by which a digital signal or sequence of numbers (acting as input) is transformed into a second sequence of numbers termed the output digital signal. Digital filters involve signals in the digital domain (discrete-time signals) and are used extensively in applications such as digital image processing, pattern recognition, and spectral analysis.

Digital Signal Processing (DSP) is concerned with the representation of signals (and information they contain) by sequences of numbers and with the transformation or processing of such signal representations by numeric computational procedures. In order to be considered a DSP microcontroller, a part must be able to quickly multiply two values, and add the result to an accumulator register. this is a minimum requirement. "Quickly" implies MAC (Multiply and Accumulate). Typically, the multiply and accumulate path operates on 16-bit values with a 32-bit result. Figure 1 shows a typical Digital Signal Processing hardware used in digital filtering.

Although XA currently does not have a hardware MAC unit, it is quite suitable for some DSP applications, due to its relatively high computational power, and high I/O throughput. This application note is intended to demonstrate such DSP power of the XA through implementation of FIR and IIR digital filters. It is to be noted, though, that this application note is not intended as a learning tool for DSP. It is assumed that the reader is familiar with DSP and filtering basics.

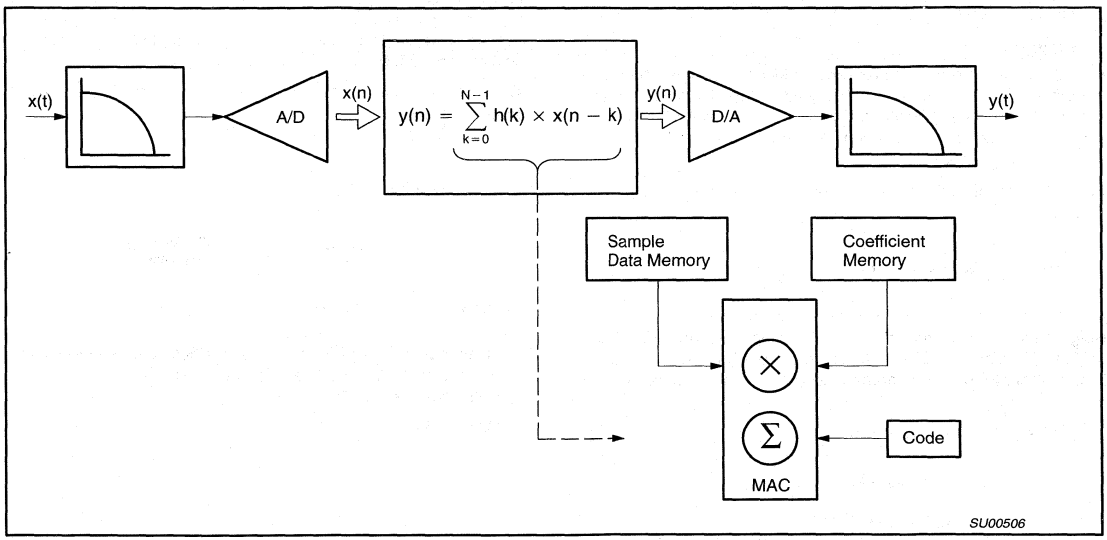


Figure 1. Typical DSP Hardware

Filter Algorithms

For a large variety of applications, digital filters are usually based on the following relationship between the filter input sequence x(n) and filter output sequence y(n);

$$y(n) = \sum_{k=0}^{N} a_k \times y(n-k) + \sum_{k=0}^{M} b_k \times x(n-k) \quad (1)$$

where a_k and b_k represent constant coefficients and N and M represent the number of input samples.

Equation (1) is referred to as a linear constant coefficient difference equation. Two classes of filters can be represented by such equations:

3. Finite Impulse Response (FIR) filters, and
4. Infinite Impulse Response (IIR) filters.

This applications note describes the implementation of the FIR class of digital filters on the XA.

FIR Filters

FIR filters are preferred in lower order solutions, and since they do not employ feedback (output values used in the calculation of newer output values), they exhibit naturally bounded response. They are simpler to implement, and require one RAM location and one coefficient for each order.

For FIR filters, all of the a_k in equation (1) is zero. Therefore (1) reduces to:

$$y(n) = \sum_{k=0}^{M} b_k \times x(n-k) \quad (2)$$

As a result, the output of an FIR filter is simply a finite length weighted sum of the present and previous inputs to the filter. If the unit-sample response of the filter is denoted as h(n), then from (2), it is seen that h(n) = b(n). Therefore, (2) is sometimes written as:

$$y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k) \quad (3)$$

where N = length of the filter = M+1.

Digital Filter Implementation

As described above, a digital filter (FIR or IIR) could then be implemented by multiplying a vector of sampled signals with another vector of constants (coefficients) and adding the results to a register. The vectors involved in the filter process are derived from transformation of an S domain transfer function into the sampled Z domain.

The Multiply-Accumulate (MAC) Function

The MAC speed applies both to finite impulse response (FIR) and finite impulse response (IIR) filters. The complexity of the filter response dictates the number MAC operations required per sample period.

A multiply-accumulate step performs the following:

- Read a 16-bit sample data (pointed to by a register)
- Increment the sample data pointer by 2
- Read a 16-bit coefficient (pointed to by another register)
- Increment the coefficient register pointer by 2
- Sign Multiply (16-bit) data and coefficient to yield a 32-bit result
- Add the result to the contents of a 32-bit register pair for accumulate.

This accumulator should be initialized to zero before calculating each output. It is assumed that the algorithm cannot overflow the accumulator, either by reducing significant bits of samples and/or constants or the number of accumulations.

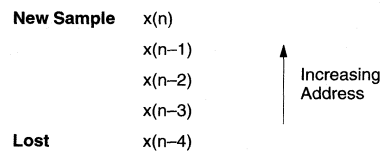
All these above MAC operations take place in addition to a buffer management routine that maintains an updated database for the filters samples, and system coefficients.

Buffer Management

In order to effectively perform the task of buffer management, the processor should be able to quickly "shift" data (or pointers) in a data array which contains a series of input samples. New data is going in, and oldest sample is disposed off.

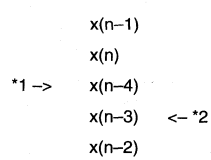
There are few ways to maintain and manage this database. They are as follows:

1. Linear Buffer



Linear buffer management requires the data to physically move down towards the oldest sample, then the newest sample is written into the top (FIFO style).

2. Circular Buffer



*1 At the beginning of filter pass, input pointer points to the oldest (n–4) sample, new sample is stored there.
*2 At the end of the filter pass, pointer now points to the next oldest sample (n–3).

Circular buffer management requires a test to make sure a buffer pointer increment does not move the pointer beyond the "tail" (end) of the buffer. If so, the pointer must be reset to the "head" (beginning) of the buffer.

Selecting one approach versus another depends mainly on the overhead involved with this task over the plain Multiply-Accumulate and loop control operations, and may vary based on the processor architecture, storage access time and other factors.

MAC Implementation on the XA

An efficient loop for memory mapped vectors is presented below. The loop entry is at an even address, to reduce the fetch overhead after branch to beginning of the loop. Arrays are accessed using the indirect-autoincrement addressing mode.

```
.incld fir.h
  MAC_LOOP:
       mov.w    R3, [R1+]     ; read sample vector entry
       mov.w    R4, [R2+]     ; read coefficient vector entry
       mul.W    R3, R4        ; multiply
       add      R5, R3        ; accumulate into
       addc     R6, R4        ; a 32 bit register pair(R5:R6)
                              ; this serves as the RRP
       djnz     R0, MAC_LOOP ;
                              ;decrement loop counter and
                              ;branch to MAC_LOOP
```

The loop contains 13 bytes and takes 32 clocks (including branch penalty) per iteration (1.6µS at 20MHz and 1.07µS at 30.0MHz).

The following section analyzes the digital filter performance, including initialization, I/O, MAC operations and sample vector buffer management.

An N-Point FIR Filter Implementation on XA

The FIR filter maintains a list of a fixed number N of recent samples. At each iteration, a new sample is taken, replacing the oldest sample on the list. This list represents a sampled vector. It is then multiplied by an N constant's vector to yield the current output.

As mentioned earlier, there are 2 register pointers fetching data samples and coefficient and feeding it to the ALU for 16-bit signed multiply with the 32-bit result being added to the MAC result register pair (RRP). In addition, a buffer management routine updates the sample data buffer each sample period.

The following sample codes show the mechanism for running filters on successive samples. It reflects the simplest data structures and list management, to simulate an output of a high level compiler.

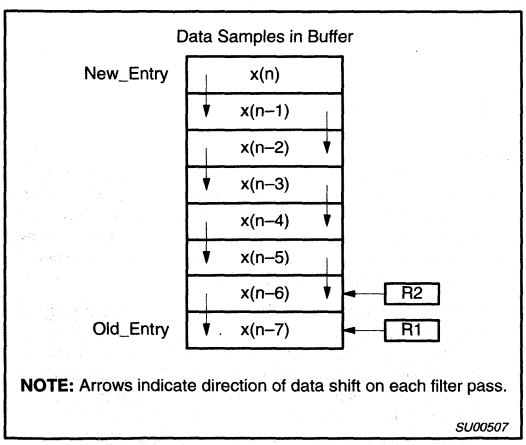


Figure 2. Buffer Management for FIR Filter

FIR Algorithm in XZ

```
;Preliminary initialization for first filter pass:
.incldfir.h
  Start_FIR:
    mov  R0, #N-1           ; N = number of entries in the list
                            ; = loop counter
    mov  R1, #Old_Entry     ; compiler uses 2 pointers
    mov  R2, #Old_Entry+2   ;

Shft_Smpl:
    mov.w  [R1+], [R2+]
    djnz   R0, Shft_Smpl
                            ; - SAMPLE FROM A/D PORT
    mov    R0, A2D          ; input from port
    and    R0, #mask        ; mask upper bits (for N-bit A/D)
    mov    New_Entry, R0    ; add to list

Mac_init:                   ; MULTIPLY ACCUMULATE
    mov    R0, #N           ; N = number of entries in the list
                            ; = loop counter
    mov    R1, #Old_Entry   ; pointer to sample vector
    mov    R2, #Coef_Entry  ; pointer to coefficient vector
    xor    R5, R5           ; zero to accumulator
    xor    R6, R6           ; zero to accumulator

MAC_LOOP:
    mov.w  R3, [R1+]        ; read sample from list to reg
    mov.w  R4, [R2+]        ; read constant from list to reg
    mul.w  R3, R4           ; multiply
    add    R5, R3           ; accumulate in RRP
    addc   R6, R4           ; complete 32 bit add
    djnz   R0, MAC_LOOP

ACC_corr:                   ; - NORMALIZE RESULT BY SHIFTING
    asl    R5, #norm**      ; correction for non-significant LSBs
                            ; for eight 10 bit samples and 16 bit
                            ; constants, #norm=3
                            ; i.e. take only most significant 29 bits of the result
                            ; [16+10 + 3 (for 8 iterations)]
                            ; - OUTPUT TO D2A PORT
    mov    DAC, R6          ; send to DAC

        A total of 62 bytes and 370 clocks for this FIR algorithm.
```

** For N=8, 10 bit A/D, 16 bit filter coefficients; 8, 12, 16 bits clock very similar performance.

Total time for an 8-point filter at 20 MHz is 19.0 microseconds and 12.7 microseconds at 30 MHz. This would translate to a maximum sampling rate of 52 KHz at 20 MHz and 78 KHz at 30 MHz clock. If this filter algorithm is interrupt driven, then additional 20 clocks would be required for latency, which would then translate to 50 KHz maximum sampling rate at 20 MHz and 75 KHz at 30 MHz. This puts the XA in the bandwidth of Audio Signal Processing (44.1 KHz) applications.

NOTES:

1. The above FIR algorithms are assembled with "asmxa rev 1.4", the first XA absolute assembler for verification. It is to be noted in this context, that this assembler is a beta-site tool and still under evaluation. The syntax used in the assembler might be subjected to change. The functionality of the code is not checked at this stage using any simulator or ICE.
2. It is possible in the above MAC operation to extend the length of the accumulator to accommodate more iterations and higher precision (greater than 10-bit A/D) sample values with some additional overhead, e.g., using 'ADDC Rn, R6H", etc., after the 32-bit accumulate, where Rn is a byte-size register to increase the length of the accumulator to accommodate more accumulations and higher precision (greater than 10-bit A/D) sample values.

Author's Note

All addresses and constants assumed 16 bit for generality. Performance is calculated for a work-aligned branch targets which is mandated in the XA architecture for performance reasons. Misalignment will result in addition of NOPs by the assembler causing penalty in both code density and execution times. It is also to be mentioned that this is not the fastest executable code for the XA. A good programmer can combine the two loops into one, and data can be kept in registers. For low order filter implementation, code can be written in-line, and can utilize direct addressing mode for samples array.

This code was written in a way that reflects minimum expected optimization form a compiler (local loop optimization only), and it shows the expected speed for code written in a high level language, without rewriting routines in assembly language. also, this is not the ultimate performance for the XA architecture. The register banks can be used to store coefficients and samples, resulting in slightly faster execution time.

Author's Acknowledgement

The author recognizes the following Philips Semiconductors XA team members for their review and inputs on this article:

Greg Goodhue, Ori Mizrahi-Shalom, and Ata Khan.

References

XA User Guide — Philips Semiconductors
Digital Signal Processing — Rosenbaum

SP floating point math with XA

AN701

Author: Santanu Roy, MCO Applications Group, Sunnyvale, California

IEEE SINGLE PRECISION FLOATING POINT ARITHMETIC WITH XA

Introduction

This application note is intended to implement Single Precision Floating Point Arithmetic package using the new Philips Semiconductors XA microcontroller. The goal is to have this package as a part of the run-time math library for the XA when the cross-compiler is developed. The package is based upon the IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985). This package, however, is not a conforming implementation of the said standard. The differences between the XA implementation and the standard are listed later in this report. Also, this package does not include routines for conversion between Integers to Floating Point and vice versa.

The following four standard Single Precision (SP) arithmetic operations have been implemented in this package:

1. **FPADD** Addition of two SP floating point numbers.
2. **FPSUB** Subtraction of two SP floating point numbers.
3. **FPMUL** Multiplication of two SP floating point numbers.
4. **FPDIV** Division of two SP floating point numbers.

The following section discusses the representation of FLP numbers. Then the differences between the XA implementation and the IEEE standard are described. This is followed by a description of the algorithms used in the computations. Appendix A is a user reference section for converting a floating point number into IEEE format and Appendix B is a listing of the code.

Note that this application note assumes that the reader is familiar with the IEEE Binary Floating-Point standard.

IEEE Floating Point Formats

The basic format sizes for floating-point numbers, 32 bits and 64 bits, were selected for efficient calculation of array elements in byte-addressable memories. For the 32-bit format, precision was deemed the most important criterion, hence the choice of radix 2 instead of octal or hexadecimal. Other characteristics include not representing the leading significand bit in normalized numbers, a minimally acceptable exponent range which uses 8 bits, and exponent bias which allows the reciprocal of all normalized numbers to be represented without overflow. For the 64-bit format, the main consideration was range.

Representation of FLP Number

The IEEE binary floating point number is represented in the following format:

$FP = \pm \text{Significand} \times \text{Base}^{\text{Characteristic}}$

The specification of a binary FP number involves two parts:
A Significand or Mantissa
and a Characteristic or Exponent.

The Mantissa is signed fixed point number and the Exponent is a signed integer. Mantissa or Significand is that component of a binary FLP number which consists of an explicit or implicit leading bit to the left of its binary point and a fraction field to the right of the binary point. Exponent signifies the power to which 2 is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent. The IEEE standard specifies that a single precision Floating Point number should be represented in 32 bits as shown in Figure 1.

Figure 1.

The significance of each of these fields is as follows:

1. **SIGN** — This 1–bit field is the sign of the Mantissa. '0' indicates a *positive* and '1' indicates a *negative* number.
2. **EXPONENT** — This is a 8–bit field. The width of this field determines the range of the FP number. The exponent is represented as a biased value with a bias of 127 decimal. The bias value is added to exponents in order to keep them always positive and is represented by

   $2^{n-1} - 1,$

   where n = number of bits in the binary exponent.
3. **MANTISSA** — This is a 23-bit field representing the fractional part. The width of this field determines the precision for the FP number. For normalized FP numbers (see below), a MSB of '1' is assumed and not represented. Thus, for normalized numbers, the value of the mantissa is 1.Mantissa. This provides an effective precision of 24–bits for the mantissa.

   If dealt with normalized numbers only (as the XA implementation does), then the MSB of the Mantissa need not be explicitly represented as per IEEE standard specification.The normalized significand lies in the range shown below.

   $1.0 < \text{Normalized Mantissa} < 2.0$

   Given the values of Sign, Exponent, and Mantissa, the value of the FP number is obtained as follows:

(i) If $0 < \text{Exp} < 255$, then

$FP = (-1)^{SIGN} \times 2^{EXP-127} \times 1.\text{MANTISSA}$

(ii) If Exp = 0, then FP = 0

(iii) If Exp = 255, and Mantissa =/= 0, then FP = Invalid Number (NaN or Not a Number).

The above format for single precision binary FP numbers provides for the representation in the range -3.4×10^{38} to -1.75×10^{-38}, 0, and 1.75×10^{-38} to 3.4×10^{38}. The accuracy is between 7 and 8 decimal digits.

Differences with the IEEE Standards

The IEEE standard specifies a comprehensive list of operations and representations for FLP numbers. Since an implementation that fully conforms to this standard would lead to an excessive amount of overhead, a number of features in the standard were omitted. This section describes the differences between the implemented package and the standard.

1. **Omission of –0** — The IEEE standard requires that both + and – 0 be represented, and arithmetic carried out using both. The implementation does not represent –0.
2. **Omission of infinity arithmetic** — The IEEE standard provides for the representation of + and – infinity, and requires that valid arithmetic operations be carried out on infinity.
3. **Omission of Quiet NaN** — The IEEE standard provides for both Quiet and Signalling NaNs. A signalling NaN can be produced as

the result of overflow during an arithmetic operation. If the NaN is passed as input to further FLP routines, then these routines would produce another NaN as output. The routines will also set the Invalid Operation Flag, and call the user FLP error trap routine at address FPTRAP.

4. **Omission of denormalized numbers** — These are FLP numbers with a biased exponent, E of zero and non-zero mantissa F. Such denormalized numbers are useful in providing gradual underflow to 0. These are not represented in the XA implementation. Instead, if the result of a computation cannot be represented as a normalized number within the allowable exponent range, then an underflow is signalled, the result is set to 0, and the user FLP error trap routine at address FPTRAP is called.
5. **Omission of Inexact Result Exponent** — The IEEE standard requires that an Inexact Result Exception be signalled when the round result of an operation is not exact, or it overflows without an overflow trap. This feature is not provided.
6. **Biased Rounding to Nearest** — The IEEE standard requires that rounding to the nearest be provided as the default rounding mode. Further, the rounding is required to be unbiased. The XA implementation provides biased rounding to nearest only, e.g., suppose the result of an operation is .b1b2b3nnn and needs to be rounded to 3 binary digits. Then if nnn is 0YY, the round to nearest result is .b1b2b3. If nnn is 1YY, with at least one of the Y's being 1, then the result is .b1b2b3 + 0.001. Finally, if nnn is 100, it is a tie situation. In such a case, the IEEE standard requires that the rounded result be such that its LSB is 0. The XA implementation, on the other hand, will round the result in such a case to .b1b2b3 + 0.001.

DESCRIPTION OF ALGORITHMS

General Considerations

The XA implementation of the SP floating point package consists of a series of subroutines. The subroutines have been written and tested using Microsoft C however, not been tested with the XA C cross-compiler and also not optimized for code efficiency. The executable could be run under DOS in any IBM compatible PC. It is a menu driven routine that enables the user to select any of the 4 Floating Point routines. The menu also includes a "HELP" item designed to provide some standard SP floating point numbers, their operations and results as a quick reference.

The Arithmetic subroutines that compute F1 (Op) F2, where Op is +, –, *, or / expect that F1 and F2 are in IEEE format. Each of F1 and F2 consists of two 16-bit words organized as follows:

Fn–HI : Sign(1) Biased exponent(7) MSB of Mantissa(8)

Fn–LO : Least Significant word of Mantissa(16)

Exception Handling

The following types of exception can occur during the course of computation.

Invalid Operand—This exception occurs if one of the input is a NaN.

Exponent Overflow—This occurs if the result of a computation is such that its rounded result is finite and not an invalid result but its exponent is too large to represent in the floating point format, i.e., exponent has a biased value of 255 or more.

Exponent Underflow—This occurs if the result of a computation is such that its exponent is 0 or less.

Divide-by-zero—This exception occurs if the FLP divide routine is called with F2 being 0.

The package signals exceptions in 2 ways. First, a word at address ERRFLG is maintained that registers the history of the exception conditions. Bits 0–3 of this word are used for the same.

Bit 0 – Exponent overflow detect

Bit 1 – Exponent Underflow detect ERRFLG

Bit 2 – Illegal Operand detect

Bit 3 – Divide–by–0 detect

ERRFLG

*	*	*	*	DBZ	IOP	EUF	EOV

This bits are never cleared by the FLP package, and can be examined by the user software to determine the exception conditions that occurred during the course of a computation. If it is the responsibility of the user software to initialize this word before calling any of the floating point routines.

The second method that the package uses to signal exceptions is to call a user floating point exception handler whenever such conditions occurs. The corresponding exception bit in ERRFLG is set before calling the handler. The starting address of the handler should be defined by the symbol FPTRAP.

Unpacked FLP Format

The IEEE standard FLP format described earlier is very cumbersome to deal with during computation. This is primarily because of splitting of the mantissa between the 2 words. The subroutine in the package unpack the input IEEE FLP numbers into an internal representation, do the computations using this representation, and finally pack the result into the IEEE format before returning to the calling program. The unpacking is done by the subroutine FUNPAK and the packing by the subroutine FPAK. The unpacked format consists of 3 words and is organized as follows:

Unpacked FLP

Fn-Exponent (8-bit biased)	Fn-Sign (extended to 8-bits)
MS 16-bits of Mantissa (implicit 1 is present as MSB)	
LS 8-bits of Mantissa	Eight 0's

Since all computations are carried out in this format, note that the result is actually known to 32 bits. This 32-bit mantissa is rounded to 24 bits before packed to the IEEE format.

Algorithms

All the arithmetic algorithms first check for easy cases when either F1 or F2 is zero or a NaN. The result in these cases is immediately available. The description of the algorithms below is for those cases when neither F1 or F2 is 0 or a NaN. Also, in order to keep the algorithm description simple, the check for underflow/overflow at the various stages is not shown. The documentation in the program, the flowcharts given below, and the theory as described in the references should allow these programs to be easily maintained.

FPADD AND FPSUB

Before a floating point add/subtract instruction is executed, the 2 operands in normalized form

The processing steps are as follows:

1. Compare the 2 exponents.
2. Align the mantissas by equalizing their exponents.
3. Compute result sign as the XOR of the signs of the 2 numbers.
4. Add/Subtract the mantissas.

   For subtract, FP2 is complemented and added with FP1, i.e., FSUB = FP1 + (–FP2).
5. Normalize the resulting sum/difference.
6. Pack the exponent, sign and mantissa in IEEE format and return.

FMULT = FP1 * FP2

Floating-point multiplication is accomplished by multiplying the mantissas of the 2 operands and adding their corresponding exponents. Exponent overflow or underflow may occur when true addition is performed on 2 exponents of the same sign.

The processing steps are as follows:

1. Add the 2 exponents and subtract 7FH (IEE bias of 127_{10}) to yield the result exponent.

   Result Exponent = FP1_EXP + FP2_EXP – 127
2. XOR the sign bits to get the result sign.

   Result Sign = FP1_SIGN XOR FP2_SIGN
3. Compute FP1_HI x FP2_HI = C1_HI.C1_LO.
4. Compute FP1_HI x FP2_LO = C0_HI.C0_LO.
5. Add C0_HI + C1_LO = C2_LO.
   If more than 16-bits, then C1_HI += 1.
6. Compute FP1_LO x FP2_HI = C3_HI.C3_LO.
7. Add C3_HI + C2_LO = C4_LO.
   If more than 16-bits, then C1_HI += 1.
8. Normalize mantissa. If MSB of C1_HI =/= 1,
   then result exponent += 1 else left shift C1_HI.C4_LO.
9. Round C1_HI.C4_LO to get result mantissa.
10. Pack the exponent, sign and mantissa in IEEE format and return.

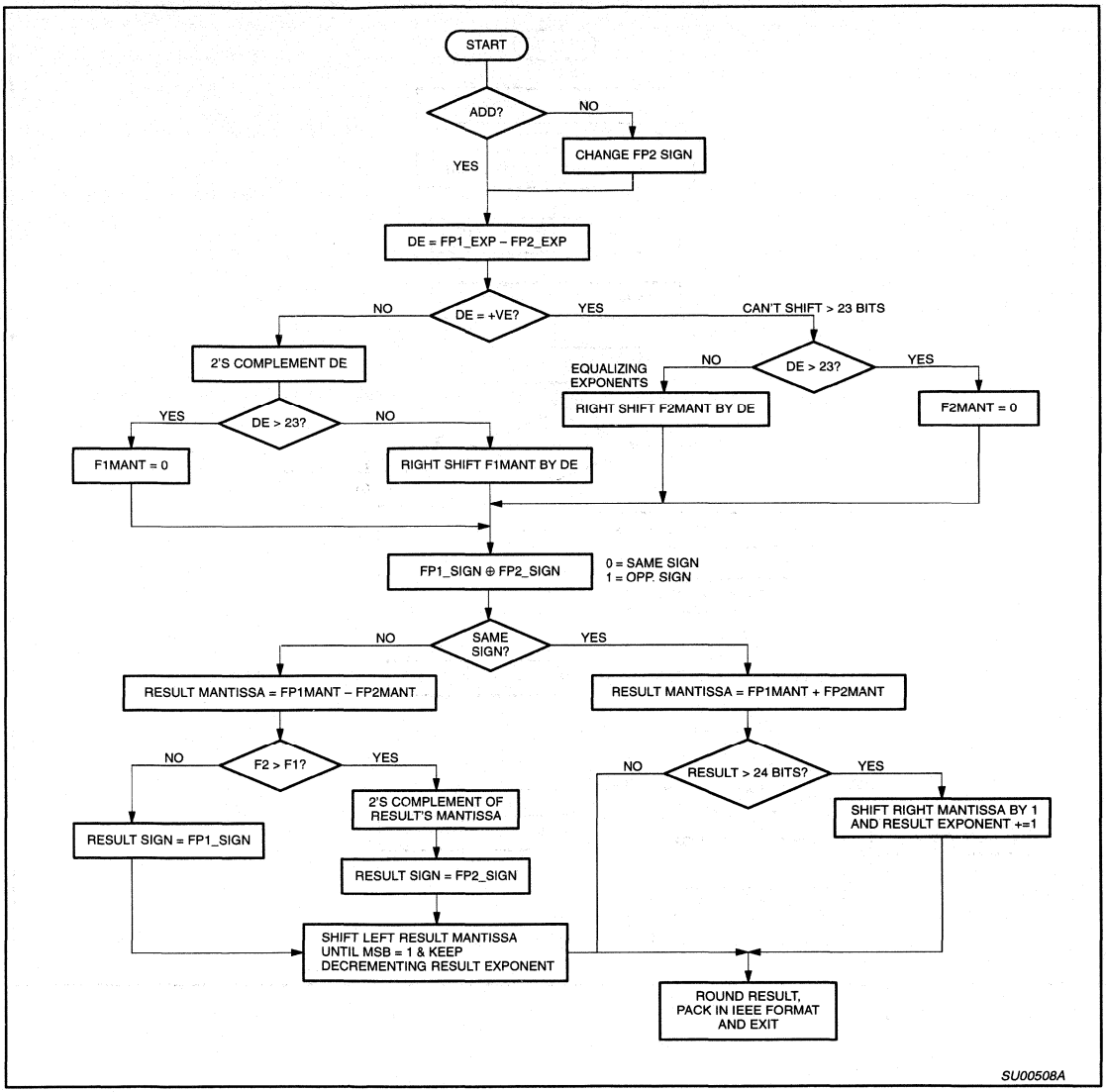
FPDIV = FP1/FP2

The way a floating-point DIVIDE instruction is executed is analogous to that of a Floating Point Multiply, except that mantissa multiplication is replaced by mantissa division and the exponent addition by exponent subtraction. Exponent overflow or underflow may occur when true addition is performed on the 2 exponents of opposite signs. The scheme must avoid the situation of having a divisor which is smaller than dividend mantissa, including the special case of a 0 divisor. With this constraint, the post normalization is unnecessary in FLP division as long as pre-normalization was conducted to avoid quotient overflow.

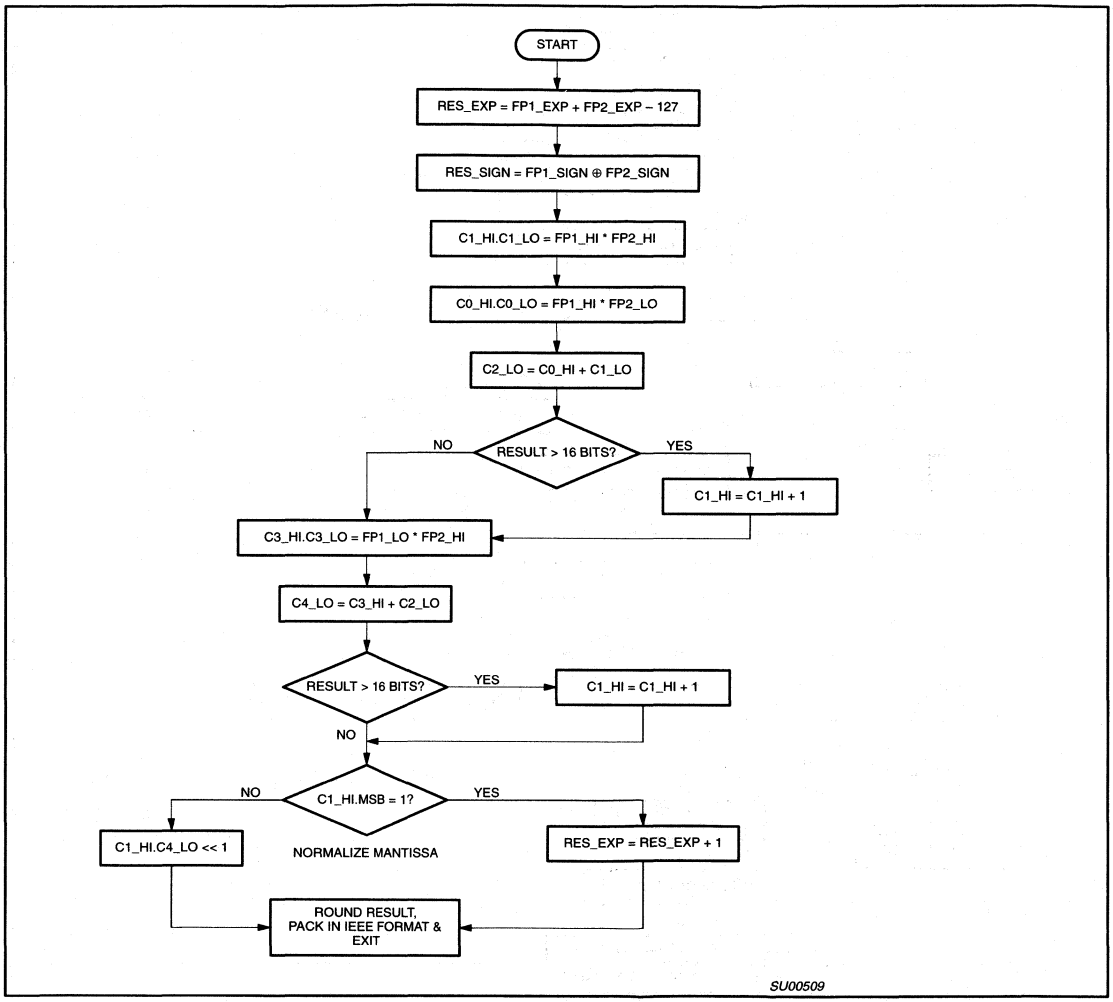
The processing steps are as follows:

1. Compare FP1_HI and FP2_HI.
   If FP2_HI > FP1_HI, then go to step 3, else go to step 2.
2. Shift right FP1 and FP1_EXP += 1.
3. Compute FP1_EXP – FP2_EXP + 127 to get C_EXP.
4. Compute FP1_SIGN XOR FP2_SIGN to get result sign, i.e., C_SIGN
5. Compute FP1_HI x FP2_LO = M1_HI.M1_LO.
6. Divide M1_HI.M1_LO / FP2_HI = M2_HI (Quotient)
7. Do a true subtract FP1_LO – M2_HI = M3_LO.
   If result –ve then go to step 8 else FP1_HI –= 1 and go to step 8.
8. Divide FP1_HI.M3_LO / FP2_HI = C1_HI (Quotient) + R1 (remainder)
9. Divide R1.0000 / FP2_HI = C1_LO (Quotient)
10. If MSB of C1_HI = 1, then go to step 11, else shift left C1_HI.C1_LO, C_EXP –= 1, and go to step 11.
11. Round C1_HI.C1_LO to get C_HI.C_LO, go to step 12
12. Pack the exponent, sign and mantissa in IEEE format and return.

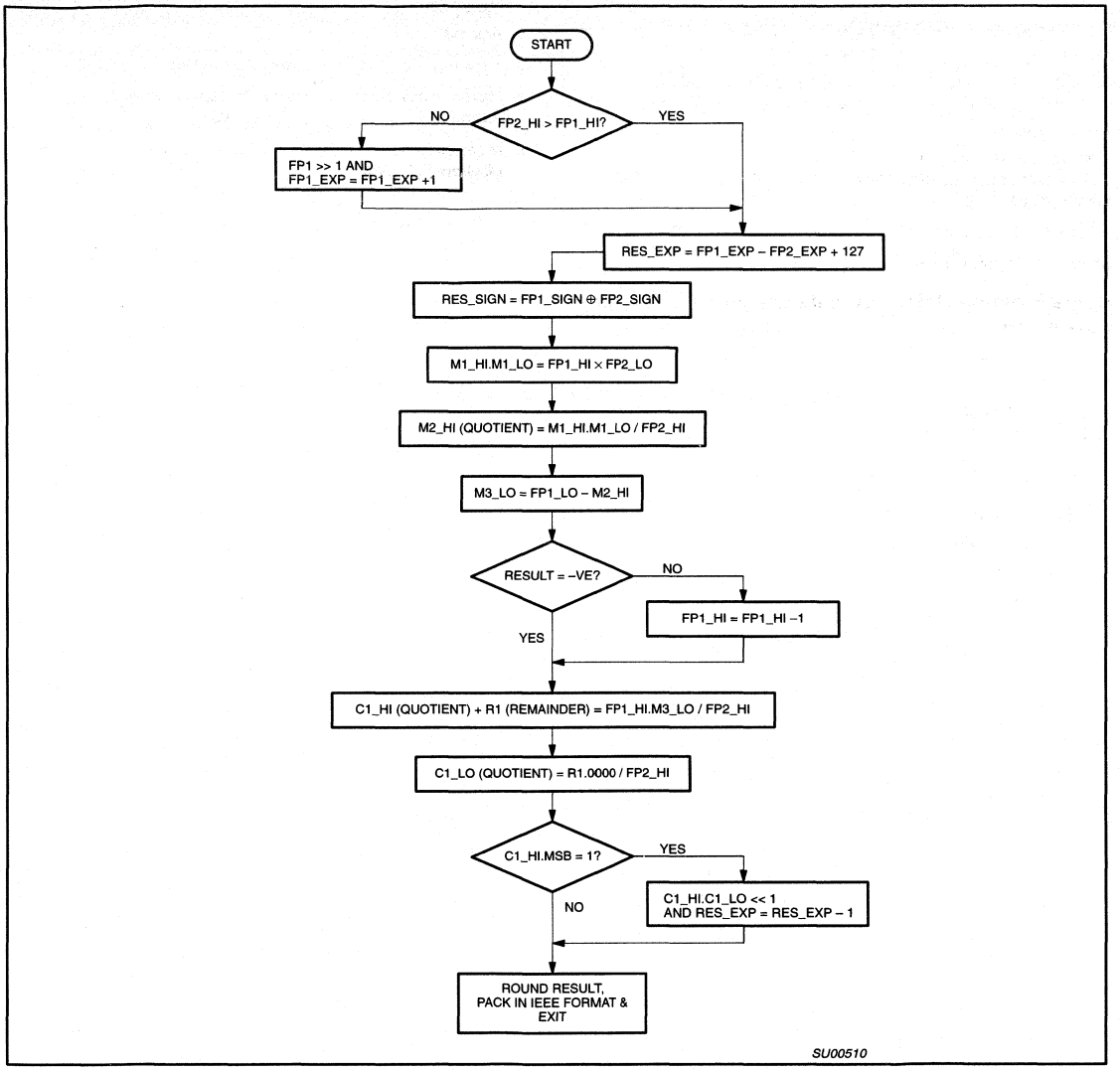
FPADD AND FPSUB



FPMULT


START
RES_EXP = FP1_EXP + FP2_EXP – 127
RES_SIGN = FP1_SIGN ⊕ FP2_SIGN
C1_HI.C1_LO = FP1_HI * FP2_HI
C0_HI.C0_LO = FP1_HI * FP2_LO
C2_LO = C0_HI + C1_LO
RESULT > 16 BITS?
NO
YES
C1_HI = C1_HI + 1
C3_HI.C3_LO = FP1_LO * FP2_HI
C4_LO = C3_HI + C2_LO
RESULT > 16 BITS?
YES
C1_HI = C1_HI + 1
NO
C1_HI.MSB = 1?
NO
YES
C1_HI.C4_LO << 1
NORMALIZE MANTISSA
RES_EXP = RES_EXP + 1
ROUND RESULT,
PACK IN IEEE FORMAT &
EXIT
SU00509

FPDIV


START
FP2_HI > FP1_HI?
NO
YES
FP1 >> 1 AND
FP1_EXP = FP1_EXP +1
RES_EXP = FP1_EXP – FP2_EXP + 127
RES_SIGN = FP1_SIGN ⊕ FP2_SIGN
M1_HI.M1_LO = FP1_HI × FP2_LO
M2_HI (QUOTIENT) = M1_HI.M1_LO / FP2_HI
M3_LO = FP1_LO – M2_HI
RESULT = –VE?
NO
FP1_HI = FP1_HI –1
YES
C1_HI (QUOTIENT) + R1 (REMAINDER) = FP1_HI.M3_LO / FP2_HI
C1_LO (QUOTIENT) = R1.0000 / FP2_HI
C1_HI.MSB = 1?
YES
C1_HI.C1_LO << 1
AND RES_EXP = RES_EXP – 1
NO
ROUND RESULT,
PACK IN IEEE FORMAT &
EXIT
SU00510

APPENDIX A

Conversion of Floating Point Numbers

In general IEEE FP = +– mantissa X 2 ^{exponent}

Format = Sign bit (1). Biased Exponent bits (8). Normalized Mantissa bits (23), e.g., convert 1.0 to a 32–bit IEEE Floating Point:

1 = 1.0 X 2^0;
Biased Exponent = 0 + 127 = 127 = 7F Hex
The 24-bit normalized mantissa = 10000000000000000000000;
Sign = positive = 0;

So, IEEE 1.0 = 0 0111 1111 00000000000000000000000

which is 3f80 0000 hex & so on.

Some Floating Point Numbers are given below for user reference:

–1.0 = BF80 0000;
+1.0 = 3F80 0000

–0.25 = BE80 0000;
+0.25 = 3E80 0000

–0.50 = BF00 0000;
+0.50 = 3F00 0000

6.250 = 40C8 0000;
1.625 = 3FD0 0000;

12.0 = 4140 0000

References

1. IEEE Draft 8.0 on *A Proposed Standard for Binary Floating-point Arithmetic, 1981*
2. K.Hwang, *Computer Arithmetic, John–Wiley and Sons, 1979*
3. *Microprocessor System Design Concepts, Nikitas A. Alexandridis, Computer Sc.Press, 1984*
4. *Microprocessors and Digital Systems, Douglas V. Hall, McGraw-Hill, 1980*

APPENDIX B

```
#include <stdio.h>
#include "fpp.h"



void main(void)

{

unsigned int fprtn;
unsigned short j;
FILE *fp;


start:
       while(1)       // clear RAM
 {
                  for(j=0; j<6;j++)
                  {
                      tmp1[j] = 0;
                      tmp2[j] = 0;
                  }


                  // Menu

                  printf("\n\n\n\n");
                  printf("              XA FLOATING POINT ARITHMETIC FUNCTION MENU\n");
                  printf("             ------------------------------------------\n");
                  printf("\n");
                  printf("               A .............Floating Point Add\n");
                  printf("               B .............Floating Point Subtract\n");
                  printf("               C .............Floating Point Multiply\n");
                  printf("               D .............Floating Point Divide\n");
                  printf("               H .............Help File\n");
                  printf("               J .............Error Register Status\n");
                  printf("               Q .............Exit Menu\n\n\n\n\n");
                  printf("               ==>");


                  fprtn = getche();   /* wait for user i/p */

                  getch();   // wait for <CR>
                  printf("\n\n");


                  /* Select Floating Point Routine */

                  switch(fprtn)
                  {
                      case 'A' :
                      case 'a' :
                                    printf("Floating Point Addition in Progrss....\n\n\n");
                                    getnum();
                                    fpadd();
                                    break;
```

```
                case 'B' :
                case 'b' :
                        printf("Floating Point Subtraction in Progrss....\n\n\n");
                        printf("\n\n\n\n");
                        getnum();
                        fpsub();
                        break;

                case 'C':
                case 'c':
                        printf("Floating Point Multiplication in Progress....\n\n\n");
                        printf("\n\n\n\n");
                        getnum();
                        fpmult();
                        break;



                case 'D':
                case 'd':
                        printf("Floating Point Divide in Progress....\n\n\n");
                        printf("\n\n\n\n");
                        getnum();
                        fpdiv();
                        break;

                case 'H':
                case 'h':
                        if( (fp = fopen("help","r")) == NULL)
                        printf("can't open flpdat for read\n");

                        fcopy(fp,stdout);
                        fclose(fp);
                        printf("Hit any key to continue ...");
                        getch();
                        goto start;


                case 'J':
                case 'j':
                        show_err_reg();
                        printf("Hit any key to continue ...");
                        getch();
                        goto start;
                case 'Q':
                case 'q':
                        printf("******Hit Ctrl+C to exit ....!!!!*******\n");
                        getch();


                default:
                        printf("   ************************************************\n");
                        printf("   *  UNKNOWN COMMAND, GOODBYE!!                  *\n");
                        printf("   ************************************************\n");
                        goto start;

  }

 }

}
```

```
/* Exception Handling Routines */

void divbyz (void)

{
                ERRFLG |= 8;            /* set the DIVBY0 bit (#3) */
                fp_lo = 0;                          /* Return  NaN */
                fp_hi = NANH;
                fptrap();               /*  exception handler */
                return;
}


/* Illegal Operand - one of the numbers is a INVALID. */

void fnan(void)

{

                ERRFLG |= 4;            /* set the NAN operand bit */
                fp_lo = NANL;
                fp_hi = NANH;           /* return NAN in fp_lo and fp_hi */
                fptrap();
                return;
}



void undflw(void)               /* exponent underflow */

{
                ERRFLG |= 2;            /* set the exponent underflow bit */
                fp_lo = 0;              /* clear FP */
                fp_hi = 0;
                fptrap();
                return;
}

void ovflw(void)                /* exponent overflow */

{
                ERRFLG |= 1;            /* set the exponent overflow bit */
                fp_lo = 0;
                fp_hi = NANH;
                fptrap();
                return;
}


/* Subroutine to check if a single precision FLP # stored in the
   IEEE flp format in registers fp_lo and fp_hi is 'INVALID'
   Returns 0 if number != INVALID and
   Returns 1 if Number == INVALID
*/

unsigned int  fnanchk()

/* Subroutine to check if a SP floating point number is a NaN
   Returns 1, if YES, and 0 if NOT */

{
                long tmp;

                tmp = fp_hi;
                tmp = tmp >> 1;         /* Shift left fp_hi by 1 */
                if(tmp > 0xfeff)        /* feff + 1 = ff00 */
                return 1;               /* biased exp >= 255 & f =/= 0 */
                return                  /* OK */
```

```
}
/* Subroutine to check if a single precision FLP # stored in the
   IEEE flp format in registers fp_lo and fp_hi is 'ZERO'
   Returns 0 if number != and
   Returns 1 if Number == 0
   Note : fp "0" = 1.0 x 2 e -127  i.e if biased exp = 0, fp = 0
*/

char zchk(void)

{
                unsigned long tmp;
                tmp = fp_hi;

                tmp = tmp << 1;         /* Shift left fp_hi by 1 */

                if(tmp > 0x00ff)
                return 0;
                return 1;
}


/* subroutine to unpack a SP IEEE formatted FP # and held in regs.fp_hi
   and fp_lo.  The unpacked format occupies 3 words & is organized as
   follows:

   WORD2 :  eeeeeeee sssssss      -> biased-exp.sign
   WORD1 :  1mmmmmmm mmmmmmmm     -> 16 MSB of Mantissa (m23 : m16)
   WORD0 :  mmmmmmmm 00000000     -> 8 LSB of Mantissa.zeros

   e7:0 - 8-bit exponent in excess-127 format
   s7:0 - sign bit -> 0x00 = +ve, 0xff = -ve ;
   m23:0 - normalized mantissa i.e 1.0000...
*/

char* funpak(fparray)                   // returns ptr. to a character array to fpadd
int *fparray;                           //pointer to the flp. array passed by fpadd

{

                char  sign;
                unsigned short i= 0,j = 0;

                     if(k=2) k=0;
                     else
                     k ++;
                     flpar[i] = 0;   // clear lo-byte of fp0


                flpar[++i] = fparray[j] & 0x00ff; /* fp0_hi = m0:m7 */
                flpar[++i] = (fparray[j++] & 0xff00) >> 8; /* fp1_lo = m8:m15 */


                flpar[++i] = fparray[j] & 0x007f; /* m16:m22 */


                flpar[i] |= 0x80;                 /* set bit7 -> normlz. bit */


                sign = (fparray[j] & 0x8000) >> 15; /* check sign bit */
                                                    /* fp2_lo = 8 sign bits */

                  if (sign)                         /*  -ve number ? */

                    flpar[++i] = 0xff;
                  else                              /* yes */
                    flpar[++i] = 0x00;              /* no */

      flpar[++i] = (fparray[j] & 0x7f80) >> 7;      /**/
                return (flpar);                     /* return the ptr to the array */
}
```

```
/* subroutine to pack a SP held in 3 words flpar0:3 into IEEE format.
   The packed format is stored in fp_hi & fp_lo as follows:
   fp_hi :  seeeeeee  emmmmmmm      -> sign.biased-exp.7 MSBs of mantissa
   fp_lo :  mmmmmmmm mmmmmmmm       -> 16 LSBs of Mantissa

               Result Stored in tmp2[i]; i=0:5
               tmp2[0] = 0x00; tmp[1] = m0:m7; tmp[2] = m8:m15;
               tmp[3] = 1.m16:m22; tmp[4] = sign7:0; tmp[5] = exp7:0
*/


void fpak(void)

{
               int i=1;
               unsigned int sign,tmpc;
               long ltmp1,ltmp2;


               fp_lo =  tmp2[i++];                  /* get m0:m7 */
               tmpc = tmp2[i++];
               fp_lo = fp_lo | (tmpc << 8);         /* get m8:m15 */


               fp_hi = tmp2[i++];                   /* get 1.m22:m16 */
               ltmp1 = (fp_hi & 0x007f);
               ltmp1 = ltmp1 << 16;                 /* mask & shift */


               sign = tmp2[i++];                    /* save sign-bit(s) */
               ltmp2 = tmp2[i];                     /* exponent */
               ltmp2 = ltmp2 << 23;
               ltmp2 = ltmp2 | ltmp1;

               if(sign)
               ltmp2 = ltmp2 | 0x80000000;

               printf("\n\nThe IEEE packed result is:");
               printf("%lx\n",ltmp2);
}


/* This routine rounds up the 32-bit mantissa resulted from FP operations
   to a 24-bit number */

void fround(void)
{

    unsigned int i=3, ctmp;
    long tmpl;

               tmpl = tmp2[i--];    /* 1.m22:16 */
               tmpl = tmpl << 8;
               ctmp = tmp2[i--];    /* m15:8 */
               tmpl = tmpl|ctmp;
               tmpl = tmpl << 8;
               ctmp = tmp2[i--];    /* m7:0 */
               ctmp = tmp2[i];

                   if(ctmp & 0x80)   /* if bit 7 i not set in the LSB */

                               tmpl += 0x0100;  /* Increase next byte by 1 */

                               ctmp = tmp2[5];

                                 if(tmpl & 0x1000000)    /* carry out of MSB? */
                                 {
                                                         tmpl = tmpl >> 1;
                                                         ctmp++;           /* exp = exp+1 */

                                                           if(ctmp > 255)

                                                                     tmp2[5] = 0xff;
```

```
                                        }

                                                        tmp2[1] = (tmp1 & 0x000000ff);
                                                        tmp2[2] = (tmp1 & 0x0000ff00) >> 8;
                                                        tmp2[3] = (tmp1 & 0x00ff0000) >> 16;
}


/* User supplied FLP Trap Routine */

void fptrap(void)

{
                printf("\n\nException Occurance !!!\n\n");

                printf("Error Flag Register = %x",ERRFLG);
                /* User exception handler

                ....
                ....
                ....
                ....
                ....
                ....
                ....
                */

}

void show_err_reg(void)

{
    printf("\n The Error Flag Register Bit Map is as follows :");
    printf("\n ---------------------------------------\n\n");
    printf("        ---------------------------------------\n");
    printf("       | - | - | - | - | DBZ | IOP | EUF | EOV |\n");
    printf("        ---------------------------------------\n\n\n");
    printf(" where DBZ = Divide by Zero exception\n");
    printf("       IOP = NaN or Invlaid operand\n");
    printf("       EUF = Exponent Underflow\n");
    printf(" and   EOV = Exponent Overflow\n\n\n");


    printf("The status of the register after the operation is :");
    printf("0x%x\n\n",ERRFLG);
 }
/* FPADD - Floating Point Add
   It is assumed two  floating point numbers F1 & F2 are in IEEE format
   Format :
   Fn = fpnh (s.e7:0.m22:16) + fpnl (m15:0)  -> In registers
*/


int fpadd()

{               // 1

long dmant, flpm1, flpm2, lnfp, tlong;
int  *flpn, temp1, temp2;
char dexp, i=0,j=0,k=5,t=0, exp1, exp2;

                exp1 = tmp1[k];   /* get exponent of 1st */

                exp2 = tmp2[k];   /* get exponent of 2nd */

                dexp = (exp1 - exp2);           /* difference in exponent */

                printf("\n\n");
                printf ("DEXP =  %x\n", dexp);
```

```
/* CASE 1: */
                        if(dexp > 0 && dexp < 23)      /* f1exp > f2exp */
  {  //2


                        tmp = dexp;

                /* if exp > 23, can't shift mantissa
                                                        more than 23-bits */


                        while(tmp--)
                        {
                          for(i=0; i<=3; i++)
                          {
                          tmp2[i] = tmp2[i] >> 1;
                          }
                         }



                         i = 4;
                         temp1 = tmp1[i] & 0x00ff;  /* get the fp1 sign byte */
                         printf("\nFP1 SIGN BITS = %x", temp1);

                         temp2 = tmp2[i] & 0x00ff;  /* get the fp2 sign byte */
                         printf("\nFP2 SIGN BITS = %x", temp2);



loop_here:
                         i = 4;
                         rsign = (tmp1[i] ^ tmp2[i]);  /* ex-or sign bits */

                         if(rsign != 0)                /* if different sign */

                 { //44
                         printf("\nFPs ARE OF DIFFERENT SIGNS!!\n");

                         flpm1 = getmant(0);
                         flpm2 = getmant(1);

                         dmant = flpm1 - flpm2;

                         if (flpm1 > flpm2)            /* F1man > F2man */

                {  //22

                         printf("\n");
                         printf("FP1 MANTISSA GREATER THAN FP2 MNATISSA\n");
                         printf("\n\n");

                         tmp2[4] = tmp1[4];           /* result sign = sign of F1 */
                         rsign = tmp1[4];             /* stored in FP2 sign byte */


                 /* Shift left mantissa till MSB = 1 */

                        for(k=0; k <= 23  && ((dmant & 0x00800000) == 0); k++)
                             {
                                                  dmant = dmant << 1;
                                                  exp1 = exp1 - 1;
                             }


                        /* save result in tmp2[i] array */
                                         tmp2[5] = exp1;
                                         tmp2[4] = rsign;
                                         tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                         tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                         tmp2[1] = (dmant & 0x000000ff);
```

```
                printf("RESULT EXPONENT : %x\n", exp1);
                printf("RESULT MANTISSA (NORMLZD) = %lx\n", dmant);
                printf("RESULT SIGN = %x\n", rsign & 0x1);

                                froundl();   /* round the result */
                                        fpak();                    /* Pak & leave */

                                        printf("Hit any key to continue ...");
                                        getch();
                                        return 0;

        } //22
                else if (flpm1<flpm2)              /* F2man > F1man */

        { //23
                                printf(" FP2 MANTISSA GREATER THAN FP1 MANTISSA \n");


                                dmant = ~dmant; /* 2'S COMPLEMENT */
                                dmant++;
                                exp2 = tmp2[5];     /* res.exp = F2.exp */
                                rsign = tmp2[4];
                                tlong = dmant;

                while(!(tlong & 0x800000))
                                { dmant = dmant << 1;
                                        tlong = dmant;
                                        exp2--;
                                }

                               dmant = dmant & 0xffffff;



                             /* save result in tmp2[i] array */
                                tmp2[5] = rexp;
                                tmp2[4] = rsign;
                                tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                tmp2[1] = (dmant & 0x000000ff);

                             printf("\n\n");
                             printf("THE RESULT MANTISSA (NORMLZD) IS = %lx\n",dmant);
                             printf("THE RESULT EXPONENT IS = %x\n", exp2);
                             printf("THE RESULT SIGN IS = %x\n", rsign & 0x1);

                                  froundl();
                                  fpak();
                                  printf("Hit any key to continue ...");
                                  getch();
                                                                  return(0);
        } //23



} // 44

        else if(rsign == 0) // same sign, so ex-OR is 0
```

```
    { //55
                        printf("\n");
                        printf("FPs ARE OF SAME SIGN!!\n");

                                    flpm1 = getmant(0);
                                    flpm2 = getmant(1);
                                    dmant = flpm1 + flpm2;
                                    rsign = tmp1[4];
                                    rexp = exp1;

                                    tlong = dmant;
                                    tlong = tlong & 0x01000000;// check if carry set


                                    if(tlong)
                                  {
                                  dmant = dmant >> 1;
                                  rexp = rexp + 1;
                                  }

                                  /* save result in tmp2[i] array */
                                    tmp2[5] = rexp;
                                    tmp2[4] = rsign;
                                    tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                    tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                    tmp2[1] = (dmant & 0x0000ff);


                                  printf("\n\n");
                                  printf("THE RESULT MANTISSA(NORMLZD) IS = %lx\n",dmant);
                                  printf("THE RESULT EXPONENT IS = %x\n", rexp);
                                  printf("THE RESULT SIGN IS = %x\n", rsign & 1);


                                    fround();
                                    fpak();       /* pack in IEEE format */

                        printf("Hit any key to continue ...");
                                  getch();

                                    return(0);
    } //55


} // 2



                        else if (dexp > 23)

{ //99
                                  printf("DIFFERENCE IN EXPONENT IS  GREATER THAN 23\n");

                                  for(i = 0; i<= 3; i++)
                                  tmp2[i] = 0;
                                  goto loop_here;
} //99



    else if (dexp < 0)
```

```
{//6
                printf("DIFFERENCE IN EXPONENT IS NEGETIVE\n");
                printf("FP2 EXPONENT GREATER THAN  FP1 EXPONENT\n");

                tmp = ~dexp ;       /* 2's complement */
                tmp++;

        printf("2's COMPLEMENT OF DEXP = %d\n", tmp);


                                    if(tmp < 23)
     {
                                    while(tmp--)
                                    {
                                      for(i = 0; i<= 3; i++)
                                      {
                                      tmp1[i] = tmp1[i] >> 1;
                                      }
                  }
                     goto loop_here;
     }

                                      else if(tmp > 23)                 /* dexp > 23 */
    {
                                      for(i=0; i<= 3; i++)
                                      tmp1[i] = 0;                      /* F1mant = 0 */
                                      goto loop_here;
    }

  } //6
                    else if(dexp == 0)                                  /* shift done */
                      printf("FPs GOT SAME EXPONENT!!\n");
                    goto loop_here;

} // 1


/* Get the mantissa for both FP in 24-bit format */


long getmant(val)
 unsigned short val;

{
    long lnfp, flpm;
    unsigned short i;
                                    i= 1;
                                    lnfp = 0;
                                    flpm = 0;

    if (!val)
                    {

                                    lnfp = tmp1[i];
                                    flpm |= (lnfp & 0x000000ff);

                                    lnfp = 0;
                                    i++;
                                    lnfp = tmp1[i];
                                    lnfp = lnfp << 8;
                                    flpm |= lnfp & 0x0000ff00;


                                    lnfp = 0;
                                    i++;
                                    lnfp = tmp1[i];
                                    lnfp = (lnfp << 16);
                                    flpm |= (lnfp & 0x00ff0000);


                                    flpm = flpm & 0x00ffffff;

                                    printf("\n FP1 MANTISSA = %lx\n", flpm);

                    }
```

```
                else
                   {
                                    i=1;
                                    flpm = 0;
                                    lnfp = 0;
                                    lnfp = tmp2[i++];
                                    flpm |= (lnfp & 0x000000ff);

                                    lnfp = 0;
                                    lnfp = tmp2[i++];
                                    lnfp = (lnfp << 8);
                                    flpm |= (lnfp & 0x0000ff00);

                                    lnfp = 0;
                                    lnfp = tmp2[i];
                                    lnfp = (lnfp << 16);
                                    flpm |= (lnfp & 0x00ff0000);

                                    flpm = (flpm & 0x00ffffff);

                                    printf(" FP2 MANTISSA = %lx\n", flpm);

                   }
      return (flpm);
}


int fpsub()
{
   unsigned char fp2_sign;

   fp2_sign = tmp2[4];
   fp2_sign = ~fp2_sign;

   tmp2[4] = fp2_sign;
   fpadd();
}


void getnum()

{
unsigned int fp[3], px;
unsigned short i,j,k;

printf("Type the lo-word of FP#1 in IEEE format :");
                scanf("%x",&px);
                i= 0;
                fp[i] = *&px;

                printf("Type the hi-word of FP#1 in IEEE format :");
                scanf("%x",&px);
                fp[++i] = *&px;


                funpak(fp);  // pass the array ptr. to unpack routine

                for(k=0,j=0; k<=5; j++,k++)
                {
                tmp1[k] = flpar[j];
                }
                printf("\n");


                i=0;
                printf("\n\n");
                printf("Type the lo-word of FP#2 in IEEE format :");
                scanf("%x",&px);
                fp[i] = *&px;
```

```
                printf("Type the hi-word of FP#2 in IEEE format :");
                scanf("%x",&px);
                fp[++i] = *&px;
                funpak(fp);

                for(k=0,j=0; k<=5; j++,k++)
                {
                tmp2[k] = flpar[j];
                }

}

void fpmult()
{

    int fp1_exp,fp2_exp, res_sign;
    unsigned int fp1_hi, fp1_lo, fp2_hi, fp2_lo, tmp;
    unsigned int c0_hi,c0_lo,c1_hi, c1_lo, c2_lo,c3_hi, c3_lo, c4_lo;
    int res_exp;
    long ltmp;


    fp1_exp = tmp1[5];
    fp2_exp = tmp2[5];

                res_exp = (long)(fp1_exp + fp2_exp -127); // result exponent


                                if(res_exp < 0)      // underflow
                                undflw();


                                if (res_exp > 255)
                                ovflw();


                res_sign = tmp1[4] ^ tmp2[4];  // XOR = result sign


                 tmp = tmp1[3] << 8;    /* 1.m22:8 = FP_HI*/
                 tmp = tmp & 0xff00;
                 tmp |= tmp1[2];

                 fp1_hi = tmp;
                 fp_hi = fp1_hi;

                 tmp = fnanchk();

                                if(tmp)
                                fnan();        // F1 is a NaN


                 tmp = tmp2[3] << 8;    /* 1.m22:8 = FP_LO */
                 tmp = tmp & 0xff00;
                 tmp |= tmp2[2];

                 fp2_hi = tmp;
                 fp_hi = fp2_hi;

                                tmp = fnanchk();

                                if(tmp)
                                fnan();        // F2 is a NaN


                 tmp = zchk();  // Check for F2 = 0

                                if(tmp)  {
                                printf("\nResult is = 0 as Multiplier is 0\n");    // F2 = 0
                                goto endprog; }

                 fp_hi = fp1_hi;
```

```
tmp = zchk();  // Check for F1 = 0

                if(tmp) {
                printf("\nResult is = 0 as Multiplicand is 0\n");    // F1 = 0
                goto endprog;  }


tmp = tmp1[1] << 8;    /* m7:0.0000 */
tmp = tmp & 0xff00;
tmp |= tmp1[0];

fp1_lo = tmp;

tmp = tmp2[1] << 8;
tmp = tmp & 0xff00;
tmp |= tmp2[0];

fp2_lo = tmp;

ltmp =  (long)fp1_hi * (long)fp2_hi;  /* FP1_HI * FP2_HI */


c1_hi = (ltmp &  0xffff0000) >> 16;
c1_lo = ltmp & 0x0000ffff;

ltmp = (long)fp1_hi * (long)fp2_lo;
c0_hi = (ltmp  & 0xffff0000) >> 16;
c0_lo = ltmp &0x0000ffff;

ltmp  = c0_hi + c1_lo;

              if(ltmp & 0x10000)
                  c1_hi++;

c2_lo = ltmp & 0xffff;


ltmp = (long)fp1_lo * (long)fp2_hi;

c3_hi = (ltmp & 0xffff0000) >> 16;
c3_lo = ltmp & 0x0000ffff;

ltmp = c3_hi + c2_lo;

             if(ltmp & 0x10000)
                 c1_hi++;


c4_lo = ltmp & 0xffff;


ltmp = c1_hi;
ltmp = ltmp <<8;
ltmp = (ltmp & 0xffffff00) | c4_lo;

if(!(c1_hi & 0x8000))
ltmp = ltmp << 1;

else
res_exp++;

if(res_exp > 254)
ovflw();


/* save result in tmp2[i] array */
                tmp2[5] = res_exp;
                tmp2[4] = res_sign;
                tmp2[3] = (ltmp & 0x00ff0000) >> 16;
                tmp2[2] = (ltmp & 0x0000ff00) >> 8;
                tmp2[1] = (ltmp & 0x000000ff);
```

```
                printf("\n\n");
                printf("RESULT EXPONENT : %x\n", res_exp);
                printf("RESULT MANTISSA (NORMLZD) = %lx\n", ltmp);
                printf("RESULT SIGN = %x\n", res_sign & 1);

                found();


              /* final check of exponent */

                tmp = tmp2[5];

                        if(!tmp)
                        undflw();  /* exponent underflow */

                        if (tmp > 254)
                        ovflw();

                        fpak();
endprog:
                printf("Hit any key to continue ...");
                        getch();
}


void fpdiv(void)

{
    unsigned int tmp, fp1_hi, fp2_hi, fp1_lo, fp2_lo, c1_hi, c1_lo;
    unsigned char i, c1_exp, c1_sign, fp1_sign, fp2_sign;
    long ltmp, rem, lfpm1, lfpm2, tmplng;
    unsigned int m1_hi, m1_lo, m2_hi, m2_lo, m3_hi, m3_lo;
    signed int fp1_exp, fp2_exp, dexp;

                tmp = tmp1[3] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp1[2];

                fp1_hi = tmp;
                fp_hi = fp1_hi;
                tmp = fnanchk();

                        if(tmp)
                        fnan();         // F1 is a NaN

                tmp = tmp2[3] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp2[2];

                fp2_hi = tmp;
                fp_hi = fp2_hi;

                tmp = fnanchk();

                        if(tmp)
                        fnan();         // F2 is a NaN


                        tmp = zchk();   // Check for F2 = 0

                        if(tmp)  {
                        divbyz();       // F2 = 0
                        printf("\nException as a result of Division by 0\n");
                        goto exit;
                        }

                        fp_hi = fp1_hi;
                        tmp = zchk();   // Check for F1 = 0

                        if(tmp)         // Result = 0
                        {
                        printf("\nResult of Division of a zero Dividend is = 0\n");
                        goto exit;
                        }
```

```
                tmp = tmp1[1] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp1[0];

                fp1_lo = tmp;


                tmp = tmp2[1] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp2[0];

                fp2_lo = tmp;


                lfpm1 = fp1_hi;
                lfpm1 = lfpm1 << 16;
                lfpm1 |= fp1_lo;

                lfpm2 = fp2_hi;
                lfpm2 = lfpm2 << 16;
                lfpm2 |= fp2_lo;


/* Exponent bits */
                fp1_exp = tmp1[5];

                fp2_exp = tmp2[5];


/* Sign bits */
                fp1_sign = tmp1[4];
                fp2_sign = tmp2[4];


    /* Ensure that fp2_hi > fp1_hi */


    if(fp1_hi > fp2_hi)          // compare fp1_hi & fp2_hi

                {
                lfpm1 >> 1;
                fp1_exp++;
                }

   /* else := good */


     fp1_hi = (lfpm1 & 0xffff0000) >> 16;
     fp1_lo = lfpm1 & 0x0000ffff;


     dexp = (fp1_exp - fp2_exp);  // difference in exponent (2's compl)


     c1_exp = dexp + 127;
     c1_sign = fp1_sign + fp2_sign;

     ltmp = (long)fp1_hi*(long)fp2_lo;  // m1_hi.m1_lo

     m1_hi = (ltmp & 0xffff0000) >> 15;
     m1_lo = (ltmp & 0x0000ffff);


     m2_hi = ltmp / (long)fp2_hi;   // Quotient
     m3_lo = fp1_lo - m2_hi;

     if( m2_hi > fp1_lo)    // B = 0 i.e C = 1
                fp1_hi--;
                ltmp = (unsigned long)fp1_hi;
                ltmp = ltmp << 16;
                ltmp = ltmp|m3_lo;
                tmplng = ltmp;
```

```
                ltmp = ltmp /(unsigned long)fp2_hi;  // Quotient
                c1_hi = ltmp & 0x0000ffff;
                ltmp = tmplng;
                ltmp =  ltmp % (unsigned long)fp2_hi; // remainder


                ltmp = ltmp << 16;
                ltmp = ltmp & 0xffff0000;
                c1_lo = ltmp/(unsigned long)fp2_hi;


                if(c1_hi & 0x8000)

                {
                   c1_hi << 1;
                   c1_lo << 1;
                   c1_exp -= 1;
                }


                   ltmp = c1_hi;
                   ltmp = ltmp << 16;
                   ltmp = ltmp | c1_lo;


                   /* save result in tmp2[i] array */
                                    tmp2[5] = c1_exp;
                                    tmp2[4] = c1_sign;
                                    tmp2[3] = (ltmp & 0xff000000) >> 24;
                                    tmp2[2] = (ltmp & 0x00ff0000) >> 16;
                                    tmp2[1] = (ltmp & 0x0000ff00) >> 8;
                                    tmp2[0] = ltmp & 0x000000ff;


                   printf("\n\n");
                   printf("RESULT EXPONENT : %x\n", c1_exp);
                   printf("RESULT MANTISSA (NORMLZD) = %lx\n", ltmp);
                   printf("RESULT SIGN = %x\n", c1_sign & 1);


                                    fround();   // round up results in IEEE format

 /* final check of exponent */

                   tmp = tmp2[5];

                                    if(!tmp)
                                    undflw();  /* exponent underflow */

                                    else if (tmp > 0xfe)
                                    ovflw;     /* exponent overflow */


                                      fpak();  // pack in IEEE format
exit:
                   printf("Hit any key to continue ...");
                                      getch();

}

void fcopy(FILE *ifp, FILE *ofp)
{
    int c;
    while ((c=getc(ifp)) != EOF)
                putc(c,ofp);
}
```

High level language support in XA AN702

Author: Santanu Roy, Philips Semiconductors, MCO Applications Group, Sunnyvale, California

Introduction

High Level Language (HLL) support is becoming a key feature in modern day microcontroller architecture. The reason is highly visible. It is easier to code a processor in a high-level platform than in conventional assembly because it is portable, i.e., it is not tied to any one machine. Also, the advantage of coding in a high-level language is because it is modular and re-usable which speeds up any code development process considerably.

In recent years, C has been "the language" of choice for all engineers. Thus almost all modern day microcontrollers are designed with C-language support in mind. This article highlights some of the architectural features of Philips XA microcontroller that has been designed to support such languages specifically C.

Supporting HLL

One of the tasks that an architect has to confront is the determination of exactly what instructions should form the functional instruction of a microcontroller to meet high-level language support. An answer to this is to provide an operation code for each functional operation in a high-level programming language. Thus operation codes will exist for +, –, *, /, and so on. Special provision is made for operation on arrays, and all operations that can be applied to data types in a high-level language are directly supported in the architecture. An instruction set ideally should contain only instructions that are used in a HLL, and not implement any non-functional instructions, i.e., instruction that is not expressed as a verb or operator in a high-level language. Thus "LOAD", "STORE", and so on which are not statements made in high-level languages are redundant and only adds to architectural overheads.

An instruction word consists of a single op-code and an operand address for each HLL variable involved in the operation. *Op-codes are symmetric in that they are applicable to any type of addressing and any data type.*

Some general criteria for an ideal architecture could be:

1. Only one instruction should be executed for most common HLL operators.
2. There should be only one memory reference for each referenced operand.
3. There should be explicit addressing only for operands whose location cannot be inferred by recent processing activity, and address should be short.
4. Instructions should be compact, and densely coded.

The XA Architecture

The XA is a register based machine. Hence most variables could be stored in these fast storage registers for high code density and fast execution. However, the beauty of the XA architecture is that, it is optimized for internal memory as well for high throughput and code density, e.g., a register-register ALU operation takes 2 bytes and 3 clocks and the same ALU operation between register-memory (indirectly addressed) is 2 bytes and 4 clocks. So, a large set of variables could be stored in memory with very little loss in performance. Additionally, hooks like "burst mode", etc., are provided to speed up external memory access as well.

Data Types and Sizes

XA directly supports the following basic data types as used in C:

character (char) – signed and unsigned bytes
integer (int) – signed and unsigned words

Constants – Supported as byte/word (char/int) immediate data in the instructions, e.g., ADD R0, #1234 etc. The range is +32,767 to –32,768 for signed and 0 to 65,536 for unsigned word/integer constants, +127 to –128 for signed or 0 to 255 for unsigned bytes/char.

For "short" qualifier, the range is +7 to –8 as used with instructions MOVS and ADDS.

A "long" qualifier to integer is implemented by the compiler by extending (signed/unsigned) the word to the next higher address(+1). In addition to the above,

Bit – This special data type is also supported to access the different bit addressable space in the machine.

Note: All signed data are represented in 2's complement form in the XA.

Type conversion

All operations are performed under natural data sizes, e.g., MULU.b does a 8x8 unsigned multiply of 2 bytes, MULU.w does the same but with 2 word-size operands. So when operands of different types appear in an expression, they are converted to a common type by the compiler, e.g., operation between a *char* (byte) and an *integer* (word) is promoted to *integer-integer*, etc.

Arrays

XA supports addressing byte and word arrays in memory as required by C or any HLL. Offset and auto-increment addressing modes in XA allow easy access and manipulation of array elements. Offsets are signed values of 8 or 16 bits and are used depending on the size of the array.

Static Variables

Static variables unlike automatic provide permanent storage in a function. This means these variables are stored in memory rather than being a part of run-time stack. A wide variety of memory addressing modes are supported in the XA to provide easy access to static variables in memory. In addition to several indirect addressing modes (auto-increment offset) the XA supports direct access to the first 1K of the memory space in each segment. This is ideal for addressing static variables, and has found to generate extremely dense code. A listing of operations to access static variables is given below for reference:

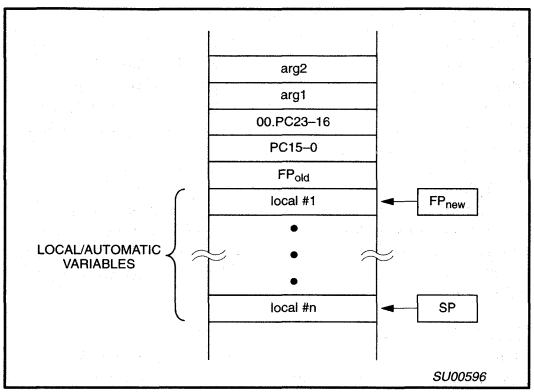
Table 1. Access to Static Variables

ADDRESSING MODES
Rd, direct
direct, Rd
Rd, [Rs+]
[Rs+], Rd
Rd, [Rs+]
Rd, [Rs+Offset8/16]
[Rs+Offset8/16], Rd
direct, direct
[Rd], #immediate
direct, #immediate
[Rd+], #immediate
[Rd+], [Rs+]

Automatic/Dynamic Variables

Within a function, a typical compiler maintains a Frame Pointer (FP), which is used to access function arguments and local automatic variables. To call a function, a compiler pushes arguments onto the stack in reverse order, (the PUSH instruction decrements the SP by 2 each time it is executed, calls the function, then increments the SP by the number of bytes pushed. For instance, to call a function with two one-word arguments, the XA-compiler generates code to do the following:

```
PUSH arg2          ; (SP –=2)
PUSH arg1          ; (SP –=2)
CALL (subroutine)  ;
ADDS SP,4          ; (SP +=4)
```



The CALL instruction pushes the current PC onto the stack. Because all stack pushes are 16-bits in XA, any 8-bit function argument is automatically promoted to word.

Upon function entry, the compiler creates new stack and frame pointers by computing:

```
PUSH FP (old)
FP (new) = SP
SP = SP – Framesize;
```

where "Framesize" is the space required for all local automatic variables. If the frame size is odd, the compiler always rounds it up to the next even number. If there are 2 arguments and 2 local variables, then the frame size is 4 and the stack looks like this:

```
FP+8 second argument
FP+6 first argument
FP+4 return address
FP+2 old FP
FP–0 first local variables
FP–2 second local variable
FP–4 next free stack location (same as SP)
```

If a function argument is defined to be an 8-bit type, then only the lower 8-bits of the value pushed by the caller are to inside the called function.

Upon function exit, the compiler restores the SP and FP to their original value by executing the following:

```
SP = FP
POP FP
RET
```

The return instruction RET sets the new PC by popping the saved PC off the stack.

Because there are so many registers in XA (unlike 8051), any of them could be assigned to hold the FP. Access to variables in the stack space is easily achieved through the indirect-offset addressing modes (signed 8 or 16) with respect to the stack pointer. In almost all the cases the variables pushed onto the stack could be accessed using only a signed 8-bit offset present in XA. The function arguments and variables could be moved in and out of the stack in a single PUSH/POP multiple instructions permitted in XA. In fact up to 8 words or 16 bytes of such information could be moved in and out of the XA stack with one instruction, which increases code density to a large extent during procedure calls and context switching. For example, if register variables are in R1,R2,R3, and R4, a single "PUSH R1,R2,R3,R4" instruction will be generated by the XA-compiler. A corresponding function exit will have a "POP R1,R2,R3,R4" for restoring the variables.

All automatic class of variables will be allocated on run-time stack. The XA has full complement of addressing modes on SP to handle dynamic variables in the stack. Table 2 shows some of the XA addressing modes that could be used for such access.

Table 2.

ANSI-C	XA	Comments
SP–>Offset	R+Offset8/16	
*SP	[R]	
SP+	[R+]	Pop

Operators for HLL support

The structure for op-codes of an ideal architecture should be stated in terms of number of operands required and the relationship between the operands. The structure should be oriented toward efficient coding of an instruction that will support programs written in a HLL with minimum compilation. The XA instruction set is designed to handle such efficiency as reflected in Table 3. The set of instructions that supports the general/basic addressing modes are used to describe HLL support in this table.

Table 3. Mapping of XA ALU Operations to C Operators

ANSI C Operator (op)	XA Op-codes
+= , += + C()	ADD, ADDC
–= , –= – C()	SUB, SUBB
< , <= , == , >= , > , != (s/u)	CMP
&= , |= , ^=	AND , OR , XOR

Data movement in C is given by "=" which is the "MOV" instruction in XA. The MOV instruction not only has the general/basic addressing modes, it also has some additional addressing modes for C-code optimization for memory transfer operations like direct-direct, direct-indirect, indirect-autoincrement – indirect-autoincrement.

Table 4 of two operand case **A = A op B** or **B = A op B** is shown below.

Table 4.

ANSI C	XA
C-operations	**Equivalent XA-operations**
R op = R	R, R
R op= *R	R, [R]
R op= *R++	R, [R+]
R op= direct	R, direct
R op= R–>offset	R, [R+offset]
*R op= R	[R], R
*R++ op= R	[R++], R
direct op= R	direct, R
R–>offset op= R	[R+offset], R
R op= constant	R, #constant
*R op= constant	[R], #constant
*R++ op= constant	[R++], #constant
direct op= constant	direct, #constant
R–>offset op= constant	[R+offset], #constant

The three operand cases A = B op C may regularly be translated as:

```
A = B;
A op= C;
```

exception to above is

```
*R++ = B op C is equivalent to
*R = B;
*R++ op = C;
```

Typical/Frequently used C-code **A = B op C** involves operations that will fetch operands from memory, register, and as immediate data which is embedded in the instruction. The XA has the following choices for operand placements for such three operand operations.

Case 1:
If A = register,

then B and C in A = B and A op= C could have the following choices

(i) Register i.e., R = R and R op= R

(ii) Memory i.e., R = Memory and R op= Memory

where Memory = [R] , direct, [R+], [R+Offset]

(iii) Immediate i.e., R = Immediate and R op= Immediate

Case 2:
If A = Memory

where Memory = [R], direct, [R+], [R+Offset]

then B and C in A = B and A op= C could have the following choices:

(i) Register i.e., Memory = Register and Memory op= Register

(ii) Immediate i.e., Memory = Immediate and Memory op= Immediate.

(iii) Memory i.e., Memory = Memory ([R+], and direct modes only) *for B*

The above indicates that virtually all C operations involving two and three operands could be very efficiently translated in XA assembly code (in two operand cases, it is one-to-one) using a cross-compiler.

NULL DETECT/STRING TERMINATOR

Checking for "0" at the end of a string is natural in XA with the MOV instruction. The Z flag is set whenever such a condition occurs. This is especially important in string copy operations where the loop ends whenever a end of string or '\0' occurs which is reflected in the status flag "Z" in XA. The following lists such C-code and equivalent XA instructions.

```
while ((c=getch()) != '\0')      Label:   MOV [R+], memory
buffer[i++] = c;                          BNE      Label
```

Coding Relational Operations

Performing relational evaluation between two operands A and B in C-language involves fetching operands (a) in memory (b) in register or (c) an immediate value, evaluating the condition and then taking appropriate actions which typically involves a branch-if-true or branch-if-false operations

The operand(s) in memory again could be addressed as direct, indirect, indirect-autoincrement, indirect-offset, etc. The XA provides one-to-one translations of such operations.

Typically such C-statements are as follows:

```
if (A cmp_op B)                  CMP A, B
{ body} /* true */               Bxx LABEL; branch if false
                                 body
                                 LABEL:

if (A cmp_op B)                  CMP A, B
{ body 1 }                       Bxx L1 ; branch if false
else                             body 1
{ body 2 }                       JMP L2
                                 L1: body 2
                                 L2:

while (A cmp_op B)               L1: CMP A, B
{ body }                         Bxx L2 ; branch if false
                                 body
                                 JMP L1
                                 L2:
```

Coding Bitwise Operations

C provides 6 operators for bit manipulation. These are & (Logical AND), | (Logical OR), ^ (Logical-XOR), << (Logical Shift-Left), >> (Logical Shift-right), and ~ (one's complement). There is one-to-one equivalence in XA for such operation class:

(a) & – AND,
(b) | – OR, ^ – XOR,
(c) << – ASL,
(d) >> – LSR, and
(e) ~ – CPL.

Compiler Optimization

Some special cases of Multiply and Divide where the multiplier and divisor could be assumed to a power of 2, following translation could be expected from the compiler during optimization which speeds up code execution and make code denser.

Language extensions to XA could be written as the pre-processor macros of the XA C-compiler as shown in Table 5.

Table 5.

C-code	XA code
R *= R	R <<= R
R *= Constant	R <<= Constant
R /= R	R >>= R
R /= Constant	R >>= Constant

ROLC(R,R) – for rotate left through carry, ROL (R,R) and ROL (R, constant) – for rotate lefts, etc. Same holds for ADDC and SUBB also.

Reentrancy

In a multi-tasking or nested interrupt environment, some system or library subroutines may be activated dynamically. These subroutines require duplication of the variable area of the subroutine per each active copy, utilizing essentially *dynamic memory allocation.*

The allocation of the dynamic area is done by a system service call. The dynamic area is allocated either out of the reserved system memory, when large memory exists in the system, or on the stack, when memory is very limited. In the latter case, the stack pointer is adjusted, to reflect the extra bytes reserved. It will be readjusted just prior to returning from the subroutine.

The subroutine code accesses variables using [R+offset] addressing mode. The register is referred to as a Static Base Register or Frame Pointer.

Since the application stack is separate from the interrupt stack, there's no problem with interrupting the dynamic allocation/de-allocation and application stack pointer adjustments.

Floating Point Support

Although the XA does not have a floating point unit, it has special instructions to provide an extensive support for floating point operations. Instructions like NORM (normalize), SEXT (sign extend), ASL, ASR (Arithmetic shifts) and status flag like "N" (sign), all aid in floating point support. Floating point library routines implementing (IEEE or ANSI) floating point provided with compilers could extensively use such instructions for increased code density and throughput in XA.

Dynamic Code Link/Relocability

The XA allows for dynamic code linking through extensive use of FCALL (Far Call 24-bit addressing). This makes code developed for XA highly portable/relocatable in memory.

Simple relocatable code however could use CALL rel16 and CALL [R] addressing modes which is limited to 64K address.

System Interface

When used for RTOS, system mode with its protected features could be extensively used for system management routines/operating System service e.g., *printf etc* and application task switching. This could be easily done in XA through a TRAP # instruction set up by the compiler requesting system service by the application task. In the event of task switching, a system service call sets up the environment for the new task via the resource access privileges of the task, application stack etc.

Author's Acknowledgement

The author recognizes the following Philips Semiconductor XA team members for their review and inputs on this article:

Ata Khan, Ori Mizrahi-Shalom, and Frank Lee

References:

XA User Guide – Philips Semiconductors

XA benchmark versus the architectures 68000, 80C196, and 80C51

AN703

Author: Santanu Roy

BACKGROUND

A computer benchmark is a "program" that is used to determine relative computer core performance by evaluating benchmark execution time by that core. In the brainstorm on microcontrollers for automotive applications, an assembler functional *benchmark for engine management*, which is a typical example of embedded high-end microcontrol, was created. This report gives worked out routines of the functions if they were implemented in assembler language of the compared controllers: Motorola 68000, Intel 80C196, Philips 80C552 and Philips XA. The total execution times of a program "engine cycle" (engine stroke) are calculated and the required program code is estimated for each controller.

Evaluation of performance in a High Level Language (HLL) like C would be preferable, but it is difficult to realize as "the best" compilers for all cores involved then should be used.

This document is generated based on the report number DPE88187. It outlines code density and execution times of the XA, based on most recent information. The execution times are given in terms of both clock cycles and time units. Although XA can run at speed of 30 MHz @ 5.0 Volts, for sake of fairness, all cores are evaluated for running at 16.00 MHz. This is reasonable for comparing the cores at the same level of technology.

A separate section is included in this benchmark for "Bit manipulation" function benchmark results only. This (bit-test) routine is a stand alone one and should not be considered as a part of *engine management* routine.

BENCHMARK RESULTS AND CONCLUSIONS

Relative performance on a line

The table below presents the most important result of the assembler benchmark evaluation. It pictures the relative performance of the compared core instruction set on a scale where XA=1.0. Also appended is the performance charts-execution and code density of all the processors.

Total exec.times/core(µs) for all routines (with *occurrences)

5,942 1,560 1089.24 402.6

PERFORMANCE RATIO	8051	68000	80C196	XA
8051	1.0	3.81	5.45	14.7
68000	0.34	1.0	1.43	3.85
80C196	0.18	0.7	1.0	2.7
XA	0.068	0.26	0.37	1.0

Table 1. XA instruction set execution times and bytes/function

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.94	15.8	18
ADD/SUB	50	0.38	19	4
CMP 24b	13	1.06	13.78	9
CAN 16b	40	0.563	22.52	5
INTPLIN	20	1.98	41.3	14
INTERR	10	6.1	61	41
BRANCH	10		153.1	

XA totals : 335.5 µs
including 20% statistics : 402.6 µs

Table 2. 68000 instruction set execution times and bytes/function

FUNCTION	OC*	68000		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(μs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	4.4	52.8	2
FDIV	4	13.4	53.6	16
ADD/SUB	50	2.75	137.5	12
CMP 24b	13	3.2	41.6	14
CAN 16b	40	2.7	108	14
INTPLIN	20	7.5	150	14
INTERR	10	21.9	219	92
BRANCH	10		537.5	

68000 totals : 1,300 μs
including 20% statistics : 1,560 μs

Table 3. 80C196 instruction set execution times and bytes/function

FUNCTION	OC*	80C196		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(μs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.75	21	3
FDIV	4	9.5	38	19
ADD/SUB	50	1.25	62.5	7
CMP 24b	13	4.25	55.2	14
CAN 16b	40	2.5	100	6
INTPLIN	20	6.4	128	18
INTERR	10	12.8	128	58
BRANCH	10		375	

80C196 totals : 907.7 μs
including 20% statistics : 1,089.24 μs

Table 4. 8051 instruction set execution times and bytes/function

FUNCTION	OC*	8051		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(μs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	37.5	450	58
FDIV	4	451.5	1806	96
ADD/SUB	50	7.5	375	19
CMP 24b	13	9.98	129.74	22
CAN 16b	40	9	360	14
INTPLIN	20	25.8	516	20
INTERR	10	31.5	315	70
BRANCH	10		1000	

8051 totals : 4,951.74 μs
including 20% statistics : 5,942 μs

Table 5. Total benchmark execution time results

MICROCONTROLLER CORE	EXECUTION TIME (µs)
XA	402.6
68000	1560
80C196	1089.24
8051	5942

As the total activity has to be completed in one machine stroke of 2 ms, the XA, and the 80C196 will be able to meet the application requirements. The 80C552 originally was assumed to complete the functions over more than one stroke.

Best efficiency is of the XA and the 80C196. The 80C196 includes 3-parameter instructions that reduce the instruction count per function and it has JB/JBN instructions. It also uses half-word (1-byte) codes for frequently used instructions.

The lower code efficiency of the 8051 instruction set can mainly be explained by the "accumulator bottleneck" which is not present in XA: most data has to be transported to and from the accumulator be fore add/sub/cmp can be done, operations on words require 4 "MOV" instructions and 2 data execution instructions. The efficient JB and JBN instructions compensate this for a great part.

BENCHMARK LIMITATIONS

Like all benchmarks, the automotive engine management assembler functional benchmark has some weakness that limit validity of its results.

1. Control in a special (automotive, engine) environment is evaluated.
2. Occurrences of operation overheads are based on estimations.
3. Occurrences of functions are based on estimations.
4. Functions are implemented in assembler, not in a HLL like C.
5. Routines may contain assembler implementation errors.
6. All cores are evaluated at 16.0 MHz

Control in a special environment is evaluated (automotive, engine)

The core performance evaluation is based on a single specialized case. All benchmark implementations are fractions of the automotive engine management PCB83C552 demonstration program.

It can be advocated that the automotive engine control task gives a good example of a typical high demanding control environment, where many >= 16 bit calculations have to be done.

Occurrences of overheads are based on estimations

The assembler functional benchmark is not a full implementation of a program. Arbitrary choosing location for storage of parameters in register file or (external) memory, for instance, has for some instruction set a considerable effect on the total execution time.

For the different core parameter storage is chosen where possible using the core facilities to have minimum access overhead.

Occurrences of functions based on estimations

Occurrences is estimated on basis of experience of the automotive group. In a real implementation of an engine controller accents may shift. As most functions already include some "instruction mix", the effect of changes in occurrences is limited.

Functions are implemented in assembler, not in a HLL like C

Control programs for embedded systems get larger, have to provide more facilities and have to be realized in shorter development times. The only way to do this is to program in a HLL like C. Efficient C-language program implementation requires different features from microcontrollers than assembly programs. Results of this assembler benchmark evaluation therefore have a restricted value for ranking microcontroller performances for future HLL applications.

Benchmark ranking on basis of HLL like C requires good C-compilers of all the devices involved are needed. The quality of the C-compilers really has to be the best there is: HLL benchmarking measures not only the micro characteristics, but even more the compiler ability to use these qualities. As these are not available for all the micros evaluated, all routines are worked out only in assembly.

Routines may contain assembler implementation errors

Assembler routine implementations are made after a short study of the micro specifications and are not checked by assembling or debugging in real hardware environment.

It can be rather safely said that a complete system setup and program debug to correct errors would not lead to considerable differences in performance results. Deviations in function occurrences and overheads may have a more significant effect on performance ratios.

All cores are evaluated at 16.0 MHz

A 16.0 MHz internal clock frequency seems a reasonable choice for comparing the cores at the same level of technology.

XA benchmark versus the architectures 68000, 80C196, and 80C51

ASSEMBLER FUNCTIONAL BENCHMARK FOR AUTOMOTIVE ENGINE MANAGEMENT

This benchmark is a functional benchmark: it is a collection of functions to be executed in an automotive engine management program. It would be preferable to implement the complete control program in assembler and evaluate it in a real hardware environment, but this is not practical as every implementation requires many man-months to realize.

To implement the assembly functional benchmark for automotive engine management correctly the "rules and details" described in this section have to be followed carefully.

The assembler functional benchmark embraces all activity to be completed in 1 program cycle that corresponds with 1 engine stroke of 2 ms. The benchmark execution time will be calculated as the sum of the products of functions and their occurrence rates in 1 calculation cycle.

Branches are evaluated separately as "branch penalties" have considerable effect of program execution efficiency. Estimated (branch count)*(average branch time) is added to the function execution times.

The relative estimated overhead for statistics does not contribute to the evaluation of speed performance ratios, but they have to be considered when looking at the total execution time required / engine stroke cycle. therefore the real total execution time is multiplied with the statistics overhead factor (1.2*).

NO.	FUNCTION DESCRIPTION	OCCURRENCES
1	16×16 Multiply	12
2	Floating Point divide (16:16)	4
3	Add/Subtract (24)	50
4	Compare (24)	13
5	CAN cmp/mov 10*8	80
6	Linear Interpolation (8*8)	20
7	Interrupts	10
8	Program control branches	500
9	Statistics (20%)	1.2 *

FUNCTION PARAMETER ALLOCATION

Most functions are very short in exec. time, so that the function parameter data access method has great effect on the total time. Thus it is to be considered carefully.

Some core features a large register files (XA, 80C196) in which variables can be stored, others with few registers (68000) have to store all data in memory.

For the XA/80C196 processor, data stored in the lower part of register file, or in SFRs for I/O, can be accessed using "direct" addressing, but table data, used, e.g., for 3 bye compare, is stored in "external memory".

The 68000 assume data in memory (or memory mapped I/O) as not enough data registers are available. All 68000 memory data has to be accessed using long-absolute addressing: 68000 short addresses are relative to memory address 0000 and are therefore not useful.

For more complex functions 16*16 multiply, Floating point division and interpolation, data is assumed to be already in registers.

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair.

Divide (16:16) "floating point"

The floating point division is entered with parameters in registers:

a divisor, a dividend and an "exponent" that determines the position of the fraction point in the result.

Floating point binary 16/16 division is a function that is normally not included in HLL compilers as it requires separate algorithms for exponent control and accuracy is limited. For assembler control algorithms, floating point division can be quite efficient as it is much faster than normal "real" number calculations (where no "floating point accelerator" hardware is available).

Compare 24-bit variables

Note that 24-bit compare is very efficient for "real" 16-bit and 8-bit) controllers, but for automotive engine timers, 24-bit seems a good solution.

Compare must give possibility to decide >, < or =. For 68000, and 80C196 instruction set LT, EQ and GT are included in the cc after CMP.

CAN move and compares

For service of the CAN serial interface, it is estimated that 40* (2 byte compares + branch) have to be done. Devices with 16-bit bus assumes word access. An average branch is included in the CAN compare function.

Linear Interpolation (8*8)

The interpolation routine is entered with 3 register parameters:

1. Table position address
2. X fraction
3. Y fraction

The routine first interpolates using the X fraction the values of F(x.x, y) between F(x,y) ....V(x+1, y) and of F(x.x, y+1) between F(x, y+1) .... F(x+1, y+1). From F(x.x, y) and F(x.x, y+1) the value of F(x.x, y.y) is interpolated using the fraction of y.

The table is organized as 16 linear arrays of 16 x-values, so that an V(x,y) can be accessed with table origin address +x+16*y = "Table Position Address". In x-direction the interpolation can be done between the "Table Position" value and next position (+1). Interpolation in y-direction is done by looking at "Table Position" + 16.

For linear interpolation time the 2-dimensional interpolation time and byte count are divided by 3 to include some "overhead" into linear interpolation.

Interrupts

The average interrupt routine overhead includes the following stages:

a. Interrupt recognition and return

b. 1 * (long) branch

c. 2 * jump (short) on bit

d. 1 * call (long) and subroutine return

e. 2* set bit and 2 * clear bit

f. 5 * POP and 5 * PUSH (or move multiple) [free 5 registers for local use]

g. 1 * mov #xxx, PST

Program Control Overheads

For a given algorithm, the Program Control Overheads consisting of a number of decisions (branches) and subroutine calls is independent of the instruction set used, except for cases where functions can be replaced by complex instructions. The most important exception cases, MPY words and Floating Point Division are handled in this benchmark separately.

Most 16-bit cores use more pipeline stages so that taken branches add branch time penalty for these CPU's due to pipeline flush. This effect can be found in the branch execution time tables.

More efficient data operations and pipeline penalty of the more complex instruction set of 16-bit cores lead to considerable higher relative time used for branch instructions.

To incorporate the influence of branches in the benchmark the number of branches to be included must be estimated. For byte and bit routines, branches occur more frequent. Average branch time of 25% may be a good guess. For the automotive engine management benchmark that executes in approx. 5000/µS (on 8051) results in +/- 1250 /µS or 625 branches. As a part of the branches already taken account for in the compare functions the number of additional program control branches is estimated 500 branches.

To estimate the average branch execution time, an estimated relative occurrence of the branch types has to be made.

Table 6. Estimated relative occurrence of the branch types

	TYPE	RELATIVE	ABSOLUTE OCCURRENCE
Absolute Jumps	AJMP/JMP	20%	100
Subroutine calls	ACALL/JSR	20%	100
Jump on condition (rel)	Bcc/Jcc	40%	200
Jump on bit (rel)	JB/JBN	20%	100

Statistic Routine Overheads

Statistic routines are estimated as relative program overheads, only to get an indication of the required total processing time in a real engine management application. "Statistics" are mainly arithmetic routines to determine table corrections. They use about 20% of the total time.

XA BENCHMARK RESULTS

The following analysis assumes worst case operation. At any point in time, only 2 bytes are available in the instruction Queue. An instruction longer than 2 bytes requires additional code read cycle.

APPENDIX 1

XA Function Implementations

XA reference: *XA User's Manual 1994*

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair. The MUL.w R,R is encoded in the XA instruction set as a 2 byte instruction. The exact optimization for this instruction (such as skip over 1's and 0's) has not been concluded at this point, and the execution time may be data dependent and shorter than one outlined here.

The basic algorithm utilizes 2-bit Booth recoding. Instruction fetch and Decode time overlaps the execution of the preceding instruction (except when following a taken branch), so it is ignored. The total execution time is either 11 or 12 clocks, including operand fetch and write back (1 clock is dependent on critical path analysis).

A1.1: 16×16 Multiply

```
                                                          Bytes    Clocks
        MUL.w   R0, R1                                      2      12 (0.75 µs)
```

A1.2: Floating Point 16x16 Divide:

The algorithm here follows the one outlined for the 80C196.

```
Arguments:      R4 = Dividend (extend into R5 for 32 bits)
                R6 = Divisor Mantissa
                R0 = Divisor  Exponent

                                                          Bytes    Clocks
FPDIV:
        ADDS    R6, # 0             ; Add short format      2       3
        BEQ     L1                  ; Check for DIVBY0      2       3  (not taken)
                                    ;
SGNXTD_AND_SHFT:                    ;
        SEXT    R5                  ; Sign extend into R5   2       3
        ASL     R4, R0              ; 13 position shifts    2       11
                                    ;
DIV:                                ;
        DIV.d   R4, R6              ; Divide 32x16 signed   2       21
        BOV     L1                  ; Branch on Overflow    2       6  (taken)
        RET                         ; Normal termination    2       8
                                    ;
L1:                                 ;
        MOVS    R4, # -1            ; Overflow - Max Result 2       3  (not  executed)
        RET                         ;                       2       8

                                                           18      63 (3.94 µs)
```

A1.3: Extended 32-bit subtract

```
; R5:R4 = Minuend
; R3:R2 = Subtrahend
        SUB     R4, R2                                      2       3
        SUBB    R5, R3                                      2       3

                                                            4       6  (0.38 µs)
```

A1.4: Compare 24-bit Variables

Only minimum execution time is considered here. An average branch is included after compare. The table data, used for 3 byte compare, is stored in memory.

```
                                                        Bytes   Clocks
        CMP.b       R1L, R2L    ; direct addressing       2       4
        BNE         L1          ; average (6t/3nt)        2       4.5
        CMP.w       R0, mem2    ;                         3       4
L1:
        CMP.w       R0, mem1    ;                         3       4
        Bxx         LABEL1      ; average                 2       4.5
LABEL1:
                                ; xx -> GT or LT or EQ
                                                          9      17 (1.06 µS)
```

A1.5: CAN Move and Compare

Application:

For service of CAN (Controller Area Network) serial Interface it is estimated that 40* (2 byte compares + branch) have to be done. One parameter is in register, the other in internal memory. Again, minimum execution times are considered.

```
                                                        Bytes   Clocks
        CMP         R0, mem0    ;                         3       4
        Bxx         LABEL       ; average                 2       4.5
                                                          5       9 (0.563 µS)
```

A1.6: Linear Interpolation

```
Arguments:
            R0 = Table Base (assumed < 400 Hex)
            R2 = Fraction 1
            R4 = Fraction 2
            R6 = Result

                                                        Bytes   Clocks
LIN_INT:
            MOV         R6, [R0+]       ;               2       4
            MOV         R1, [R0]        ;               2       3
            SUB         R1, R6          ;               2       3
            MULU.w      R6, R2          ;               2       12
            MOV.b       R1H, R1L        ;               2       3
            MOVS.b      R1L,#0          ;               2       3
            ADD         R6, R1          ;               2       3
            ADD         R0, #15         ;               2       3
            MOV         R1, [R0+]       ;               2       4
            MOV         R5, [R0]        ;               2       3
            SUB         R5, R1          ;               2       3
            MULU.w      R5, R2          ;               2       12
            MOV.b       R1H, R1L        ;               2       3
            MOVS.b      R1L,#0          ;               2       3
            ADD         R1, R5          ;               2       3
            SUB         R1, R6          ;               2       3
            MULU.w      R1, R4          ;               2       12
            MOV.b       R1H, R1L        ;               2       3
            MOVS.b      R1L,#0          ;               2       3
            ADD         R6, R1          ;               2       3
            RET                         ;               2       6
                                                       42      95 (5.94 µS)
```

Linear Interpolation (2 dim. time / 3) = 14 bytes, 1.98 µS

A1.7: Interrupt Overhead

Note: Interrupt overhead, as defined in the benchmark, applies to performance calculations. It does not consider the interrupt latency associated with completing the current instruction.

All transfers are to / from internal memory, all addresses are 16-bit long.
{
Saves 2 words on stack = 4 clks
Prefetching ISR = 3 clks
Overhead through Interrupt Controller = 3 clks (allow synch + avoid metastability)
i.e., total = 10 clks
}

```
Interrupt Accept/Return
JMP     rel16              ; uncond. x 2                    0/2     10+8
Bxx     bit, rel8          ; Branch on bit test x 2         3x2     6x2
CALL    rel16              ; Long Call (PZ assumed)         2x2     4.5x2
RET                        ; Subroutine return              3       4
SETB    bit                ; Set bit x 2                    2       6
CLR     bit                ; Clear bit  x 2                 3x2     4x2
PUSH    Rlist (5)          ; 5 PUSH Multiple                3x2     4x2
POP     Rlist (5)          ; 5 POP Multiple                 2       15
MOV     PSWL, #data8       ; imm. byte to PSWL              2       12
MOV     PSWH, #data8       ; needs 2 for 8-bit sfr          4       3
                           ; bus                            4       3
                                                            41      98 (6.1 µS)
```

A1.8: Program Overhead

Branches are assumed taken 70% of the time, all addresses are external. Code is assumed a run-time trace, code size cannot be calculated; based on the same approach taken for 80C196, code size is 1400 bytes.

```
JMP     rel16          ; Long branch  x 100          3x100   6 x 100
CALL    rel16          ; Call  x 100 (Page 0)        3x50    4 x 50
RET                    ; Subroutine return x 100     2x100   6 x 50
Bxx     rel8           ; Condl. short branch x 100   2x200   4.5 x 200
JB/JNBbit, rel8        ; Bit test & branch x 100     2x100   4.5 x 100

                                                     1400     2,450
                                                              (153.1 µS)
```

A1.9: XA TOTALS

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.94	15.8	18
ADD/SUB	50	0.38	19	4
CMP 24b	13	1.06	13.78	9
CAN 16b	40	0.563	22.52	5
INTPLIN	20	5.94	118.8	42
INTERR	10	6.1	61	41
BRANCH	10		153.1	

Conclusion:

An assumption is made that XA code is in first 64K (PZ) as the 80196 has a 64K address space only.

APPENDIX 2

8051 Function Implementations

8051 reference: *Single chip 8-bit microcontrollers PCB83C552 Users manual 1988*

A2.1: 80C51 Multiply 16×16

The 80C51 core performs 8-bit multiply only. A 16×16 multiply has to be done by splitting X and Y into XH, XL and YH, YL so that:

P3..P0 = (XH*256+XL)*(YH*256+YL) =
XH*YH*65536+(XH*YL+XL*YH)*256+XL*YL

```
                          Clocks  Bytes
MPY:
        MOV     R1,XH       2       3
        MOV     R2,XL       2       3
        MOV     R3,YH       2       3
        MOV     R3,YL       2       3
        MOV     A,R2        1       1       ;XL
        MOV     B,R4        1       3       ;YL
        MUL     AB          4       1
        MOV     P0,A        1       2       ; Lowest multiply result byte
        MOV     A,R4        1       1       ;YL
        MOV     R4,B        2       3       ; XL*YL upper byte (*256)
        MOV     B,R1        2       3       ;XH
        MUL     A,B         4       1       ;XL*YL
        ADD     A,R4        1       1
        MOV     R4,A        1       1       ;upper (Xl*YL)+lower(XH*YL) in R2
        MOV     A,B         1       2
        ADDC    A,#0        1       2
        XCH     A,R2        1       1       ;XL upper (XH*YL) in R2
        MOV     B,R3        3       2       ;YH
        MUL     A,B         4       1       ;XL*YH
        ADD     A,R4        1       1
        MOV     P1,A        1       2
        MOV     A,B         1       2
        ADDC    A,R2        1       1
        MOV     R2,A        1       1
        MOV     A,R3        1       1
        MOV     B,R1        2       3
        MUL     AB          4       1
        ADD     A,R2        1       1
        MOV     P2,A        1       2
        MOV     A,B         1       2
        ADDC    A,#0        1       2
        MOV     P3,A        1       2

        Total               50      58
```

50 clocks = 50*12 = 600 clocks (37.5 µs @ 16.0 Mhz)

8051 MPY 16×16 (MPY Bytes) 50 clocks = 37.5 µs / 58 bytes

A2.2: 8051 Divide (16/16) "floating point"

Divide (R6, R7) (dividend) by (R4,R5) (divisor) with (R0) bits after the fraction point.

Alignment of MSBits of operand in R6.7 and R4.7 using R0 as bit counter.

```
                                  Clocks  Bytes
FDV:
        INC     R0                  1       1
        INC     R0                  1       1
        MOV     R3,#0               1       2
        MOV     R2,#0               1       2
        CLR     C                   1       1
        CLR     F0                  1       2
        MOV     A,R4                1       1
        JB      ACC.7, L2           2       3
        JNZ     L1                  2       2
        MOV     A,R5                1       1
        JZ      LX                  2       2
L1:
        MOV     A,R5                1       1
        RCL     A                   1       1
        MOV     R5,A                1       1
        MOV     A,R4                1       1
        RCL     A                   1       1
        MOV     R4,A                1       1
        INC     R0                  1       1
        JNB     ACC.7, L1           2       3
L2:
        MOV     A,R6                1       1
        JB      ACC.7, L6           2       3
L3:
        MOV     A,R7
        RLC     A                   1       1
        MOV     R7,A                1       1
        MOV     A,R6                1       1
        RLC     A                   1       1
        MOV     R6,A                1       1
        DJNZ    R0, $+4             2       2
        AJMP    LX                  2=0     3
        JNB     ACC.7,L3            2       3
        AJMP    L6                  2       3
L4:
        MOV     A,R3
        RLC     A                   1       1
        MOV     R3,A                1       1
        MOV     A,R2                1       1
        RLC     A                   1       1
        MOV     R2,A                1       1
        JNC     L5                  2       2
        MOV     R2,#0FFH            1       1
        MOV     R3,#0FFH            1       1
        SJMP    LX                  1       1
L5:
        CLR     C                   1       1
        MOV     A, R7               1       1

        RLC     A                   1       1
        MOV     R7,A                1       1
        MOV     A,R6                1       1
        RLC     A                   1       1
        MOV     R6,A                1       1
        JNC     L5                  1       1
        MOV     F0,C                1       2
```

```
L6:
        CLR     C               1       1
        MOV     A,R7            1       1
        SUBB    A,R4            1       1
        JNC     L7              2       2
        JNB     F0,L8           2       3
        CPL     C               1       1
L7:
        MOV     R6,A            1       1
        MOV     A, R1           1       1
        MOV     R7,A            1       1
L8:
        CPL     C               1       1
        DJNZ    R0,L4           2       2
        MOV     A,R3            1       1
        ADD     A,#0            1       1
        MOV     R3,A            1       1
        MOV     A,R2            1       1
        ADD     A,#0            1       2
        MOV     R2,A            1       1
LX:
        RET                     2       1
```

Total 96 bytes 13 branch instructions (=35 bytes== 36%)

Timing : 3 divide cases :		**subtracts**	**shifts**	**total**	**average**
1. R0=0E, 8-bit/14 bit	-->	15-8+2=9	8+2=9	32 subtracts	11
2. R0=08, 12-bit/14 bit	-->	8-4+4=8	4+4=8	17+11 shifts	6+4
3. R0=10, 11-bit/12 bit	-->	16-5+4=15	5+5		

17+4*9+6*10+(15.5+10*31.5)+8=451.5 clocks = 338.6 µS

8051 UFDIV 16/16 (sub/sft) : 338.6 clocks = 451.5 µs, 96 bytes.

A2.3: 8051 Add/Sub

```
                        Bytes   Clocks
ADS:
        CLR     C               1       1
        MOV     A,X0            1       2
        SUBB    A,Y0            1       2
        MOV     Z0,A            1       2
        MOV     A,X1            1       2
        SUBB    A,Y1            1       2
        MOV     Z0,A            1       2
        MOV     A,X2            1       2
        SUBB    A,#0            1       2
        MOV     Z2,A            1       2

                                10      19
```

8051 ADD/SUB in reg file 10 clocks = 7.5 µs, 19 bytes

8051 CMP enabling JZ JNZ JC JNC

The 8051 decisions made with branches are one of these three :

```
        JC      lt              2       2
        JC                      2       2
        JZ      eq              2       2
        JC                      2       2
        JNZ     gt              2       2
```

8051 compare decision branches take average : 10/3 clocks => 2.5 µs

A2.4: 8051 CMP 3 byte compare

```
                          Bytes  Clocks
CM3:
        CLR     C           1      1
        MOV     A,X2        1      2
        SUBB    A,Y2        1      2
        MOV     R0,A        1      2
        MOV     A,X1        1      2
        SUBB    A,Y1        1      2
        ORL     R0,A        1      2
        MOV     A,X2        1      2
        SUBB    AY2         1      2
        Orl     A,R0        1      2
        Jcc     xxxx        3.33   3.33

                           10     19
```

8051 CMP 3 byte data in reg file 13.3 clocks = 9.975 µs, 22.3 bytes

A2.5: 8051 2-byte CAN compares

```
                          Bytes  Clocks
CAN:
        MOV     DPTR,aX1    2      3        ; one compare src in X-RAM
        MOVX  A, @DPTR      1      2
        CJNE    A,Y1        1      2
        MOV     DPTR,aX2    1      2        ; one compare src in X-RAM
        MOVX  A,@DPTR       1      2
        CJNE    A,Y2        2      3

                           12     14
```

8051 CAN CMP XRAM/Direct 9 µs, 14 bytes

A2.6: 8051 2-dimensional interpolation

```
At the start registers are prepared
A       : position in table (x+16*y)
DPTR    : Start address of table (aligned at 256 byte boundary)
R0      : x-fraction   R1      : y-fraction
Result  : ACC registers used : ACC,R0,R1,R2,R4,R5,R6
```

			Clocks	Bytes	
INT:					
	MOV	DPL,A	1	2	;POS X,Y
	ACALL	GVAL	2	2	
	MOV	R4,A	1	1	
	MOV	A,DPL	1	2	
	ADD	A,#15	1	2	
	MOV	DPL,A	1	2	
	ACALL	GVAL	2	2	
	MOV	REG6,R4	1	2	
	MOV	B,R1	1	2	
	ACALL	INTP	1	2	
	RET		2	1	
GVAL:					
	MOVX	A,@DPTR	2	1	
	MOV	R6,A	1	1	
	INC	DPL	2	1	
	MOVX	A,@DPTR	2	1	
	MOV	B,R0	1	2	
INTP:					
	CLR	SF	1	2	
	CLR	C	1	1	
	SUBB	A,R6	1	1	
	JNC	INT1	2	2	
	CPL	A	1	1	
	INC	A	1	1	
	SETB	SF	1	2	
INT1:					
	MUL	A,B	4	1	
	XCH	A,B	1	2	
	CLR	C	1	1	
	RRC	A	1	1	
	XCH	A,B	1	2	
	XCH	A,B	1	2	
	CLR	C	1	1	
	RRC	A	1	1	
	XCH	A,B	1	2	
	JB	SF,INT2	2	3	
	ADDC	A,R6	1	2	
	RET		2	1	
INT2:					
	XCH	A,R6	1	2	
	SUBB	A,R6	1	2	
	RET		2	1	

Total 2-dim. interpolation : 15+2*(8+24)+24=103 clocks = 77.25 µs, 59 bytes

8051 Linear interpolation : (2–dim. intp time /3) = 103/3 =25.75 µs, 20 bytes

A2.7: 8051 Interrupt Overhead

```
                                Bytes  Clocks
a.      interrupt                 2      2 (vector)
        RETI                      2      1
b.      AJMP 2*                   4      4
c.      JB      2*                4      6
d.      ACALL                     2      2
        RET                       2      1
e.      SETB 2*                   2      4
        CLRB 2*                   2      4
f.      POP 5*                   10     10
        PUSH 5*                  10     10
g.      MOV  1*                   2      2

                                 42     46
```

8051 Interrupt Overhead 42 clocks = 31.5 µs

A2.8: 8051 Program Overhead

TYPE	OCCURRENCE	8051		BYTES	
LJMP/JMP	100	2	200	3	300
LCALL/JSR	100	2	200	3	300
Jcc/Bcc	200	2	400	3	600
JB/JBN	100	2	200	3	300
total cylces µsec			1000 750		1500

A2.9: 8051 Totals

FUNCTION	OC*	8051	
		EXEC	*OC
1. MPY	12	37.5	450
2. FDIV	4	338.6	1354.4
3. ADD/SUB	50	7.5	375
4. CMP 24b	13	9.98	129.74
5. CAN 16b	40	9	360
6. INTPLIN	20	25.8	516
7. INTERR	10	31.5	315
8. BRANCH	10		750

8051 totals : 4250.14 µs
including 20% statistics : 5,100.2 µs

APPENDIX 3

68000 implementations

68000 reference: *SC68000 microprocessor users manual*

(Motorola copyright; Philips edition 12NC: 4822 873 30116)

A3.1: 68000 16x16 Multiply

The 68000 can use 1 <ea> with MUL and move a long word result.

```
MUL     R0,R1                   2       70
```

total: 4.375 µs, 2bytes

A3.2: Floating point division 16:16

```
(R0) Accuracy,          (R1)/(R2)    R1 result
```

```
                                Bytes  Clocks
FDV:
        EXT.l   R1              2       4
        TST     R2              2       4
        BEQ     L1              2       10/8
        ASL     R0,R1           2       32
        DIVU    R2,R1           2       140
        BVC     L2              2       10/8
L1:
        MOVI    #-1,R1          2       4
L2:
        RTS                     2       16
```

total : 214 clocks or 13.375 µs, 16 bytes

A3.3: Add/Sub

```
                                Bytes  Clocks
ADDS:
        MOV.l   A,R0            6       20
        ADD.l   R0,C            6       48
```

total : 44 clocks or 2.75 µs, 12 bytes

A3.4: Compares 24 (=32) bit

```
                                Bytes  Clocks
CMPl:
        MOV.l   X,R0            6       20
        CMP.l   Y,Rn            6       22
        BLT/EQ/GT (av) 2        9
```

total : 51 clocks or 3.19 µs, 14 bytes

A3.5: CAN move and compares (16-bit)

```
                                Bytes  Clocks
CMPw:
        MOV.w   X,R0            6       16
        CMP.w   Y,Rn            6       18
        BLT/EQ/GT (av)          2       9
```

total : 43 clocks or 2.69 µs, 14 bytes

A3.6: 2-dimensional interpolation

A0 : table position, R0 : fraction1, R1 : fraction 2 , R2 : result, R3, R4

```
                                        Bytes  Clocks
CMPw:
        MOV.w           (A0), R2          2       8
        ADDQ.l          #1,A0             2       8
        MOV.l           (A0), R3          2       8
        SUB.w           R2,R3             2       4
        MULu            R0,R3             2      74
        ASR.l           #8,R3             2      28
        ADD.w           R3,R2             2       4
        ADDI.l          #15,A0            4       8
        MOV.w           (A0),R3           2       8
        ADDQ.l          #1,A0             2       8
        MOV.w           (A0),R4           2       8
        SUB.w           R3,R4             2       4
        MULu            R0,R4             2      74
        ASR.l           #8,R4             2      28
        ADD.w           R4,R3             2       4
        SUB.w           R2,R3             2       4
        MULu            R1,R3             2      40
        ASR.l           #8,R3             2      22
        ADD.w           R3,R2             2       4
        RTS                               2      16
```

total : 362 clocks or 22.62 µs, 42 bytes

Linear interpolation is 2-dim. interpolation /3 :

1-dim. interpolation 7.54 µs, 14 bytes

A3.7: 68000 Interrupt Overhead

```
                                Clocks  Bytes
a.      interrupt                 44      4
        RETI                      20      2
b.      JMP 2*                    24     24
c.      BTST+BNE 2*               60     16
d.      BSR                       18      4
        RTS                       16      2
e.      BSET/BCLR 4*              96     24
f.      MOVEM 2* n=5              64     12
g.      MOVI    #xx,CCR            8      4
                                  350     92
```

68000 INTerrupt overhead 350 clocks = 21.87 µs, 92 bytes

A3.8: 68000 Program Overhead

For the 68000, the JB/JBN branches have to be constructed :

		Clocks	Bytes
MOV.w	ABS.l,Rn	12	6
ANDI.w	#bitmask,Rn	8	4
BEQ/BNE	rel.address	10	2

total JB/JNB execution : 34 clocks, 12 bytes

Now the absolute (estimated) branch time can be calculated, taking the core difference in account.

TYPE	OCCURRENCE	68000		BYTES	
LJMP/JMP	100	12	1200	6	600
LCALL/JSR	100	20	2000	8	800
Jcc/Bcc	200	10	2000	2	400
JB/JBN	100	34	3400	12	1200
total cycles µsec			8600 537.5		3000

A3.9: 68000 Totals

FUNCTION	OC*	68000	
		EXEC	*OC
1. MPY	12	4.4	52.8
2. FDIV	4	13.4	53.6
3. ADD/SUB	50	2.75	137.5
4. CMP 24b	13	3.2	41.6
5. CAN 16b	40	2.7	216
6. INTPLIN	20	7.5	150
7. INTERR	10	21.9	219
8. BRANCH	10		537.5

68000 totals : 1,300 µs
including 20% statistics : 1,560 µs

APPENDIX 4

80C196 function implementations

80C196 reference: *Embedded controller handbook* vol II-16 bit

A4.1: 80C196 Unsigned multiply P=X*Y (16x16)

```
                                  Bytes  Clocks
MUL             R0,R1               3      28
```

total: 1.75 µs, 3 bytes

A4.2: Floating point division 16:16

```
(R0) Accuracy, (R4)/(R8)   R4 result

                                  Bytes  Clocks
FDV:
        EXT     R4                  2      4
        AND     R8,#FFFF            4      5
        JE      L1                  2      8/4
        SHLL    R4,R0               3      20
        DIVU    R8,R4               3      24
        JNV     L2                  2      4/8
L1:
        LD      R4,#FFFF            2      5
L2:
        RET                         1      11
```

total: 76 clocks or 9.5 µs, 19 bytes

A4.3: Add/Sub

```
                                  Bytes  Clocks
ADDS:
        SUB     R5,R1,R3            3      5
        SUBB    R4,R0,R2            4      5
```

total : 10 clocks or 1.25 µs, 7 bytes

A4.4: 80C196 "3-byte compare"

```
                                  Bytes  Clocks
        CMP     Rn,Y1               5      9
        BNE     L1                  2      4/8
        CMP     Rm,Y2               5      9
L1:
        BLT/EQ/GT  (av)             2      4/8
```

Average total: 34 clocks or 4.25 µs, 14 bytes

A4.5: CAN move and compares (16-bit)

```
                                  Bytes  Clocks
        CMP     Rx,Y                4      9
        BLT/EQ/GT (av)              2      6
```

total : 15 clocks or 2.5 µs, 6 bytes

A4.6: 80C196 2-dimensional interpolation using in-line linear interpolations

R0 : table position, R2=fraction1, R4=fraction2, R6=result, R8, R10

		Bytes	Clocks
LD	R6,[R0]+	3	6
LD	R8,[R0]+	3	5
SUB	R8,R6	3	4
MULU	R8,R2	3	14
SHRAL	R8,#8	3	15
ADD	R6,R8	3	4
ADD	R0,#15	4	6
LD	R8,[R0]+	3	6
LD	R6,[R0]	3	5
SUB	R10,R8	3	4
MULU	R10,R2	3	14
SHRAL	R10,#8	3	15
ADD	R8,R10	3	4
SUB	R8,R6	3	4
MULU	R8,R4	3	14
SHRAL	R8,#8	3	15
ADD	R6,R8	3	4
RET		1	14

total : 153 clocks or 19.1 µs, 53 bytes

Linear interpolation is 2-dim. interpolation /3 :

1-dim. interpolation 6.4 µs, 18 bytes

A4.7 80C196 Interrupt Overhead

		Clocks	Bytes
a.	interrupt /RTE	27	2
b.	LJMP 2*	14	6
c.	JB 2*av.7	14	6
d.	CALL/RTS	22	4
e.	BSET/BCLR 4*	28	16
f.	POP 5*	40	10
	PUSH 5*	55	10
g.	MOVI #xx,CCR	5	4
		205	**58**

80C196 INTerrupt overhead 205 clocks = 12.8 µs, 58 bytes

A4.8: 80C196 Program Overhead

TYPE	OCCURRENCE	68000		BYTES	
LJMP	100	7	700	3	300
LCALL/RET	100	22	2200	4	400
Jcc/Bcc	200	7	1400	2	400
JB/JBN	100	7	700	3	300
total cycles µsec			6000 375		1400

80C196 totals : 958.1 µs
including 20% statistics : 1150 µs

FUNCTION	OC*	80C196	
		EXEC	*OC
1. MPY	12	1.75	21
2. FDIV	4	9.5	38
3. ADD/SUB	50	1.25	62.5
4. CMP 24b	13	4.25	55.2
5. CAN 16b	40	1.88	150.4
6. INTPLIN	20	6.4	128
7. INTERR	10	12.8	128
8. BRANCH	10		375

BIT MANIPULATION

Copy a bit from one location to another in memory. Complement the bit in the new location

Note: Assumed that memory is on-chip and directly addressed.

Bit "x" of mem0 needs to be copied to bit "y" of mem1.

XA

```
        CLR     C               ; clear Carry                   3       4
        ORL     C, /bitm        ; compl. bit and save in C      3       4
        MOV     bitn, C         ; move mem0.x -> mem1.y         3       4

                                                                9       12
                                                                        (0.75 µS)
```

Intel 80C196

Note : States = clock (period)/ 2

Move complement of bit "m" to "n" in memory

R3 = memory byte having bit "m"
R4 = memory byte having bit "n"
R0 = Used as bit-mask register
R1 = position of "m" in mem0
R2 = position of "n" in mem1

```
                                                        Bytes   States
        LD      R0, 1               ; Load 1 in Reg
        SHLB    R0, R2              ; position of bit "n    3       16
                                    ; in R2
        NOTB    R0                  ; complement            2       4
        JBC     R3,bitm, L1         ; test bit "m" polarity 3       7 (av)
        ANDB    R4, R0              ; reset "n" if "m" = 0  3       4
L1:
        ORB             R4, R0      ; set "m" otherwise     3       either/or

                                                            14      31 (3.88 µS)
```

Motorola 68000

```
                                                        Bytes   States
        BTST    bitm                ; Test bit              2       4
        BEQ     L1                  ; Branch if reset       2       6
        BCLR    bitn                ; Test bit and clear (~m = 0) 2 4
        .......
        .......
L1:     BFSET   bitn                ; Test bit and set (~m = 1)   2 either/or

                                                            8       14  (0.88 µS)
```

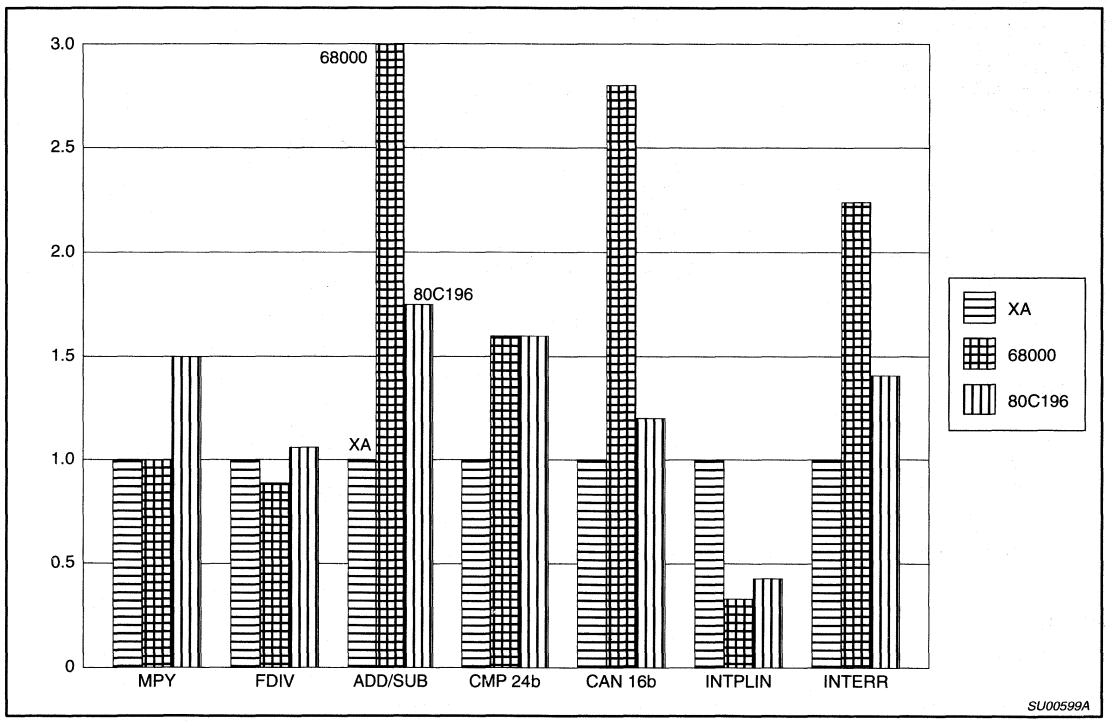
8051 Bit-test

```
        MOV     C, bitm                                     2       12
        CPL     C                                           1       12
        MOV     bitn, C                                     2       24

                                                            5       48 (3.0 µS)
```

XA CODE DENSITY RESULTS

Graph showing performance with respect to 68000, and 80C196 cores normalized with respect to XA. The 80C51 is included just for reference.

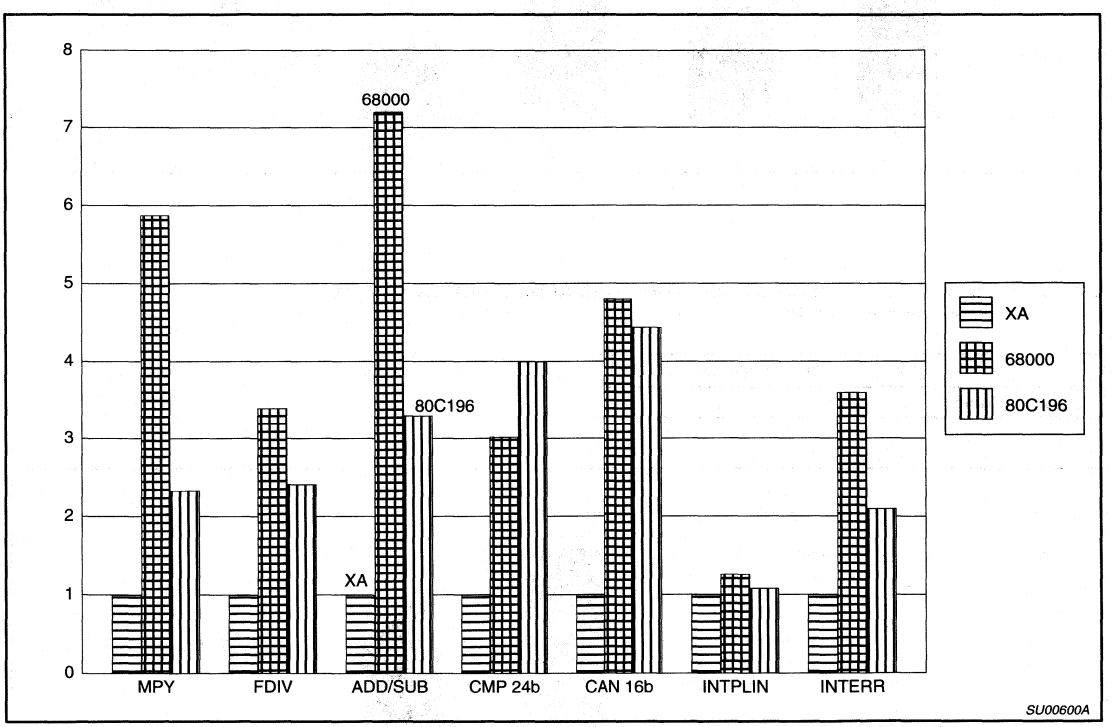
	XA	68000	80C196	8051
MPY	1	1	1.5	1
FDIV	1	0.89	1.06	5.33
ADD/SUB	1	3	1.75	2.5
CMP 24b	1	1.6	1.6	1
CAN 16b	1	2.8	1.2	1.5
INTPLIN	1	0.33	0.43	0.33
INTERR	1	2.24	1.41	1.71



XA EXECUTION TIME RESULTS

Graph showing performance with respect to 68000, and 80C196 cores normalized with respect to XA. The 80C51 is included just for reference.

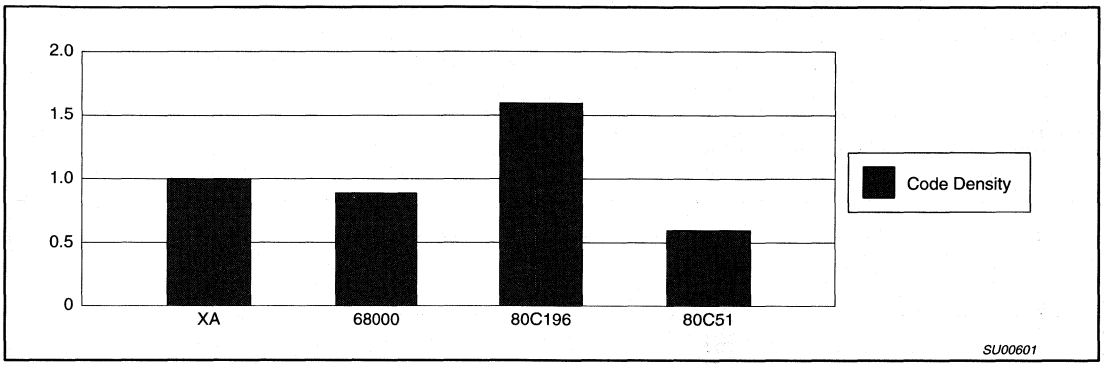
	XA	68000	80C196	8051
MPY	1	5.87	2.33	50
FDIV	1	3.4	2.41	86
ADD/SUB	1	7.2	3.3	19.74
CMP 24b	1	3.02	4	9.41
CAN 16b	1	4.8	4.44	15.98
INTPLIN	1	1.26	1.08	4.34
INTERR	1	3.6	2.1	5.16



BIT TEST BENCHMARK: CODE DENSITY NORMALIZED WITH XA (=1.0)

The 80C51 is shown here only for reference.

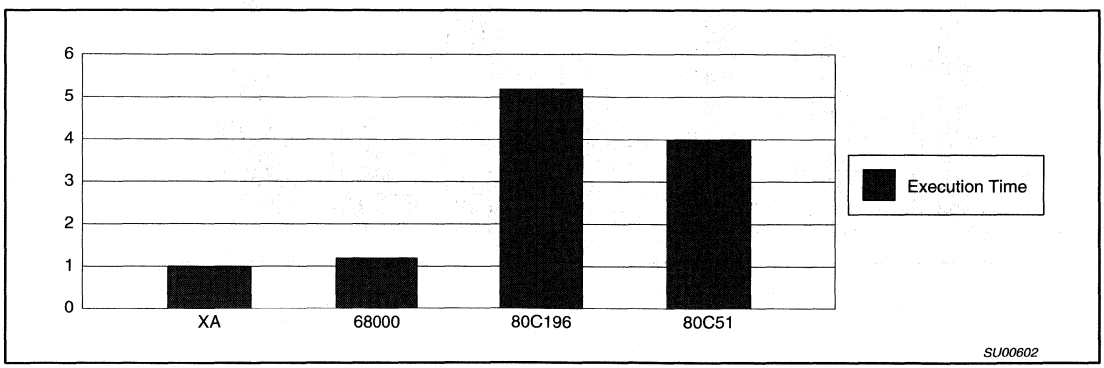
	XA	68000	80C196	8051
Code Density	1	0.89	1.6	0.6



BIT TEST BENCHMARK: EXECUTION TIME NORMALIZED WITH XA (=1.0)

The 80C51 is shown here only for reference.

	XA	68000	80C196	8051
Execution Time	1	1.2	5.2	4



An upward migration path for the 80C51: the Philips XA architecture

AN704

Author: Greg Goodhue

The 80C51 is arguably the most used 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this processor. The "XA" (Extended Architecture) microcontroller, developed by Philips Semiconductors, is a high performance 16-bit processor that retains source code compatibility with the original 80C51. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with a higher performance 16-bit controller. This provides an upward mobility path to a 16-bit controller for 80C51 users that has not previously existed, while also bringing a low cost, high performance, general purpose 16-bit controller to the market. How can a modern 16-bit controller provide compatibility with the venerable 80C51 without badly compromising the architecture and performance?

DESIGN TRADEOFFS

Many tradeoffs must be made and considerations taken into account when creating an upward compatible processor that must also be high performance and low cost. Among the areas to be considered are the processor's memory map and means of accessing memory, instruction set and methods of instruction execution, stack operation, interrupts, and special features added to enhance particular functions, such as multi-tasking, exception handling, and debugging features.

The goal of source code compatibility, rather than object code compatibility, was adopted for a number of reasons. First, absolute upward compatibility with an existing processor is by definition impossible if one of the goals of the new processor is to generally improve performance. By doing the same things in less time, the time related attributes of previously written code change.

Another consideration has to do with the fact that the 80C51 used all but one of the 256 opcodes available with an 8-bit opcode field. Adding more than a few new instructions or a new data type (such as 16-bit operations) would result in a very inefficient instruction encoding, and inefficient execution as well, for those new functions.

Creating a new instruction set that includes an exact copy of the 80C51 instruction set as a subset would also be very inefficient, since some subset of many new operations would act as duplicates of 80C51 instructions. For instance, a more powerful ADD instruction that can add any byte or word register to any other register is a superset of the 80C51 instruction to add a register to the accumulator. In such a case, there is no good argument to duplicate the original instruction precisely.

An 80C51 "mode" on an otherwise totally new (and therefore incompatible) processor was also considered. However, this approach would result in having in effect 2 processors on one chip, which would be confusing and not very cost effective. Mixing new, more efficient code with existing 80C51 code would require switching modes often, which would be very cumbersome and potentially hazardous. If a mode switch was skipped by accident in some seldom executed code sequence, the processor could suddenly find itself executing code using the wrong instruction set!

HOW IS IT DONE?

The team that created the XA architecture at Philips followed several rules in order to insure that 80C51 compatibility goals were met. First, translation for all (or nearly all) 80C51 instructions would be one to one. Multi-instruction combinations that could result in problems if split by an interrupt or otherwise compromise the integrity of the translation would be avoided. This has the effect of producing a simple, straightforward, and easily checkable translation.

Second, most 80C51 instructions should be a subset of new XA instructions. If that is not possible or doesn't make sense in a particular case, the original 80C51 instruction would be included "as-is", even though it might not fit the basic XA architecture's philosophy.

Third, XA register, code memory, data memory, and Special Function Register addressing would be a superset of the 80C51 equivalents. The same idea applies to other features that are part of the CPU.

Finally, some compromises to these compatibility rules are allowed in cases where keeping absolute compatibility would adversely affect system cost, high level language support, or performance. The cost (in engineering time) of dealing with any incompatibilities must be kept to a minimum. Preferably, the issue should not even be noticeable to most customers.

An upward migration path for the 80C51: the Philips XA architecture

AN704

MEMORY MAPPING

At the root of any potential compatibility between the XA and the 80C51 is the memory map. The XA takes a simple but effective approach to this issue: its memory map is a superset of the 80C51 memory map. Modes of addressing memory likewise duplicate the modes available on the 80C51, adds new modes, and enhances some of the old ones.

In translating 80C51 source code to the XA, particular registers are used to represent the accumulator (A) and the data pointer (DPTR). Although the XA can use any of the 14 general purpose byte registers int he register file as an accumulator, the 80C51 has some features that require the accumulator to be a specific byte register. These are primarily the parity flag and a few special instructions that intrinsically reference the accumulator in a way that could not be generalized in the XA. The latter are, specifically, instructions like: JZ, JNZ, MOVC A,@A+DPTR, MOVC @A+PC, and JMP @A+DPTR. Figure 1 shows the register file of the first XA derivative (the XA architecture can support some additional registers not implemented in the first part) and the registers used for 80C51 translation.

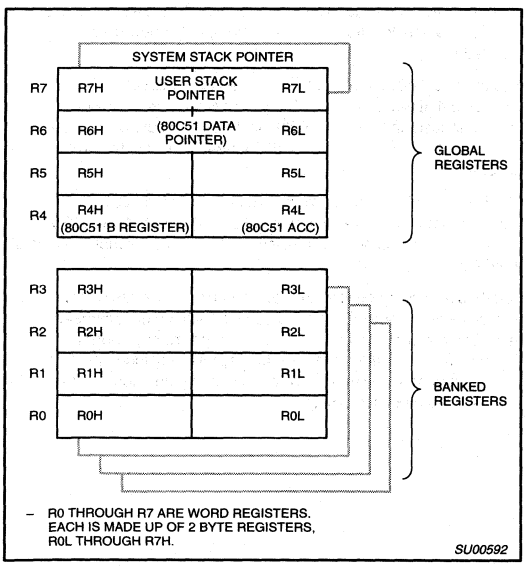


Figure 1. XA Register File

An alternate program status word (PSW) was created on the XA to duplicate the 80C51 PSW and contains the P (parity) flag as well as the F1 and F0 user defined flags that are not found in the native XA PSW. The XA PSW, on the other hand, adds some new status flags and system controls to expand its capabilities.

The XA register file duplicates the 4 banks of 8 bytes that are found in the 80C51. An 80C51 compatibility mode determines whether these locations appear both as registers and as the lower 32 bytes of data memory as they do on the 80C51. The more standard scheme of keeping the register file separate from the data memory is the default on the XA. Besides being "cleaner", the separation of the register file from data memory allows for a higher performance implementation of the XA processor core at some point in the future if and when 80C51 compatibility is no longer required. Figure 2 shows the overlap of data memory and the register file in compatibility mode. This shows only this one aspect of the XA memory map, not a general view of the memory.

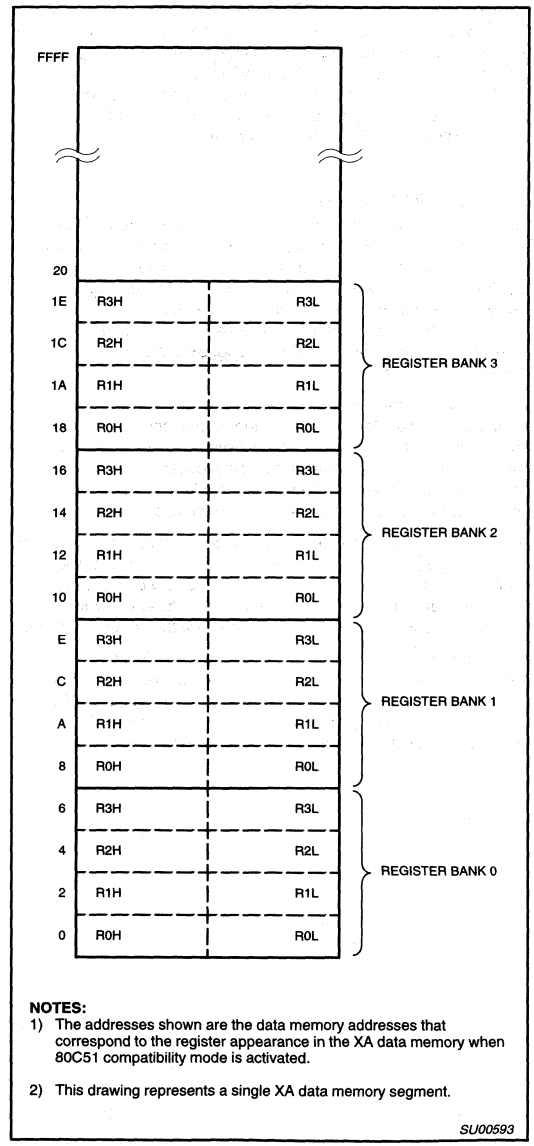


Figure 2. XA Register File and Data Memory Overlap

A second aspect of XA memory addressing is also controlled by the aforementioned 80C51 compatibility mode. In the XA, indirect memory accesses normally make use of a 16-bit pointer register, which may be any of the word registers in the register file. The 80C51, however, allows only the 2 single-byte registers R0 and R1 to used for indirect references. The XA is forced use the first 2 single-byte registers in the currently selected bank as byte pointers rather than word pointers when the 80C51 compatibility mode is activated. Thus, translated 80C51 code typically must be run with the compatibility mode activated.

The data memory map for a single XA data segment looks just like the entire data memory map for an 80C51. This leads to the possibility of using a single XA to perform the function of several 80C51s, with a separate data segment and code area allocated to a task that was originally performed by one 80C51. The XA includes hardware support for multi-tasking operation in order to allow for this and other interesting possibilities.

The XA retains the direct and indirect addressing modes of the 80C51, although both are greatly expanded in capability, as shown in figure 3. The direct data addressing has been increased to use up to 1K bytes of data memory. Indirect addressing is done in 64K byte segments, for a total of up to 16 megabytes. Both types of addressing seamlessly switch from internal to external data memory wherever the boundary exists between the two for a particular chip. In this manner, the processor stack may also be extended off-chip up to nearly 64K bytes if necessary. Because of the seamless internal to external memory transition, the XA would not normally attempt off-chip data accesses at the low memory addresses that correspond to the on-chip data RAM. For that reason, the 80C51 MOVX instruction is included on the XA in order to allow translated code to run without changes in the external memory address map. This works because MOVX always forces data to be read from off-chip memory.

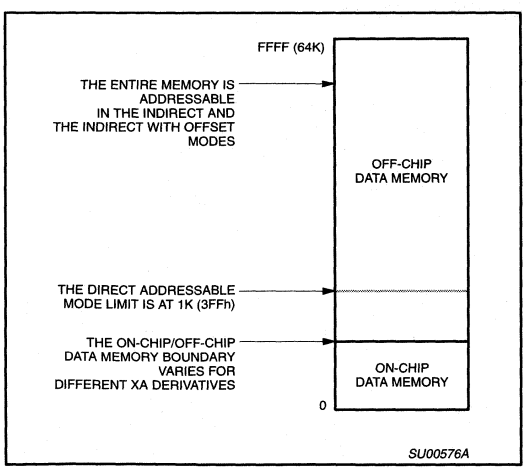


Figure 3. XA Memory Addressing

On the 80C51, the special function registers (SFRs) were mapped into the direct address space starting at location 128, through the end of that space at location 255. Since the 80C51 only allowed SFR access by direct addresses, where the entire address is encoded into the instruction, the XA does not need to duplicate its SFRs in exactly the same area or at the same specific addresses. In order to simplify the memory map, expand the SFR space, and expand the directly addressed data space, the XA defines a totally separate SFR space that is not logically related to the rest of data memory. To translate 80C51 source code, the original SFR name is kept in the translated code, unless the name was changed for some reason. In any case, as long as the reference is by name, a code translator need not try to determine which SFR it is, or where it belongs on a particular XA derivative. If 80C51 source code for some reason references an SFR by its address, a code translator might attempt to look it up in an SFR map for the 80C51 derivative to which the code was targeted.

A second mode control in the XA applies to 80C51 translated code, although it may be used in pure XA applications as well. This is the Page Zero, or PZ, mode. This mode forces the XA to only allow 64K of address space in both the data and code memories. The purpose is to reduce the overhead required to support the extra address space if it is not needed, such as in "single-chip" systems that do not use any off-chip data or program. Besides saving stack space for 24-bit subroutine and interrupt return addresses (reduced to 16 bits in PZ mode), overall XA operation is faster by having smaller stack pushes and pops. Since the 80C51 supported only 64K of code and data space, translated 80C51 code will likely fit into the same category.

There are other changes in the processor stack on the XA, besides the need to save 24 bits of return address when not running in the Page Zero mode. First, a great deal of extra hardware in the processor would be required to allow both byte and word pushes and pops on the stack, especially since word operations could then sometimes be mis-aligned from word address boundaries in the data memory, so stack operations on the XA are always done in word increments. Mis-aligned word operations, aside from being difficult to implement, would be very inefficient, since they would have to be split up into multiple byte operations. This means that translated 80C51 code run on the XA will tend to use somewhat more stack space than it did originally. The automatic save of the PSW during interrupts on the XA might also increase stack usage in some cases, since a few 80C51 programs may have been able to omit saving the PSW during interrupt processing.

Secondly, the XA stack has been altered so that the direction of growth is downward, conforming to the industry standard for stack operation on 16-bit processors. There is also a necessary relationship between the stack growth direction and the order in which the bytes of a word are stored in memory for a processor that is capable of stack relative addressing, as can be done with the XA. This relationship required that the stack grow downward since data on the XA is stored in memory with the low order byte of a word at the lower address (sometimes referred to as Little Endian storage order).

These differences in stack operation may require some changes to be made by the user for any 80C51 source code translated to the XA. In most cases, the change would be limited to choosing a different starting address for the stack.

A look at interrupt processing presents some other issues for 80C51 compatibility. In order to allow more powerful handling of interrupts, the XA has to make some compromises. Besides the previously mentioned fact that the PSW is automatically saved on the stack, which would have been done explicitly in 80C51 interrupt service code, the return address on the stack is also different if Page Zero mode is not active. So, any code written for the 80C51 which relied

in some manner on manipulating the return address on the stack, or on the PSW not being saved and restored automatically, will require modification. Both of these situations should be very rare. The standard (non-Page Zero mode) XA interrupt stack frame is shown in Figure 4.

CPU FEATURES

Another difference in interrupt processing is that the XA uses a more efficient and flexible vector table for interrupts and exceptions instead of the fixed vector scheme of the 80C51. The vector table must reside at the bottom of the code memory, since this is the only region that is guaranteed to always exist in a system that uses on-chip ROM or EPROM for the program. Thus, during 80C51 code translation, code found at the 80C51 interrupt service locations must be moved to another location. Of course, an interrupt vector table must be added to any translated 80C51 program that makes use of interrupts, and a reset vector entry must be created for all XA programs.

A major enhancement to the XA is the addition of a general purpose interrupt priority scheme that can support up to 15 levels, compared to only 2 on standard 80C51 parts, and up to 4 on enhanced parts. This addition, however, requires some changes in the way interrupt priorities are handled. Two-priority interrupt systems on 80C51 derivatives used a single bit in a priority register to select the two levels. Four-priority systems extended this to two bits, but in 2 different registers for each interrupt source. Extending that approach to 15 levels would entail 4 bits in 4 different registers for each interrupt source, which is getting a bit ridiculous. For the XA, a more reasonable approach was taken: 4 bits in a single register control the priority of each interrupt source. Priorities for 2 separate interrupts are contained in each 8-bit priority register.

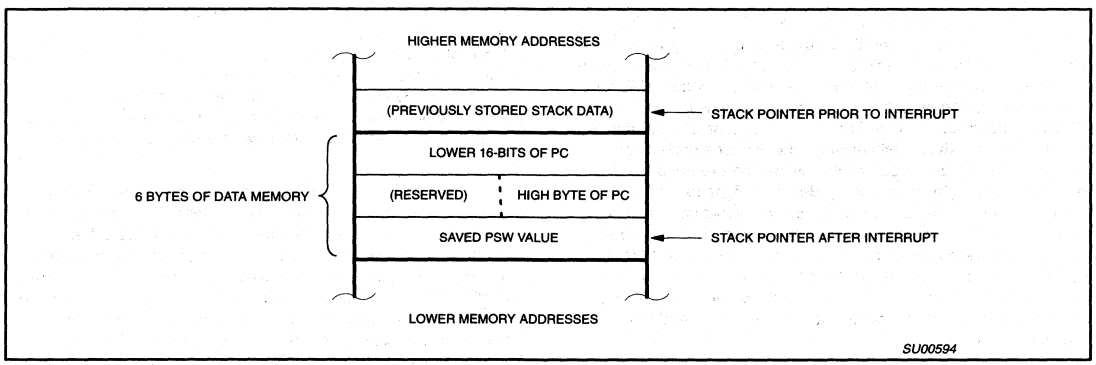


Figure 4. Standard Interrupt Stack Frame on the XA

PERIPHERALS, ON AND OFF-CHIP

Another subject to look at is hardware compatibility. While complete hardware compatibility with the 80C51 was not a primary goal during the XA architecture development, hardware compatibility was retained whenever possible and practical. This particularly concerns peripheral devices such as UARTs, Timers, etc., and the processor's external bus system.

In the case of peripherals that are the same as those customarily found on the 80C51, these have been made to function as close as possible to the original, with some transparent enhancements such as framing error detection, overrun detection, and break detection in the UARTs. One exception to this general compatibility is that timer mode 0 of the standard timers 0 and 1, which is the rarely used 8048 compatible timer mode, has been replaced with a much more useful 16-bit auto-reload mode. In the future, further enhanced peripheral functions will likely lead eventually to completely new implementations that are not backward compatible with the 80C51.

Since there is no supposed relationship between the original oscillator frequency of an 80C51 system and a similar XA system using translated code, the exact relationship of peripheral speeds to the oscillator need not be preserved. For more flexibility in timer rates and therefore UART baud rates, the XA timers and some other peripherals are operated from a special clock whose rate is user programmable. The choices are the CPU clock divided by 4, 16, or 64, giving a wide range of uses. This function, like anything else in an application that is time critical, will need to be visited by the user when translated 80C51 code is used to drive XA peripherals.

The standard XA external bus interface includes all of the familiar 80C51 bus signals: ALE, $\overline{PSEN}$, $\overline{RD}$, $\overline{WR}$, $\overline{EA}$, the multiplexed address and data bus, and address-only lines. However, some additional signals have been added and changes have been made in some of the details. For instance, the XA supports both 8-bit and 16-bit bus widths, using a second write signal to distinguish byte writes on a 16-bit bus. A WAIT line allows external circuitry to insert wait states into bus cycles for slow peripherals or program memories.

The largest change in the XA bus from the 80C51 is in the mapping of the multiplexed address and data lines. The 80C51 has a somewhat inefficient mapping that requires an ALE (Address Latch Enable) cycle in order to latch the least significant bits of an address for all external bus cycles. This was not a concern for the 80C51 due to its machine cycle timing, which allowed plenty of time for an ALE pulse. For the XA, which has no extra cycles during instruction execution, any extra strobes required on the bus during code fetches will likely take away time that could be used to execute instructions. As a result, the XA drives the 4 lower address lines directly, and does not require them to be latched. This means that the XA can fetch as many as 16 bytes of code between ALE cycles. The multiplexed address and data bus begins with the fifth address line (A4), paired with the first data line (D0), and continues to the width of the bus, either 8 or 16 bits. Above that will be more always-driven address lines, if more are needed by the application. Since the XA allows programming the number of address lines, those above the multiplexed portion of the bus need not be driven by the XA if they are not needed, leaving them free for other functions.

These changes mean that an XA device may be made pin compatible with a similar 80C51 derivative if the external bus is not used. Small changes to the external hardware must be made if the external bus is in use. Internally programmable bus cycle timing control on the XA allows programming the duration of all of the bus cycles, allowing nearly all memory and peripheral devices to be used on the XA bus without the need for an external WAIT state generator or any other additional circuitry.

INSTRUCTIONS REVISITED

The earlier mentioned goal of the XA to map nearly every 80C51 instruction to a single XA instruction was met. Just one 80C51 instruction cannot be replaced by single XA instruction. That instruction is XCHD (exchange digit), a seldom used 80C51 instruction. This unusual instruction exchanges the lower nibble of the 80C51 accumulator with a nibble at an internal RAM address pointed to by byte register R0 or R1. The XA would have required additional special circuitry in order to support this operation. As a result, it was decided to allow a multi-instruction sequence in this case, since the instruction is rarely used. The sequence used to replace XCHD is:

```
PUSH  R4H          ; save temporary register.
MOV   R4H, (Ri)    ; get second operand.
RR    R4H, #4      ; swap one byte.
RR    R4L, #4      ; swap second byte (the "A" register).
RL    R4, #4       ; swap word, result is swapped nibbles in A
                     and R4H.
MOV   (Ri), R4H    ; store result.
POP   R4H          ; restore temporary register.
```

Some additional code may be needed if an application requires this sequence to be un-interruptable for some reason. All other 80C51 instructions translate one-to-one to XA instructions. Since the XA instruction set and memory model are a superset of the 80C51, and since most mnemonics and names were kept the same, 80C51 code translated for the XA looks nearly the same as the original. Some examples are shown below.

Table 1. Examples of 80C51 to XA Source Code Translation

TYPE OF OPERATION	80C51 SOURCE CODE		XA SOURCE CODE	
Move immediate to SFR.	MOV	TCON,#00h	MOV.B	TCON,#00h
Move direct address to accumulator.	MOV	A,TstDat	MOV.B	R4L,TstDat
Move register to register.	MOV	R5,A		
Arithmetic with 2 registers.	ADD	A,R1	ADD.B	R4L,R0H
Arithmetic with register and immediate.	SUBB	A,#'0'	SUBB.B	R4L,#'0'
Increment a register.	INC	R0	ADDS.B	R0L,#1
Test a register.	CJNE	A,#'0',Cmd1	CJNE.B	R4L,#'0',Cmd1
Clear a bit.	CLR	RxFlag	CLR	RxFlag
Set a bit.	SETB	EX1	SETB	EX1
Test a bit.	JNB	RcvRdy,Wait	JNB	RcvRdy,Wait
Subroutine call.	ACALL	Test	CALL	Test
Subroutine return	RET		RET	
Push register onto stack.	PUSH	ACC	PUSH.B	R4L
Pop register from stack.	POP	ACC	POP.B	R4L

Details of instruction translation for the entire 80C51 instruction set are available in the Philips XA User Guide.

One side effect of source code compatibility of the XA with the 80C51 is that the number of bytes required to encode some instructions changes between the two processors. In most cases, this is not a major concern, however it does raise issues with the translated code for some situations. A simple example of this is that a conditional branch could have the target address move out of range when translated code is re-assembled. This should be a rare occurrence since the range of short relative branches on the XA has been doubled to 256 bytes forward or backward. The same issue does not exist for farther jumps and calls since the XA extends that range to beyond the entire 80C51 address range.

The precise length of a branch instruction is of concern in certain cases, such as a table of jump instructions entered using the JMP @A+DPTR instruction of the 80C51. The XA instruction set includes this jump, but does not include a 2-byte replacement for the 80C51 AJMP instruction which is often used in jump tables. The user will have to make small changes to the indexing into such a table if it is translated to run on the XA.

A similar issue can arise for a translation of the 80C51 instruction MOVC A,@A+PC, since the distance from this instruction to the lookup table that it is accessing may change. The solution is the same as for JMP @A+DPTR: some user intervention to adjust the table index.

User intervention will also be needed in any case where the timing of instructions in the original 80C51 code is of importance. The XA reduces the execution time of each instruction to the minimum possible with its internal hardware implementation. Also, instructions are normally fetched into a small queue prior to being needed to continue execution, which can lend additional uncertainty to execution times. The execution time of loops or the time between particular instructions can be calculated and adjusted by the use of NOPs, delay loops, or other means of matching timing. Also, any variable execution timing of the same code due to it being entered in different ways can be handled with certain coding techniques. An example would be a loop that is entered by "falling through" the preceding code on the first instance and branching back to be repeated on subsequent occasions. The branch back takes extra time not seen on the first entrance to the code due to the necessity of "flushing" the queue on a branch. The solution in this case is to add a branch instruction prior to the loop branching to the first instruction of the loop. Then, each cycle through the loop acquires the same timing. Of course, a simple source code translator cannot sense such cases and attempt to deal with them automatically.

AN EXAMPLE

As an example of translating 80C51 source code into XA source code, an actual piece of 80C51 code from a working application was taken and translated using the rules that were presented above. The results of the simple one-to-one translation are shown below.

Table 2. Sample 80C51 Routines Translated for the XA

Original 80C51 source code:

```
; Sets up UART and Timer (for baud rate generation), prints a string,
; and prints a hexadecimal value.

Start:  MOV     SCON,#42h       ; Set UART for 8-bit variable rate.
        MOV     TMOD,#20h       ; Set Timer1 for 8-bit auto-reload.
        MOV     TCON,#00h       ; Stop timer 1 and clear flag.
        MOV     TL1,#0FDh       ; Set timer for 9600 baud @ 11.0592 MHz.
        MOV     TH1,#0FDh       ; Set reload register for same rate.
        MOV     A,PCON          ; Make sure SMOD bit in PCON is
        CLR     ACC.7           ;  cleared for this baud rate.
        MOV     PCON,A
        SETB    TR1             ; Start timer

        MOV     DPTR,#Msg1      ; Send a stored message.
        ACALL   Msg

        MOV     A,P1            ; Send Port 1 value as hexadecimal.
        ACALL   PrByte
        .       .
        .       .
        .       .

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

; Print byte routine: print ACC contents as ASCII
;  hexadecimal.

PrByte: PUSH    ACC
        SWAP    A
        ACALL   HexAsc
        ACALL   XmtByte
        POP     ACC
        ACALL   HexAsc          ; Print nibble in ACC as ASCII hex.
        ACALL   XmtByte
        RET

; Hexadecimal to ASCII conversion routine.
; Converts a nibble to ASCII hex.

HexAsc: ANL     A,#0FH
        JNB     ACC.3,NoAdj
        JB      ACC.2,Adj
        JNB     ACC.1,NoAdj
Adj:    ADD     A,#07H
NoAdj:  ADD     A,#30H
        RET
```

Translated XA source code:

```
Start:  MOV.B   SCON,#42h
        MOV.B   TMOD,#20h
        MOV.B   TCON,#00h
        MOV.B   TL1,#0FDh
        MOV.B   TH1,#0FDh
        MOV.B   R4L,PCON
        CLR     R4L.7
        MOV.B   PCON,R4L
        SETB    TR1

        MOV.W   R6,#Msg1
        CALL    Msg

        MOV.B   R4L,P1
        CALL    PrByte
        .       .
        .       .
        .       .

PrByte: PUSH.B  ACC
        RL.B    R4L,#4
        CALL    HexAsc
        CALL    XmtByte
        POP.B   ACC
        CALL    HexAsc
        CALL    XmtByte
        RET

HexAsc: AND.B   R4L,#0FH
        JNB     R4L.3,NoAdj
        JB      R4L.2,Adj
        JNB     R4L.1,NoAdj
Adj:    ADD.B   R4L,#07H
NoAdj:  ADD.B   R4L,#30H
        RET
```

Original 80C51 source code:

```
; Message string transmit routine.

Msg:    PUSH    ACC
        MOV     R0,#0           ; R0 is character pointer (string
MsgL:   MOV     A,R0            ;  length is limited to 256 bytes).
        MOVC    A,@A+DPTR       ; Get byte to send.
        CJNE    A,#0,Send       ; End of string is indicated by a 0.
        POP     ACC
        RET

Send:   ACALL   XmtByte         ; Send a character.
        INC     R0              ; Next character.
        SJMP    MsgL

Msg1:   DB      0Dh,0Ah,0Dh,0Ah
        DB      'Port 1 value = ', 0

; Wait for UART ready, then send a byte.

XmtByte: JNB    TI,$
        CLR     TI
        MOV     SBUF,A
        RET
```

Translated XA source code:

```
Msg:    PUSH.B  ACC
        MOV.B   R0L,#0
MsgL:   MOV.B   R4L,R0L
        MOVC.B  A,[A+DPTR]
        CJNE.B  R4L,#0,Send
        POP.B   ACC
        RET

Send:   CALL    XmtByte
        ADDS.B  R0L,#1
        BR      MsgL

Msg1:   DB      0Dh,0Ah,0Dh,0Ah
        DB      'Port 1 value = ',0

XmtByte: JNB    TI,$
        CLR     TI
        MOV.B   SBUF,R4L
        RET
```

The translated XA code looks very much like the 80C51 source code and can easily be read by anyone familiar with the original program. Statistics for this example are shown in the following table.

Table 3. Statistics on Sample 80C51 to XA Code Translation

STATISTIC	80C51 CODE	XA TRANSLATION	COMMENTS
Bytes to encode	107	151	– Includes NOPs added for branch alignment on XA.
Clocks to execute	840	212	– Raw execution time for instructions in code, without flow analysis. Conditional branch times calculated as if half taken, half not taken.
Time to execute @ 20 MHz	42 sec	10.6 sec	– A 4x speed improvement for a simple translation with no optimization.

SOME XA ENHANCEMENTS

The subject of this article has been how the new Philips XA microcontroller architecture supports upward compatibility with the 80C51. The XA adds quite a bit to the equation beyond mere 80C51 compatibility, which has barely been touched upon here. In addition to high performance and very compact instruction encoding, the XA is specifically designed for high level language support for compilers such as C, has many features to support multi-tasking, with protected features and separate memory spaces, many 32-bit operations in addition to general 16-bit arithmetic, and greatly enhanced interrupt processing, to name a few. A complete description of all of these features and many more may be found in the XA User Guide and data sheets for specific parts.

THE UPWARD SPIRAL

Many openings have been left in the XA architecture for even more enhancements in the future, such as full pipelining, complete 32-bit operation support, or a faster peripheral bus. The XA is the foundation of a new microcontroller derivative family in a manner similar to the very popular 80C51 family. Many other advanced microcontroller architectures have been brought to market since the 80C51 was designed years ago. But until now, none has allowed the enormous quantities of 80C51 code that users have on file to be re-used with minimal effort on a state-of-the-art 16-bit processor. With the Philips XA, that is now possible, while getting the benefit of a modern 16-bit processor with few compromises.

XA benchmark vs. the MCS251 AN705

BACKGROUND

A computer benchmark is a "program" that is used to determine relative computer core performance by evaluating benchmark execution time of the core. In a brainstorm sessionon microcontrollers for automotive applications, an assembler functional *benchmark for engine management*, which is a typical example of embedded high-end microcontrol was created. This report summarizes the functions implemented in assembler language of the compared controllers: Intel MCS251, and Philips XA. The total execution times of a program "engine cycle" (engine stroke) are calculated and the required program code is estimated for each controller.

Evaluation of performance in a High Level Language (HLL) like C would be preferable, but it is difficult to realize as "the best" compilers for all cores involved then should be used.

This document outlines code density and execution times of the XA, based on the most recent information. The execution times are given in terms of both clock cycles and time units. Although the XA can run at a much higher speed than the MCS251, for the sake of fairness, both cores are evaluated running at 16.00 MHz. This is a reasonable assumption for comparing the cores at the same level of technology.

Because of the pipeline architectures of the MCS251 and the XA, the benchmarks are run on actual silicon.

BENCHMARK RESULTS AND CONCLUSIONS

Relative performance on a line

The table below presents the most important result of the assembler benchmark evaluation. It pictures the relative performance of the compared core instruction set on a scale where XA=1.0. Also appended is the performance charts–execution and code density of all the processors.

Total exec.times/core(µs) for all routines (with *occurrences)

	938.75	359.86
Performance ratio	**MCS251**	**XA**
MCS251	1.0	2.61
XA	0.383	1.0

Table 1. XA instruction set execution times and bytes/function

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.0	12	18
ADD/SUB	50	0.375	18.75	4
CMP 24b	13	1.25	16.25	9
CAN 16b	80	0.562	44.96	5
INTPLIN	20	2.04	40.8	42
BRANCH	1		158.13	

XA totals : 299.89 µs
including 20% statistics : 359.86 µs

Table 2. MCS251 instruction set execution times and bytes/function

FUNCTION	OC*	MCS251		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.53	18.36	2
FDIV	4	30.125	120.6	25
ADD/SUB	50	0.641	32.05	2
CMP 24b	13	3.375	43.88	12
CAN 16b	80	1.625	130	6
INTPLIN	20	6.12	122.4	60
BRANCH	1		315.0	

MCS251 totals : 782.29 µs
including 20% statistics : 938.75 µs

Table 3. Total benchmark execution time results

MICROCONTROLLER CORE	EXECUTION TIME (µs)
Philips XA-G3	359.86
Intel MCS251	938.75

Benchmark limitations

Like all benchmarks, the automotive engine management assembler functional benchmark has some weakness that limit validity of its results.

1. Control in a special (automotive, engine) environment is evaluated.
2. Occurrences of operation overheads are based on estimations.
3. Occurrences of functions are based on estimations.
4. Functions are implemented in assembler, not in a HLL like C.
5. Routines may contain assembler implementation errors.
6. Cores are evaluated at 16.0 MHz

Control in a special environment is evaluated (automotive, engine)

The core performance evaluation is based on a single specialized case. All benchmark implementations are fractions of the automotive engine management PCB83C552 demonstration program.

It can be advocated that the automotive engine control task gives a good example of a typical high demanding control environment, where many >= 16 bit calculations have to be done.

Occurrences of overheads are based on estimations

The assembler functional benchmark is not a full implementation of a program. Arbitrary choosing location for storage of parameters in register file or (external) memory, for instance, has for some instruction set a considerable effect on the total execution time.

For the different core parameter storage is chosen where possible using the core facilities to have minimum access overhead.

Occurrences of functions based on estimations

Occurrences is estimated on basis of experience of the automotive group. In a real implementation of an engine controller accents may shift. As most functions already include some "instruction mix", the effect of changes in occurrences is limited.

Functions are implemented in assembler, not in a HLL like C.

Control programs for embedded systems get larger, have to provide more facilities and have to be realized in shorter development times. The only way to do this is to program in a HLL like C. Efficient C–language program implementation requires different features from microcontrollers than assembly programs. Results of this assembler benchmark evaluation therefore have a restricted value for ranking microcontroller performances for future HLL applications.

Benchmark ranking on basis of HLL like C requires good C–compilers of all the devices involved are needed. The quality of the C–compilers really has to be the best there is : HLL benchmarking measures not only the micro characteristics, but even more the compiler ability to use these qualities. As these are not available for all the micros evaluated, all routines are worked out only in assembly.

All cores are evaluated at 16.0 MHz

A 16.0 MHz internal clock frequency seems a reasonable choice for comparing the cores at the same level of technology:

Assembler functional benchmark for automotive engine management

This benchmark is a functional benchmark: it is a collection of functions to be executed in an automotive engine management program. To implement the assembly functional benchmark for automotive engine management correctly the "rules and details" described in this section have to be followed carefully.

The assembler functional benchmark embraces all activity to be completed in 1 program cycle that corresponds with 1 engine stroke of 2 ms. The benchmark execution time will be calculated as the sum of the products of functions and their occurrence rates in 1 calculation cycle.

Branches are evaluated separately as "branch penalties" have considerable effect of program execution efficiency. Estimated (branch count)*(average branch time) is added to the function execution times.

The relative estimated overhead for statistics does not contribute to the evaluation of speed performance ratios, but they have to be considered when looking at the total execution time required / engine stroke cycle. therefore the real total execution time is multiplied with the statistics overhead factor (1.2*).

NO.	FUNCTION DESCRIPTION	OCCURRENCES
1	16×16 Multiply	12
2	Floating Point divide (16:16)	4
3	Add/Subtract (24)	50
4	Compare (24)	13
5	CAN cmp/mov 10*8	80
6	Linear Interpolation (8*8)	20
7	Program control branches	500
8	Statistics (20%)	1.2 *

Function Parameter Allocation

Most functions are very short in exec. time, so that the function parameter data access method has great effect on the total time. Thus it is to be considered carefully. Both XA and MCS251SB have register files in which variables can be stored.

For the XA and 251SB processors, data is stored in the lower part of register file, or in sfrs for I/O, can be accessed using "direct"addressing, but table data, used e.g. for 3 byte compare, is stored in "external memory". For more complex functions 16*16 multiply, Floating point division and interpolation, data is assumed to be already in registers.

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32–bit result written into a register pair.

Divide (16:16) "floating point"

The floating point division is entered with parameters in registers:

a divisor, a dividend and an "exponent" that determines the position of the fraction point in the result.

Floating point binary 16/16 division is a function that is normally not included in HLL compilers as it requires separate algorithms for exponent control and accuracy is limited. For assembler control algorithms, floating point division can be quite efficient as it is much faster than normal "real" number calculations (where no "floating point accelerator" hardware is available).

Compare 24–bit variables

Note that 24–bit compare is very efficient for "real" 16–bit and 8–bit) controllers, but for automotive engine timers, 24–bit seems a good solution. Compare must give possibility to decide >, < or =. An average branch is included in the function.

CAN move and compares

For service of the CAN serial interface, it is estimated that 40* (2 byte compares + branch) have to be done. Devices with 16–bit bus assumes word access. An average branch is included in the CAN compare function.

Linear Interpolation (8*8)

The interpolation routine is entered with 3 register parameters:

1. Table position address
2. X fraction
3. Y fraction

The routine first interpolates using the X fraction the values of F(x.x, y) between F(x,y) ....V(x+1, y) and of F(x.x, y+1) between F(x, y+1) .... F(x+1, y+1). From F(x.x, y) and F(x.x, y+1) the value of F(x.x, y.y) is interpolated using the fraction of y.

The table is organized as 16 linear arrays of 16 x–values, so that an V(x,y) can be accessed with table origin address +x+16*y = "Table Position Address". In x–direction the interpolation can be done between the "Table Position" value and next position (+1). Interpolation in y–direction is done by looking at "Table Position" + 16.

For linear interpolation time the 2–dimensional interpolation time and byte count are divided by 3 to include some "overhead" into linear interpolation.

Program Control Overheads

For a given algorithm, the "program control overhead" consisting of a number of decisions (=branches) and subroutine calls is independent of the instruction set used, except for cases where functions can be replaced by complex instructions. The most important exception cases, MPY words and Floating Point Division are handled in this benchmark separately.

Most 16–bit cores use more pipeline stages so that taken branches add branch time penalty for these CPU's due to pipeline flush. This effect can be found in the branch execution time tables.

More efficient data operations and pipeline penalty of the more complex instruction set of 16–bit cores lead to considerable higher relative time used for branch instructions.

To incorporate the influence of branches in the benchmark the number of branches to be included must be estimated. For byte and bit routines, branches occur more frequent. Average branch time of 25% may be a good guess. For the automotive engine management benchmark that executes in approx. 5000/µS (on 8051) results in +/– 1250 /µS or 625 branches. As a part of the branches already taken account for in the compare functions the number of additional program control branches is estimated 500 branches.

To estimate the average branch execution time, an estimated relative occurrence of the branch types has to be made.

Table 4. Estimated relative occurrence of the branch types

	TYPE	RELATIVE	ABSOLUTE OCCURRENCE
Absolute Jumps	AJMP/JMP	20%	100
Subroutine calls	ACALL/JSR	20%	100
Jump on condition (rel)	Bcc/Jcc	40%	200
Jump on bit (rel)	JB/JBN	20%	100

Statistic Routine Overheads

Statistic routines are estimated as relative program overheads, only to get an indication of the required total processing time in a real engine management application. "Statistics" are mainly arithmetic routines to determine table corrections. They use about 20% of the total time.

XA BENCHMARK RESULTS

The following analysis assumes worst case operation. At any point in time, only 2 bytes are available in the instruction Queue. An instruction longer than 2 bytes requires additional code read cycle.

APPENDIX 1

XA Function Implementations

XA reference: *XA User's Manual 1994*

A1.1: 16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair.

```
MUL.w                R0, R1          ; result is in register pair R1:R0
```

2 Bytes, 12 clocks ==> 0.75 µs

A1.2: Floating Point 16x16 Divide:

```
;The floating point division is entered with parameters in registers:
;Arguments:     R4 = Dividend (extend into R5 for 32 bits)
;               R6 = Divisor Mantissa
;               R0 = Divisor  Exponent

FPDIV:
        ADDS            R6, # 0         ; Add short format
        BEQ             L1              ; divby 0 chk - if z=1, go to L1

SGNXTD_AND_SHFT:
        SEXT.W          R5              ; Sign extend into R5
        ASL             R4, R0L         ; 13 position shifts (average)

DIV:                                    ;
        DIV.d           R4, R6          ; Divide 32x16 signed
        BOV             L1              ; Branch on Overflow
        RET                             ; Normal termination

L1:
        MOVS            R4, # -1        ; Overflow - Max Result
        RET
```

18 Bytes, 48 clocks ==> 3.0 µs

A1.3: Extended 32-bit subtract

```
;       R5:R4 = Minuend
;       R3:R2 = Subtrahend

        SUB.w           R4, R2
        SUBB.w          R5, R3
```

4 Bytes, 6 clocks ==> 0.375 µs

A1.4: Compare 24-bit Variables

An average branch is included after compare.

The table data, used for 3 byte compare, is stored in "memory".

```
CMP:
        CMP.B   R1L, R2L                ;
        BNE     L1                      ;

L1:
        CMP.W   R0, mem1                ;
        BGT     LABEL1                  ;
                                        ;

LABEL1:
;       xx -> GT or LT or EQ
```

9 Bytes, 20 clocks (average - branch always taken and not taken) ==> 1.25 µs

A1.5: CAN Compare and Move

Application: For service of CAN (Controller Area Network) serial Interface it is estimated that 80* (2 byte compares + branch) have to be done. One parameter is in register, the other in internal memory.

```
CAN:
        CMP             R0, mem0        ; mem0 = $10H           3
        BGT             LABEL           ;                       2

LABEL:
```

5 Bytes, 9 clocks (average) ==> 0.563 µs

A1.6: Linear Interpolation

```
Arguments:
                R0 = Table Base (assumed < 400 Hex)
                R2 = Fraction 1
                R4 = Fraction 2
                R6 = Result

LIN_INT:
                MOV             R2, [R5+]       ;               2
                MOV             R0, [R5]        ;               2
                SUB             R0, R2          ;               2
                MULU.w          R2, R6          ;               2
                MOV.b           R0H, R0L        ;               2
                MOVS.b          R0L,#0          ;               2
                ADD             R2, R1          ;               2
                ADD             R5, #15         ;               2
                MOV             R0, [R5+]       ;               2
                MOV             R4, [R5]        ;               2
                SUB             R4, R0          ;               2
                MULU.w          R4, R6          ;               2
                MOV.b           R0H, R0L        ;               2
                MOVS.b          R0L,#0          ;               2
                ADD             R0, R4          ;               2
                SUB             R0, R2          ;               2
                MULU.w          R0, R5          ;               2
                MOV.b           R0H, R0L        ;               2
                MOVS.b          R0L,#0          ;               2
                ADD             R2, R0          ;               2
                RET                             ;               2
                                                ;               42
```

42 Bytes, 98 clocks ==> 6.125 µs
Linear Interpolation (2 dim. time / 3) = 42 bytes, 2.04 µs

A1.8: Program Overhead

Branches are assumed taken 70% of the time, all addresses are external. Code is assumed a run–time trace, code size cannot be calculated.

TYPE	OCCURRENCE	XA	BYTES
JMP rel16	100	6 600	3 300
CALL rel16	100	4 400	3 300
Bxx rel8	200	5.1 1020	2 400
JNB bit,rel8	100	5.1 510	2 200
total cylces µsec		2,530 158.13	1,200

A1.9: XA Totals

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.0	12	18
ADD/SUB	50	0.375	18.75	4
CMP 24b	13	1.25	16.25	16
CAN 16b	80	0.562	44.96	8
INTPLIN	20	2.04	40.8	14
BRANCH	1		158.3	1200

```
XA total/µs:                299.89 µs
including 20% statistics:   359.86 µs
```

Note:

An assumption is made that XA code is in first 64K (PZ), that is, only 64K address space is used.

APPENDIX 2

MCS251 Implementations

MCS251 reference: *"MCS251SB Embedded microcontroller users manual"*, February 1995.
All data are taken using the Kiel Development Board using a 251SB 16.0 MHz part.

A2.1: MCS251SB 16×16 Multiply

```
;The MCS251 can do only unsigned multiply. So, there will be some overhead for testing
;the sign of the result.

MUL             R0,R1

;Total: 2 bytes, 24 clocks ==> 1.5 µs
```

A2.2: Floating point division 16:16

```
; Arguments:    WR4 = 16-bit Dividend
;               WR6 = 16-bit Divisor Mantissa
;               WR0 = Divisor Exponent

FPDIV:
        ADD     WR2,#0                  ;                       4
        JE      L1                      ;                       2
                                        ;
SGNXTD_AND_SHFT:
        MOVS    WR6,R5                  ;                       2

SHFT_LOOP:
        SLL     WR4                     ;NO ARITH SLL ?         2
        DJNZ        R0,SHFT_LOOP    ;DOES 1 BIT AT A TIME       3

DIVISION:
        DIV     WR4,WR2                 ;                       2
        JB      OV,L1                   ;IF OVFLW BIT IS SET    4
        RET                             ;NORMAL TERMN.          1

L1:                                     ;
        MOV     WR4, #-1                ; OVFL - MAX RESULT     4 (not exc)
        RET                             ;                       1

; Totals: 25 bytes, 482 clocks ==> 30.125 µs
```

A2.3: Add/Sub

```
;       DR0 = Minuend
;       DR4 = Subtrahend

SUB     DR0,DR4                         ;

; Totals: 2 bytes, 10 clocks ==> 0.625 µs
```

A2.4: Compares 24 (=32) bit

```
COMPARE:
        MOV     WR0,60H                 ;memory                 3
        MOV     WR2,50H                 ;memory                 3
        CMP     DR0,DR4                 ;                       2
        JE      CMP_EQUALS              ;                       2
        SJMP    CMP_APPROX              ;                       2
CMP_EQUALS:
CMP_APPROX:

; Totals: 12 bytes,   54 clocks (branch average)  ==> 2.375 µs
```

A2.5: CAN move and compares (16-bit)

```
COMPARE:
        CMP     WR0,mem0                ;mem0 = 40H             4 bytes, 6 clocks
        JNE     THERE                   ; 2 bytes 2t/8nt
THERE:

; Totals: 6 bytes, 10 clocks ==> 0.625 µs
```

A2.6: 2-dimensional interpolation

```
;Arguments:
;               XAR0 = Table Base (assumed < 400 Hex)
;               XAR2 = Fraction 1
;               XAR4 = Fraction 2
;               XAR6 = Result
;               XAR1 = temporary1
;               XAR0 = temporary2
;               XAR5 = temporary3
;
;               WR0 = Table Base (assumed < 400 Hex)
;               WR2 = Fraction 1
;               WR4 = Fraction 2
;               WR6 = Result
;               WR8 = temporary1 = XAR1
;               WR10 = temporary2 = XAR0
;               WR12 = temporary3 = XAR5

LIN_INT:
        MOV     WR6,@WR10               ; 3     6
        ADD     WR10,#2                 ; 4     6
        MOV     WR8,@WR10               ; 3     6
        SUB     WR8,WR6                 ; 2     4
        MUL     WR6,WR2                 ; 2     22
        MOV     R2,R1                   ; 2     2
        MOV     R1,#0                   ; 3     4
        ADD     WR6,WR8                 ; 2     4
        ADD     WR10,#15                ; 4     6
        MOV     WR8,@WR10               ; 3     6
        ADD     WR10,#2                 ; 4     6
        MOV     WR12,@WR10              ; 3     6
        SUB     WR12,WR8                ; 2     4
        MUL     WR12,WR2                ; 2     22
        MOV     R2,R1                   ; 2     2
        MOV     R1,#0                   ; 3     4
        ADD     WR8,WR12                ; 2     4
        SUB     WR8,WR6                 ; 2     4
        MUL     WR8,WR4                 ; 2     22
        MOV     R2,R1                   ; 2     2
        MOV     R1,#0                   ; 3     4
        ADD     WR6,WR8                 ; 2     4
        RET                             ; 1     12

; Totals:  58 bytes, 274 clocks ==> 17.125 µs
; Linear Interpolation (2 dim. time / 3) = 60 bytes, 5.71 µs
```

A2.7: MCS251 Program Overhead

TYPE	OCCURRENCE	MCS251	BYTES
LJMP addr16	100	8 800	4 400
LCALL addr16	100	18 1800	3 300
JLE rel	200	6.8 1360	2 400
JNB rel	100	10.8 1080	4 400
total cylces µsec		5040 315.0	1500

A2.8: MCS251 Totals

FUNCTION	OC*	MCS251		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.53	18.36	2
FDIV	4	30.125	120.6	25
ADD/SUB	50	0.641	32.05	2
CMP 24b	13	3.375	43.88	12
CAN 16b	80	1.625	130	6
INTPLN	20	6.12	122.4	60
BRANCH	1		315.0	

```
MCS251 total/µs:            782.29 µs
including 20% statistics:   938.75 µs
```

EXECUTION TIME PERFORMANCE

Actual execution times/function

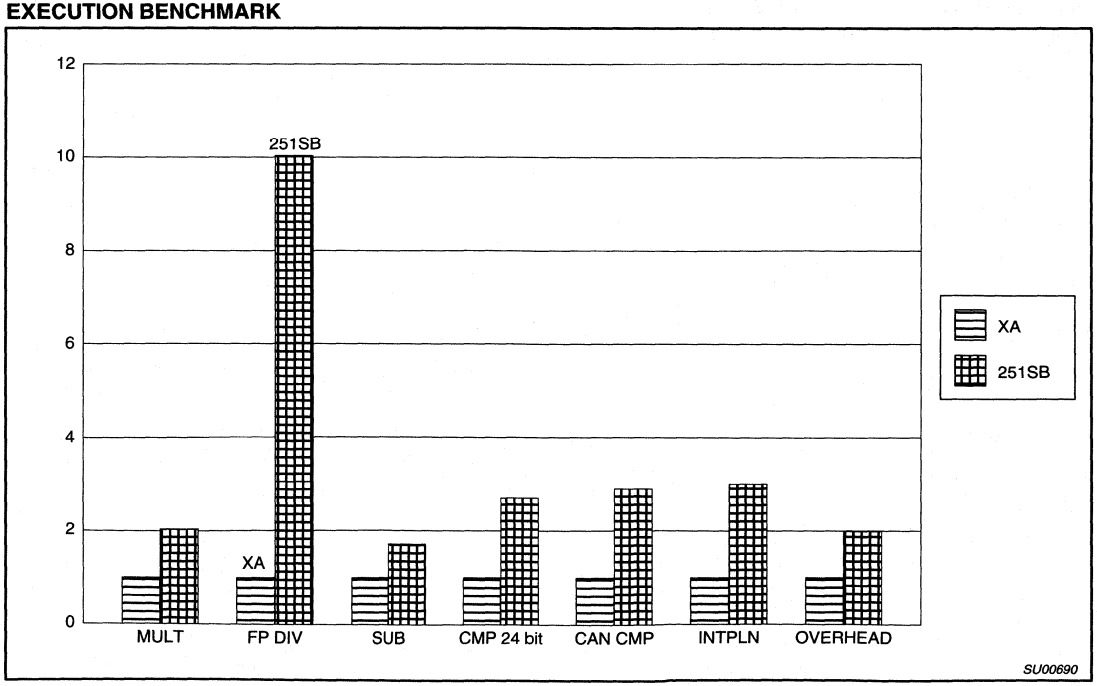
FUNCTIONS	XA	251SB
MULT	0.75	1.53 *
FP DIV	3	30.125
SUB	0.375	0.641
CMP 24 bIT	1.25	3.375
CAN CMP	0.562	1.625
INTPLN	2.04	6.12
OVERHEAD	158.13	315

* Only for unsigned, extra overhead for sign needs to be added.

Normalized timings/function

FUNCTIONS	XA-G3	251SB
MULT	1	2.04
FP DIV	1	10.04
SUB	1	1.71
CMP 24 bIT	1	2.7
CAN CMP	1	2.89
INTPLN	1	3
OVERHEAD	1	1.99

EXECUTION BENCHMARK



BENCHMARK OF CODE DENSITY

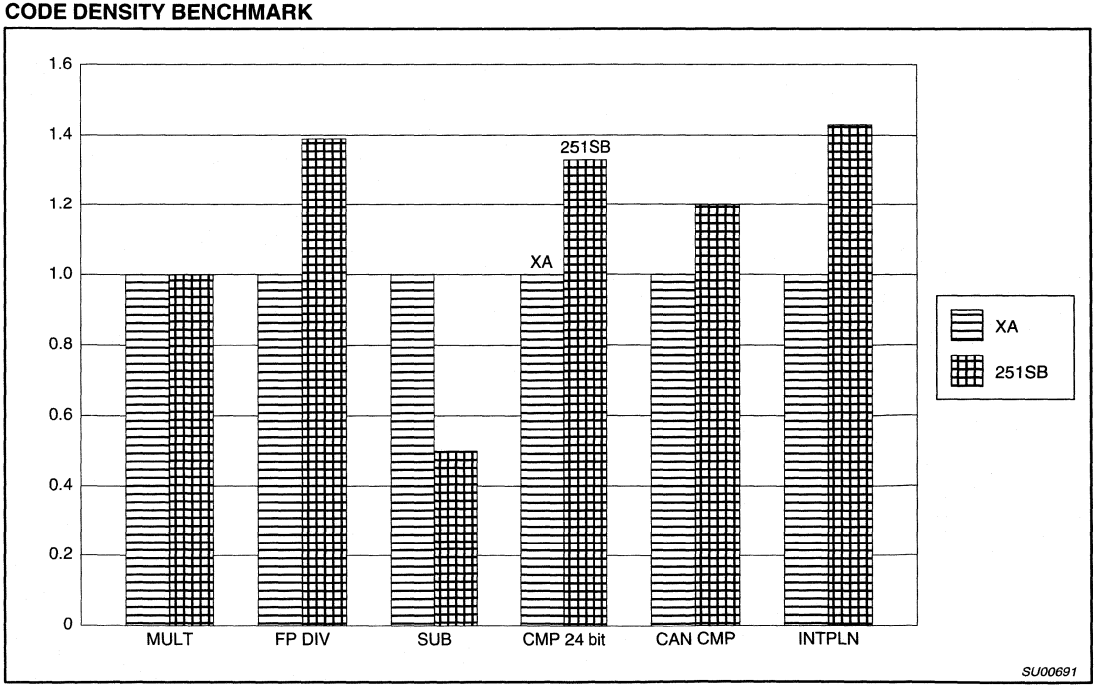
Actual code density performance

FUNCTIONS	XA-G3	251SB
MULT	2	2
FP DIV	18	25
SUB	4	2
CMP 24 bIT	9	12
CAN CMP	5	6
INTPLN	42	60

Normalized w.r.t. XA

FUNCTIONS	XA-G3	251SB
MULT	1	1
FP DIV	1	1.39
SUB	1	0.5
CMP 24 bIT	1	1.33
CAN CMP	1	1.2
INTPLN	1	1.43

CODE DENSITY BENCHMARK



BM1.ASM

```
$include xa-g3.equ
$include bm.inc

 ;16x16 signed multiply



        org $0
        dw $8f00,start
;

        org $200

start:

         setp_15
         MUL.w   R0, R1   ;      2
         rstp_15
         br      start


;Totals = 2 Bytes, 12 clocks (0.75 uS)
```

BM2.ASM

```
;$listing_min
$include xa-g3.equ
$include bm.inc


        org $0
        dw $8f00,start                          Bytes      Clocks
;

        org $200
;r6= divisor mantissa
;r0=divisor exponent
;r4=dividend (extended to r5 for 32-bits)

start:
        movs.b  r61,#2          ; some value > 0
        mov.b   r01,#13         ;
        mov.w   r4,#$200        ;
        mov.w   r6,#$100        ;
        call    FPDIV           ;
        br      start


FPDIV:
        setp_15
        ADDS    R6, # 0         ; Add short format              2
        BEQ     L1              ; divby 0 chk                   2
                                ;- if z=1, go to L1
                                ;
SGNXTD_AND_SHFT:                ;
        SEXT.W  R5              ; Sign extend into R5           2
        ASL     R4, R0L         ; 13 position shifts (average)  2
                                ;
DIV:                            ;
        DIV.d   R4, R6          ; Divide 32x16 signed           2
        BOV             L1      ; Branch on Overflow            2
        rstp_15                 ;

        RET                     ;                               2
                                ;
L1:                             ;
        MOVS    R4, # -1        ; Overflow - Max Result         2
        rstp_15
        RET                                                     2

;Totals = 18 Bytes, 48 clocks (averages for branches) i.e 3.0 uS at 16.0 MHz
```

BM3.ASM

```
;$listing_min
$include xa-g3.equ
$include bm.inc


        org $0
        dw $8f00,start
;                                                   Bytes       Clocks

        org $200

start:
        MOV             R4,#$200
        MOV             R5,#$210
        MOV             R2,#$100
        MOV             R3,#$110

        setp_15

;Extended 32-bit subtract

        SUB     R4, R2          ;       2
        SUBB    R5, R3          ;       2

        rstp_15
        br      start

;Totals= 4 Bytes and 6 clocks  (0.375 uS) at 16.00 MHz
```

BM4.ASM

```
$include xa-g3.equ
$include bm.inc

mem1    equ             $20

        org $0
        dw $8f00,start
;;Compare 24-bit Variables                                  Bytes      Clocks

        org $200

start:
        mov     R2L,#$40        ;  one parameter is register
        mov     mem1,#$1000     ;  and one in memory
        mov     R1L,#$50        ;
        mov     R0,#$5000       ;

CMP:
        setp_15
        CMP.B   R1L, R2L        ;               2
        BNE     L1              ;               2
                                ;


L1:
        CMP.W   R0, mem1        ;               3
        BGT     LABEL1          ;  average      2

LABEL1:
;       xx -> GT or LT or EQ
          rstp_15
          br      start

;Totals= 9 Bytes and 20 clocks i.e 1.25 uS at 16.00 MHz
```

BM5.ASM

```
$include xa-g3.equ
$include bm.inc


;A1.5
;CAN Move and Compare
;one parameter in register, the other in memory

mem0            equ             $10


        org $0
        dw $8f00,start
;
  Bytes Clocks

        org $200
start:
        mov     mem0,#$100
        mov     R0,#$50

CMPR:
        setp_15
        CMP     R0, mem0        ;       3
        BGT     LABEL           ;       2

LABEL:

        rstp_15
        br      start


;Totals = 5 Bytes and 9 clocks (average for branches)
;or 0.563 uS at 16.00 MHz
```

BM6.ASM

```
$include xa-g3.equ
$include bm.inc

mem1    equ             $20

        org $0
        dw $8f00,start


;Linear Interpolation

;Arguments:
;               R4 = Table Base (assumed < 400 Hex)
;               R6 = Fraction 1
;               R5 = Fraction 2
;               R2 = Result




        org $200
start:
        mov     r7,#$100        ;safe
        movs    scr,#1          ;page 0
        mov     R5,#$120
        mov     R2,#$12F
        mov     R4,#$80
        mov.w   $120,#$45
        call    LIN_INT
        rstp_15
        br      start

LIN_INT:
        setp_15                         ;
        MOV     R2, [R5+]               ;       2
        MOV     R0, [R5]                ;       2
        SUB     R0, R2                  ;       2
        MULU.w  R2, R6                  ;       2
        MOV.b   R0H, R0L                ;       2
        MOVS.b  R0L,#0                  ;       2
        ADD     R2, R1                  ;       2
        ADD     R5, #15                 ;       2
        MOV     R0, [R5+]               ;       2
        MOV     R4, [R5]                ;       2
        SUB     R4, R0                  ;       2
        MULU.w  R4, R6                  ;       2
        MOV.b   R0H, R0L                ;       2
        MOVS.b  R0L,#0                  ;       2
        ADD     R0, R4                  ;       2
        SUB     R0, R2                  ;       2
        MULU.w  R0, R5                  ;       2
        MOV.b   R0H, R0L                ;       2
        MOVS.b  R0L,#0                  ;       2
        ADD     R2, R0                  ;       2
        RET                             ;       2
;Totals = 42 bytes and 98 clocks i.e 6.125 us at 16.00 MHz
;For 2-dim interpolation, exec. time = 6.13/3 = 2.04 us
```

BM1.A51

```
$TITLE(bm1.a51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM1 SEGMENT CODE
        RSEG ?PR?BM1

; 16x16 '251 Multiply
;
;
;
test:
        T_START
        MUL     WR0,WR2          ;   2
        T_END
;stall:
        sjmp     test

;Totals:  2 bytes, 24.5 clocks ==> 1.53 uS

        END
```

BM2.A51

```
$TITLE(bm2.a51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM2 SEGMENT CODE
        RSEG ?PR?BM2

; 251 Floating Point 16x16 Divide, 16:16
;
; Note: the '251 may have a shift-by-n, but I can;t seem to find it!
; If there is one, the '251 results would likely improve.
;
; Arguments:    WR4 = 16-bit Dividend
;               WR2 = 16-bit Divisor Mantissa
;               WR0 = Divisor Exponent

test:
        mov  r0,#13
        mov  wr4,#200H
        mov  wr2,#100H
        call FPDIV
                                        ; return here
stall:
        jmp test


FPDIV:
        T_START
        add     wr2,#0                  ;                       4
        je      l1                      ;                       2
                                        ;
SGNXTD_AND_SHFT:
        movs    wr6,r5                  ;                       2
SHFT_LOOP:
        sll     wr4                     ;No arith sll ?         2
        djnz    r0,SHFT_LOOP            ;does 1 bit at a time   3

DIVISION:
        div     wr4,wr2                 ;                       2
        jb      OV,L1                   ;if ovflw bit is set    4
        T_END
        ret                             ; Normal termination    1
L1:                                     ;
        mov     wr4, #-1                ; Overflow - Max Result 4

        T_END
        ret

        END


;Totals: 25 bytes, 482 clocks ==>  20.125 uS

;Note : The shift instructions are taking 10 clocks in the MCS251 part
;instead of 2 clocks as specified in the manual. No idea why !!!
;For sign divide in MCS 251, there will be a considerable overhead involved
```

BM3.A51

```
$TITLE (BM3.A51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM3 SEGMENT CODE
        RSEG ?PR?BM3

;; Extended 32-bit subtract
; Z = X - Y
;
; entry: DW(X) in DR0
;         DW(Y) in DR4
; exit:  DW(Z) in DR0
;
;
SUBTR:
        T_START
        SUB     DR0,DR4                 ;    2
        T_END
        sjmp    SUBTR
        END

; Totals: 2 bytes, 10.25 clocks ==> 0.641 uS at 16.00 MHz
```

BM4.A51

```
$TITLE (BM4.A51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM4 SEGMENT CODE
        RSEG ?PR?BM4

; Compare 24-bit Variables

; The '251 really uses fewer instruction for a 3 byte compare because it
;
test:
        mov     wr4,#4000H
        mov     wr6,#2000H
        mov     60H,wr6
        mov     50H,wr4

compare:
        T_START
        MOV     WR0,60H         ;               3
        MOV     WR2,50H         ;               3
        CMP     DR0,DR4         ;               2
        JE      CMP_EQUALS      ;               2
        SJMP    CMP_APPROX      ;               2

; Totals: 12 bytes, 54 clocks (average) ==> 3.375 uS
CMP_EQUALS:
CMP_APPROX:
        T_END
        sjmp    compare
        END
```

BM5.A51

```
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM5 SEGMENT CODE
        RSEG ?PR?BM5

; CAN COMPARE
;1 parameter in register, the other in memory

test:
         MOV     WR0,#2000H
         MOV     WR4,#3000H
         MOV     40H,WR4
compare:
        T_START
        CMP     WR0,40H          ;    4
        JNE     THERE            ;    2

THERE:
        T_END
        jmp test

        end


;
; Totals: 6 bytes, 26 clocks (average branches) ==> 1.625 uS at 16 MHz
;
```

BM6.A51

```
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM6 SEGMENT CODE
        RSEG ?PR?BM6

;;Linear Interpolation

;Arguments:
;               XAR0 = Table Base (assumed < 400 Hex)
;               XAR2 = Fraction 1
;               XAR4 = Fraction 2
;               XAR6 = Result
;               XAR1 = temporary1
;               XAR0 = temporary2
;               XAR5 = temporary3
;
;               WR0 = Table Base (assumed < 400 Hex)
;               WR2 = Fraction 1
;               WR4 = Fraction 2
;               WR6 = Result
;               WR8 = temporary1 = XAR1
;               WR10 = temporary2 = XAR0
;               WR12 = temporary3 = XAR5

;
test:
        call    LIN_INT
        T_END                           ; return here
stall:
        jmp test

LIN_INT:
        T_START

        MOV             WR6,@WR10       ;                       3
        ADD             WR10,#2         ;                       4

        MOV             WR8,@WR10       ;;                      3
        SUB             WR8,WR6         ;;                      2
        MUL             WR6,WR2         ;                       2

        MOV             R2,R1           ;                       2
        MOV             R1,#0           ;                       3

        ADD             WR6,WR8         ;;                      2
        ADD             WR10,#15        ;                       4
        MOV             WR8,@WR10       ;;                      3

        ADD             WR10,#2         ;                       4

        MOV             WR12,@WR10      ;;                      3
        SUB             WR12,WR8        ;;                      2
        MUL             WR12,WR2        ;;                      2
        MOV             R2,R1           ;                       2
        MOV             R1,#0           ;                       4
        ADD             WR8,WR12        ;;                      2
        SUB             WR8,WR6         ;;                      2
        MUL             WR8,WR4         ;;                      2
        MOV             R2,R1           ;                       2
        MOV             R1,#0           ;                       4
        ADD             WR6,WR8         ;;                      2
        RET                             ;

        END

; Totals:  60 bytes, 294 clocks ==>18.36 uS at 16.00 MHz
```

Programmable peripherals using the PSD311 with the Philips XA

AN707

Author: Lane Hauck; with permission from WaferScale Incorporated.

Introduction

The Philips Semiconductors P51XA-G3 is the first of a new breed of fast, inexpensive 16-bit processors designed for high performance, high integration, and family growth. Although the P51XA (XA) family is promoted as a modern version of the venerable 8-bit 8051, it actually outperforms most of today's 16-bit embedded processors by a wide margin.

The XA is available in the usual array of OTP, ROMless and mask ROM versions so the cost/performance benefit that has made WSI PSD3XX chips attractive to embedded system designers applies the to XA. A typical system can be built using the ROMless version of the XA and a PSD311 for less cost than the OTP version of the XA.

Connection of a PSD3XX to the XA is not straightforward, due to the fact that the XA address and data lines are multiplexed in a manner unlike all other CPU chips that the PSD family is designed to support. This application note identifies the interface issues and solves them one by one to achieve an efficient XA–PSD interface.

The WSI PSD3XX devices can be used either with multiplexed address/data buses or with separate address and data buses. Multiplexed buses have the advantage that fewer PSD pins are required for the CPU interface, leaving more PSD pins available for general purpose system use. This application note addresses multiplexed bus connection of the XA and the PSD311.

The XA-PSD Marriage: Almost Perfect

The Philips XA designers took a radical departure from the 8051 bus architecture by bringing out the address lines A0–A3 on dedicated pins. These addresses are not multiplexed, which means that they do not require an ALE pulse to separate the address information from the data information. This allows up to 16 byte fetches on an 8-bit external bus with only one ALE pulse – the address is latched, the first byte is read or written, and then A0–A3 are incremented and the subsequent bytes are accessed.

This non-multiplexing of A0–A3 also allows very quick access of 16-bit operands on an 8-bit bus, because the time required to fetch the second byte can be as low as 20% of the normal ALE-R/W cycle time. This innovative timing allows external 8-bit bus systems to run nearly as fast as external 16-bit bus systems.

The XA gives very precise control (via internal programmable registers) of its bus timing. You can set the width of the ALE signal, and the positions and widths of the RD and WR signals. Given the inherent speed of the XA and the capability to fine-tune its bus timing, a word fetch using an 8-bit external bus can be significantly faster than other 16-bit CPUs that use a 16-bit external bus.

But...

For all the reasons it makes sense to buy the "ROMless" version of a CPU like the 8031 and attach a PSD chip for a lower system cost, it likewise makes sense to use a PSD chip with the ROMless XA. But there's a hitch. PSD3XX chips expect to see the low 8 bits of address and data multiplexed together, i.e., AD7–AD0. But the XA uses a different multiplexing arrangement, as shown in Table 1.

Table 1. Address-Data Multiplexing Schemes

CONVENTIONAL		XA	
A15		A15	
A14		A14	
A13		A13	
A12		A12	
A11		A11	D7
A10		A10	D6
A9		A9	D5
A8		A8	D4
A7	D7	A7	D3
A6	D6	A6	D2
A5	D5	A5	D1
A4	D4	A4	D0
A3	D3	A3	
A2	D2	A2	
A1	D1	A1	
A0	D0	A0	

As illustrated in Table 1, data lines D0 – D7 are multiplexed with A4 – A11 on the XA, not with A0–A7 as the PSD devices expect.

Basic Strategy

Given Table 1, how should the XA buses be connected to a PSD? In principle, it is possible to scramble address and data lines, as long as the scrambling is accounted for in the system design. For example, if you scramble address lines connected to a RAM, the scramble occurs for both writes and reads, so the effect is transparent to the system. However, address scrambling is not transparent in a device like a ROM that stores data at predetermined locations. When a CPU sends out an address to fetch an interrupt vector or execute one step of a program, it expects the data to be at that absolute address, not somewhere else due to scrambled address lines.

The first interface consideration is that the XA data lines must be connected to the corresponding PSD311 data lines. This dictates that XA A4/D0–A11/D7 must be connected to PSD311 AD0–AD7 as shown in the highlighted portion of Table 2.

Table 2. XA–PSD311 Bus Connection

XA	PSD311
A15	A15
A14	A14
A13	A13
A12	A12
A11/D7	AD7
A10/D6	AD6
A09/D5	AD5
A08/D4	AD4
A07/D3	AD3
A06/D2	AD2
A05/D1	AD1
A04/D0	AD0
A03	A11
A02	A10
A01	A09
A00	A08

As shown in Table 2, the upper four address lines A15–A12 are connected straight across. Because the data lines D7–D0 must line up, the CPU address lines A11–A4 must be connected to the PSD A7–A0. Then the remaining CPU lines A3–A0 connect to PSD A11–A8.

This address scramble must be accommodated for in the system design. There are three areas to consider: the EPROM, the IO port control registers, and the RAM.

EPROM

XA code is usually supplied to a device programmer using a file format called "Intel HEX", and files of this type generally have the extension "HEX". A HEX file is supplied to the WSI PSDsoft software, which combines it with PSD configuration information and writes out a new hex file with an "OBJ" extension.

A standard HEX file associates data with absolute addresses. Because of the address line scrambling shown in Table 2, a standard XA HEX file will not work. For example, if the XA sends out the address 0x1234, the EPROM location accessed within the PSD311 will actually be 0x1423. To account for this, we need a program that reads the XA HEX file, stores the data in memory in address–scrambled order, and then writes a new HEX file with the data residing at the scrambled addresses.

Appendix A is the source code for a C program to accomplish this address translation. It was compiled on the Borland C++ compiler Version 3.1 using the LARGE memory model. The SCRAMBLE.CPP and SCRAMBLE.EXE files are available on the WSI BBS. The source code is included in case you have any trouble running the program — you can freely adapt it to suit your purposes or cater to the whims of your particular C compiler.

To use the utility, place your XA HEX file and the SCRAMBLE.EXE file in the same directory, and type "SCRAMBLE myfile.hex" where myfile.hex is the HEX file to be scrambled. The SCRAMBLE program writes out a new file in address–scrambled order with the same filename and the "HX2" extension — in this example, myfile.HX2.

I/O Port Control Registers

The PSD311 port control registers appear at byte offset 2–7 from a programmable base address. The base address is set by the equation you write for the CSIOP output in the Programmable Address Decoder (PAD). If this base address is positioned at a 4 Kilobyte boundary, only the address lines A15–A12 participate in the decoding. These addresses are not scrambled, so there is a direct mapping of the equation you write for CSIOP and the memory space which the block of I/O Port Control Registers inhabit.

The address lines that participate in selection of the IO control registers, CPU A2–A0, **are** scrambled: CPU A0 is PSD A8, CPU A1 is PSD A9, and CPU A2 is PSD A10 (Table 2). Therefore the register offsets are translated as shown in Table 3.

Table 3. I/O Port Register Mapping

CPU REGISTER OFFSET	ACTUAL PSD ADDRESS
2 (Port A Pin Register)	0x20
3 (Port B Pin Register)	0x30
4 (Port A Direction Register)	0x40
5 (Port B Direction Register)	0x50
6 (Port A Data Register)	0x60
7 (Port B Data Register)	0x70

Table 3 indicates that to access the Port A direction register, for example, the byte at (BASE+0x40) must be accessed. This might be accomplished with the following XA code fragment, which sets PA0 and PA1 to outputs, and PA2–PA7 to inputs:

```
Apins   equ     $20
Bpins   equ     $30
Adir    equ     $40
Bdir    equ     $50
PortA   equ     $60
PortB   equ     $70
BASE    equ     $C000

mov     r0, #BASE
mov.b   [r0+Adir], #00000011b       ; 1=out, 0=in
```

Programmable peripherals using the PSD311 with the Philips XA

AN707

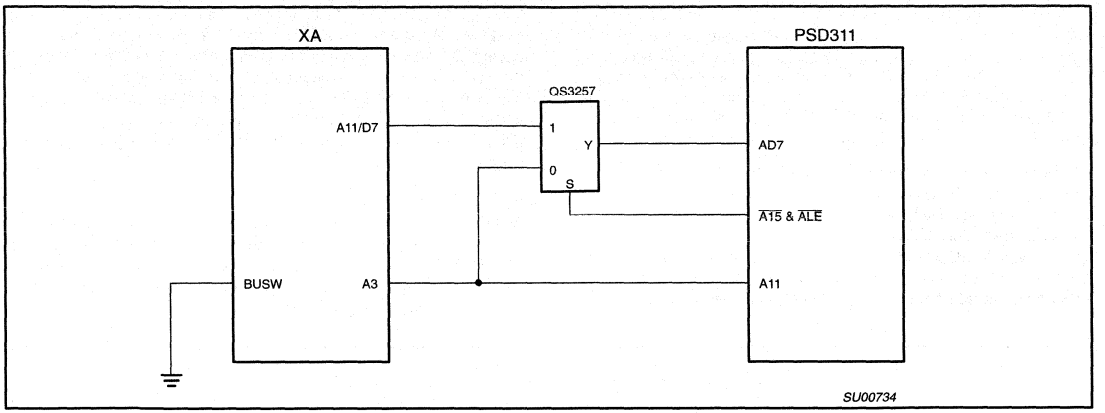


Figure 1. How to Convert A11D7 to A3D7

RAM

At first glance, it might appear that the PSD RAM is the easiest portion of the PSD to accommodate the scrambled address lines. After all, if the CPU writes to address XYZ, and unbeknownst to the CPU, it instead writes to address ABC, when the CPU tries to retrieve the data at XYZ it (again unknowingly) retrieves the data at ABC, which is the correct data. In other words, as long as the same scrambling occurs on a read-write device for both reads and writes, everything is copacetic.

A problem arises, however, because the connection shown in Table 2 connects CPU A3 to PSD A11, and CPU A11 (actually A11/D7) to PSD AD7. Why is this a problem? The RAM size in the PSD is 2 kilobytes, requiring eleven CPU address lines A0–A10. But look where CPU A03 is connected — it's to PSD A11, which is not used in the RAM addressing. Therefore, as far as the PSD311 RAM is concerned, it is missing CPU A03. Furthermore, the signal connected to the PSD311 A7 pin, CPU A11, is superfluous for RAM access.

The net result is that if the connections are made exactly as shown in Table 2, only half of the RAM would be addressable, and every eight bytes would repeat! This would tend to make the software people very unhappy, especially if they put data like the system stack in the PSD RAM.

The solution is to change the CPU "A11/D7" signal to "A3/D7". This change connects all eleven active CPU address lines A0–A10 to all eleven active PSD RAM address lines A0–A10, albeit in scrambled order (which is OK for a RAM). This is accomplished by the circuit shown in Figure 1. A15 is used as a RAM select signal to tell the circuit when to do the A11–A3 swap. The Address swap should be done for RAM accesses only, because A11 is required for EPROM addressing. In order to swap only the address and not the data portion of the multiplexed A11/D7 signal, the ALE signal is used as a qualifier.

The QS3257 is a quad bi-directional multiplexor made by Quality Semiconductor and others. In this circuit, A15 is used as the RAM chip select. When A15 goes HI to select the RAM, the MUX connects XA A3 to PSD311 AD7, but only for the ALE (address) portion of the cycle. When ALE de-asserts, the MUX re-connects XA A11/D7 to the PSD311 AD7 to connect the D7 signals together. The mux must be bi-directional to allow read-write access on D7. Note that the XA BUSW pin is tied low to support an 8-bit bus system at power-on.

How do we develop the logic for driving the MUX select (S) signal? Using the PSD311 PAD, of course. If the RAM is to be positioned within an 8K block, rather than the 32K block decoded by A15 alone, the other address lines A14–A12 may be used in the mux control equations. Appendix B is a PSDabel listing showing the mux select signal as 'mux', which uses PB0. Appendix C is the PSDsoft configuration file for the design.

Figure 2 is a scope photo of ALE, $\overline{\text{WRITE}}$, $\overline{\text{READ}}$ and address line A0. Figure 3 shows the timing for the MUX select signal. The measurements for Figures 2 and 3 were taken using a 30 MHz XA system, with the following bus timing parameters:

ALEW	1	[1.5 clock ALE pulse]
WM1	1	[long write pulse]
WM0	1	[1 clock data hold time for write]
DWA	3	[5 clock ALE–WR cycle]
DW	3	[4 clock WR cycle]
DRA	2	[4 clock ALE–RD cycle]
DR	3	[4 clock RD cycle]
CRA	2	[4 clock ALE–PSEN cycle]
CR	3	[4 clock PSEN cycle]

The XA listing in Appendix D gives the startup code that establishes the above bus timing plus other chip configuration data, and then runs a continuous loop to produce the waveforms shown in Figures 2 and 3.

Figure 2 illustrates two consecutive XA bus cycles. In the first cycle, the XA writes a 16-bit word by issuing two consecutive byte writes. Notice that address A0 changes from an odd address to an even address midway through the cycle (between write pulses) and a single ALE pulse is issued for both byte writes. The PSD3XX family devices work properly with the single ALE pulse because the addresses A8–A11, which are connected to XA addresses A0–A3, **are not latched in the PSD3XX.** PSD devices (PSD4XX/5XX) that latch all of the address lines would not work in this application, since they would not pick up the address change on A0 without a second ALE pulse.

Figure 3 shows the timing for the multiplexor select signal.

Because the first cycle writes data to memory outside the PSD311 RAM (A15=0), the mux select signal is high throughout the write cycle. The second ALE pulse corresponds to a read operation from RAM (A15=1). In this cycle the mux-S signal switches low, feeding A3 into the AD7 pin in place of A11. A3 is latched by the falling edge of ALE, the mux switches back to normal operation, and CPU D7 is connected to PSD AD7 for the remainder of the read operation.

The $\overline{RD}$ and A0 traces in Figure 2 illustrate the basic bus timing for the PSD311. The PSD311 access time can be determined by examining the read cycle which starts at the center division of the scope diagram. The XA reads the first byte by issuing the address of the first byte (A0=LO) and an ALE pulse. The RAM address is valid about 10 nanoseconds after the mux-S signal switches LO (to account for the 3257 mux switching time), and this address is latched inside the PSD311 by the falling edge of ALE. The XA reads the byte just before A0 switches from LO to HI, which starts the second RAM access cycle. (Remember that "A0" is actually A8 in the PSD311, which is not latched). The access time required for the first byte read (mux-S LO to A0 LO-HI transition) is about 100 nsec, and the access time required for the second byte read (A0 HI to $\overline{RD}$ going HI) is about 120 nsec. Thus a PSD311-90 is a good choice for this design.

Performance

As the bus timing waveforms of Figures 2 and 3 demonstrate, an 8-bit bus connection of the Philips Semiconductors XA CPU and the WSI PSD311 gives a very high performance system. Using fairly conservative timing, a word (double byte) read or write takes 400 nanoseconds using the PSD311-30.

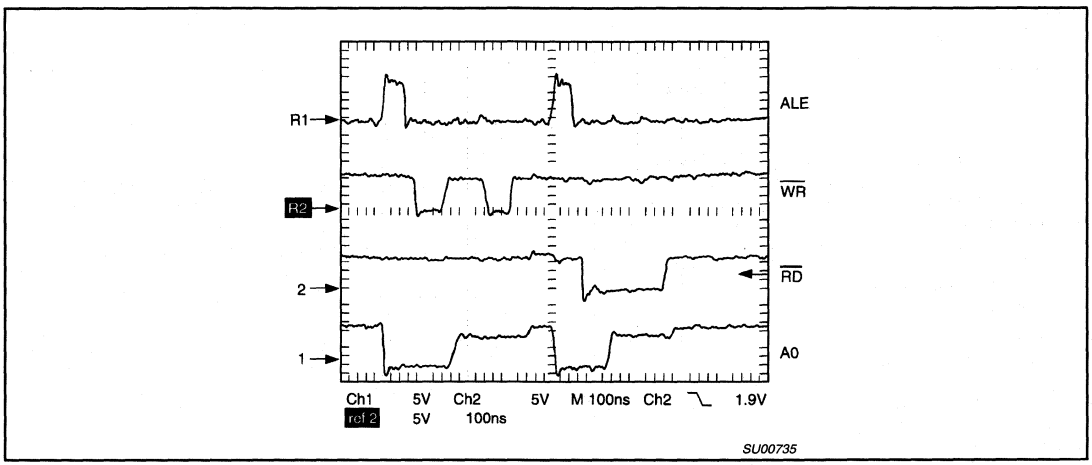


Figure 2. MUX Timing

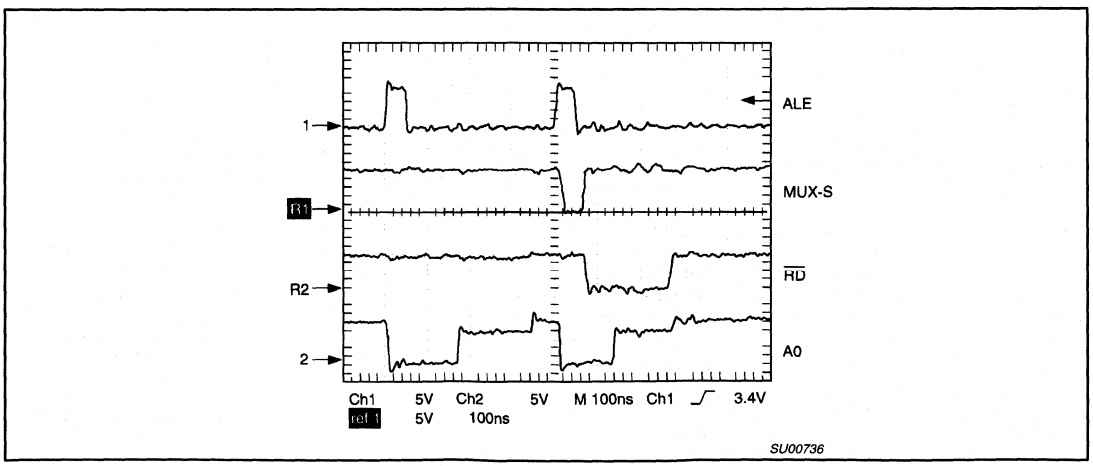


Figure 3. Multiplexor Select Signal (MUX-S)

Appendix A:
C Listing for SCRAMBLE Program

Scramble.cpp 9–12–95 Lane Hauck

This program is used to modify a standard Intel Hex file (.hex) so that it can be used to load a WaferScale PSD311 that is connected to a Philips Semiconductors XA microprocessor. Because the XA does not multiplex AD7–AD0, but instead multiplexes A11D7–A4D0, the addresses to the PSD311 must be scrambled for the data stored in the PSD311 ROM.

Typically the input hex file will be the output of a 51XA linker.

The program reads an Intel hex file, scrambles addresses, and writes a new Intel hex file with an "hx2" extension.

Invoke with: scram <infile.hex>.
Outputs file: "infile.hx2".

Scramble order:

A15 A14 A13 A12 A11 A10 A09 A08 A07 A06 A05 A04 A03 A02 A01 A00
A15 A14 A13 A12 A07 A06 A05 A04 A03 A02 A01 A00 A11 A10 A09 A08

Hex ABCD becomes ADBC

Intel hex format:
(A) Data Record
: cc aaaa 00 [data] cs CR LF

: Colon
cc # data bytes (2 chars)
aaaa load addr (4 chars)
00 record type=data record
data 2 times cc chars
cs 2's compl of checksum (binary values, not ASCII codes) includes cc,aaaa,00,data
CR carriage ret
LF line feed

(B) End Record
: 00 aaaa 01 cs CR LF

: Colon
00 no data bytes
aaaa program start address
01 indicates an END record
cs checksum of 00,aaaa,01

```
**********************************************************************************************************/

#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <dir.h>
#define ROMSIZE 32768L     // PSD311 ROM size

// function prototypes
int         a2d(int  a);
int         scramble(int  inaddr) ;

// global variables
int huge   inarray[ROMSIZE];
FILE                *out;

int main(int argc, char *argv[])
{

FILE                *in;
unsigned int        pos,j,k,a,b,c,d,e,f,m,data;
char                outfilename[12];
char                *ptr;
int                 ch;
unsigned int        count,addr,scradd,csum;
char                string[16];
```

```
// check for two command line items: "scramble", outfilename
if (argc ! = 2)
        {
        printf("\nERROR: Usage: SCRAMBLE outfile\n");
        sound(100);                     /* a little razz sound */
        delay(200);
        nosound();
        return 1;
        }
/* open the file given in the command line */
        if ((in=fopen(argv[1],"rt")) == NULL)
        {
        printf("Cannot open input file..%s\n",argv[1]);
        return 1;
        }
printf("File-%s-opened!\n",argv[1]);
// open a file with input file name plus '.hx2' extension
strcpy(outfilename,argv[1]);    // make a copy of filename
ptr=strchr(outfilename,'.');    // ptr --> '.'
pos=ptr-outfilename;            // position of period
outfilename[++pos]='h';         // replace extension
outfilename[++pos]='x';
outfilename[++pos]='2';
if ((out=fopen(outfilename,"wt")) == NULL)
        {
        printf("Cannot open output file..%s\n",outfilename);
        return 1;
        }
printf("File-%s-opened!\n",outfilename);
for (j=0; j<ROMSIZE; j++)
        {
        inarray[j]=0xFF;
        }
while (!feof(in))
        {
        ch=fgetc(in);
        if (ch==':')
                {
                csum=0;
                a=fgetc(in);
                b=fgetc(in);
                count=16*a2d(a)+a2d(b);
                if (count!=0)                           // ignore end record
                        {
                        csum+=count;
                        c=fgetc(in);
                        d=fgetc(in);
                        e=fgetc(in);
                        f=fgetc(in);
                        addr=4096*a2d(c)+256*a2d(d)+16*a2d(e)+a2d(f);
                        csum+=addr;
                        a=fgetc(in);            // should be two zero bytes
                        b=fgetc(in);
                        data=16*a2d(a)+a2d(b);
                        csum+=data;                     // (checks for 00 byte)
                        for (j=0; j<count; j++)
                                {
                                a=fgetc(in);            // data byte first digit
                                b=fgetc(in);            // data byte second digit
                                data=16*a2d(a)+a2d(b);
                                scradd=scramble(addr);
                                inarray[scradd]=data;
```

```
                                        csum+=data;         // NOTE: csum not checked
                                        addr++;             // here for debug/checkout only
                                        }
                                csum=255–(csum&0x00FF);  // 8–bit, 2's complement
                                }
                        }
                else;
                }
// Write the new hex file
addr=0;
for (j=0; j<=1023; j++)                 // 1024 lines of 32 bytes each
                {
                csum=0;
                fputs(":20",out);
                csum+=addr;
                sprintf(string,"%04X",addr);
                fputs(string,out);
                fputs("00",out);
                for (k=0; k<=15; k++)
                        {
                        for (m=0; m<=1; m++)
                                {
                                data=inarray[addr];
                                csum+=data;
                                sprintf(string,"%02X",data);
                                fputs(string,out);
                                addr++;
                                }
                        }
                csum=255–(csum&0x00FF);         // 8–bit, 2's complement
                sprintf(string,"%02X",csum);    // 2 chars in checksum
                fputs(string,out);
                fputs("\n",out);
                }
                fputs(":00000001FF\n",out);
sound(1000);                                    // a pleasant little sound...
delay(20);
sound(500);
delay(20);
nosound();
fclose(in);
fclose(out);
return 0;
}
// Scramble routine:  Change address ABCD to ADBC
int     scramble(int inaddr)
{
int     outaddr=0;
        outaddr = inaddr            &   0xF000  // A
                | (inaddr <<8)      &   0x0F00  // D
                | (inaddr >>4)      &   0x00FF; // BC
return(outaddr);
}
// ASCII to hex digit conversion
// converts ASCII char to integer 0–15
int a2d(int x)
{
if (x>=65 && x<=70) // A to F
        x–=55;  // –65 makes it 0–5, –55 makes it 10–15
else
        x–=48;  // "0" is ascii 48
return(x);
}
```

Appendix B:
PSDAbel File for MUX Control Signal

```
module xa311
title 'xa311';

mux                     pin 11;          " PB0
nA000                   pin 40;          " PC0–/CS8
ale,nRD,nWR             pin 13,22,2;
a15,a14,a13,a12,a11     pin 39,38,37,36,35;
es0,es1,es2,es3         node 140,141,142,143;
es4,es5,es6,es7         node 144,145,146,147;
rs0,csiop          node 124,125;

equations

es0 =    !a15 & !a14 & !a13 & !a12 ;    " EPROM address map
es1 =    !a15 & !a14 & !a13 &  a12 ;
es2 =    !a15 & !a14 &  a13 & !a12 ;
es3 =    !a15 & !a14 &  a13 &  a12 ;
es4 =    !a15 &  a14 & !a13 & !a12 ;
es5 =    !a15 &  a14 & !a13 &  a12 ;
es6 =    !a15 &  a14 &  a13 & !a12 ;
es7 =    !a15 &  a14 &  a13 &  a12 ;
rs0 =     a15 & !a14 & !a13 & !a12 ;    " RAM select
csiop = !a15 & !a14 &  a13 & !a12 ;     " IOCTL select

mux   = !(a15 & ale);                          " a11–a3 mux control
!nA000 = a15 & !a14 & a13 & !a12;       " FPGA chip select

test_vectors

([a15,a14,a13,a12] -> [rs0,csiop,nA000])

 [ 0,  0,  0,  0 ] -> [ 0,   0,   1  ]; " nothing selected

 [ 0,  0,  1,  0 ] -> [ 0,   1,   1  ]; " IO  at 2000
 [ 1,  0,  0,  0 ] -> [ 1,   0,   1  ]; " RAM at 8000
 [ 1,  0,  1,  0 ] -> [ 0,   0,   0  ]; " FPGA at A000

test_vectors

([a15,ale] -> [mux])

 [ 0,  0 ] -> [ 1 ];
 [ 0,  1 ] -> [ 1 ];
 [ 1,  0 ] -> [ 1 ];
 [ 1,  1 ] -> [ 0 ];      " mux low only for address (ALE) time

end xa311
```

Appendix C: PSDSoft Configuration File

```
**********************************************************************************************************
WSI – PSDsoft Version 2.10
Output of PSD Configurations
**********************************************************************************************************
  PROJECT:        xa311     DATE :   10/24/1995
  DEVICE:         PSD311    TIME :   18:31:39
**********************************************************************
```

==== Bus Interface ====

Data bus width	= 8–Bits
Address/Data Mode	= Multiplexed
ALE/AS signal	= Active High
Read/Write signals	= /WR,/RD,/PSEN
Memory space setting for EPROM	= Program space only (/PSEN)
Security bit	= OFF
Power–down capability	= OFF
EPROM low power mode	= OFF
Active–level of RESET signal	= LOW

==== Other Configurations ====

Port A : ADDRESS/IO Mode

Pin	IO/Address	CMOS/OD Output
PA0	IO	CMOS
PA1	IO	CMOS
PA2	IO	CMOS
PA3	IO	CMOS
PA4	IO	CMOS
PA5	IO	CMOS
PA6	IO	CMOS
PA7	IO	CMOS

Port B :

Pin	CMOS/OD Output
PB0	CMOS
PB1	CMOS
PB2	CMOS
PB3	CMOS
PB4	CMOS
PB5	CMOS
PB6	CMOS
PB7	CMOS

Appendix D:
XA Listing for Figures 2 and 3

```
; RAMTEST.ASM
$include xa–g3.equ
$pagewidth 132t
;
; PSD311 control registers
;
DDRA    equ             $40
DDRB    equ             $50
PortA   equ             $60
PortB   equ             $70
PinsA   equ             $20
PinsB   equ             $30
;
;
        org     0               ; System exceptions:
        dw      $8f00, Start    ; Reset PSW, Reset vector
; ============================
;  Begin initialization code.
; ============================
        org     100
;
Start:
        mov     R7, #$100       ; initialize stack pointer
;-------------------------------------------------
; SCR, System Configuration Register
;-------------------------------------------------
;               76543210
;               0000            ; reserved
;                   00          ; PT1:PT0 = 00 for periph osc/4
;                     0         ; XA mode
;                      1        ; Page 0 mode, uses 16–bit addresses
SCRval  equ     00000001q
;-------------------------------------------------
; WDCON, Watch Dog Timer Control Register
;-------------------------------------------------
;               76543210
;               000             ; Prescaler divisor is TCLK*32*2
;                  00           ; reserved
;                    0          ; WDRUN is OFF
;                     0         ; input bit WDTOF
;                      0        ; reserved
WDCONval equ    00000000q
;-------------------------------------------------
; BCR, Bus Control Register
;-------------------------------------------------
;               76543210
;               000             ; reserved
;                  1            ; WAITD: disable EA/WAIT pin
;                    0          ; Bus Disable OFF (bus enabled)
;                     001       ; bc2:0 –> 8–bit data bus, 16–bit address bus
BCRval  equ     00010001q
;-------------------------------------------------
```

```
; Bus Timing Registers
;-------------------------------------------------
; BTRH
;-------------------------------------------------
;                   76543210
;                   11              ; DW=3  for 4 clock write-w/o-ALE cycle
;                     11            ; DWA=3 for 5 clock ALE-write cycle
;                       11          ; DR=3  for 4 clock read-w/o-ALE cycle
;                         10        ; DRA=2 for 4 clock ALE-read cycle
BTRHval  equ      11111110q
;-------------------------------------------------
; BTRL
;-------------------------------------------------
;                   76543210
;                   1               ; WM1=1  for 2 clock write pulse
;                    1              ; WM0=1  for 1 clock write hold time
;                      1            ; ALEW=1 for 1.5 clock ALE width
;                       0           ; (reserved)
;                         11        ; CR=3   for 4 clock PSEN cycle
;                           10      ; CRA=2  for 4 clock ALE-PSEN cycle
BTRLval  equ      11101110q
;
         mov.b    scr,#SCRval       ; (see above for bit assignments)
         mov.b    wdcon,#WDCONval   ; (see above for bit assignments)
;        mov.b    wfeed1,#$a5       ; Feed watchdog so new config takes effect.
;        mov.b    wfeed2,#$5a
         mov.b    bcr,#BCRval       ; (see above for bit assignments)
         mov.b    btrh,#BTRHval     ; (see above for bit assignments)
         mov.b    btrl,#BTRLval     ; (see above for bit assignments)
;-------------------------------------------------
; Configure the IO port drivers
;-------------------------------------------------
         mov.b    p0cfga,#11111111q ; Configure port0 for bus(11)
         mov.b    p0cfgb,#11111111q
         mov.b    p1cfga,#11111111q ; Configure p14-p17 for quasi-bidirec(10),
         mov.b    p1cfgb,#00001111q ;   A3-A0 for push-pull (11).
         mov.b    p2cfga,#11111111q ; Configure port2 for push-pull (11)
         mov.b    p2cfgb,#11111111q
         mov.b    p3cfga,#11111111q ; Configure p35-p30 for quasi-bidirec(10),
         mov.b    p3cfgb,#11000000q ;   WR(p36), RD(p37) for push-pull (11).
;
; End of initialization, begin user code.
;
                  mov      r1,#$8000; RAM
                  mov      r2,#$7000; not RAM
                  mov      r3,#$00FF
wr1:              mov.w    [r2],r3   ; word write to outside RAM
                  mov.w    r3,[r1]   ; word read from RAM
                  br       wr1

         END
```

Translating 8051 assembly code to XA AN708

CONTENTS

Translating 8051 assembly code to XA AN708

1. INTRODUCTION

What if...

- You've been given the task of translating your company's flagship 80C51 application to XA; your manager gives you an XA data manual and some diskettes—and until tomorrow to get it done?
- You're going to be translating lots and lots of 8051 code to XA over the next few months, and you want to set up the most effective system for doing so?
- You want to learn the XA architecture by translating an example 8051 application?

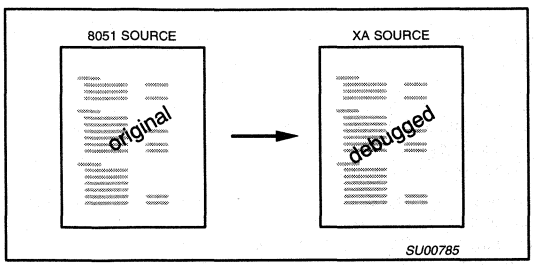


Figure 1.

Relax! Translating production 80C51 source code to debugged XA code is very practical—although we cannot guarantee you'll make your deadline—and this application note brings together the information you'll need.

We're going to generally assume that you're dealing with 80C51 source code you've never seen before.

We will also assume you're a reasonably skilled 80C51 programmer and that you've already studied the basics of XA architecture. We're also going to assume you're able to use Windows™ in general, and you've familiarized yourself with the XA development tools. Although using the simulator or emulator is a key part of verifying your translations, teaching you to use these is beyond the scope of this note.

Source code for the examples may be found on the Philips ftp site.

1.1 The questions you are probably asking right now

How is this application note organized?
We're going to describe a stepwise procedure for translating 80C51 code to working XA code, concentrating mostly on hardware-independent issues. Later on, we'll deal with some very specific issues in detail (section 3: Special Topics), so we recommend you read through all this material before starting to do any translation work.

How long will it take to translate an 8051 application?
That depends on the application and the code. We've seen code from experienced 8051 programmers who've used every conceivable 8051 coding trick to squeeze out the last drop of functionality. We expect this kind of 8051 code will be the most difficult, especially when the code uses target hardware-specific tricks. On the other hand, some 8051 code, especially small applications designed to be easily maintainable, aren't particularly difficult and you may be able to do a complete translation in a matter of a few hours.

Is translation automatic?
Not completely. While the XA does have an equivalent for every 8051 instruction, some code cannot be automatically translated. You'll have to intervene manually.

Is translation a single-pass process?
If you are really expert and you spend enough time, you can probably translate simple to moderately complex applications in a single pass. We don't generally recommend this approach, however, especially when you are starting out. It's probably more productive to iterate several times, using all the development tools available to produce the most robust translation.

Will XA code be bigger?
In all likelyhood, it will be somewhat to significantly larger than the original 8051 code. However, you'll be well-compensated by a significant increase in generality and functionality –the XA instructions are bigger, but they do more– so your XA code will be much easier to maintain and expand.

Will XA code be faster?
Based on the same clock rate, XA code will execute significantly faster than the 80C51 code.

What about 8051 and XA derivatives?
This application note is about translating "general" 8051 code to a "generalized" XA. We'll look briefly at the essential peripheral devices, like the UART and timers, but we're not going to look at other subsystem-specific programs.

What about memory and I/O expansion?
We won't deal with specific memory and I/O issues, but we will comment that the XA is flexible enough to deal with almost any kind of memory and I/O interfaces. The only exception to this rule is some very 8051-specific external interfacing which you'll very likely need to re-engineer anyway.

What's the standard for 8051 code syntax?
The code translater expects the MetaLink ASM51 assembler syntax. This assembler is available for free on the Philips BBS, and it has become the de-facto standard for 8051 and derivatives. If your 8051 source code is based on a different standard, we'll have some suggestions.

What's special about translating your own 8051 code?
As you'll see below, we generally recommend that you deal with mechanical translation issues first and worry about structural changes later. If you're translating 8051 code that's very familiar to you, it is very easy to get distracted by making structural changes too early: you know the constraints of the original design and you'll likely be eager to overcome them by using the significantly greater freedom of the XA architecture.

What is the biggest difference between 8051 and XA afffecting the translation process?
Most 8051 programmers building complex applications spend a significant amount of time reconciling application requirements to 8051 architectural capabilities. When you translate this code to the XA, you'll find many familiar architectural concepts but far fewer constraints on them.

.Windows is a tradmard of Microsoft, Inc.

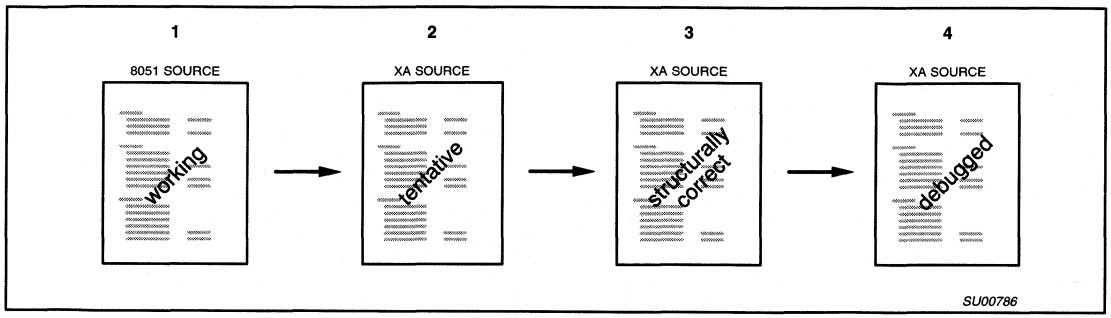


Figure 2. Translation Process

1.2 Overview of the translation process

Let's take a more detailed look at the overall translation process (Figure 2). There are four distinct source versions:

1. The working 8051 source (a copy, not your original source, which you must preserve).
2. A tentative translation of the original 8051 source into XA assembly.
3. The structurally correct XA source.
4. The debugged XA source.

The method of producing each of these source versions is similar to the standard program development cycle, with a single additional step, as shown in Figure 3:

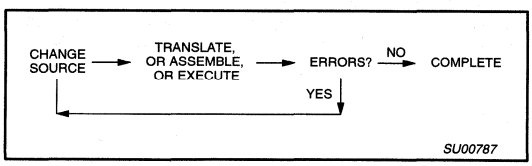


Figure 3.

We recommend that you first make a copy of your original 8051 source. We'll call this the "working" 8051 source. In some cases we've found it convenient to make changes to this source, in particular, when we've found ourselves re-running the translator multiple times for particularly difficult source programs. Be sure to protect your original 8051 source carefully; this is so important we'll remind you to do so several times.

You can see that the edit-assemble cycle and the edit-assemble-execute is preceded by an edit-translate cycle. As we will show below, this extra step is easy. After you complete your work with the translator, you'll have tentative XA source. You'll edit and assemble the tentative XA source, you'll have structurally correct XA source code. You'll run the simulator or emulator and edit and reassemble until your application is proven.

In a few cases, especially with complex applications, you may find it useful to return to an earlier source version. For example, you may encounter some translation issues while debugging your XA source that are best handled by returning to the working 8051 source and repeating the intervening steps. Because of the number of source versions in this process and the potential for returning to an earlier step, we've found that careful organization of the translation process can be very helpful.

1.3 What you'll need

Here's a checklist of what you need to get started translating your 8051 application to XA:

- ☐ The XA Data Handbook (IC25 or its successor); especially Section 2, Chapter 9 and AN704.
- ☐ An 8051 reference, preferably one that's familiar to you.
- ☐ A Windows-based computer
- ☐ The Macraigor XA Development Environment, which includes a translator, an assembler, a simulator, and an optional single-chip XA emulator.
- ☐ The MetaLink ASM51 8051 assembler and its manual, both available on the Philips BBS
- ☐ Macraigor Development hardware is helpful, but optional
- ☐ Your favorite word processor and text utilities.
- ☐ Any and all documentation about the original application. An as-implemented memory map is invaluable. (If you don't have one, we recommend you immediately prepare one by examining the original source code).
- ☐ Design specs for your XA target, particularly the memory map.

1.4 Preparation

Read this application note thoroughly. Section 3, "Special Topics" describes some specific problems you may encounter, and you should have these in mind from the start. Then scan through your original 80C51 source to see how many potential translation issues you can identify.

We recommend that you prepare by choosing a file labeling system to distinguish among the original 8051 source, intermediate translations, and the final XA source file. We have found the following scheme to be useful:

app.asm the original 80C51source file (Not to be modified!)

tapp.asm the working 80C51 source, which may be modified

tapp.1 an intermediate translation

tapp.2 ...

tapp.xa the current XA assembler source

Note that the translator identifies 80C51 source files and activates the 8051 to XA Translator option in the LANGUAGES menu whenever you use a file with a ".asm" extension. The default extension for translated files is ".xa". You may want to keep copies of intermediate translations or originals.

Choose a method that is comfortable for you and that's consistent with the size of the job. This seems like a trivial matter, but we've found that a little extra bookkeeping care can make the translating job much easier.

Second, we recommend that you make sure that all your tools are installed and are functional before attempting translation:

1. Assemble your original source file with the Metalink assembler; note its assembled size.
2. Assemble and simulate one of the examples that accompany the XA tools.
3. Translate one of the example 8051 files.

In other words, we suggest you are comfortable with, and sure of, your tools before you start actual translation work.

1.5 About the original application

Note what we haven't suggested: a detailed examination of the original application.

We're not sure this is necessary!

Intense study of someone else's code can be so incredibly boring or discouraging that you might balk at going through with the translation. You'll have to determine in each case exactly how much detailed knowledge of the application is actually necessary to do a translation. As we've already mentioned, it almost all cases you should start with a memory map of the original application, but it isn't always necessary to have everything documented.

For example, we've been successful translating a moderately complex application, TinyBASIC, without really attempting to understand how the code works, rather, by just attending to the mechanical translation details.

Ultimately, of course, it is up to you how much you need to know about the original application before doing the translation. In the ideal case, you already have full and clear documentation of your application's algorithm (what it does) and the implementation (how it does it). In practice, unfortunately, you may have to re-develop this documentation or translate without it.

1.6 Translation decisions you must make

Although the translator does a good job, there will be some grey areas where you will have to decide about specific issues, and possibly make manual changes to the translated code. Here are the issues:

Retaining or Replacing 80C51-like instructions

The XA instruction set includes a number of instructions for compatibility with 80C51, even though the XA architecture provides improved alternatives.

Instructions like "JMP [A+DPTR]", for example, are supported in the XA, but the full functionality may not translate directly. The translator leaves a warning to mark each use of one of these instructions. You'll either have to check each case carefully and make adjustments to make sure the translated code will function correctly, or replace the entire construction with one or more native XA instructions.

In general, we recommend you replace these 80C51-style instructions with native XA instructions whenever it is practical. We'll give you more information about this issue in following sections.

Recoding obscure but functional 80C51-style coding

There is an additional class of 80C51 instructions and constructions that translate directly into XA instructions. These instructions generate no warning messages from the translator because the resulting XA code, while often obscure, will function correctly. One example is a common 80C51 compare-tree construction.

We can't make a general recommendation in this case, but we'll admit our bias towards recoding into native XA code whenever possible. The following sections will give you more information on this issue.

Using Native versus Compatibility Mode on the XA

The XA System Configuration Register (SCR) CM bit controls the 8051 Compatibility Mode. At reset, SCR.CM is set to zero and the XA operates in "native" XA mode. Setting SCR.CM to "1" makes the XA register model and indirect register addressing mirrors the 80C51 model.

The effects of the CM bit setting are given in Table 1.

Table 1.

FEATURE	IN NATIVE MODE: CM=0	IN COMPATIBILITY MODE: CM=1
registers (R0, R1 ...)	Accessible only as registers	Accessible as registers or as the first 32 bytes of data memory.
R0, R1 indirect addressing	uses 16-bit pointers: R0, R1	Uses 8-bit pointers: R0L, R0H

In other words, if you do nothing, the XA will operate in native mode. The first 32 bytes of data memory will be accessible only as such and won't be overlaid by registers. You'll be able to use r0 and r1 (and any other register) as a 16-bit pointer.

If you insert an instruction that sets SCR.CM=1, translated code that depends on the overlaid memory and register mapping shown in Table 2 will continue to work, and both r0 and r1 will function as 8 bit indirect pointers. This maintains "pure" code compatibility for translation but effectively wastes 32 bytes of internal RAM data memory.

Table 2.

ORIGINAL 8051 REFERENCE	TRANSLATED XA REFERENCE	OVERLAID DATA MEMORY ADDRESS
R0 (RB0)	R0l (RB0)	0
R1 (RB0)	R0h (RB0)	1
R2 (RB0)	R1l (RB0)	2
R3 (RB0)	R1h (RB0)	3
...		
R7 (RB0)	R3h (RB0)	7
R0 (RB1)	R0l (RB1)	8
...		
R5 (RB3)	R2h (RB3)	29
R6 (RB3)	R3l (RB3)	30
R7 (RB3)	R3h (RB3)	31

* RB = Register Bank.

NOTE: The setting of SCR.CM has no effect on instructions labeled "...included for 80C51 compatibility", e.g., "JMP [A+DPTR]" These instructions function as described in the *XA User Guide* no matter what the setting of SCR.CM.

What are the pros and cons of compatability mode versus native mode?
By design, enabling compatability mode produces the greatest chance of translated code working with the least necessity of manual intervention.

In practice, the deciding factor for choosing is, in our experience, the degree of memory map rearrangement you do and the resulting effects on indirect addressing in the application. Specifically: if your XA memory map requires the use of 16-bit pointers, you'll need to use native mode.

Here's a list of factors to consider:

- Using compatability mode....
  - translated code is more likely to run correctly with minimal manual intervention
  - preserves register assignments
  - preserves direct addressing to registers commonly used in 8051 programs
- Using native mode...
  - is sometimes necessary due to changes in the memory map
  - supports cleaner, more efficient XA code
  - generally requires more manual changes to the translated code
  - frees up 32 bytes of on-chip RAM

Using 24-bit versus 16-bit ("Page 0") addressing
You can save some data and hardware resources if you choose 16-bit addressing; clearly this option is available only to applications with addressing requirements below 64K bytes.

The System Configuration Register (SCR) PZ bit controls the XA's Page 0 mode. At reset, SCR.PZ is set to "0" and the XA uses standard 24-bit XA addressing. If SCR.PZ is set to "1", the XA maintains only 16 bits of address data throughout.

For XA targets implementing a memory space of 64K or less, using Page 0 mode can result in some resource savings because the XA will only PUSH and POP 16-bit values for subroutine calls and returns, respectively, insread of 32-bits in standard operation. See the *XA User Guide* sections 4.3.1 and 4.3.2 for more details

Your choice of 24-bit versus 16-bit addressing has no direct effect on the translation process, but you should be generally aware of the implications of each alternative. We've chosen to set Page 0 mode in our start-up code example (Appendix 1).

2. THE TRANSLATION PROCESS

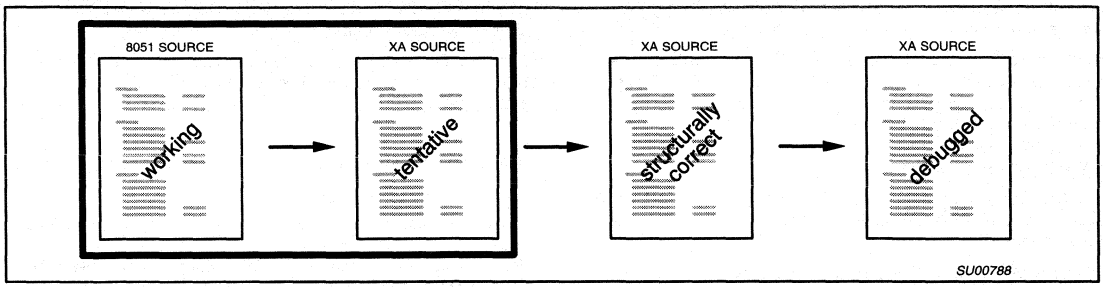


Figure 4.

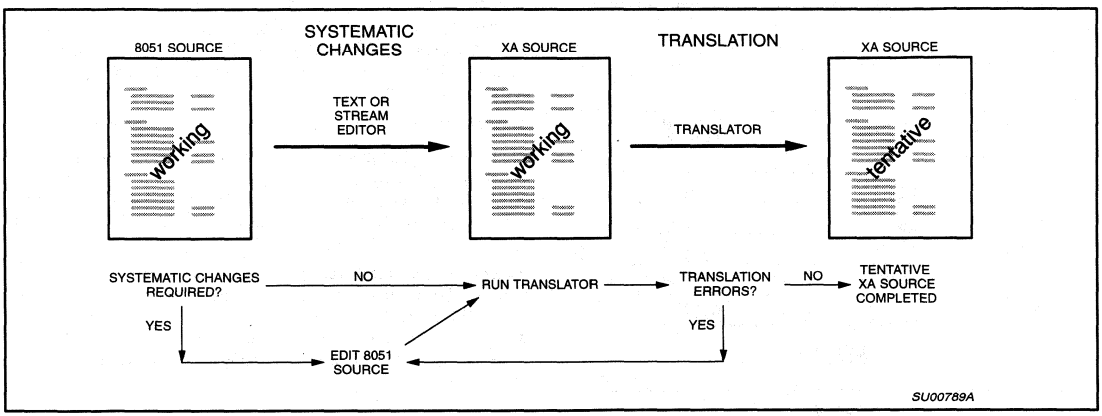


Figure 5.

2.1 Starting translation: Producing tentative XA source

Let's get started by looking at the process of turning 8051 source into tentative XA source code. Figure 5 tells the story. You'll edit, translate, and repeat if necessary until the translator gives no error messages. Don't worry about warnings placed in the translated source code just yet.

2.1.1 Systematic changes

We've chosen the term "systematic changes" to describe anything you might do to change the 8051 source overall.

Do you need to make systematic changes at this stage? The answer depends on the assembler you've been using, the complexity of the code, and the degree to which special directives and other assembler features are present in the source code.

At this point, we'll give you an easy answer that will serve for most purposes: "no". Just go ahead and translate. You'll find out in subsequent steps if this answer was right.

We might as well remind you right now: **Whatever you do, make sure you keep a protected copy of your original 8051 source code somewhere**. It is all too easy to make modifications to the original source file—as you will see, the working 8051 source file may change during this process—and you'll lose valuable information.

2.1.2 Translating

Translating is the soul of simplicity: the XA Development Environment recognizes any file with a ".asm" file as being possibly 8051 assembly source.

1. Open the source file (for example, "`try.asm`")
2. Select "Languages --> 8051 to XA Translator."
3. Respond "Yes" or "No" as you prefer to the query "Include 8051 Code in Output?"
4. The XA translator...
   a. translates your file, leaving it unchanged,
   b. makes a new XA source file with the same name and an "`.xa`" extension ("`try.xa`"), and
   c. automatically saves a copy of the XA source file.

If you choose to include 8051 code in the translator output you'll see the original 8051 instruction, commented out, adjacent to the corresponding XA instruction. This can be very helpful in some cases but the output file can be difficult to read.

We recommend standardizing on the ".xa" extension for all XA assembly source files to distinguish them clearly.

2.1.3 How the translator works

The translator looks at each input source line individually. The translator looks at each source line and attempts to identify it as either a) translatable 8051 code or b) anything else.

If the translator sees 80C51 code, it translates it according to the rules described in Chapter 9 of the *XA User Guide*. The XA was designed to have direct correspondences with 80C51 whenever possible, and most of the translations are probably what you'd expect. The register translations are so critical that we want to restate them here (Figure 6).

All other source lines—that is, anything other than translatable 8051 code—are passed through the translator and appear in the output file unchanged.

The translator doesn't attempt to "understand" what the code is doing; in specific, it doesn't look at more than one source file line or more than one actual instruction at a time. However, the translator warns you when some specific multi-line constructs might go wrong after translation.

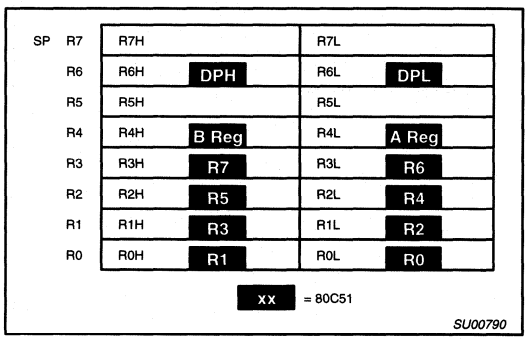


Figure 6.

2.1.4 Interpreting translator results

In most cases, the translator will run without error, and you'll see an open window with the translated source file. This file will contain inserted lines and warnings.

The translator always inserts a line that controls the pagewidth, in order to produce generally convenient displays within source and listing windows:

```
$pagewidth 132t
```

If necessary, the translator inserts an "include" reference, as follows,

```
$include XA-G3.EQU
```

so that any XA register or other reference it produced in translated code is correctly resolved. (If no such references are made, this "include" directive is omitted.) The default location of this file is the PROGRAM subdirectory of the XA Development Tools. You may want to use a different file—for example, for a different XA derivative—or use your own copy of this file.

The translator inserts warnings adjacent to source lines that are translated correctly, but which might not produce the desired results without further intervention. We will take a look at resolving these warnings later in this document.

The translator displays any serious errors in a standard dialog box.

What produces translator errors? In general, when the translator encounters a source line containing what looks like translatable 8051 code, it attempts to do the translation. If the translator finds something about the code that's not expected, it will flag an error for this line.

Fortunately, translator errors are very rare and occur only if the translator finds something entirely unexpected, like source code for an entirely different processor. You'll have to take a look at each error message and decide what to do about them on a case-by-case basis. A large number of errors probably means something very basic is wrong.

2.1.5 Systematic changes revisited

Let's revisit the issue of systematic changes.

You'll need to make systematic changes to your original 8051 source file whenever there are consistent differences between your source and the expectations of the XA Development Tools, especially the translator.

We have found that the easiest way to discover the need for systematic changes is to proceed as if they are not necessary.

If you see significant numbers of similar translator warning messages or XA assembler errors, consider changing your 8051 working source to avoid them.

NOTE: We are suggesting you change the 8051 source file to avoid problems downstream in the translation process. It is for this reason that we use the label "working 8051 source" and we remind you to carefully preserve your original 8051 source file.

If systematic changes are necessary, you may want to work outside the XA Development Environment in order to use the familar, and likely more advanced, features of your favorite text editor. We've found using a UNIX™-style stream editor such as **sed** or **awk** can be very useful, especially in the case of large-scale syntax changes to big source files. It is well beyond the scope of this application note to describe such programs.

We've found the following to require attention:

- 8051 Source code intended for some assembler other than MetaLink

  If you discover large numbers of translation warnings due to unrecognized directives, consider scanning your 8051 working source for directives. Compare them to the XA Development Environment Assembler (see "Help" → "XA Assembler"→ "Directives") and make indicated changes. Some directives don't have equivalents in the XA tools; comment these out or remove them.

  If you see large number of XA assembler errors due to syntax errors, you may have to make significant changes to your working source. Compare the syntax of your source to that described in the XA Development environment ("Help" → "XA Assembler).

- Include files

  The translator doesn't handle all forms of "include" and may or may not pass through a given include file directive. It may be practical in some cases to remove "include" instances and insert the "include" file contents. Otherwise, you may change the existing syntax in your working 8051 source file to one of the alternatives accepted by the XA tools:

  ```
  #include file.ext
  ```

  or

  ```
  $include file.ext
  ```

  and translate the files individually.

Once more we will remind you to keep a protected copy of the 80C51 original source!

2.1.6 What you've got: "tentative XA source"

After you've taken care of translator warnings and any errors, the output file you've got is "tentative" because:

- It doesn't have the proper XA startup vector or any other XA interrupt vector.
- It uses an 8051-style stack.
- It may contain warning messages.
- It might contain illegal or unrecognizable syntax with respect to the XA assembler.
- No comments contain anything about XA implementation.
- If you chose to have 8051 instructions included in the output, there are lots of lines you'll probably want to remove.

We'll see how to resolve these problems in the next section.

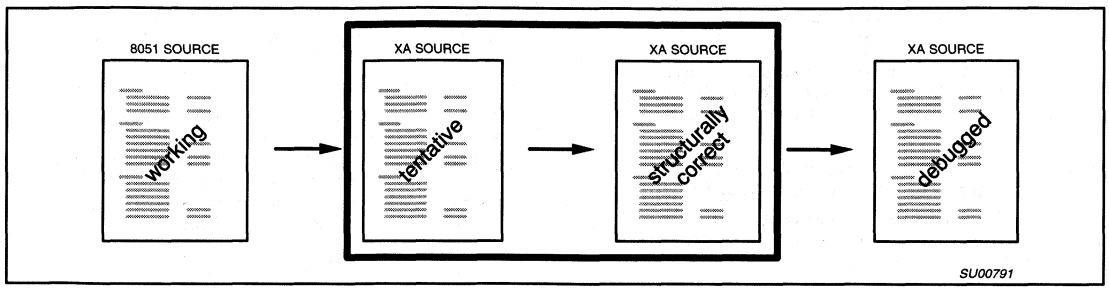


Figure 7.

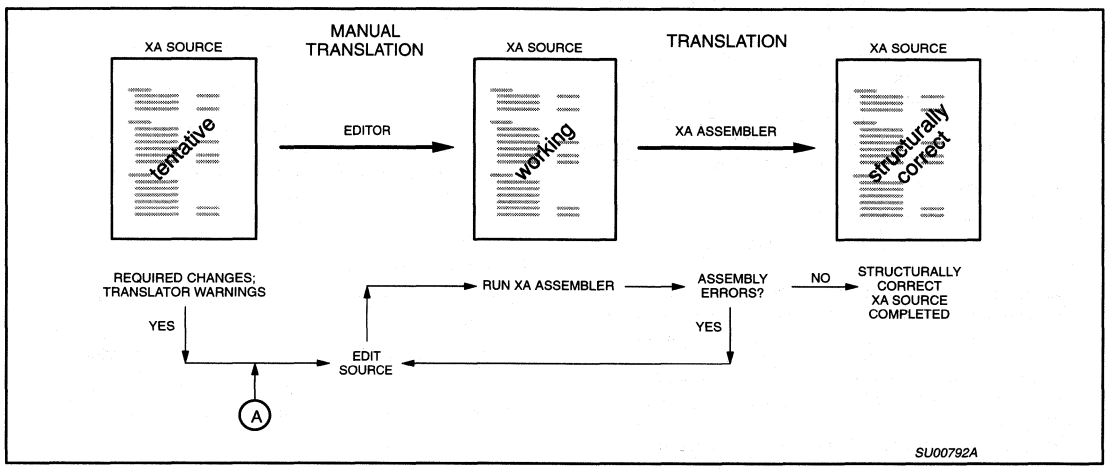


Figure 8.

2.2 Continuing translation: Producing structurally correct XA code

The next step of translation, producing structurally correct XA code from tentative XA code, is summarized in Figure 8.

The diagram item "required changes; translator warnings" denotes the following:

- Installing an XA startup vector and any other required XA interrupt vectors.
- Implementing an XA stack.
- Removing or resolving warning messages.

Later on in this section we will offer a few comments on the remaining issues of tentative XA source: resolving illegal or unrecognizable XA syntax, adding comments about the XA implementation, and removing 8051 instruction comments optionally included by the translator.

Although this application note is targeted to dealing with software translation issues, we'll also cover the key areas of hardware preparation for real XA hardware. If you are just simulating for now, please feel free to maintain code for real hardware; it won't have any effect on the simulator, and you'll be prepared for eventual use on a real XA.

2.2.1 XA startup vector (simple case)

Practiced 8051 programmers usually have a standard program template readily available. In its essence—ignoring interrupts—it looks something like this:

```
    org 0
    ljmp    start                   ; first executed instruction

    org 040h
  start:
    mov     SP,#stack_bottom        ; initialization code
    ...
```

If you've studied the XA, you know that the equivalent minimum startup looks like this:

```
    org 0
    dw      8F00H                   ; initial PSW
    dw      start                   ; reset interrupt vector
    ...

    org 0120h
  start:
    mov     SP,#stack_top           ; initialization code
    ...
```

(See Appendix 2 for a complete listing of a standard XA initialization template.)

As you can see, the XA startup is based on a pair of word vectors at the beginning of code memory, the first taken as the initial Program Status Word (PSW) value, and the second as the initial Program Counter (PC). There is no way for the XA Development Environment translator to automatically translate the 80C51 startup to the XA form, so you'll have to do it yourself: Replace the initial jump with two "dw" define words, the first setting the initial PSW (8f00H is the recommended default) and the second containing the address of the first instruction to execute.

Use this code as a guide for simple cases, such as evaluating code using the XA Development Environment XA simulator. The next section discusses more complex cases. We'll take up the differences in stack initialization further below.

NOTE: It may be useful to make these changes to your working 8051 source file, especially if you return to the translation step more than once.

2.2.2 XA startup and interrupt vectors (more complex case)

Practiced 8051 programmers usually have a vector template for simple applications that looks like the following:

```
    CSEG

    org 0
    ljmp start                      ; Reset Vector

    org 03H
    reti                            ; EXT 0 interrupt: not used

    org 0BH
    reti                            ; Timer 0 interrupt: not used

    org 13H
    reti                            ; EXT1 interrupt: not used

    org 1BH
    reti                            ; Timer 1 interrupt: not used

    org 23H
    ljmp SerialISR                  ; Serial port interrupt

    org 40H
  start:
    mov     SP,#stack_bottom        ; initialization code
```

(This example includes all the interrupt vectors on the core 80C51; derivates differ.)

Drawing again on the complete startup template given in Appendix 2, here's an expanded template that includes the key interrupt vectors for the XA:

```
    dw      08f00H, Start           ; Reset PSW, vector
    dw      08f00H, BreakVec        ; breakpoint PSW, vector
    dw      08f00H, TraceVec        ; trace PSW, vector
    dw      08f00H, StkOvfVec       ; stack overflow PSW, vector
    dw      08f00H, Div0Vec         ; divide by 0 PSW, vector
    dw      08f00H, URetiVec        ; user reti PSW, vector

    org     040H                    ; (TRAP 0-15 exceptions omitted)

    org     080H                    ; Event interrupts:
    dw      08900H, ExtInt0Vec      ; external interrupt 0
    dw      08900H, Timer0Vec       ; timer 0 interrupt
    dw      08900, ExtInt1Vec       ; external interrupt 1
    dw      08900H, Timer1Vec       ; timer 1 interrupt
    dw      08900H, Timer2Vec       ; timer 2 interrupt

    org     090H
    dw      08900H, Rxd0Vec         ; Serial port 0 receive
    dw      08900H, Txd0Vec         ; Serial port 0 transmit
    dw      08900H, Rxd1Vec         ; Serial port 1 receive
    dw      08900H, Txd1Vec         ; Serial port 1 transmit

    org     0100H                   ; (Software interrupts omitted)


    org     00120H                  ; Start of executable code area.

BreakVec:
TraceVec:
StkOvfVec:
Div0Vec:
URetiVec:
ExtInt0Vec:
Timer0Vec:
ExtInt1Vec:
Timer1Vec:
Timer2Vec:
Rxd0Vec:
Txd0Vec:
Rxd1Vec:
Txd1Vec:

    reti    ; Location to route interrupts/exceptions with no specific
            ; handler code. This could prevent lockup particularly due
            ; to an unexpected exception such as stack overflow if
            ; there was no vector or handler whatsoever.
            ; (labels for traps and software interrupts omitted)


;=========================================================================


    org     0                       ; System exception interrupts:
; Beginning of initialization code.

Start:
    mov     R7, #0100H              ; initialize stack pointer top
    ...
```

For brevity, we've omitted the traps and software interrupts and their corresponding labels for the "reti tie-off" found in the original, but we'd encourage you to include them in all code destined for execution on an actual XA target.

The purpose of this comparison is to highlight the differences in interrupt setup when translating an application between the 80C51 and the XA.

We want you to notice that the XA provides extensive support of exception, trap, and event interrupt mechanisms. Tying off unused interrupts takes a lot more "bookkeeping", but you'll recognize that this cost is well worthwhile when you start using these.

2.2.3 The XA stack

We recommend that the first instruction executed in any XA application be a stack initialization; even if this does no more than duplicate the default initialization (completely sufficient for small test programs).

Odds are that the translator will find an 80C51 stack initialization and translate it, but you'll have to manually locate any such code and replace it with XA-specific initialization, if for no other reason that the byte-sized 80C51 stack initialization will be translated to a byte operation; the translator doesn't handle SP as a special case.

Converting from an 8051-style stack to an XA stack is fairly easy if you understand all the issues. Let's take a look at them, in order of increasing complexity:

Basics

In all but the most trivial 8051 applications you'll most certainly see an instruction like this

```
mov             SP, #stackbottom
```

In other words, it may be entirely sufficient to use code like this:

```
   ENDRAM       EQU 0FFFFh                          ; last cell
STACKSIZE       EQU 200h                            ; in bytes
ISTACKPTR       EQU ENDRAM-STACKSIZE - 1            ; initial value

...

   mov          SP,ISTACKPTR                        ; recommended 1st initialization
```

You should already know the differences between the 8051 and the XA stacks:

- The 8051 stack grows upward, while the XA stack grows downward.
- The 8051 stack contains bytes, while the XA stack contains words.
- The 8051 has only one stack, the XA has a System and a User Stack.
- The 8051 stack must be located in on-board memory, while the XA System Stack must be in the first 64K of RAM, on-chip or off.
- The XA user stack may be located in any memory region.

Because 8051 stack space is generally at a premium, stack allocation in 8051 applications is usually done with great care. With any luck, your original 8051 application documentation will describe the stack allocation in detail, including the expected "high-water mark" of the maximum expected usage. If not, we encourage you to spend a little time to study and characterize how the stack is implemented.

Fortunately, stack allocation is much easier on the XA because stack space and placement is much less restricted.

In simple cases—with simple subroutine calls/returns and no interrupts—you'll be able to transform the 8051 stack allocation to a first approximation by using the following steps:

1. Use only the System Stack.
2. Declare the top of the stack in the XA at a specific RAM address.
3. Allocate as many words in the XA as the original application allocates bytes.

As the XA initializes the System Stack Pointer to 100H, that is, within on-chip memory space for all XA devices, you may be able to use the default setting for small applications. Even so, we recommend that you explicitly document the stack allocation for the translated application and explicitly set the stack pointer. Please also note that the XA provides a stack overflow exception if the stack reaches 80H. It's often a good idea to add an interrupt handler for this overflow exception so that your application can recover from stack overflow.

That said, we'll warn you that more complex 8051 applications may require considerably more special handling with respect to stack operation translation. We'll take a look at the next more complex case—handling interrupts—in a following section and then take a look at special topics later on.

The System Stack

We're going to assume that you're translating a single-threaded 8051 application to the XA for the purposes of this application note; translating a multitasked 8051 application is beyond the scope of this application note.

In most cases, Philips recommends you operate the XA initially in System Mode (see the *XA User Guide*, section 4.2.4, for example) and we'll extend that recommendation here to suggest that most translated applications be operated entirely in System Mode.

You set System Mode by setting the SM bit in the initial PSW. If you do this, your application will have full access to all XA registers, instructions, and memory spaces. Further, you'll have only one stack and one stack pointer, and you can ignore any discussion of the User Stack Pointer. R7 will always contain the System Stack Pointer, which you can simply call "the stack pointer" or SP. Throughout your code, you need to take care that no operation you perform sets PSW.SM to zero.

Interrupts add another level of complexity to the picture. When handling interrupts you must "confirm" the System Mode setting of PSW.SM by making sure that the new PSW portion of each interrupt vector contains a value that will result in PSW.SM = 1. In other words, to preserve System Mode, make sure that all values of PSW specified in interrupt vectors leave PSW.SM=0.

The System Stack and Interrupts

Let's review how interrupts are serviced: As you can see in the XA User Manual, the address of the next instruction and PSW in force at the instant of the interrupt are saved on the System Stack. (**All** interrupts—exception interrupts, event interrupts, software interrupts, and trap interrupts—use the System Stack exclusively.) The new value of the Program Counter (PC) and the new PSW are taken from the code-memory vector associated with that interrupt. Normally, you'll use the RETI instruction at the conclusion of an interrupt service routine to restore the interrupted program flow and Program Status.

Unlike the 8051, the XA PSW value is automatically saved any time an interrupt occurs, so it is unnecessary to save the PSW explicitly. (This means you can save a few bytes in translated programs where the original 8051 code did the save and restore by manually removing the explicit PUSH PSW and POP PSW instructions.)

As described in the *XA User Guide*, section 4.3.2, the amount of information processed on the stack during interrupts varies between the default 24-bit XA operation mode and the optional 16-bit Page 0 mode.

The User Stack

What about the User Stack Pointer? We recommend that you don't worry about it for the purposes of the vast majority of translated applications. Simply make sure that PSW.SM is always set. R7 will always contain the System Stack Pointer.

Of course, the XA offers support for multitasking and, for this purpose, you'll need to know how to use User and Supervisor Mode functionality. See Chapter 5 of the *XA User Manual*, and look for forthcoming applications notes on general multiasking support and running multiple virtual 8051's on a single XA.

Advanced Stack Issues

Clever uses of the 8051 stack have been a key feature of that architecture. We can think of a number of schemes you may have to recognize and handle, as well as some XA-specific issues:

- Multiple 8051 Stacks

  Some applications demand multiple stacks, and we've seen some translation problems handling these. Fortunately, these generally involve schemes in which registers other than the 8051 SP is used for stack pointers—we have generally seen the "SP stack" used only for interrupts in some applications—and so these can be handled routinely.

- 16-bit 8051 Stacks

  Likewise, some 8051 applications require 16 bit stacks, and the ones we've seen also handle these outside the scope of the hardware SP.

- Special PSW handling within interrupt service routines (ISRs)

  When you are translating an 8051 ISR, it is almost certain you'll see a "PUSH PSW" somewhere near the beginning and a matching "POP PSW"—or some equivalents—near the end. Unlike the XA, 8051 interrupt processing doesn't automatically save and restore the PSW value, and the vast majority of applications will require that the foreground's PSW be unchanged through interrupt service.

  However, it is remotely possible that an 8051 application breaks this general rule and depends on an altered value of PSW on return from an ISR.

  Here's a somewhat contrived example: an application with no arithmetic processing whatsoever and extreme storage requirements might use the CY flag to indicate the completion of some ISR-processed event to a foreground processing loop.

  The translator can't detect this special method and certainly can't defeat the standard interrupt processing peformed by the XA hardware, saving and restoring the PSW, so you'll have to modify the code manually to make this algorithm work. Fortunately, XA storage is likely to be much more plentiful and you'll probably be able to use a much more conventional technique.

  By the way, you can manually remove the PUSH/POP pair from the translated code if you want.

- Stacks extending over physical address boundaries

  80C51 stacks necessarily reside within the IRAM memory space. XA stacks may be in IRAM, in external RAM, or --as the stack grows downward across the boundary-- in both. There is no special consideration for the placement of the stack with respect to the internal/external boundary, but you should be aware that access speed may be different for the two types of memory.

2.2.4 Feeding the Watchdog

There is one critical mechanism provided on the XA that has no equivalent in the basic 80C51 architecture (but is implemented in some 80C51 derivatives): the watchdog timer. If you fail to pay attention to the watchdog timer—which is activated automatically by a hardware reset—your translated applications will crash mysteriously!

Fortunately, it is very easy to take care of the watchdog. For most developmental purposes, we recommend simply deactivating the entire subsystem early in your XA initialization code. (Right after stack pointer initialization is a good place.) We will draw again from the complete startup initialization template in Appendix 1. The watchdog timer uses a special "feeding" sequence to enable any changes to its configuration:

```
wdoff:    equ $00                  ; WDCON value to turn off WD
...

   mov.b  wdcon,#wdoff             ; Turn off watchdog timer.
   mov.b  wfeed1,#$a5              ; Feed watchdog: use new config
   mov.b  wfeed2,#$5a
```

This code sequence deactivates the watchdog subsystem and places it in a known state.

The watchdog mechanism is not implemented in the XA Tools simulator, and these instructions will have no effect.

2.2.5 Obligatory Hardware Initialization

The XA contains a number of hardware initialization registers. (Namely, BCR, BTRH, BTRL, P0CFGA, P0CFGB, P1CFGA, P1CFGB, P2CFGA, P2CFGB, P3CFGA, and P3CFGB.) The default power-up values of these are often sufficient to get your application running, but the defaults won't work in every case, and we recommend that you consider initializating these registers as obligatory among the first instructions after reset.

You'll definitely need to use non-default values to speed up external memory access and to enable external RAM access when executing code in internal code space.

Explaining how to set all these registers is well beyond the scope of this document, but we'll give a common example, taken from Appendix 1:

```
   mov.b  bcr,#waitd+bus16+adr20   ; Set up bus configuration.
   mov.b  btrh,#dw5+dwa5+dr4+dra5  ; Config bus timing to longest
                                   ; bus cycles
   mov.b  btrl,#wrpuls2+holdmin+ale05+cr4+cra5 ; short ALE, min data
                                   ; hold.

   mov.b  p0cfga,#pcfga_pp         ; Configure port0 types for bus.
   mov.b  p0cfgb,#pcfgb_pp
   mov.b  p1cfga,#p1cfga_bus       ; Configure P1 for quasi-bidirec
   mov.b  p1cfgb,#p1cfgb_bus       ; except A3 - A0 are push-pull.
   mov.b  p2cfga,#pcfgb_pp         ; Configure P2 types for bus.
   mov.b  p2cfgb,#pcfgb_pp
   mov.b  p3cfga,#p3cfga_bus       ; Configure P3 for quasi-bidirect
   mov.b  p3cfgb,#p3cfgb_bus       ; except WR, RD are push-pull.
```

This won't have any effect on the XA simulator. See section 7.3 in the *XA User Guide* for more details.

2.2.6 Disposing of Warning Messages

Warnings are written as comments to the output file. We recommend resolving all warning messages before proceeding with the translation process.

The majority of warnings are usually due to 8051 instructions that translate directly into XA "compatibility" instructions:

```
JZ/JNZ
JMP[A+DPTR]
MOVC A,[A+PC]
MOV A,[A+DPTR]
MOVC A,[A+DPTR]
```

The resulting XA code may be obscure or non-functional, and the translator flags this usage to warn you to check each instance. You'll have to understand what the original code does to do this.

A second class of warning messages are generated for bit references, since identities of bits—even those with identical functions—are different on the 8051 and the XA. Some of these problems will be identified in a later stage by the XA assembler, but you'll have to review all the bits referenced in the translated program sooner or later. No matter what, we can't over-emphasize the rule that's equally valid for 80C51 and XA coding:

"All SFR and bit references should be symbolic."

The first reason for this rule is that identically named SFRs and bits may appear at different places in different 80C51 and XA variants. Secondly, symbolic references receive some error-checking by the assembler that can't be done with explicit numeric references.

Details of Compatibility Instructions

The XA "compatibility" instructions offer you the option of continuing to use a number of highly-functional 80C51 instructions in your XA code. To do so requires that you understand, in detail, any differences that exist between the 80C51 and XA implementation of the instructions, as well as the specifics of each implementation. You may choose to adjust the code to continue to use these instructions, or recode into native XA instructions. (We generally recommend recoding whenever practical.)

Here are the details, along with alternatives for recoding into native XA instructions:

- JZ/JNZ

  JZ/JNZ in the 80C51 uses the current contents of ACC –not the result of a prior ALU operation–to determine whether the jump is taken or not. This function is duplicated in the XA using the translated ACC, R4L. Check the program logic to make sure that the the value in R4L is valid at the time the JZ/JNZ is executed.

  XA native alternative: recode to use CMP followed by Bxx.

- JMP[]

  The JMP[A+DPTR] instruction, used for implementing jump or call tables, depends on the length of each entry in the jump table. Because the lengths of translated instructions in the tables are likely to differ, the translated algorithm probably won't work without further manual intervention.

  XA native alternative: implement a vector table instead, do an indexed fetch followed by a jump-thru-register.

- MOVC A,[A+PC]

  The MOVC A,[A+PC] instruction fetches a constant byte out of code memory. Since the algorithm depends on the length of this instruction –one byte on the 80C51, and two bytes on the XA– you'll have to make an adjustment to make the translated code work correctly.

  XA native alternative: Use standard register-indirection or MOVC to get bytes or words from data or code memory, respectively.

- MOV A,[A+DPTR] and MOVC A,[A+DPTR]

  These instructions are used to fetch one of 256 bytes in a data- or code-memory table, respectively. As long as the translated A, R4L, and the translated DPTR, R6, contain the correct values, algorithms implemented with this instruction should continue to work in the XA.

  XA native alternative: Use standard register-indirection or MOVC to get bytes or words from data or code memory, respectively.

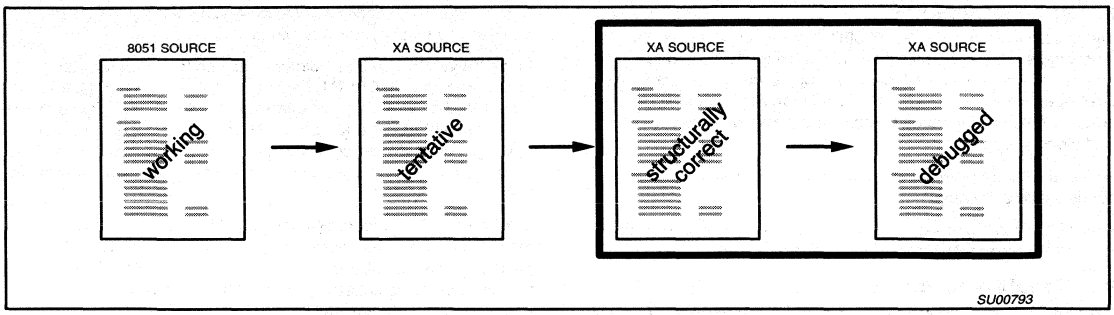


Figure 9.

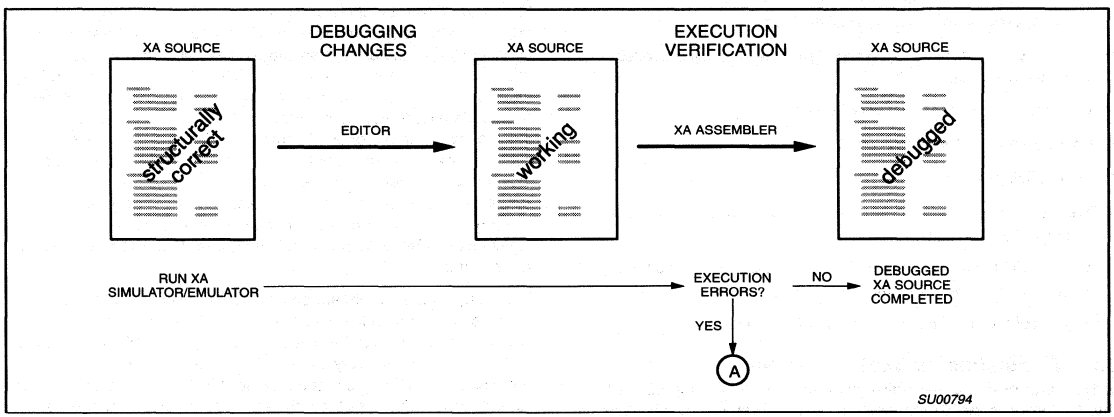


Figure 10.

2.3 The final step: Generating debugged XA code

There's good news at the final step: Transforming structurally correct XA code into debugged XA is almost identical to the standard native code-development cycle. You'll run the XA simulator or XA emulator to verify your code, make corrections, re-assemble, then re-verify, continuing until your program is proven. If you've been careful during previous steps, you shouldn't have any translation-specific problems here.

Figure 10 describes the details.

The only difference for translated code is the very slight chance you'll need to return to an early translation step to correct an unforeseen systematic error in your code. We don't expect this to happen except in extreme cases of specialized code.

As standard XA development techniques are covered by other materials, we'll cut this section short and turn to special topics.

3. SPECIAL TOPICS

We've translated a good deal of 80C51 code, ranging from short functional snippets to applications of considerable size. In the process, we're reminded of the wealth of programming tricks developed over the years by 80C51 programmers to extract all possible functionality from the architecture and to overcome limitations of early tools.

We've also seen a few cases where the 80C51 to XA Translator needs a little extra help.

We've kept notes of these, and we'll share them with you—along with our recommendations for fixes—in this section.

3.1 Trouble-making items

Here's a list from our notes of some potential troublespots. You may want to scan your 80C51 and translated source code for these. Most of these issues are covered in detail in this Application Note.

- 80C51-specific comments in translated code.
- 80C51-style increments in XA that don't affect condition codes, but possibly should.
- Any occurence of the [...+DPTR] operand.
- Any operation on the 80C51 PSW register.
- Any explicit push or pop.
- Over-terse console messages (due to extremely limited ROM size in many 8051 applications).
- Multiply/Divide algorithms, which may translate and run correctly, but very slowly if not re-written for the XA.
- Indirect references (@R0, @R1) in 80C51 code.

3.2 Translating indirect references

Indirect addressing doesn't always translate well from the 80C51 to the XA since the 80C51 implements 8-bit indirection through R0 or R1, while the XA can do 16-bit indirection through any of R0 thru R7.

For programs that are heavily dependent on the 80C51 implementation, you may consider setting SCR.CM to put the XA into 80C51 "compatibility mode" in which XA R0 and R1 function as 8 bit index registers.

In general, we recommend that you examine the issues carefully and recode if necessary to use native 16-bit addressing whenever possible. The increased generality of your code will almost always pay back the effort.

3.2.1 Indirect reference failure

In a few cases, the translator can produce XA code which won't function correctly.

For example, the XA translator translates the 8051 code sequence

```
MOV R1,PTR      ; get a pointer from IRAM
MOV A,@R1       ; load @ptr to ACC
```

to

```
MOV.B R0H,PTR   ; get a pointer from IRAM
MOV.B R4L,[R1]  ; load @ptr to ACC
```

In the 80C51 version, the IRAM location "PTR" contains an 8-bit pointer, which is fetched to R1, one of the two 80C51 index registers. (The source of the pointer is not important; this example would be equally valid if the value of R1 is loaded from any other 8-bit location.) The code uses the byte-size pointer in R1 to load another 8-bit quantity to the accumulator.

In the XA version, an 8-bit pointer is fetched to the XA register corresponding to 80C51 R1, namely R0H. For native mode, the translation has already gone astray as no byte-length index registers are implemented in the XA, and R0H isn't a proper index register anyway. The next instruct, an indirect load, fetches through R1, which is completely uninitialized by this sequence, so the translation fails.

In XA compatibility mode, byte-length pointers are available, but they are defined as the least-significant byte of either R1 or R0, so the 8-bit pointer should be loaded to either R0L or R1L. This translated sequence effectively uses R1L, which is likewise uninitialized, and the translation fails.

3.2.2 Recommendations for Indirect References

The most general solution is to search for all "@R0" and "@R1" addressing in the 8051 code or all "[R0]" and "[R1]" references in translated code. Note that translated code won't contain any references like "[R2]" through "[R7]" since no 8051 code generates these. Although there are no useful changes you can make to the 8051 code prior to translating, you may be able to see better how the code works in the original source, and it is a good idea to consider how pointers might be converted to words from bytes at the most convenient level. Make sure to distinguish between true byte-sized pointers and word-size pointers used to access external memory. Some 8051 programs keep word pointers in adjacent registers or IRAM cells, but there's no guarantee, since we've seem some that don't.

Don't forget that pointers are sometimes checked against limits, and you'll need to convert the limit checking code as well. 8051 CJNE's are translated to CJNE.B's. You may have to manually convert these to CJNE.W's. It's probably a good idea to replace the CJNE tests completely with CMP instruction followed by a conditional branch. In the 8051, the CJNE was the only way to do a non-destructive test, so it was overused—and possibly misused—in situations where a limit might be reached **or** exceeded.

3.2.3 Picking a register for indirect references

If you are manually modifying translated code—following our recommendation to recode certain 80C51-specific constructions—you may need to pick a new register for indirect addressing. Here are some pointers:

In the most general case, you can't use XA registers R0 through R3 for indirect addressing because it is possible that 8051 registers R0 through R7, which translate into byte registers in R0 through R3, namely: R0L, R0H, R1L, ...R3H, might contain useful data.

Of the group R4 through R7, R4 is best preserved for translated code that assigns R4L as "ACC" and R7 is the Stack Pointer. The chances are good that DPTR in the 8051 code was doing something useful, so it's XA twin, R6, is busy.

That leaves R5 as a good candidate. (If R5 is also busy, then you'll have to save a register temporarily; unless stack space is very limited, the stack is as good as any place. Alternatively, you may want to switch register banks if there are free registers in a different bank.)

Remember to consider that it is a standard practice in 8051 coding to assign symbolic names to registers, so you may need to take extra care to track register usage.

3.3 Translating "degenerate" 8051 source code

This section discusses some "degenerate" 8051 programming practices in this section. We use this term for some constructs because they are very 8051-specific, developed over time by programmers to make the most of the available tools and the 8051 architecture.

Most of these don't translate mechanically very well. You can manually modify the code used in around this kind of code and get it to work, but our experience indicates that it is often better to it outright with maintainable, clear XA code. We'll show you how.

3.3.1 "Here" relative addressing

Some typical 8051 source code includes branch instructions resembling the following:

```
    JB      testbit1,$+3
    JNB     testbit2,somewhere
    RET                             ; or whatever
```

Early 8051 assemblers had limited symbolic capacity and encouraged such usage. This code should have been written

```
    JB      testbit1,xyz
    JNB     testbit2,somewhere
xyz:
    RET
```

The translator cannot perform this replacement mechanically, and the XA assembler will not accept "here" relative constructs except in specific cases, so we recommend that you correct the original 8051 source code by adding labels. It is a good idea to search 8051 source code for "$-" and "$+" and resolve all these before attempting translation.

Note: The translator will not translate operands like ".+3" and ".-6". The period operator does not mean "here" in the XA assembly language, where it is reserved for bit addressing constructions, e.g., "SCR.PZ". The translator will emit an error message and leave dot-based operands alone.

3.3.2 Branch to "Here"

A very typical 8051 source code construction looks like this:

```
    JB      eventbit1,$
```

Of course, at the cost of inventing a label name, this code could have been written

```
wait:
    JB      eventbit1,wait
```

Unlike the 8051, the XA cannot branch (jump) to an odd address. If an labeled instruction prospectively occurs at an odd address, the assembler inserts a NOP before any jump target to place it at the next even address. Without a label, however, the assembler can't distinguish this instruction as a jump target, so the translator generates a label when this construction is found. The generated label is not very informative, appearing, for example, as follows:

```
XA_ADJUST_0005:
    JB      eventbit1,$
```

This guarantees that the assembler will even-align the target instruction. We recommend that you substitute a better label, at minimum; even better, consider replacing the "$" reference with the full label.

3.3.3 Branch to "here+offset" or "here-offset"

It's a common 8051 practice to use knowledge of the the length of instructions and the exact movement of the program counter during instructions to use specify branches in conditionals to a place "so many bytes from here". Without careful documentation, the code is often incomprehensible. For example, this 80C51 fragment maps alpha characters in the accumulator to upper case:

```
    CJNE    A,#'a',$+3
    JC      C_IN_1
    CJNE    A,#'z'+1,$+3
    JNC     C_IN_1
    ANL     A,#11011111B
C_IN_1:
    RET
```

This code won't translate. Even though it is obvious what this segment does, it is not immediately obvious how it does it. (See also "Compare Trees" in section 3.3.7.) To which instruction does the first CJNE branch if the branch is not taken? If you keep this construction, you'll have to figure this out.

We recommended you recode this kind of construction in clear XA code, for example as follows:

```
    CMP.B   R4L,#'a'
    BL      EXIT
    CMP.B   R4L,#'z'
    BG      C_IN_EXIT
    AND.B   R4L,#11011111B
EXIT:
    RET
```

3.3.4 In-line strings

Some 8051 assembly programs follow the practice of embedding strings within code sequences that use them. Often this is done where a canned string is to be output to the console, but it may occur in other contexts, for example in a Tiny BASIC interpreter, where the BASIC commands are sought by successive code routines and the compare strings are embedded in the code.

The print string case might look like the following:

```
    CALL    String_Out
    DB      "This string goes to the console",0
    NOP                             ; or whatever
```

The subroutine "String_Out" obtains the string address from the stack and outputs the string, in the process adjusting the stack value to that of the following instruction –here a RET. A trailing null is often used as a string delimiter. (Other schemes that delimit the string are possible, for example, a leading byte count, or setting the 80h bit of the final character, but these have no effect on the basic method.) The code at String_Out usually starts out as follows:

```
String_Out:
   POP     DPH
   POP     DPL
   MOV     a,#0
   MOVC    A,@A+DPTR
   ...
```

This method works on the 8051, and very efficiently, because it take advantage of the fact that the CALL places a full 16-bit address on the stack in a single instruction. Unfortunately, the 8051 has no corresponding 16-bit pop to anywhere else but the Program Counter (i.e., via the RET instruction). So the string pointer must be retrieved and placed in DPTR, the most useful place for it, one byte at a time.

The translator transforms this code to

```
    CALL    String_Out
    DB      "This string goes to the console",0
    NOP
....

String_Out:
   POP.B   DPH
   POP.B   DPL
   MOV.B   R4L,#0
   MOVC.B  A,[A+dptr]
   ...
```

which looks almost the same, but operates very differently. First, the XA CALL places a 16-bit return address on the stack in page 0 mode. In native XA mode, CALL generates a 24-bit return address in two 16-bit stack cells.

When it comes to retrieving the string address, there are no 8-bit stack operations on the XA, so an entire word is popped for each byte POP. If the XA is in page 0 mode, this code will remove two 16-bit words from the stack, one more than was pushed by the CALL, causing a likely fatal stack imbalance as well as incorrect data in DPTR. In native mode, this code will remove the two words pushed by the CALL, leaving the stack in balance, but DPTR will contain an incorrect value. In other words, this code won't work when translated.

What to do? The answer depends on your application.

If you are certain to use page 0 mode, the two byte POPs can be replaced by a single word POP. This will keep the stack balanced and retrieve the pointer value correctly. The return will always be to an even address, so assure that it is the correct one by placing a label on the first instruction following the string. Eliminate the special 8051-equivalent "A+DPTR" form in favor of a straight indirect code memory fetch, in the process eliminating the need to clear the accumulator, and gaining an implicit autoincrement:

```
    CALL    String_Out
    DB      "This string goes to the console",0
new_label:
   NOP
....

String_Out:
   POP.W   R6
   MOVC.B  R4L,[R6+]
   ...
```

Don't forget to remove the code that increments DPTR further below in the output subroutine.

In XA native mode, on entry to String_Out, address bits 16 thru 23 will be in a word on the top of the stack. In some simple cases you might be able to simply POP these and discard them, but resist the temptation, since this won't always work if your code moves.

In general, it is best to remove this construction entirely so there will be no mode or address dependence:

```
msg1:
   DB "This string goes to the console",0
...
   MOV     R6,#msg1
   CALL    String_Out
...
String_Out:
   MOVC.B  R4L,[R6+]
...
```

When you make this conversion, you can forget about the alignment issue, i.e. the need for a dummy label on the code following the embedded string goes away. Of course, you'll have to invent some labels for the strings and place them somewhere out of the code flow.

3.3.5 General In-line args

We've seen strings placed in-line in 80C51 code in the previous section and these at least have the somewhat redeeming value of self-documentation. Imagine finding a code sequence like the following:

```
STK_ER:
          CALL AES_ER
          DB 0FH
```

This is a bit mysterious, especially if the code following is of no particular relevance. To make a long story short, the value '0FH' is an error number, and this routine handles stack errors in a Tiny Basic interpreter. The routine AES_ER could retrieve the error number by popping the presumptive return address into DPTR and doing a MOVC.

This is a good example of the lengths to which 8051 programmers have gone to squeeze optimum performance from the architecture.

Since there are many more registers available in the XA, your translated code could simply move the error code into a free register before calling AES_ER. The error code could even be pushed on the stack:

```
    ADDC    R7,#-2
    MOV.B [R7],#0FH
```

without involving any registers. Whatever you do, we advise you to eliminate in-line arguments.

3.3.6 Program Resets

It is not uncommon to see an 8051 program instruction:

```
JMP     0000H
```

This will translate, but the result won't be correct since there's not an executable instruction at 0.

Better to use:

```
RESET
```

3.3.7 Compare trees

The 8051 construction

```
CJNE    A,#'a',$+3
JC      C_IN_1
CJNE    A,#'z'+1,$+3
JNC     C_IN_1
...
```

works because the CJNE instruction performs a subtract and sets the carry flag. This construction is the only way to "bin" numbers in the 8051 using the accumulator only. But it is often very difficult to understand.

We recommend recoding to use successive XA compares and branches.

Note that XA CJNE's don't include a form that allows comparing two register arguments, and in the XA you'll likely have more working values in registers.

3.3.8 Code Table Fetches

To do table look ups returning bytes, it's common to see 8051 code that does the following:

```
MOV    A,#index          ; or calculated
MOV    DPTR,#TABLE_BASE
MOVC   A,[A+DPTR]        ; A <-- indexed byte value
```

This code sequence will work equally well on the XA, and may often simply be retained after translation, assuming you can live with the limitations:

1. The table is in the current code page
2. You must use the simulated DPTR (R6) and the simulated accumulator, R4L
3. The table can only be 256 bytes long
4. You can only conveniently fetch bytes this way
5. This minimal form uses three register bytes
6. It's not easy to expand to use multiple indexes

Using native XA code is the alternative. Let's see what that looks like in a similar case:

```
MOV.W   Rn,#index     ; or calculated
ADD.W   Rn,#TABLE_BASE
MOVC.B  RnL,[Rn+]     ; RnL <-- indexed byte value
```

Note: we're re-using the same register for the sake of illustration, which you may not want to do in all cases. Using native code has the following advantages:

- The table is in the current code page or through ES, which can point anywhere in code space.
- You can use any registers for the index and table base.
- You can create tables up to 64K bytes long, maybe longer in some situations.
- You can fetch bytes or words, and the autoincrement makes longer fetches easy.
- This minimal form uses only two register bytes.
- It's easy to expand to use multiple indexes.

3.3.9 Using the stack for vectored execution

A sequence of 2 pushes followed by a "ret" instruction in 80C51 source code means that a new execution address has been calculated by some means; the only way to get it into the 80C51 PC is through the stack, for example:

```
..
push DPL
push DPH
ret
```

This kind of construct is used when "JMP @A+DPTR" is insufficient, for example, in the case of large or very sparse jump tables. Although we've seen cases where the translated code works correctly, it is both inefficient and obscure. We recommend recoding to use jumps through an XA register.

Threaded code requiring double-indirect jumps demands use of the single XA instruction to replace the mass of 80C51 code required to do this function.

3.4 Translating "untranslatable" 8051 source

Before you use the translator to mechanically translate 8051 code, you should be aware of a number of 8051 constructions that may translate without error, but will simply not work.

There are also some limitations on source code that the translator can handle, and it is best to manually or automatically scan your source code for these and eliminate them before attempting to use the translator.

3.4.1 PSW bit addressing

Some 8051 code takes advantage of the fact that PSW bits are bit-addressable.

None of these bits are addressable on the XA. This fact should be mechanically demonstrated by the lack of BIT definitions in "XA.EQU" (or its equivalent in your environment), but it is probably better to take care of this kind of problem explicitly, since you'll have to make some code changes right off the bat if you've got PSW bit dependence.

3.4.2 P2 addressing

On the 80C51 it was common for applications with external RAM and ROM to use P2 addressing to address external RAM data memory. This trick provides a second 16-bit memory pointer, albeit an awkward one, for accessing external memory. Obviously, this extremity is not necessary for the XA, which provides plenty of memory pointer registers. We'll describe how to recognize this trick and what to do about it.

Here's what you'll see in the 80C51 source code:

A sixteen-bit pointer is divided into two bytes. The high-order byte is written to port P2, while the low-order byte is written to or maintained in R0 or R1. This is followed by a MOVX to read or write a data byte through the 16-bit pointer.

(The trick: writes to the P2 SFR are gated to the external bus only when this doesn't interfere with P2 addessing external program memory. When the 8051 fetches an instruction from external program memory, the contents of SFR P2 are ignored –but retained– and the most significant byte of the program counter is written to the external P2 physical pins. At all other times, the SFR contents are output to the physical pins.)

To translate these references into XA external data memory references:

1. Chose a word pointer register, XA-Rn.
2. Write the value formerly written into P2 into RnH
3. Write the value formerly written into R0 or R1 into RnL
4. Perform a MOVX indexed by XA-Rn

To translate these references into plain old XA internal data references—assuming there's enough internal RAM available—substitute a MOV for the MOVX. Note that the XA will do an external memory access if the address exceeds the on-chip space.

3.4.3 R7 use

Just a reminder: the XA Stack Pointer is maintained in the XA register R7; avoid the temptation to use R7 as a general register, except in the unusual case that your application doesn't use the XA stack.

3.5 Wrap-up

We've designed the entire suite of XA Development Environment tools—translator, assembler, simulator/debugger—to make the process of translating 80C51 code to the XA to be as smooth as possible.

With the exception in a few cases of the requirement to return to the original source code and re-translate—which occurs only when there's some systematic differences between the original source and the requirements of the tools—the job is little different from any other native development and debugging job.

3.6 Trouble checklist

In trouble? Sometimes it is easy to lose perspective when you're immersed in the details of a translation. Here's a checkist to help you:

- ☐ Have you read the entire application note and studied the special topics?
- ☐ Have you organized your files and file-naming system to the necessary degree?
- ☐ Are you spending too much time understanding obscure 8051 constructs that would be better recoded into native XA code?
- ☐ Should your choice of Page 0 or compatability mode settings be reconsidered?
- ☐ Are you using the simulator effectively?
- ☐ Are you using the correct (up-to-date, variant-specific) include file (e.g., "XA–G3.equ")?
- ☐ Are you attempting to access external memory on an development tool that only supports single-chip operations?
- ☐ Are you using the symbolic register-naming capabilities of the assembler to the best advantage to translate code, by translating functionality, then assigning a specific register?
- ☐ Are you re-editing translated code too often?
- ☐ Are you making piecemeal changes when systematic changes are required?
- ☐ Would you be better off re-writing the application from scratch?
- ☐ Do have a case so special you need help from Philips Applications?

3.7 Final Comments

We've seen that 80C51 code varies enormously with respect to quality, amount of documentation, dependence on architectural "tricks", and so on. Each translation will have its own flavor. Ultimately, there's no substitute for experience, so we encourage to you start with simple cases, take them through the process to completion, and then start again with more difficult translation problems.

APPENDIX 1: STARTUP+INTERRUPT PROTOTYPES

The following code is a complete template for software startup, interrupt vectors, and hardware initialization. Extracts from this code are used in several places in the text. You may obtain this code from the Philips BBS as file "XA-SKEL.ASM".

```
$pagewidth 132t
$listing_min

;===========================================================================

; General purpose skeleton assembly file for the XA-G3

;===========================================================================

; This file may be used as a starting point to develop XA assembly
; language applications. Many definitions are included to simplify XA
; system initialization and to make the code more readable. The user
; will need to adjust the initialization values to suit a specific
; application. Some things to look at are: stack starting location,
; system configuration (8051 compatibility mode; page 0 mode, and
; peripheral timing), watchdog timer setup, bus configuration
; and timing, and port configuration (especially as regards bus
; operation).

; The default setup shown gives an XA with 8051 compatibility turned
; off, page 0 mode on, peripheral timing set to clk/4, watchdog timer
; turned off, the external bus configured to a 16-bit data width with 20
; address lines, bus timing set to the longest cycles but a short ALE
; and minimum data hold, ports; configured for quasi-bidirectional
; mode except for the pins related to bus operation, which are set to
; push-pull.

;===========================================================================

$include xa-g3.equ                      ; Model file for the XA-G3
                                        ; which defines all of the
                                        ; Special Function Registers
                                        ; and addressable bits.


$nolist
;===========================================================================
; Equates to for XA initialization and make setup code self-documenting:

; System Configuration register (SCR):
cmoff:      equ   $00   ; SCR value to turn off 8051 compatibility mode.
cmon:       equ   $02   ; SCR value to turn on 8051 compatibility mode.
page0off:   equ   $00   ; SCR value to turn off Page Zero mode.
page0on:    equ   $01   ; SCR value to turn on Page Zero mode.
time4:      equ   $00   ; SCR value for timer rate = clk / 4.
time16:     equ   $04   ; SCR value for timer rate = clk / 16.
time64:     equ   $08   ; SCR value for timer rate = clk / 64.

; Watchdog timer configuration register (WDCON):
wdoff:      equ   $00   ; WDCON value to turn off watchdog timer.
wdon:       equ   $04   ; WDCON value to turn on watchdog timer.
wdpre64:    equ   $00   ; WDCON value for prescale = 64 * TCLK.
wdpre128:   equ   $20   ; WDCON value for prescale = 128 * TCLK.
wdpre256:   equ   $40   ; WDCON value for prescale = 256 * TCLK.
wdpre512:   equ   $60   ; WDCON value for prescale = 512 * TCLK.
wdpre1K:    equ   $80   ; WDCON value for prescale = 1024 * TCLK.
wdpre2K:    equ   $a0   ; WDCON value for prescale = 2048 * TCLK.
wdpre4K:    equ   $c0   ; WDCON value for prescale = 4096 * TCLK.
wdpre8K:    equ   $e0   ; WDCON value for prescale = 8192 * TCLK.
```

```
; Bus Configuration Register (BCR)
waitd:      equ   $10   ; bcr value to activate Wait Disable.
busd:       equ   $08   ; bcr value to activate Bus Disable.
bus8:       equ   $00   ; bcr value to set 8-bit bus.
bus16:      equ   $04   ; bcr value to set 16-bit bus.
adr12:      equ   $00   ; bcr value to set 12 address lines.
adr16:      equ   $01   ; bcr value to set 16 address lines.
adr20:      equ   $02   ; bcr value to set 20 address lines.

; Bus Timing Register High (BTRH):
dw2:        equ   $00   ; data write cycle without ALE is 2 clocks.
dw3:        equ   $40   ; data write cycle without ALE is 3 clocks.
dw4:        equ   $80   ; data write cycle without ALE is 4 clocks.
dw5:        equ   $C0   ; data write cycle without ALE is 5 clocks.
dwa2:       equ   $00   ; data write cycle with ALE is 2 clocks.
dwa3:       equ   $10   ; data write cycle with ALE is 3 clocks.
dwa4:       equ   $20   ; data write cycle with ALE is 4 clocks.
dwa5:       equ   $30   ; data write cycle with ALE is 5 clocks.
dr1:        equ   $00   ; data read cycle without ALE is 1 clock.
dr2:        equ   $04   ; data read cycle without ALE is 2 clocks.
dr3:        equ   $08   ; data read cycle without ALE is 3 clocks.
dr4:        equ   $0C   ; data read cycle without ALE is 4 clocks.
dra2:       equ   $00   ; data read cycle with ALE is 2 clocks.
dra3:       equ   $01   ; data read cycle with ALE is 3 clocks.
dra4:       equ   $02   ; data read cycle with ALE is 4 clocks.
dra5:       equ   $03   ; data read cycle with ALE is 5 clocks.

; Bus Timing Register Low (BTRL):
wrpuls1:    equ   $00   ; write pulse width is 1 clock.
wrpuls2:    equ   $80   ; write pulse width is 2 clocks.
holdmin:    equ   $00   ; data hold time is minimum.
holdlong:   equ   $40   ; data hold time is 1 clock.
ale05:      equ   $0    ; ALE width is 0.5 clocks.
ale15:      equ   $1    ; ALE width is 1.5 clocks.
cr1:        equ   $00   ; data read cycle without ALE is 1 clock.
cr2:        equ   $04   ; data read cycle without ALE is 2 clocks.
cr3:        equ   $08   ; data read cycle without ALE is 3 clocks.
cr4:        equ   $0C   ; data read cycle without ALE is 4 clocks.
cra2:       equ   $00   ; data read cycle with ALE is 2 clocks.
cra3:       equ   $01   ; data read cycle with ALE is 3 clocks.
cra4:       equ   $02   ; data read cycle with ALE is 4 clocks.
cra5:       equ   $03   ; data read cycle with ALE is 5 clocks.

; Port configuration registers (PnCFGA, PnCFGB):
pcfga_pp:   equ   $ff   ; port config reg a value for push-pull output.
pcfgb_pp:   equ   $ff   ; port config reg b value for push-pull output.
pcfga_qb:   equ   $ff   ; port config reg a value for quasi-bidirect
pcfgb_qb:   equ   $00   ; port config reg b value for quasi-bidirect
pcfga_od:   equ   $00   ; port config reg a value for open drain output.
pcfgb_od:   equ   $00   ; port config reg b value for open drain output.
pcfga_z:    equ   $00   ; port config reg a value for output off.
pcfgb_z:    equ   $ff   ; port config reg b value for output off.
p1cfga_bus: equ   $ff   ; port1 config reg a=quasi, but A3-A0=push-pull.
p1cfgb_bus: equ   $0f   ; port1 config reg b=quasi, but A3-A0=push-pull.
p3cfga_bus: equ   $ff   ; port3 config reg a=quasi, but RD,WR=push-pull.
p3cfgb_bus: equ   $c0   ; port3 config reg b=quasi, but RD,WR=push-pull.

$list
```

```
;=====================================================================
; Start code space:

    org     0                       ; System exception interrupts:
    dw      $8f00, Start            ; Reset PSW, vector.
    dw      $8f00, BreakVec         ; breakpoint PSW, vector.
    dw      $8f00, TraceVec         ; trace PSW, vector.
    dw      $8f00, StkOvfVec        ; stack overflow PSW, vector.
    dw      $8f00, Div0Vec          ; divide by 0 PSW, vector.
    dw      $8f00, URetiVec         ; user reti PSW, vector.

    org     $40                     ; TRAP 0 - 15 exceptions:
    dw      $8800, Trap0Vec         ; trap 0 PSW, vector.
    dw      $8800, Trap1Vec         ; trap 1 PSW, vector.
    dw      $8800, Trap2Vec         ; trap 2 PSW, vector.
    dw      $8800, Trap3Vec         ; trap 3 PSW, vector.
    dw      $8800, Trap4Vec         ; trap 4 PSW, vector.
    dw      $8800, Trap5Vec         ; trap 5 PSW, vector.
    dw      $8800, Trap6Vec         ; trap 6 PSW, vector.
    dw      $8800, Trap7Vec         ; trap 7 PSW, vector.
    dw      $8800, Trap8Vec         ; trap 8 PSW, vector.
    dw      $8800, Trap9Vec         ; trap 9 PSW, vector.
    dw      $8800, Trap10Vec        ; trap 10 PSW, vector.
    dw      $8800, Trap11Vec        ; trap 11 PSW, vector.
    dw      $8800, Trap12Vec        ; trap 12 PSW, vector.
    dw      $8800, Trap13Vec        ; trap 13 PSW, vector.
    dw      $8800, Trap14Vec        ; trap 14 PSW, vector.
    dw      $8800, Trap15Vec        ; trap 15 PSW, vector.

    org     $80                     ; Event interrupts:
    dw      $8900, ExtInt0Vec       ; external interrupt 0.
    dw      $8900, Timer0Vec        ; timer 0 interrupt.
    dw      $8900, ExtInt1Vec       ; external interrupt 1.
    dw      $8900, Timer1Vec        ; timer 1 interrupt.
    dw      $8900, Timer2Vec        ; timer 2 interrupt.
    org     $90
    dw      $8900, Rxd0Vec          ; Serial port 0 receive.
    dw      $8900, Txd0Vec          ; Serial port 0 transmit.
    dw      $8900, Rxd1Vec          ; Serial port 1 receive.
    dw      $8900, Txd1Vec          ; Serial port 1 transmit.

    org     $100                    ; Software interrupts:
    dw      $8100, SWI1Vec          ; SWI1
    dw      $8200, SWI2Vec          ; SWI2
    dw      $8300, SWI3Vec          ; SWI3
    dw      $8400, SWI4Vec          ; SWI4
    dw      $8500, SWI5Vec          ; SWI5
    dw      $8600, SWI6Vec          ; SWI6
    dw      $8700, SWI7Vec          ; SWI7


    org     $0120                   ; Start of executable code area.
BreakVec:
TraceVec:
StkOvfVec:
Div0Vec:
URetiVec:
Trap0Vec:
Trap1Vec:
Trap2Vec:
Trap3Vec:
Trap4Vec:
Trap5Vec:
Trap6Vec:
Trap7Vec:
Trap8Vec:
```

```
Trap9Vec:
Trap10Vec:
Trap11Vec:
Trap12Vec:
Trap13Vec:
Trap14Vec:
Trap15Vec:
ExtInt0Vec:
Timer0Vec:
ExtInt1Vec:
Timer1Vec:
Timer2Vec:
Rxd0Vec:
Txd0Vec:
Rxd1Vec:
Txd1Vec:
SWI1Vec:
SWI2Vec:
SWI3Vec:
SWI4Vec:
SWI5Vec:
SWI6Vec:
SWI7Vec:

   reti     ; Location to route interrupts/exceptions with no specific
            ; handler code. This could prevent lockup particularly due
            ; to an unexpected exception such as stack overflow if
            ; there was no vector or handler whatsoever.


;=========================================================================
; Beginning of initialization code.

Start:
   mov      R7, #$100                  ; initialize stack pointer.

   mov.b    scr,#cmoff+page0on+time4   ; Set up chip configuration.

   mov.b    wdcon,#wdoff               ; Turn off watchdog timer.
   mov.b    wfeed1,#$a5                ; Feed watchdog: use new config
   mov.b    wfeed2,#$5a

   mov.b    bcr,#waitd+bus16+adr20     ; Set up bus configuration.
   mov.b    btrh,#dw5+dwa5+dr4+dra5    ; Config bus timing to longest
                                       ; bus cycles
   mov.b    btrl,#wrpuls2+holdmin+ale05+cr4+cra5 ; short ALE, min data
                                       ; hold.

   mov.b    p0cfga,#pcfga_pp           ; Configure port0 types for bus.
   mov.b    p0cfgb,#pcfgb_pp
   mov.b    p1cfga,#p1cfga_bus         ; Configure P1 for quasi-bidirectional
   mov.b    p1cfgb,#p1cfgb_bus         ; except A3 - A0 are push-pull.
   mov.b    p2cfga,#pcfgb_pp           ; Configure P2 types for bus.
   mov.b    p2cfgb,#pcfgb_pp
   mov.b    p3cfga,#p3cfga_bus         ; Configure P3 for quasi-bidirectional
   mov.b    p3cfgb,#p3cfgb_bus         ; except WR, RD are push-pull.


; End of initialization, begin user code.


end
```

APPENDIX 2: COMPARE/BRANCH SUMMARY

XA Compare/Branch Operations are summarized in the following tables:

Compare op: (destination – source)

Comparison: destination to source

op	CONDITION(S)	JUMP IF...	ALTERNATE: JUMP IF...
BG	(C OR Z) = 0	above, *unsigned*	not below nor equal, *unsigned*
BL	(C OR Z) = 1	below or equal, *unsigned*	not above, *unsigned*
BCC	C = 0	above or equal, *unsigned*	not below, *unsigned*
BCS	C = 1	below, *unsigned*	not above nor equal, *unsigned*
BGT	((N XOR V) OR Z) = 0	greater, *signed*	not less or equal, *signed*
BGE	(N XOR V) = 0)	greater or equal, *signed*	not less, *signed*
BLT	(N XOR V) = 1	less, *signed*	not greater nor equal, *signed*
BLE	((N XOR V) OR Z) = 1	less or equal, *signed*	not greater, *signed*
BCS	C = 1	carry	
BCC	C = 0	not carry	
BEQ	Z = 1	zero	equal
BNE	Z = 0	not zero	not equal
BOV	V = 1	overflow	
BNV	V= 0	not overflow	
BPL	N = 0	not sign	positive
BMI	N = 1	sign	negative

XA FLAG	MEANING
"Z"	Zero Flag
"C"	Carry Flag
"V"	Overflow Flag
"N"	Sign Flag

NOTE: XA PSW51.P is a bit-testable flag that indicates parity (=1 indicates even parity).

Following are some parallel examples of compare trees done first with 8051 CJNE operations, then with XA CJNE operations (directly translated), and XA CMP instructions followed by branches. These examples are illustrative and not intended to be exhaustive

EXAMPLE 1

8051 CJNE

```
        CJNE        dest,src,NOT_SAME          ; branch if dest ≠ src
        JMP         SAME                       ; dest = source
NOT_SAME:
        JC          DEST_SMALLER               ; dest ≠ src
SRC_SMALLER:
        ...                                    ; dest > src
        JMP DONE
DEST_SMALLER:
        ...                                    ; dest < src
        JMP DONE
SAME:
        ...                                    ; dest = source
DONE:
        ...
```

XA CJNE

```
        CJNE        dest,src,NOT_SAME          ; branch if dest ≠ src
        JMP         SAME                       ; dest = source
NOT_SAME:
        BCS         DEST_SMALLER               ; dest ≠ src
SRC_SMALLER:
        ...                                    ; dest > src
        BR          DONE

DEST_SMALLER:                                  ; dest < src
        ...
        BR          DONE
SAME:
        ...                                    ; dest = source
```

XA CMP/Bxx

```
        CMP         dest, src                  ; compare dest to src
        BEQ         SAME                       ; branch if dest = source
NOT_SAME:
        BCS         DEST_SMALLER               ; branch if dest < src
SRC_SMALLER:
        ...                                    ; dest > src
        BR          DONE

DEST_SMALLER:                                  ; dest < src
        ...
        BR          DONE
SAME:
        ...                                    ; dest = source
DONE:
        ...
```

EXAMPLE 2

8051 CJNE

```
        CJNE    dest,src,NOT_SAME       ; branch if dest ≠ src
        JMP     SAME                    ; dest = source
NOT_SAME:
        JNC     SRC_SMALLER             ; dest ≠ src
DEST_SMALLER:
        ...                             ; dest < src
        JMP     DONE
SRC_SMALLER:
        ...                             ; dest > src
        JMP     DONE
SAME:
        ...                             ; dest = source
DONE:
        ...
```

XA CJNE

```
        CJNE    dest,src,NOT_SAME       ; branch if dest ≠ src
        JMP     SAME                    ; dest = source
NOT_SAME:
        BCC     SRC_SMALLER             ; dest ≠ src
DEST_SMALLER:
        ...                             ; dest < src
        BR      DONE
SRC_SMALLER:
        ...                             ; dest > src
        BR      DONE
SAME:
        ...                             ; dest = source
DONE:
        ...
```

XA CMP/Bxx

```
        CMP     dest, src               ; compare dest to src
        BEQ     SAME                    ; branch if dest = source
NOT_SAME:
        BCC     SRC_SMALLER             ; branch if dest < src
DEST_SMALLER:
        ...                             ; dest < src
        BR      DONE
SRC_SMALLER:
        ...                             ; dest > src
        BR DONE
SAME:
        ...                             ; dest = source
DONE:
        ...
```

EXAMPLE 3

8051 CJNE

```
        CJNE        dest,src,GTE                ; branch if dest ≠ src
        JC          DEST_SMALLER                ; dest≠src; branch if dest < src
GTE:
        ...                                     ; dest >= src
        JMP         DONE
DEST_SMALLER:
        ...                                     ; dest < src

DONE:
        ...
```

XA CJNE

```
        CJNE        dest,src,NOT_SAME           ; branch if dest ≠ src
        BCS         DEST_SMALLER                ; dest≠src; branch if dest < src
GTE:
        ...                                     ; dest >= src
        BR          DONE
DEST_SMALLER:
        ...                                     ; dest < src
DONE:
        ...
```

XA CMP/Bxx

```
        CMP         dest, src                   ; compare dest to src
        BCS         DEST_SMALLER                ; branch if dest < src
GTE:
        ...                                     ; dest >= src
        BR          DONE
DEST_SMALLER:
        ...                                     ; dest < src
DONE:
        ...
```

EXAMPLE 4

8051

```
        CJNE    dest,src,LTE            ; branch if dest ≠ src
        JNC     SRC_SMALLER             ; dest≠src; branch if dest > src
LTE:
        ...                             ; dest <= src
        JM      DONE
SRC_SMALLER:
        ...                             ; dest > src
DONE:
        ...
```

XA CJNE

```
        CJNE    dest,src,LTE            ; branch if dest ≠ src
        BG      SRC_SMALLER             ; dest≠src; branch if dest > src
LTE:
        ...                             ; dest <= src
        BR      DONE
SRC_SMALLER:
        ...                             ; dest >src
DONE:
        ...
```

XA CMP/Bxx

```
        CMP     dest, src               ; compare dest to src
        BG      SRC_SMALLER             ; branch if dest > src
LTE:
        ...                             ; dest <= src
        BR      DONE
SRC_SMALLER:
        ...                             ; dest > src
DONE:
        ...
```

Reversing bits within a data byte on the XA AN709

Author: Greg Goodhue, Microcontroller Product Planning, Philips Semiconductors, Sunnyvale, CA

Implementing an algorithm to reverse the bits within a data byte is notorious for producing inefficient code on most processors. This function can serve as a case study of how to trade off code size for performance and shows some of the methods that might be employed in similar types of data conversion situations.

Here four solutions are shown to implement the byte reverse function. The first version (Listing 1) uses a very simple approach. The result is produced by shifting a bit out of the initial data value and shifting the same bit back into the result value. This is repeated in a loop for each bit. Since the two shift operations are done in opposite directions, the result value is a bit reversal of the initial value. This version is the smallest in size, using only 11 bytes. However, it takes 128 XA clock periods to complete.

Listing 2 uses the same method as the first, but "unfolds" the loop to eliminate the counting and branching overhead. What is left are the instructions from the inside of the loop repeated eight times. Unfolding the loop gives faster execution, 64 clocks in this case. The code size grows somewhat to 16 bytes.

The third method (Listing 3) uses a partial lookup table to reverse one nibble at a time and assemble the complete byte from two lookup values. In a reversed byte, the upper nibble of the result consists of the reversed bits of the lower nibble of the initial value, while the lower nibble of the result consists of the reversed bits of the upper nibble of the initial value. The code example uses each nibble of the initial value as an index into the lookup table, which provides a nibble of result data. The two partial results are then combined to produce the complete result. This version uses 42 bytes for both the code and the lookup table, but requires only 42 XA clock periods to complete.

The final method shown (Listing 4) uses a full lookup table to produce the entire result very quickly. The initial data byte is used as an index into the lookup table and the value from the table is the complete result byte. This method produces the result in only 12 XA clocks. However, the code plus the lookup table occupies a fairly large amount of code space: 264 bytes.

CONCLUSION

These examples show how code size may often be traded for execution speed, or execution speed for code size, depending on an application's requirements. This is summarized in Figure 1. Other solutions to this particular algorithm are certainly possible and other algorithms will likely have different types of solutions with different resulting tradeoffs.

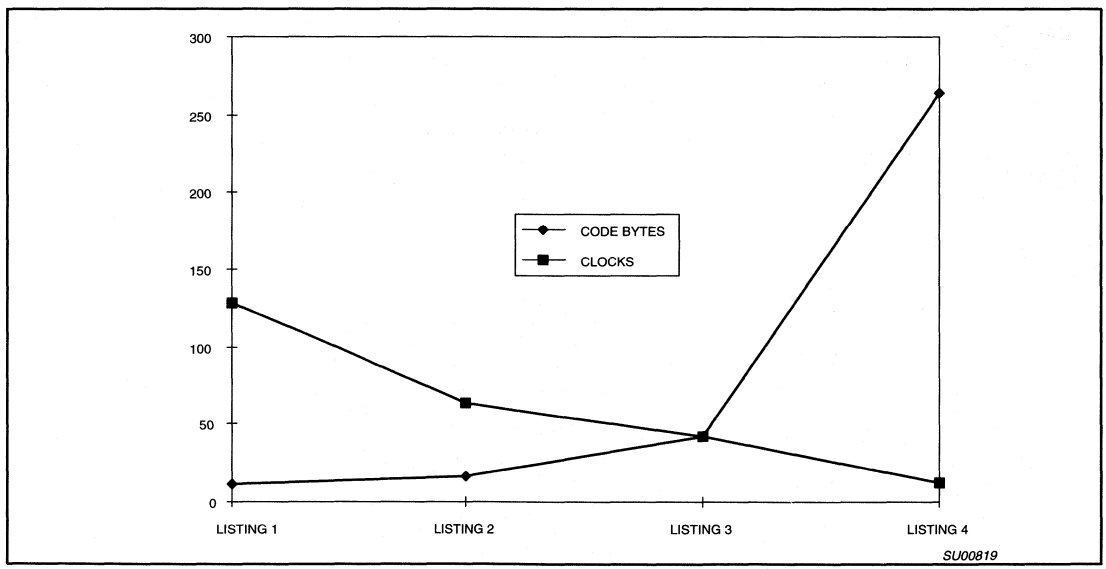


Figure 1. Tradeoff of code size to performance.

LISTING 1

```
; Listing 1) Smallest solution in terms of code space:
;     Enter with value to be reversed in R0L, result in R0H.
;     This works by shifting the register out in one direction and back in
;     the other.
          mov       count,#8             ; 3 clks
loop:     rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          djnz      count,loop           ; 8/5 clks
; total time = 8*(4+4) + 7*8 + 3+5 = 128 clocks
```

LISTING 2

```
; Listing 2) Solution 1 with the loop "unfolded".
;     Enter with value to be reversed in R0L, result in R0H.
;     This works by shifting the register out in one direction and back in
;     the other.
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
          rrc       r0l,#1               ; 4 clks
          rlc       r0h,#1               ; 4 clks
; total time = 8*(4+4) = 64 clocks
```

LISTING 3

```
; Listing 3) Fastest solution (without using a 256 byte lookup table):
;     Enter with value to reverse in R4L, result returned in R0L.
;     This works by reversing each nibble using a look up table and reversing
;     the two nibbles separately as part of the procedure.
            mov         r6,#LUT1                ; 3 clks
            push        r4l                     ; 3 clks
            and         r4l,#$0f                ; 3 clks
            movc        a,[a+dptr]              ; 6 clks
            mov         r0l,r4l                 ; 3 clks
            rr          r0l,#4                  ; 4 clks
            pop         r4l                     ; 4 clks
            and         r4l,#$f0                ; 3 clks
            rr          r4l,#4                  ; 4 clks
            movc        a,[a+dptr]              ; 6 clks
            or          r0l,r4l                 ; 3 clks
            .           .
            .           .
            .           .

; this is a nibble reverse lookup table:
LUT1:       db          $00                     ; 0000 => 0000
            db          $08                     ; 0001 => 1000
            db          $04                     ; 0010 => 0100
            db          $0C                     ; 0011 => 1100
            db          $02                     ; 0100 => 0010
            db          $0A                     ; 0101 => 1010
            db          $06                     ; 0110 => 0110
            db          $0E                     ; 0111 => 1110
            db          $01                     ; 1000 => 0001
            db          $09                     ; 1001 => 1001
            db          $05                     ; 1010 => 0101
            db          $0D                     ; 1011 => 1101
            db          $03                     ; 1100 => 0011
            db          $0B                     ; 1101 => 1011
            db          $07                     ; 1110 => 0111
            db          $0F                     ; 1111 => 1111
; total time = 42 clocks
```

LISTING 4

```
; Listing 4) Fastest solution (using a 256 byte lookup table):
;     Enter with value to reverse in R4L, result returned in R0L.
            mov         r6,#LUT2                ; 3 clks
            movc        a,[a+dptr]              ; 6 clks
            mov         r0l,r4l                 ; 3 clks
            .           .
            .           .
            .           .
; this is a byte reverse lookup table:
LUT2:
            db          $00                     ; 00000000 => 00000000
            db          $80                     ; 00000001 => 10000000
            db          $40                     ; 00000010 => 01000000
            db          $C0                     ; 00000011 => 11000000
            .           .
            .           .
            .           .
; total = 12 clocks
```

Implementing fuzzy logic control with the XA

AN710

Author: Zhimin Ding

ABSTRACT

Most control applications involve the specification of a relationship between sensor signals and actuator outputs. Fuzzy logic provides an intuitive way to accomplish that. It allows the user to use linguistic rules to specify a nonlinear mapping between sensor signals and actuator outputs, thus provide a framework for programing an embedded system. Using a multi-joint robot system as a testbed, we implemented fuzzy logic on an 8051 compatible 16-bit microcontroller—the XA. The robot controlled by the XA running the fuzzy logic algorithm is able to carry out a goal-directed motor sequencing behavior. An 8XC552 is also used to directly interface with the robot and communicate with the XA through I²C. In addition to carrying out AD/PWM conversions, the '552 also implements multiple loops of linear feedback for servo positioning and compliance control. This application note will demonstrate the implementation of Fuzzy Logic in an embedded control solution using the Philips XA microcontroller.

INTRODUCTION

Fuzzy logic was originally created as a mathematical model of human thought. It is said that fuzzy logic is able to capture the "vagueness" and "inexactness" of the concepts that we use for reasoning. In the past few decades or so, the main area of success with fuzzy logic has been in industry control. The application of fuzzy logic allows us to specify the relationship between sensor inputs and actuator outputs using "if...then..." type of linguistic rules. A fuzzy logic algorithm would be able to translate or interpolate these rules into a nonlinear mapping between sensor input signals and actuator outputs for feedback control [1]. Fuzzy logic makes it easy for a human designer to fine tune a control system through trial and error. Together with some other approaches such as artificial neural networks, genetic algorithm, etc., fuzzy logic is considered a useful tool for non-model based control system design[4].

There are a number of software products available that would allow the user to design a fuzzy controller interactively with a special graphic user interface (GUI). These tools would usually generate C codes which can to be modified to fit into a user target platform. If you have to determine the parameters of your fuzzy logic control system on trial-and-error basis, it is certainly desirable to have some kind of graphic user interface so that you do not have to go into your code and make modification here and there.

As the number of inputs to a control system increases, the number of potential useful fuzzy rules increases dramatically and it becomes increasingly desirable to use some kind of automated method for rule synthesizing. There are a variety of such methods for doing this and active research is being carried out in this area currently. For example, the combination of fuzzy logic with artificial neural networks, genetic algorithm and learning automata have proved to be effective in many applications.

In this application note, I will demonstrate the use of fuzzy logic in the XA. With a two-joint robot system as the testbed, I will discuss how to use fuzzy logic to tackle a specific control problem as well as some general programming issues related to the XA. Instead of exploring all the options that are out there, I will focus on one effective solution in this application note to get the readers quickly acquainted with the technique.

ROBUST CONTROL OF A "BUG"-LIKE ROBOT LEG

Figure 1 is a diagram of the robot leg. It is powered by two gearmotors and it has a passive foot-like structure at the end of the distal segment. We call the distal segment the "tibia", and the proximal segment the "femur" after animals. The behavioral purpose of this robot is to grab an object within the space it can reach. The location of the object is unknown to the robot and changing periodically. This is very similar to a situation when an insect walking over a very rough terrain is trying to find an object (such as a tree branch or twig) to grab onto as a foothold. In this design, range sensing such as vision is not involved in the search of the object, as is the case with insects. Insects have developed a behavior shown in Figure 2 where they use their legs as probes to actively sense where the object is and then establish a foothold onto it through a simple reflex [2]. The active sensing reflex makes the "substrate-finding" behavior quite robust.

The robot is equipped with two potentiometers which give us angular position readings for the two joints. On the two segments of the leg, strain gauges are pasted as force or touch sensors. The two strain gauges that are pasted near the junction of each actuator and the corresponding leg segment give us indications of the output torque of the two actuators. Three additional strain gauges are pasted along the distal segment (the tibia). These strain gauge readings can be decoded to determine where the touch between the leg and a external object has occurred. One of the strain gauges is pasted at the foot ankle region to signal foothold.

Our purpose of controlling this leg is to replicate the "substrate-finding" behavior described above in a robust and reliable fashion (Figure 2). The challenge of this control problem lies in the fact that the position and touch sensors do not passively tell where the object is. The robot has to carry out active search movement to find out where the object is. In such a case, there is no way to linearly combine the sensor signals (or their derivatives and integrations) to produce the desired motor movement as in PID control. It is therefore an ideal application for us to try out fuzzy logic.

1. Non-model based design is a design that does not depend on a mathematical description of the plant dynamics.

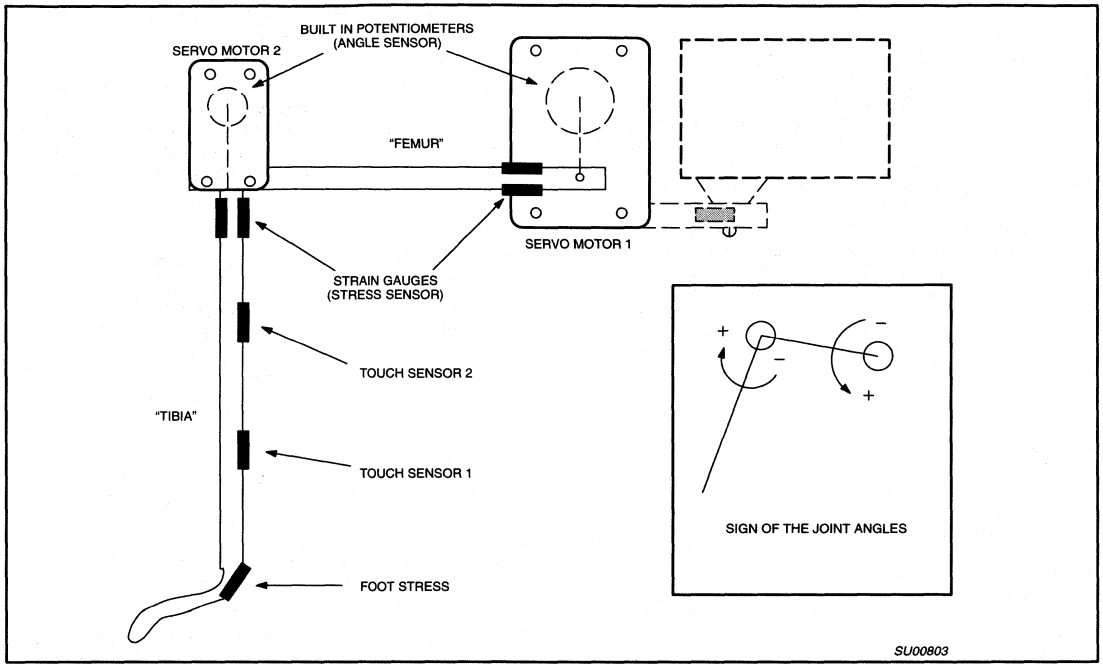


Figure 1. A two-joint robot leg.

For each axis, there is a potentiometer for angle sensing and a pair of strain gauges to measure the output torque. Additionally, there are three strain gauges on the tibia to measure the stress caused by touch or foot load. A third servo can be added to this leg to make it three degrees of freedom.

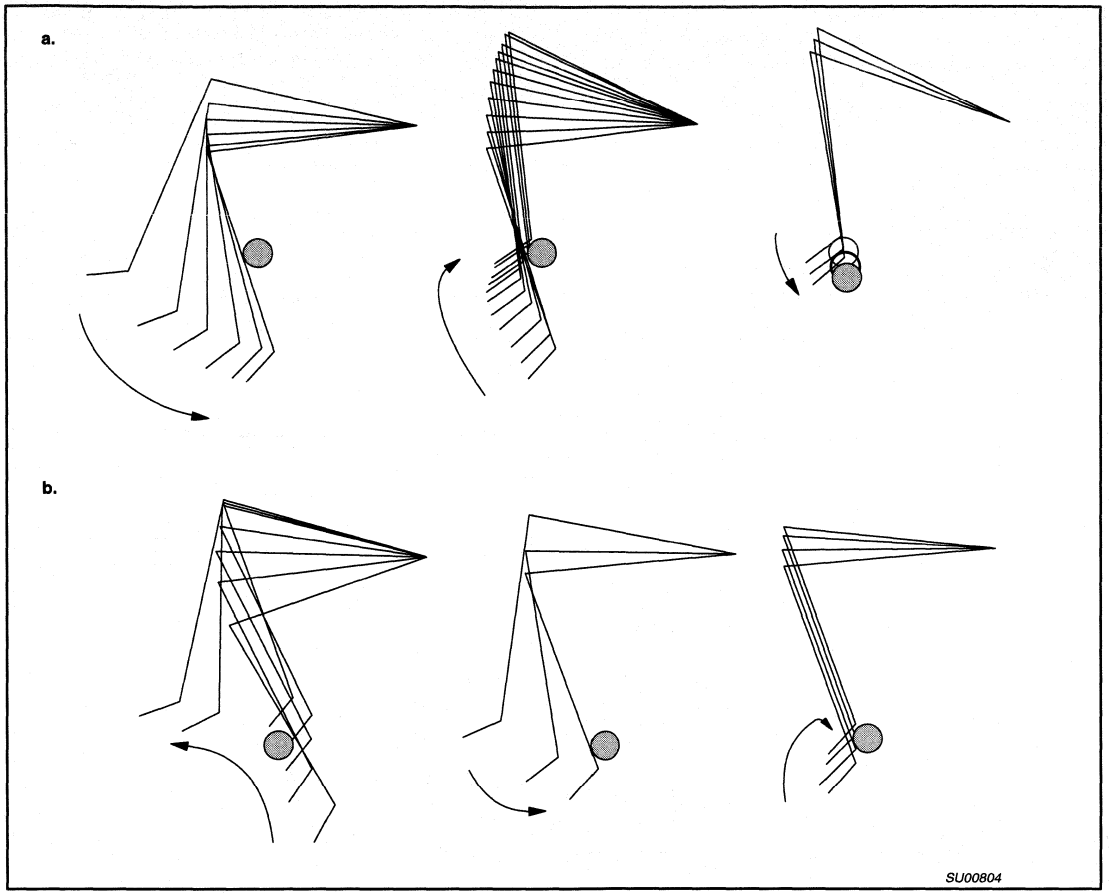


Figure 2. Digitized robot leg movement trajectories from the "substrate-finding" behavior.

a. The leg encounters an object during the downward sweep of a search cycle; once contact is made, the leg slips up until it just clears the object and then comes back down to establish a foothold.

b. The leg encounters the object during the upward sweep of a search cycle. This is a typical nonlinear control problem because one could not linearly combine the sensor signals and get the actuator output values as shown in the figure.

OUTLINE OF OUR APPROACH

As illustrated in Figure 3, two Philips microcontrollers on an I²C-bus are used to control this robot. Firstly, an 8-bit 8XC552 microcontroller is used to interface directly with the robot. In addition to carrying out the necessary A/D and PWM functions for sensor and actuator interface, this 8-bit microcontroller also implements position and force feedback to shape the actuator dynamics so it becomes a position servo with proper compliance and damping properties. This is very important because the compliance in the actuators allows the robot to carry out contact based maneuvers stably and reliably. Together with the sensors and actuators of the robot leg, the 8XC552 implements "virtual muscles" as seen from the microcontroller at the upper level, which is the 16-bit XA microcontroller running the fuzzy logic algorithm. I chose an XA as the fuzzy logic engine because of its higher arithmetics capability. The XA reads "crisp" sensor values from the 8-bit microcontroller through I²C interface and converts them into fuzzy membership grades. These values are evaluated by a set of fuzzy rules implemented in the XA in order to generate appropriate motor commands which are sent back to the '552 through I²C. With I²C, we can easily put multiple robot legs in the control system as shown in Figure 3. For example, we can put together a six-legged hexapod robot.

In this application, the use of fuzzy logic and XA is not intended to replace low level linear, classical control carried out with the 8XC552. Instead, I use fuzzy logic in an augmented and distributed fashion. Fuzzy logic in XA and linear classical control in '552 function in parallel and contribute to different aspects of the control.

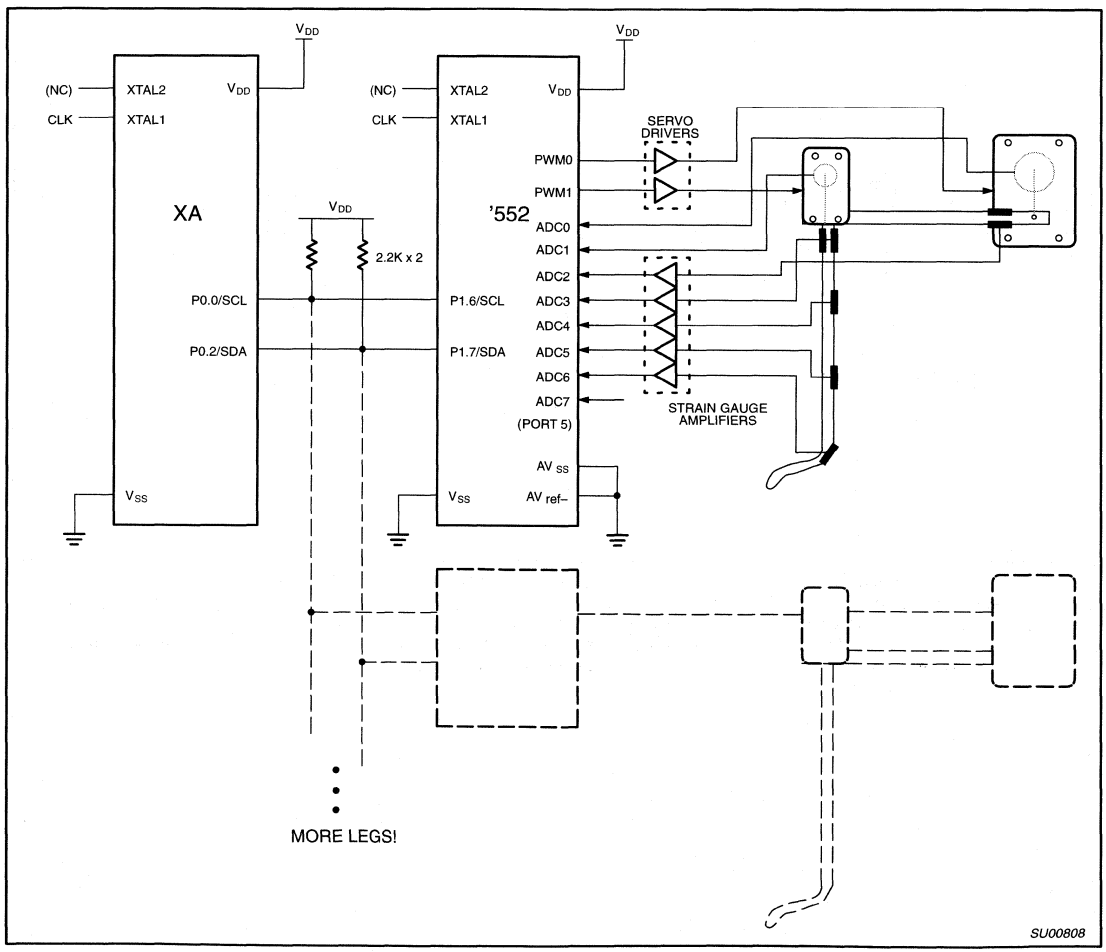


Figure 3. A diagram of the robot control circuit.

We use two microcontrollers to implement two levels of control. A 8XC552 is used to directly interface with the robot and carry out actuator level control feedback. The XA is used here to carry out fuzzy logic algorithms to control the leg movement. The two microcontrollers communicate with each other through an I²C bus. With I²C, we can easily put more than one robot leg in the same system.

THE IMPLEMENTATION OF FUZZY LOGIC ALGORITHM IN XA

In this section, I will explain some fuzzy logic related programming issues. I first explain the algorithm itself by going through the basic steps and then discuss how to implement the algorithm in the XA.

The first thing to do in a fuzzy logic control system is to translate a real sensor signal value into fuzzy membership grades. This procedure is called fuzzification. For example, if we have a sensor input value **x** within the range of 0 to 255, we want to find out to what extent is it "big" or "medium" or "small". We can assign three functions corresponding to "big", "medium" and "small" to do the translation. Those are called membership functions. As shown in Figure 4, if *x = 10*, then "**x** is small" has a truth value of 0.9; "**x** is medium" has a truth value of 0.2; "**x** is big" has a truth value of 0.1. In other words, we are mapping the value of *x = 10* into a triplet (0.9, 0.2, 0.1).

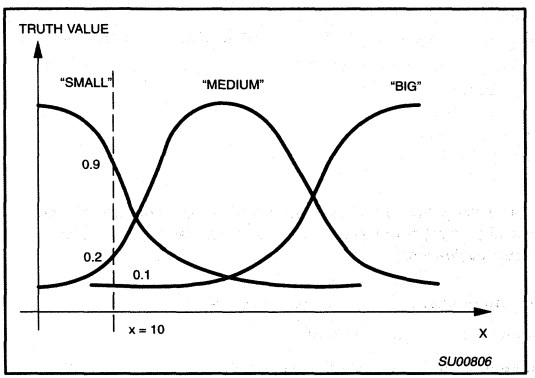


Figure 4. An illustration of the fuzzy membership functions.
For each *x* in the range of 0 to 255 (e.g., x=10), we can describe it as degrees of being "small", "medium" and "big" by using the three fuzzy membership functions.

The next thing to do at this point is to evaluate rules and find out their strengths. Suppose we have these three rules that involve input x.

- *if x is small then z is low*
- *if x is medium then z is high*
- *if x is high then z is low*

In this case, there is one "if ..." part in each rule, the strength of the rule is simply the truthfulness of the "if ..." part, which is called the antecedent. The truth values of the above three rules are 0.9, 0.2, and 0.1, respectively. If there are two or more antecedents, as in "if **x** is small and y is big", the strength of the rule is the smallest of the truthfulness of the antecedents (if the relationship between the two antecedents is an "or" instead of an "and", we would use the largest value of the truthfulness of the antecedents).

The last thing is to find out is the real value of the output **z** from rule evaluation. Before we proceed, we need to define the membership functions for **z**. For example, we can simply assign **z** = 5 to "**z** is low" and **z** = 200 to "**z** is high". These membership functions are impulse functions and they are generally called "singletons".

To find out the precise (crisp) value of **z** according to the above three rules, we simply calculate the weighted average of z-singletons according to the strength of the three rules, therefore,

$$z|_{x=10} = \frac{5 * 0.9 + 200 * 0.2 + 5 * 0.1}{0.9 + 0.2 + 0.1} \approx 38 \qquad (1)$$

What we have accomplished so far is to map *x=10* to *z= 38*. According to the three rules, we can map every point of **x** in range 0–255 to some value of **z** as shown in Figure 5. Now the reader might be wondering what difference does it make if we just implement a look-up table to describe the relationship in Figure 5. The answer is, for a one-dimensional sensor input, you can implement exactly the same sensor to actuator mapping with a table and possibly save code complexity and memory space. It is, however, not so obvious how to implement multi-dimensional sensor to actuator mapping with tables. Furthermore, the fuzzy logic method allows the user to tune the system more easily. For example, in order to change a mapping relationship between input and output, for most people, it is more intuitive to change a set of linguistic rules instead of an array of parameters in a table.[5]

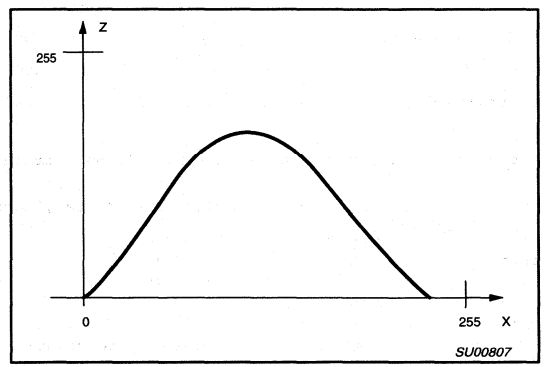


Figure 5. The curve in this figure represent a mapping relationship from *x* to *z*.
This relationship is interpolated from the above three fuzzy rules.

5. An important point has to be stressed now before we go on. A mapping relationship implemented by fuzzy logic is no different from that implemented by a mathematical function. Such a relationship is clearly defined and fully deterministic. Once the input membership functions are defined, the process of translating rules into mapping functions is strictly conventional algebra. The buzz word "fuzzy" is thus very misleading.

To implement the above algorithm in an XA, we need to consider the following issues. Since the membership functions do not usually change at run time, I use arrays stored in the code memory space to represent input membership functions. With this approach, I will not lose membership function information during power down, and also I can get membership functions of any shape. An easy way to choose the array size (dimension) for membership functions is to let the array size equal to the resolution of the AD conversion. For example, with an 8-bit A/D input, I use arrays of size 256 to represent input membership functions. Furthermore, I also use 8-bit unsigned integers to represent the membership grades so that they go from 0 to 255 instead of 0 to 1. Suppose we have multiple input channels and for each channel, we divide the domain into a number of clusters; the total number membership function would be the number of input channels times the number of clusters in each input. For the example in Figure 4, we need three arrays or membership functions to characterize input **x**. In our robot application, we need a total of 40 membership function arrays and that takes about 10K code memory space.[6]

The following is an example that shows how to perform "fuzzification". It plugs input value x into the membership function array that stands for "x is small". The XA instruction **movc A, [A+DPTR]** (code memory access with indirect addressing with an offset) is used to access membership function data. this is an 80C51 compatible instruction. In XA, A is mapped to R4L and DPTR is mapped to R6.

```
x_small:        db      $ff,$fd,$f8,$f0, ...    ;Membership function for "x is small".

x               data    10h                     ;Input value x.
antecedent      data    11h                     ;The resultant truth value of the antecedent: "x is small".

;To find out the truth value of x being "small":
                mov.w           r6,#x_small     ;Indirect pointer.
                mov.b           r4l,x           ;Offset.
                movc            A,[A+DPTR]      ;Code memory access.
                mov             antecedent,r4l  ;Return the result.
```

Once we have an appropriate way to implement membership functions, it is fairly straightforward to evaluate rules. In case there are multiple antecedents in each rule, however, we need to additionally implement some kind of **min()** or **max()** function to evaluate "and", "or" relationships. The following is a code example that implements the min() function for fuzzy rule evaluation

```
num_antecedents                 equ     4       ;Number of antecedents in a rule, e.g. 4.

antecedents                     data    20h     ;Truth values of the antecedents.

truth_rule                      data    30h     ;The resultant truth value of the rule.


;To evaluate the truth value of the rule with multiple antecedents:

                mov.w           r0,#antecedents ;Index to antecedents.

                mov.b           r1l,[r0+]
loop:           cmp             r1l,[r0]
                bl              proceed
                mov.b           r1l,[r0]
proceed:        add             r0,#1
                cmp             r0,#antecedents+num_antecedents
                bcs             loop            ;Loop "num_antecedents" times.
```

6. If cost of memory space is a concern, there are other ways to implement input membership functions. For example, we can specify a trapezoid membership function with a few key parameters instead of a 256 dimensional array. We have to then write a subroutine to map inputs into fuzzy membership grades according to these parameters.

The last part of fuzzy logic loop, the "defuzzification" process is most computationally expensive. As shown in equation (1), we need to perform a series of 8×8 to 16-bit multiplications and a 32×16 to 16 division to get one final output value. This is to assume that our sensor values and membership grades have 8-bit resolution. We need to use 32-bit (long) integer to represent the numerator of equation (1) and 16-bit integer to represent the denominator. The following is an example of the defuzzification code segment.

```
num_rules               equ     4       ;Number of rules, e.g. 4.
truth_rules             data    10h     ;The truth value of the rules (an array).

singletons              data    20h     ;The output singleton functions (an array);

z                       data    21h     ;The crisp output value.

;To perform the defuzzification process
            mov.w       r0,#0           ;Index to the rules.
            mov.b       r1h,#0          ;Clear the high order bits of r1.

            mov.w       r4,#0           ;Initialize low order bits of the numerator.

            mov.w       r5,#0           ;Initialize high order bits of the numerator.

            mov.w       r6,#0           ;Initialize the denominator.

loop:       mov.b       r1l,[r0+truth_rules]
            mov.b       r2l,[r0+singletons]
            mulu.b      r2l,r1l         ;8x8=16 multiplication.
            add.w       r4,r2           ;r4 stores the numerator.
            addc.w      r5,#0           ;add carry to higher bits.
            add.w       r6,r1           ;calculate the denominator.
            add.w       r0,#1           ;increment the index.
            cmp         r01,#num_rules
            bcs         loop

            divu.d  r4, r6              ;32/16 -> 16 unsigned division.
            mov.w   z, r4
```

The above code segments serve as examples to illustrate how to efficiently use the XA instruction to perform the basic fuzzy logic operation. One would still have to decide on how to encode rules and control the timing of the peripheral access. Most fuzzy logic controllers sample sensor inputs and update actuator outputs synchronously at fix time intervals. The XA provides a number of internal timers which can be used to control the timing of peripheral access.

IMPLEMENTATION OF A COMPLIANT ROBOT ACTUATOR THROUGH SENSORY FEEDBACK IN AN 8XC552

As stated earlier, we intend to use fuzzy logic in an augmented fashion. In this application, the low level servo control can still be handled more easily with conventional linear feedback. In this section, we focus on these low level implementation issues. Specifically, we will discuss the interface between the 8XC552 and the sensors and actuators of the robot leg.

Most robot actuators use position feedback to implement a closed-loop position servo. In order to achieve position accuracy, those actuators are usually quite rigid. Although robots powered by this kind of servo are usually able to make unconstrained movement smoothly and quickly, they become unstable and behave erratically upon contact with external objects [3]. With sufficient power, this kind of robot could also be dangerous to human operators and things around it. It is therefore often necessary to avoid any contact situation. Animal and human muscles, on the other hand, are very versatile due to the fact that they are usually compliant and more importantly, the compliance can be actively controlled.

Figure 6 illustrates the implementation of our robot actuator as a "virtual muscle". I use a DC gearmotor as the core. Each motor is integrated with a position sensor (potentiometer) and a feedback circuit that acts as a position servo. Since the gear motor is non-backdrivable, without the additional circuitry described below, the servo system is quite rigid, that is to say, the output angle is determined by the input command signal, and largely unaffected by external torques acting on the joint. To achieve actuator properties that resemble those of muscles, I add additional feedback pathways through an 8XC552 to allow us to control compliance and damping properties. The torque signal from strain gauges is fed back to the position command signal to form a compliance feedback loop. The gain of the compliance loop determines the extent to which the servo moves in response to external forces, thus establishing compliant properties (see Figure 6, the outer loop). The dynamic properties of the integrated sensor-actuator such as the compliance and the damping ratio can be controlled by adjusting the variable gains and low pass filter time constants in the compliance feedback loop. With compliant robot joint actuators, we effectively added a cushion between the robot and the objects it is in contact with and therefore get a significant improvement in contact stability [3], [4]. With an adjustable joint compliance, the robot can serve as both a contact based probe and an effector that is capable of exerting forces and maintaining positional accuracy depending on the behavior context.

The 8XC552 carries out both position and force feedback. The feedback loop is implemented in a timer interrupt service routine that is called every 0.1ms. After the 8XC552 completes the sensory feedback function for the tuning of a compliant actuator dynamics, it stores copies of all the sensor values in a buffer for access by the XA through I^2C to control the output angle and compliance of the robot joint. The 8XC552 thus implements a compliant actuator with electronically controllable compliance and presents itself as an I^2C slave to the XA.

Notice that the feedback pathways implemented thus far are strictly linear feedback loops that are intended for actuator control. This part of the feedback can be done easily without fuzzy logic[7]. On the other hand, the control at this level has to be done in hard real time to ensure dynamic stability.

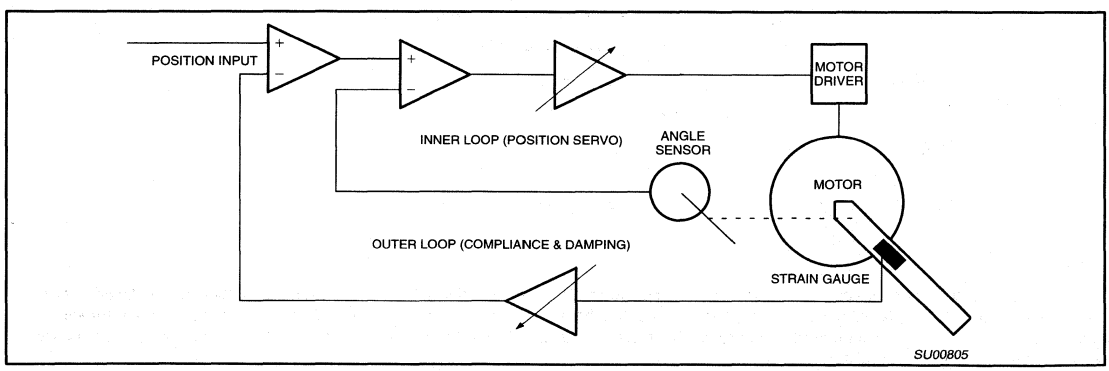


Figure 6. An integrated sensor-actuator assembly for compliant robot joint actuation.
The active compliance is accomplished through stress feedback.

7. It is possible to use fuzzy logic to make an exact linear feedback loop, but this approach would seem counter-productive.

Rule base generation

In this application, fuzzy logic is used to control the robot at higher level, that is, the coordination between joints in order to carry out meaningful motion sequence. As mentioned earlier, the main advantage of fuzzy logic is that the user can design a control system based on intuition. There are, therefore, not many rules to follow to generate the fuzzy logic rules. In this section, we discuss a few techniques that we used in this specific application.

In addition to the strain and position sensor inputs mentioned earlier, a software generated timer signal implemented in XA is fed to the fuzzy evaluator internally and this counts as another sensory input. The timer counts from 0 to 255 repeatedly and they were clustered into three fuzzy sets corresponding to the three phases of the leg searching cycle, namely "start", "probe" and "retract". This is necessary because the control of the leg involves the generation of rhythmic movement in the absence of any specific sensor inputs. The rhythmic movement ensures that the robot will engage in active searching when it is not in touch with any object. The timer input functions as a "central pattern generator".

In addition to the soft timer, there are a total of 7 sensor inputs to this system. Each of sensor values are clustered into 5 clusters. For example, a quantity ranging from 0 to 255 can be characterized by membership functions corresponding to, "very small", "small", "medium", "big", "very big". For each output, there could be as much as $5^7 * 3 = 234375$ rules. It is obviously impossible for us to manually try all the rules on "trial-and-error" basis. Notice that for this system, the signals detected from the various sensors are highly correlated. For example, when touch sensor 1 is signaling positive, touch sensor 2 is likely to signal positive also (but not vice versa). It is therefore unnecessary to try a rule like

```
.IF touch sensor 1 positive-big
   AND touch sensor 2 negative-big
   AND ...
   THEN ...
```

because this situation does not exist.

By this analysis we can reduce the number of rules significantly. Here are a set of rules that are used to give the performance shown in Figure 2. Additional rules can be put in to make the leg more versatile.

```
.IF tibia stress is zero
   AND femur stress is zero
   AND timer is start
   THEN femur output is negative-small
   AND tibia output is positive-big.

.IF tibia stress is zero
   AND femur stress is zero
   AND timer is probe
   THEN femur output is positive-big
   AND tibia output is negative-big.

.IF tibia stress is zero
   AND femur stress is zero
   AND timer is retract
   THEN femur output is negative-big
   AND tibia output is negative-big.
```

(* The above three rules are responsible for the generation of the three phased search pattern when the leg is not in touch of anything).

```
.IF touch sensor 1 is negative-small
   THEN femur output is negative-big
   AND tibia output is negative-big.

.IF touch sensor 2 is negative-small
   THEN femur output is negative-big
   AND tibia output is negative-big.

.IF tibia stress is negative-small
   THEN femur output is negative-big
   AND tibia output is negative-big.

.IF touch sensor 1 is positive-small
   THEN femur output is negative-big
   AND tibia output is zero.

.IF touch sensor 2 is positive-small
   THEN femur output is negative-big
   AND tibia output is zero.

.IF tibia stress is positive-small
   THEN femur output is negative-big
   AND tibia output is zero.

.IF femur angle is negative-big
   and tibia angle is negative-big
   THEN tibia output is positive-big.
```

(* The above rules are responsible for the retract movement when the leg is in touch with an object in a way as shown in Figure 2.)

```
.IF foot stress is negative-small
   THEN femur output is positive-small
   AND tibia output is negative-small.
```

(* This rule is responsible for the foot to keep in contact with an object by pressing onto it.)

Figure 2 gives the digitized trajectory plots of the "substrate finding" behavior performed by our robot leg. When the robot leg is not in contact with anything, it carries out a three-phased searching movement. As soon as the leg touches an object, it would generate reflexes as shown in Figure 2. For example, in Figure 2a, the tibia would press against the object while slipping upwards. As soon as the tibia just clears the object, the robot will reposition the foot on to the object and keep a pressure. If the substrate moves, the leg is able to adjust promptly to maintain contact with the substrate due to the joint compliance. Even though there is no visual guidance, with active sensing, the robot leg is able to find and grab onto any firm object quite reliably.

DISCUSSIONS

The feedback pathways in a control system can often be categorized into two classes, linear (e.g., PID control) and nonlinear, and they often serve quite different purposes. In this application, the linear feedback control loops are implemented to "tune" the robot joint dynamics in some desired fashion, i.e., position servo with some compliance, whereas the nonlinear feedback control reflexes are used to control the coordination between multiple robot joints in order to achieve a more concrete objective such as the requirement for the robot to grab and hold onto an object. The linear feedback algorithms are usually straightforward to implement but they generally have high speed requirements for stability reasons. Nonlinear feedbacks, on the other hand are usually computationally more intensive due to the requirements for interpolation (fuzzy logic algorithm does exactly that). This requirement will usually slow things down a little bit. In this paradigm, the stability and robustness of a system depends critically on the speed of the linear feedback layer and is somewhat less sensitive to the speed of the fuzzy logic loop. We envision that with our next generation of XA (XA-S3). We can integrate all of these functions into one chip. We will use the XA multi-tasking capabilities so that we implement several layers of feedback, some of which carry out simple, but fast servoing for actuator control and the others running fuzzy logic for goal-directed motor sequencing behavior.

REFERENCES

[1] Castro, J.L., Fuzzy logic controllers are universal approximators. *IEEE transactions on system, man, and cybernetics*, Vol. 25, No. 4, 629-635.

[2] Bassler, U. (1991) Interruption of searching movements of partly restrained front legs of stick insects, a model situation for the start of a stance phase? *Biol. Cybern.* 65, 507-514.

[3] Hogan, N. (1988) On the stability of manipulators performing contact tasks. *IEEE Journal of Robotics and Automation*, vol. 4, 677-686.

[4] Ding, Z., Nelson, M.E. (1995) A neural controller for single-leg substrate-finding: a first step toward agile locomotion in insects and robots. In: *Computation and Neural Systems 3* F. Eeckman and J.M. Bower, eds., Kluwer Academics Press.

[5] Data Handbook IC25: *16-bit 80C51XA Microcontrollers (eXtended Architecture)*. Philips Semiconductors, 1996.

µC/OS for the Philips XA

AN711

Author: Jean J. Labrosse

SUMMARY

A real-time kernel is software that manages the time of a microprocessor or microcontroller to ensure that all time critical events are processed as efficiently as possible. This application note describes how a real-time kernel, µC/OS works with the Philips XA microcontroller. The application note assumes that you are familiar with the XA and the C programming language.

INTRODUCTION

A real-time kernel allows your project to be divided into multiple independent elements called *tasks*. A task is a simple program which competes for CPU time. With most real-time kernels, each task is given a priority based on its importance. When you design a product using a real-time kernel you split the work to be done into tasks which are responsible for a portion of the problem. A real-time kernel also provides valuable services to your application such as time delays, system time, message passing, synchronization, mutual-exclusion and more.

Most real-time kernels are *preemptive*. A preemptive kernel ensures that the highest-priority task ready-to-run is always given control of the CPU. When an ISR (Interrupt Service Routine) makes a higher-priority task ready-to-run, the higher-priority task will be given control of the CPU as soon as all nested interrupts complete. The execution profile of a system designed using a preemptive kernel is illustrated in Figure 1. As shown, a low-priority task is executing ①.

An asynchronous event interrupts the microprocessor ②. The microprocessor services the interrupt ③ which makes a high-priority task ready for execution. Upon completion, the ISR invokes a service provided by the kernel which decides to return to the high-priority task instead of the low-priority task ④. The high-priority task executes to completion, unless it also gets interrupted ⑤. At the end of the high-priority task, the kernel resumes the low-priority task ⑥. As you can see, the kernel ensures that time critical tasks are performed first. Furthermore, execution of time critical tasks are deterministic and are almost insensitive to code changes. In fact, in many cases, you can add low-priority tasks without affecting the responsiveness of you system to high-priority tasks. During normal execution, a low-priority task can make a higher-priority task ready for execution. At that point, the kernel immediately suspends execution of the lower priority task in order to resume the higher priority one.

A real–time kernel basically performs two operations: *Scheduling* and *Context Switching*. Scheduling is the process of determining whether there is a higher priority task ready to run. When a higher–priority task needs to be executed, the kernel must save all the information needed to eventually resume the task that is being suspended. The information saved is called the *task context*. The task context generally consist of most, if not all, CPU registers. When switching to a higher priority task, the kernel perform the reverse process by loading the context of the new task into the CPU so that the task can resume execution where it left off.

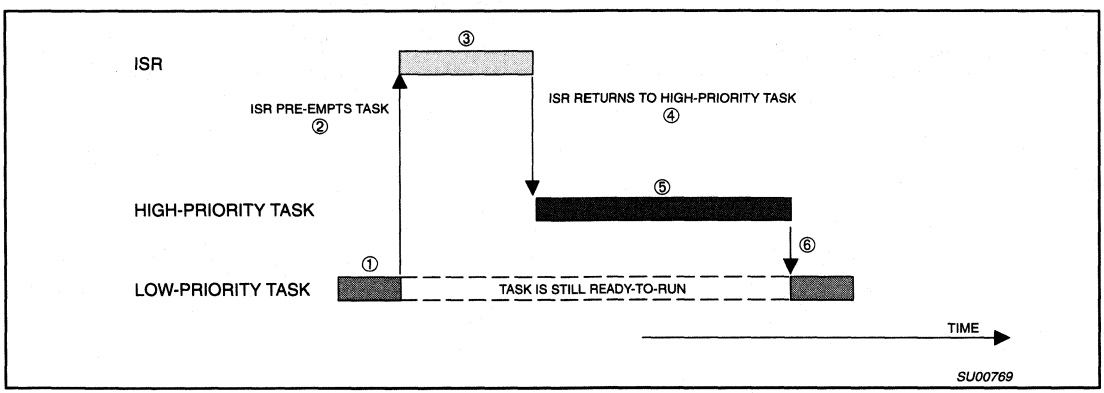


Figure 1.

THE PHILIPS XA AND REAL–TIME KERNELS

The XA has a number of interesting features which makes it particularly well suited for real–time kernels.

When you use a kernel, each task requires its own stack space. The size of the stack required for each task is application specific but basically depends on function call nesting, allocation of local variables for each function and the worst case interrupt requirements. Unlike other processors, the XA provides two stack spaces: a *System Stack* and a *User Stack*. The System Stack is automatically used when processing interrupts and exceptions. The User Stack is used by your application tasks for subroutine nesting and storage of local variables. The most important benefit of using two stacks is that you don't need to allocate extra space on the stack of each task to accommodate for interrupt nesting. This feature greatly reduces the amount of RAM needed in your product. With the XA, the total amount of RAM needed just for stacks is given by:

$$TotalRAM_{Stack} = ISRStack_{Max} + \sum_{i=1}^{n} TaskStack_i$$

The XA divide its 16 MBytes of data address into 256 *segments* of 64 Kbytes. The stack for each task can be isolated from each other by having them reside in their own segment. The XA protects each stack by preventing a task from accessing another task's stack. This feature can prevent an errant task from corrupting other tasks.

Scheduling and task–switching can eat up valuable CPU time which directly translates to overhead. A processor with an efficient instruction set such as that found on the XA helps reduce the time spent performing scheduling and context switching. For instance, the XA provides two instructions to PUSH and POP multiple registers onto and from the stack, respectively. This feature makes for a fast context switch because all seven registers (R0 through R6) can be saved and restored onto and from the stack in just 42 clock cycles whereas it would take 70 clock cycles to perform the same function with individual PUSH and POP instructions.

µC/OS

µC/OS (pronounced *micro C OS*) is a portable, ROMable, preemptive, real–time, multitasking kernel and can manage up to 63 tasks. The internals of µC/OS are described in my book called: *µC/OS, The Real–Time Kernel* [1]. The book also includes a floppy disk containing all the source code. µC/OS is written in C for sake of portability, however, microprocessor specific code is written in assembly language. Assembly language and microprocessor specific code is kept to a minimum. µC/OS is comparable in performance with many commercially available kernels. The execution time for every service provided by µC/OS (except one) is both deterministic and constant. µC/OS allows you to:

- Create and manage up to 63 tasks,
- Create and manage binary or counting semaphores,
- Delay tasks for integral number of ticks,
- Lock/Unlock the scheduler,
- Change the priority of tasks,
- Delete tasks,
- Suspend and resume tasks and,
- Send messages from an ISR or a task to other tasks.

USING µC/OS

µC/OS requires that you call `OSInit()` before you start using any of the other services provided by µC/OS. After calling `OSInit()` you will need to create at least one task before you start multitasking (i.e., before calling `OSStart()`). All tasks managed by µC/OS needs to be **created**. You create a task by simply calling a service provided by µC/OS (described later). You need to create each task in order to prepare them for multitasking. If you want, you can create all your tasks before calling `OSStart()`. Once multitasking starts, µC/OS will start executing the highest priority task that has been created. You should note that interrupts will be enabled as soon as the first task starts execution. Your `main()` function will thus look as shown in Figure 2.

A task under µC/OS must always be written as an infinite loop as shown in Figure 3. When your task first executes, it will be passed an argument (`pdata`) which can be made to point to task specific data when the task is created. If you don't use this feature, you should simply equate `pdata` to `pdata` as shown below to prevent the compiler from generating a warning. Even though a task is an infinite loop, it must not use up all of the CPU's time. To allow other tasks to get a chance to execute, you have to write each task such that the task either suspends itself until some amount of time expires, wait for a semaphore, wait for a message from either another task or an ISR or simply suspend itself indefinitely until explicitly resumed by another task or an ISR. µC/OS provides services to accomplish this.

A task is created by calling the `OSTaskCreate()` function. `OSTaskCreate()` requires four arguments as shown in the function prototype of Figure 4.

`task` is a pointer to the task you are creating. *`pdata`* is a pointer to an optional argument that you can pass to your task when it begins execution. This feature allows you to write a generic task which is personalized based on arguments passed to it. For example, you can design a generic serial port driver task which gets passed a pointer to a structure defining the ports parameters such as the address of the port, its interrupt vector, the baud rate etc. *`pstk`* is a pointer to the task's top–of–stack. Finally, *`prio`* is the task's priority. With µC/OS, each task must have a unique priority. The smaller the priority number, the more important the task is. In other words, a task having a priority of 10 is more important than a task with a priority of 20.

With µC/OS, each task can have a different stack size. This feature greatly reduces the amount of RAM needed because a task with a small stack requirement doesn't get penalized because another task in your system requires a large amount of stack space. You should note that you can locate a task's stack just about anywhere in the XA's address space. This is accomplished by specifying the task's top–of–stack through a constant (or a #define) as shown in the two examples of Figure 5.

Here, I located *task1*'s stack at the top of page 0 while *task2*'s stack will start at offset 0xF700 of page 7 and grow downwards from there. When locating stacks using constants, you must be careful that the linker does not locate data at these memory locations. If needed, you can also locate the stacks of multiple tasks in the same page using the same technique.

`OSTaskCreate()` returns a value back to its caller to notify it about whether the task creation was successful or not. When a task is created, µC/OS assigns a *Task Control Block (TCB)* to the task. The TCB is used by µC/OS to store the priority of the task, the current state of the task (ready, waiting for an event, delayed, etc.), the current location of the task's top–of–stack and, other kernel related data.

Table 1 shows the function prototypes of the services provided by µC/OS, V1.09. The prototypes are shown in tabular form for sake of discussion. The actual prototype of `OSTimeDly()` for example is actually:

```
void OSTimeDly(UWORD ticks);
```

You will notice that every function starts with the letters 'OS'. This makes it easier for you to know that the function call is related to a kernel service (i.e., an **O**perating **S**ystem call). Also, the function naming convention groups services by functions: 'OSTask...' are task management functions, 'OSTime...' are time management functions, etc. Another item you should notice is that non–standard data types are in upper–case: UBYTE, UWORD, ULONG and OS_EVENT. UBYTE, UWORD and ULONG represent an *unsigned–byte* (8–bit), an *unsigned–word* (16–bit), and an *unsigned–long* (32–bit), respectively. OS_EVENT is a typedef'ed data structure declared in UCOS.H and is used to hold information related to semaphore, message mailboxes and message queues. Your application will in fact have to declare storage for a pointer to this data structure as follows:

```
far OS_EVENT *MySem;
```

The 'far' attribute is specific to the HI–TECH compiler (described later) and indicates that the pointer `MySem` will be able to access the OS_EVENT data structure which may be located in another bank. OS_EVENT is used in the same capacity as the FILE data–type used in standard C library. OSSemCreate(), `OSMboxCreate()` and `OSQCreate()` return a pointer which is used to identify the semaphore, mailbox or queue, respectively.

```
void main(void)
{
    /* Perform XA Initializations                          */
    OSInit();
    /* Create at least one task by calling OSTaskCreate()  */
    OSStart();
}
```

SU00770

Figure 2.

```
void UserTask(far void *pdata)
{
    pdata = pdata;
    /* User task initialization                            */
    while (1) {
        /* User code goes here                             */
        /* You MUST invoke a service provided by µC/OS to: */
        /* ... a) Delay the task for 'n' ticks             */
        /* ... b) Wait on a semaphore                      */
        /* ... c) Wait for a message from a task or an ISR */
        /* ... d) Suspend execution of this task           */
    }
}
```

SU00771

Figure 3.

```
UBYTE OSTaskCreate(void (*task)(far void *pd),
                   far void *pdata,
                   far void *pstk,
                   UBYTE prio);
```

SU00772

Figure 4.

```
UBYTE OSTaskCreate(task1, pdata1, (far void *)0x00FFFE, prio1);
UBYTE OSTaskCreate(task2, pdata2, (far void *)0x07F700, prio2);
```

SU00773

Figure 5.

Table 1. µC/OS V1.09
Philips XA, Large Model

RETURN VALUE	FUNCTION NAME	ARGUMENT #1	ARGUMENT #2	ARGUMENT #3	ARGUMENT #4	CALLED FROM...
Initialization						
void	OSInit	void	–	–	–	main()
void	OSStart	void	–	–	–	main()
Task Management						
UBYTE	OSTaskCreate	void (task)(far void *pd)	far void *pdata	far void *pstk	UBYTE prio	main() or Task
UBYTE	OSTaskDel	UBYTE prio	–	–	–	Task
UBYTE	OSTaskDelReq	UBYTE prio	–	–	–	Task
UBYTE	OSTaskChangePrio			–	–	Task
UBYTE	OSTaskSuspend	UBYTE prio	–	–	–	Task or ISR
UBYTE	OSTaskResume	UBYTE prio	–	–	–	Task or ISR
void	OSSchedLock	void	–	–	–	Task or ISR
void	OSSchedUnlock	void	–	–	–	Task or ISR
Time Management						
void	OSTimeDly	UWORD ticks	–	–	–	Task
UBYTE	OSTimeDlyResume	UBYTE prio	–	–	–	Task
void	OSTimeSet	ULONG ticks	–	–	–	Task or ISR
ULONG	OSTimeGet	void	–	–	–	Task or ISR
Semaphore Management						
far OS_EVENT *	OSSemCreate	UWORD value	–	–	–	Task
UWORD	OSSemAccept	far OS_EVENT *pevent	–	–	–	Task or ISR
UBYTE	OSSemPost	far OS_EVENT *pevent	–	–	–	Task or ISR
void	OSSemPend	far OS_EVENT *pevent	UWORD timeout	UBYTE *err		Task
Message Mailbox Management						
far OS_EVENT *	OSMboxCreate	far void *msg	–	–	–	Task
far void *	OSMboxAccept	far OS_EVENT *pevent	–	–	–	Task or ISR
UBYTE	OSMboxPost	far OS_EVENT *pevent	far void *msg	–	–	Task or ISR
far void *	OSMboxPend	far OS_EVENT *pevent	UWORD timeout	UBYTE *err	–	Task
Message Queue Management						
far OS_EVENT *	OSQCreate	far void **start	UBYTE size	–	–	Task
far void *	OSQAccept	far OS_EVENT *pevent	–	–	–	Task or ISR
UBYTE	OSQPost	far OS_EVENT *pevent	far void *msg	–	–	Task or ISR
far void *	OSQPend	far OS_EVENT *pevent	UWORD timeout	UBYTE *err	–	Task
Interrupt Management						
void	OSIntEnter	void	–	–	–	ISR
void	OSIntExit	void	–	–	–	ISR

µC/OS AND THE PHILIPS XA

µC/OS (V1.09) was ported to the XA using the *HI–TECH C XA* tool chain and the complete source code for both µC/OS and the port to the XA's *Large Memory Model* are available from Philips Semiconductors, Inc. The large memory model allows you to write very large applications (up to 16 Mbytes of code) and access a lot of data memory (up to 16 Mbytes). The XA port has been tested on the Future Design, Inc. XTEND–G3 evaluation board and the test code provided with the port is assumed to run on this target. The test code can, however, be easily modified to support other environments. The large model requires that your XTEND board has at least two pages of data RAM. In other words, you must have more than 64K bytes of RAM in the XA's addressable data area. This can be easily accomplished by replacing the two 32K bytes data RAM chips with two 128K bytes chips.

A number of assumptions have been made about how µC/OS uses the XA. µC/OS will run the XA in *Native Mode.* This allows the compiler to use as many new features of the XA as possible and does not make any effort to be backwards compatible with the 80C51.

Your application code and most of µC/OS services will be executing in *User* mode. The XA will automatically be placed in *System* mode when either an interrupt or an exception occurs or, when µC/OS performs a context switch. Each of your application task will require its own stack space in *Banked RAM* while all interrupts will share the system mode stack. µC/OS allows you to specify a different stack size for each task. In other words, µC/OS doesn't require that the stack for each task be the same size. This feature prevents you from wasting valuable RAM when the stack requirements for each task varies.

µC/OS will only manipulate the registers in bank #0. If your application code changes register bank, you will need to ensure that your code restores register bank #0 prior to using any of µC/OS's services.

µC/OS requires a periodic interrupt source to maintain system time and provide time delay and timeout services. This periodic interrupt is called a *System Tick* and needs to occur between 10 and 100 times per second. The system tick can be generated by using any of the XA's three internal timers or externally through the INT0 or INT1 inputs. For lack of a better choice, I used timer #0 and configured it for a 100 Hz tick rate. The tick interrupt vectors to an assembly language function called `OSTickISR`. If you application requires the use of all of the XA's timers then you will have to find another source for the 'ticker'. For example, if your system is powered from a power grid, you can bring the line frequency (50 or 60 Hz) in through either INT0 or INT1.

µC/OS also requires one of the 16 TRAP vectors in order to perform a context switch. I decided to use `TRAP #15` which is defined in the C macro `OS_TASK_SW()`. A context switch will force the XA into system mode and push the return address and the PSW onto the system stack.

As previously mentioned, you must prepare your tasks for multitasking by calling `OSTaskCreate()`. `OSTaskCreate()` builds the stack frame for the task being created as illustrated in Figure 6. DS:USP indicates that once the task gets to execute, the stack will be located in the bank selected by the DS register at an offset supplied by the USP. You should note that pointers in the large model are 32–bits but, only the least significant 24 bits are used. `OSTaskCreate()` first sets up the stack to make it look as if your task has just been called by another C function ①. In other words, when your task first executes, it will think it was called by another function since the stack pointer will point as shown in ②. `OSTaskCreate()` then simulates the stacking order of a `PUSHU R0-R6` instruction ③ which is needed for a context switch. The initial value of each register is set to the values shown for debugging purposes and can thus be changed as needed. Next, `OSTaskCreate()` stacks both the ES register and the SSEL register ④. Even though both the ES and SSEL registers are 8–bit, they are stacked as two 16–bit values because all XA stacking operations are 16–bit. The SSEL register is initialized to 0x80 to allow your task to read and write data anywhere in the 16–MBytes data address space. You may not want to change the initial value of the SSEL register because the compiler will not know that write through the ES register is not allowed (run–time) but, it will generate code (compile–time) as if it was. During an interrupt or a context switch, the XA pushes the PC and the PSW onto the system stack. The stacking order of these registers as shown on the stack frame of Figure 6 is reversed because `OSTaskCreate()` simulates a move of these registers from the system stack to the user stack ⑤.

As previously mentioned, multitasking starts when you call `OSStart()`. Figure 7 illustrates the process. `OSStart()` finds the TCB of the highest priority task that you created, loads the pointer `OSTCBHighRdy` to point to that TCB ① and calls the assembly language function `OSStartHighRdy`. `OSStartHighRdy` loads the USP and the DS register from the task's TCB ② and then moves the start address of your task, along with the PSW from the user stack to the system stack ③. `OSStartHighRdy` then pops the remaining registers from the user stack ④ and finally, `OSStartHighRdy` executes a return from interrupt which loads the PC and PSW from the system stack ⑤ into the XA. Because the PSW was initialized to 0x0000, the XA will now execute the first instructions of your task in user mode with all interrupts enabled.

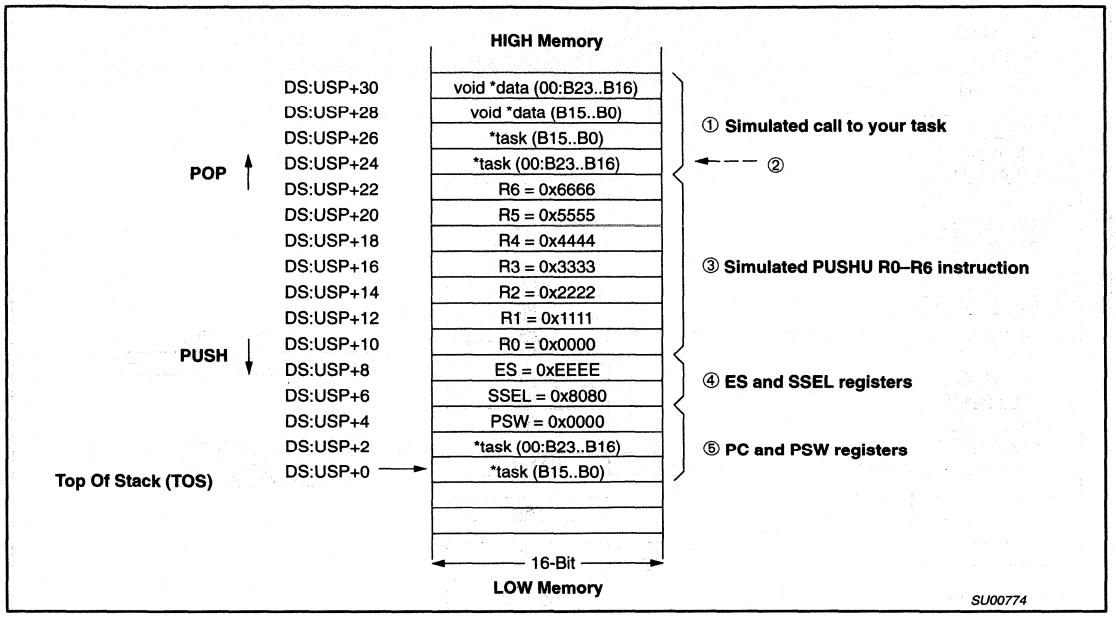

HIGH Memory
DS:USP+30
DS:USP+28
DS:USP+26
DS:USP+24
DS:USP+22
DS:USP+20
DS:USP+18
DS:USP+16
DS:USP+14
DS:USP+12
DS:USP+10
DS:USP+8
DS:USP+6
DS:USP+4
DS:USP+2
DS:USP+0
POP
PUSH
Top Of Stack (TOS)
void *data (00:B23..B16)
void *data (B15..B0)
*task (B15..B0)
*task (00:B23..B16)
R6 = 0x6666
R5 = 0x5555
R4 = 0x4444
R3 = 0x3333
R2 = 0x2222
R1 = 0x1111
R0 = 0x0000
ES = 0xEEEE
SSEL = 0x8080
PSW = 0x0000
*task (00:B23..B16)
*task (B15..B0)
16-Bit
LOW Memory
① Simulated call to your task
②
③ Simulated PUSHU R0–R6 instruction
④ ES and SSEL registers
⑤ PC and PSW registers
SU00774

Figure 6.

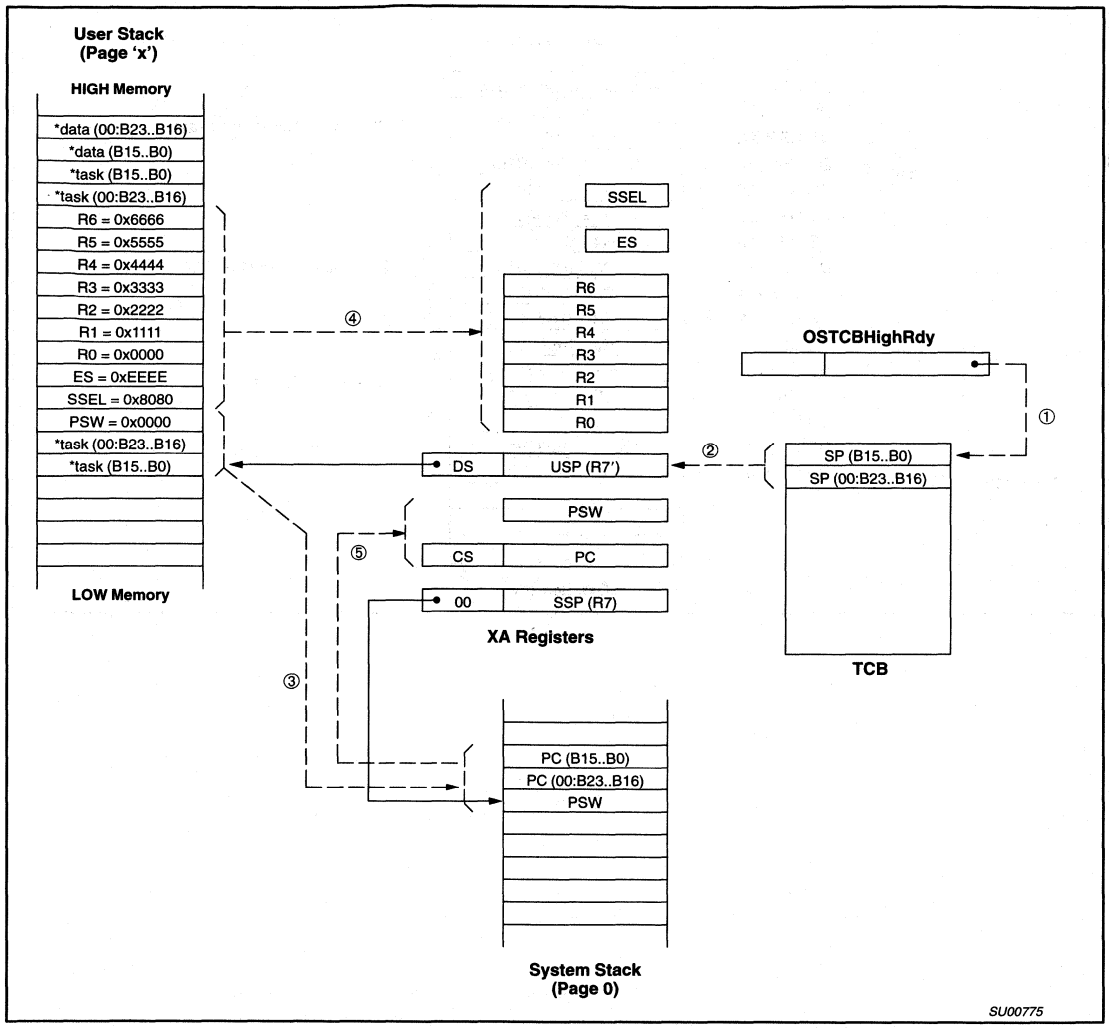

User Stack
(Page 'x')
HIGH Memory
*data (00:B23..B16)
*data (B15..B0)
*task (B15..B0)
*task (00:B23..B16)
R6 = 0x6666
R5 = 0x5555
R4 = 0x4444
R3 = 0x3333
R2 = 0x2222
R1 = 0x1111
R0 = 0x0000
ES = 0xEEEE
SSEL = 0x8080
PSW = 0x0000
*task (00:B23..B16)
*task (B15..B0)
LOW Memory
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
④
OSTCBHighRdy
①
②
SP (B15..B0)
SP (00:B23..B16)
DS
USP (R7')
PSW
⑤
CS
PC
00
SSP (R7)
XA Registers
TCB
③
PC (B15..B0)
PC (00:B23..B16)
PSW
System Stack
(Page 0)
SU00775

Figure 7.

CONTEXT SWITCHING WITH µC/OS

Because µC/OS is a preemptive kernel, it always executes the highest priority task that is ready to run. As your tasks execute they will eventually invoke a service provided by µC/OS to either wait for time to expire, wait on a semaphore or wait for a message from another task or an ISR. A context switch will result when the outcome of the service is such that the currently running task cannot continue execution. For example, Figure 8 shows what happens when a task decides to delay itself for a number of ticks. In ①, the task calls `OSTimeDly()` which is a service provided by µC/OS. `OSTimeDly()` places the task in a list of tasks waiting for time to expire ②. Because the task is no longer able to execute, the scheduler (`OSSched()`) is invoked to find the next most important task to run ③. A context switch is performed by issuing a TRAP #15 instruction ④. The function `OSCtxSw()` is written entirely in assembly language because it directly manipulates XA registers. All execution times are shown assuming a 24 MHz crystal and the large model. The highest priority task executes at the completion of the XA's RETI instruction ⑤.

The work done by `OSCtxSw()` is illustrated in Figure 9. The scheduler loads `OSTCBHighRdy` with the address of the new task's TCB ① before invoking `OSCtxSw()` which is done through the `TRAP #15` instruction. `OSTCBCur` already points to the TCB of the task to suspend. The `TRAP #15` instruction automatically pushes the return address and the PSW onto the system stack ②. `OSCtxSw()` starts off by saving the remainder of the XA's registers onto the user stack ③ and then, moves the saved PSW and PC from the system stack to the user stack ④. The final step in saving the context of the task to be suspended is to store the top-of-stack into the current task's TCB ⑤. The second half of the context switch operation restores the context of the new task. This is performed in the following four steps. First, the user stack pointer is loaded with the new task's top–of–stack ⑥. Second, the PC and PSW of the task to resume is moved from the user stack to the system stack ⑦. Third, the remainder of the XA's registers are restored from the user stack ⑧. Finally, a return from interrupt instruction (RETI) is executed ⑨ to retrieve the new task's PC and PSW from the system stack which causes the new task to resume execution where it left off. As shown in Figure 7, a context switch for the large model takes only about 10µs at 24MHz.

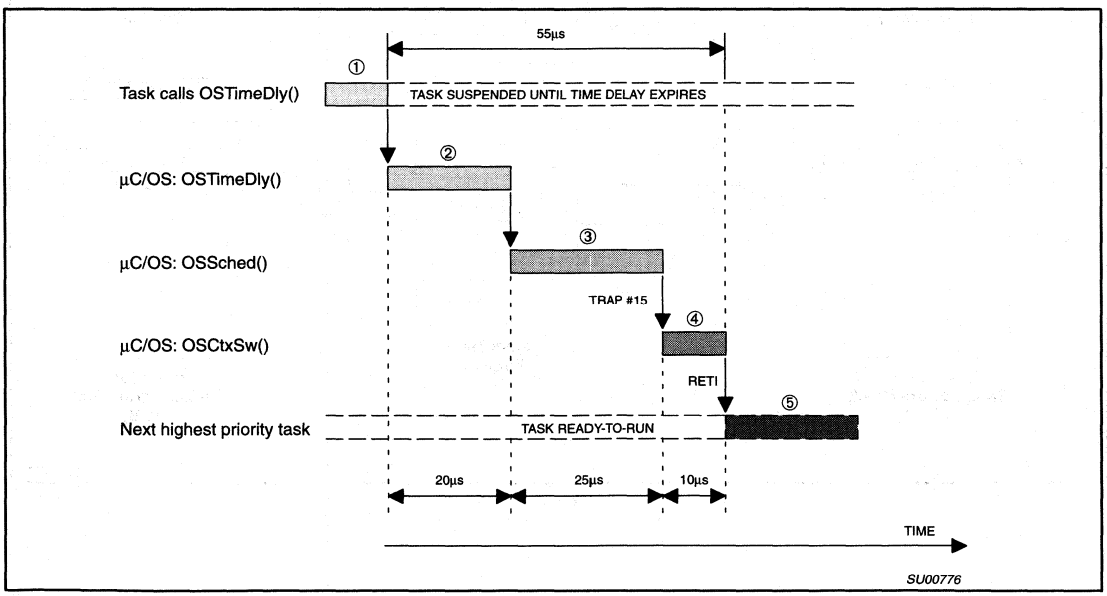


Figure 8.

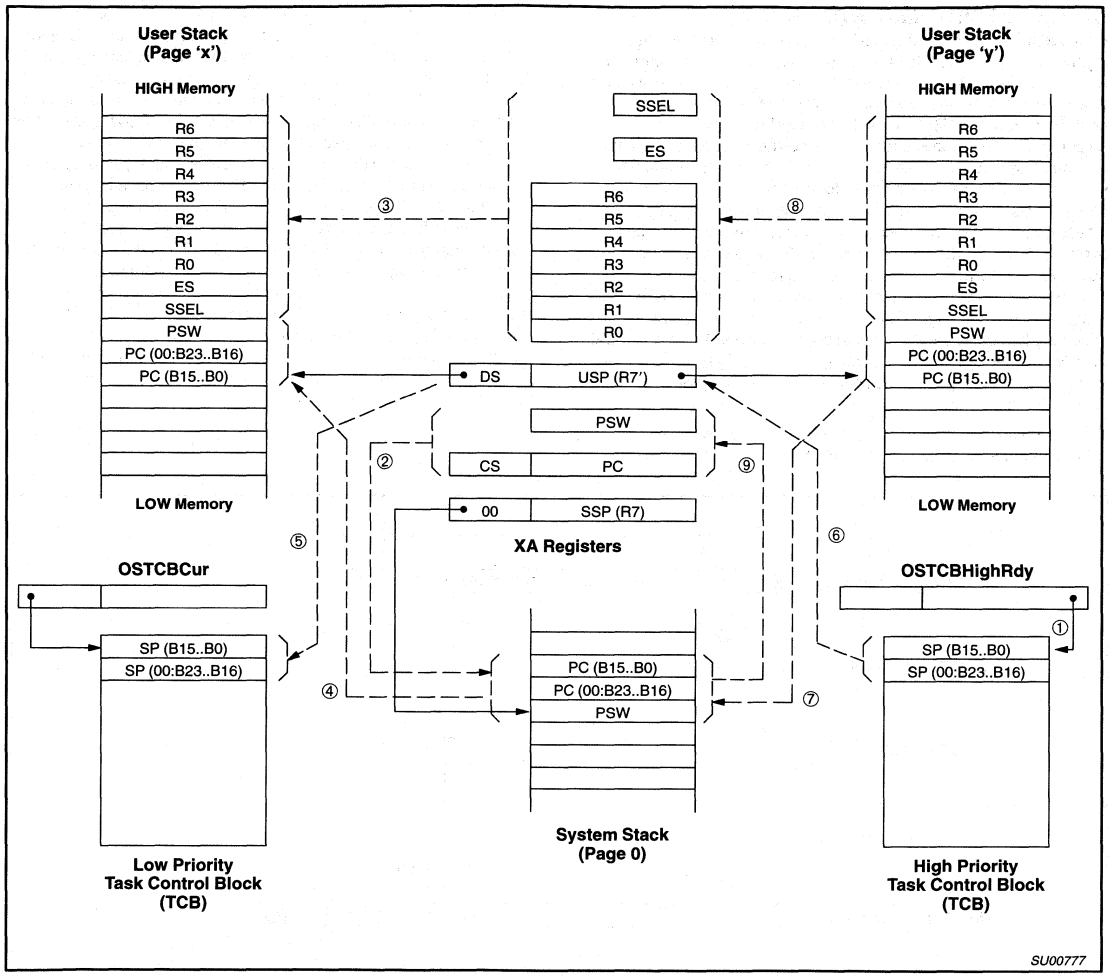

User Stack
(Page 'x')
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (00:B23..B16)
PC (B15..B0)
LOW Memory
User Stack
(Page 'y')
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (00:B23..B16)
PC (B15..B0)
LOW Memory
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
DS
USP (R7')
PSW
CS
PC
00
SSP (R7)
XA Registers
OSTCBCur
SP (B15..B0)
SP (00:B23..B16)
OSTCBHighRdy
SP (B15..B0)
SP (00:B23..B16)
PC (B15..B0)
PC (00:B23..B16)
PSW
System Stack
(Page 0)
Low Priority
Task Control Block
(TCB)
High Priority
Task Control Block
(TCB)
①
②
③
④
⑤
⑥
⑦
⑧
⑨
SU00777

Figure 9.

INTERRUPT SERVICE ROUTINES (ISRS) AND µC/OS

Under µC/OS, you must write your ISRs in assembly language as shown in Figure 10. This code assumes you are using the HI–TECH C compiler and assembler. You must always save all the registers at the beginning of the ISR and restore them at the completion of the ISR. You must also always notify µC/OS when you are starting to process an ISR by calling `OSIntEnter()`. `OSIntEnter()` simply increments an interrupt nesting counter and thus takes very little time to execute. You can either process the interrupting device direcly in assembly language or, call a C handler as shown in Figure 9. Note that you are responsible for clearing the interrupt. When you are done processing the interrupt, you must call `OSIntExit()`. `OSIntExit()` decrements the nesting counter and, when the nesting counter reaches 0, all interrupts have nested and the scheduler is invoked to determine whether the ISR needs to return to the interrupted task or, whether a higher priority task has been made ready to run by one of the ISRs. If there is a higher priority task, µC/OS will need to perform a context switch to return to the more important task.

The stack frames (system and user) during an interrupt is shown in Figure 11. Items ①, ② and ③ are performed at the beginning of your ISR. When you call `OSIntExit()`, the return address is pushed onto the system stack. `OSIntExit()` creates a local variable on the stack by saving R4. If a context switch is needed, `OSIntExit()` calls `OSIntCtxSw()` which also causes its return address to be pushed onto the stack. In order for `OSIntCtxSw()` to properly perform a context switch, the stack pointer (R7) needs to be adjusted so that it points as shown in Figure 11, ④. The adjustment value of the stack pointer depends on the compiler model and the compiler options selected. The value is, however, at least 8 (for the large model) because of the two return addresses. If your application crashes you may want to make sure that you have the proper value for this constant. The rest of the context switch is exactly the same as previously discussed.

```
        signat      _YourISR,24
        global      _OSIntEnter
        global      _YourISRHandler
        global      _OSIntExit
_YourISR:
        pushu       r0,r1,r2,r3,r4,r5,r6   ; Save R0 through R6
        pushu.b     442h                   ; Save ES
        pushu.b     403h                   ; Save SSEL
        pop         r2                     ; Move PSW to user stack
        pushu       r2
        pop         r2                     ; Move PCH to user stack
        pushu       r2
        pop         r2                     ; Move PCL to user stack
        pushu       r2
;
        fcall       _OSIntEnter            ; Notify uC/OS of ISR begin
        mov.b       401h,#80h              ; Allow interrupt nesting
        fcall       _YourISRHandler        ; Call your C ISR handler
        fcall       _OSIntExit             ; Notify uC/OS of ISR end
;
        popu        r2                     ; Move PCL to system stack
        push        r2
        popu        r2                     ; Move PCH to system stack
        push        r2
        popu        r2                     ; Move PSW to system stack
        push        r2
        popu.b      403h                   ; Restore SSEL
        popu.b      442h                   ; Restore ES
        popu        r0,r1,r2,r3,r4,r5,r6   ; Restore R0 through R6
;
        reti                               ; Return from Interrupt
```

SU00778

Figure 10.

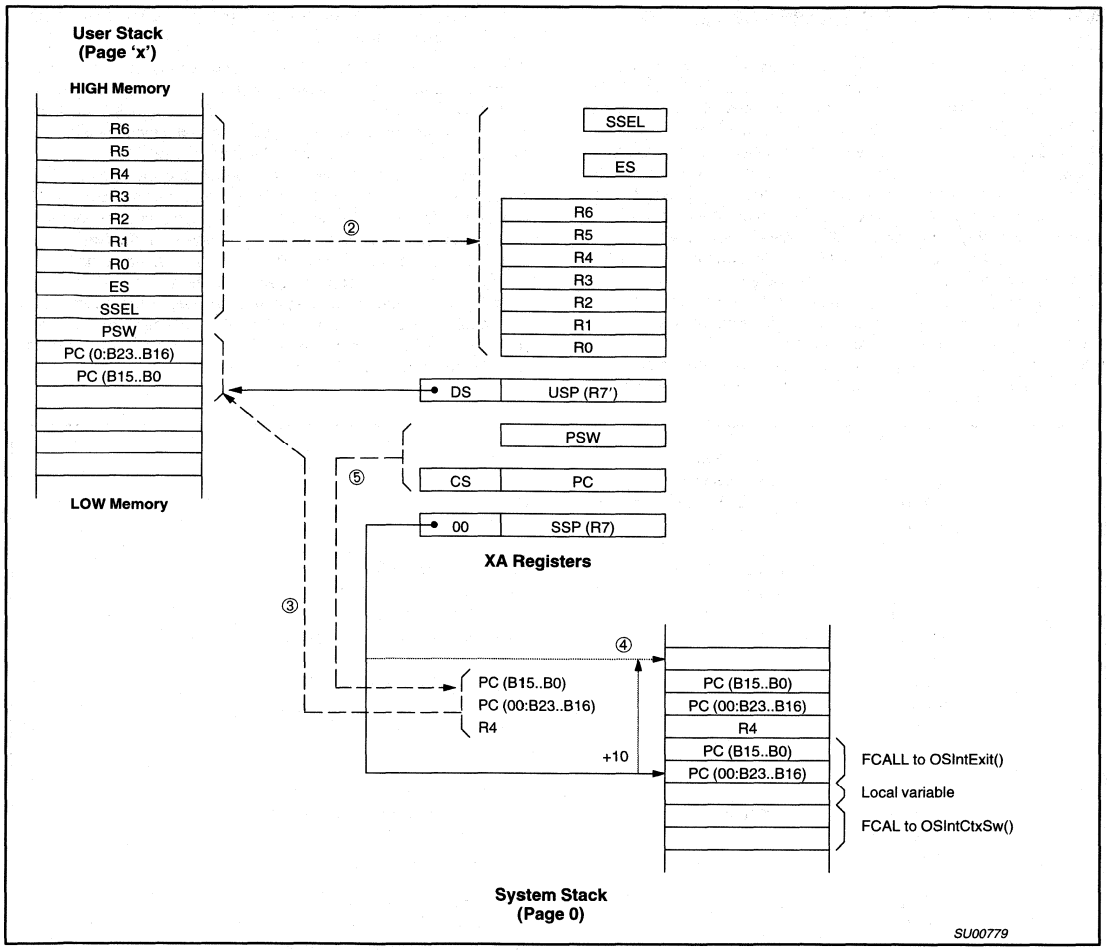

User Stack
(Page 'x')
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (0:B23..B16)
PC (B15..B0
LOW Memory
②
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
DS
USP (R7')
PSW
⑤
CS
PC
00
SSP (R7)
XA Registers
③
④
PC (B15..B0)
PC (00:B23..B16)
R4
+10
PC (B15..B0)
PC (00:B23..B16)
R4
PC (B15..B0)
PC (00:B23..B16)
FCALL to OSIntExit()
Local variable
FCAL to OSIntCtxSw()
System Stack
(Page 0)
SU00779

Figure 11.

REFERENCES

[1] *µC/OS, The Real–Time Kernel*
Jean J. Labrosse
R&D Books, 1992
ISBN 0–87930–444–8

[2] *Embedded Systems Building Blocks, Complete and Ready–to–Use Modules in C*
Jean J. Labrosse
R&D Books, 1995
ISBN 0–87930–440–5

[3] *16–bit 80C51XA Microcontrollers (eXtended Architecture)*
Philips Semiconductors, Inc.
Data Book IC25, 1996

CONTACTS

HI–TECH Software
P.O. Box 103 Alderley
QLD 4051, Australia
+61 7 3300 5011
+61 7 3300 5246 (FAX)
WEB: http://www.hitech.com.au
e–mail: hitech@hitech.com.au

Jean J. Labrosse
9540 N.W. 9th Court
Plantation, FL 33324
954–472–5094
e–mail: 72644.3724@compuserve.com

Philips Semiconductors, Inc.
811 E. Arques Avenue
Sunnyvale, CA 94088–3409
(408) 991–2000
WEB: http://www.semiconductors.philips.com
FTP: ftp://ftp.ibsystems.com/pub/philips–mcu/bbs/xa/rtos
BBS: 800–451–6644 or 408–991–2406
or +31–40–2721102 (Europe)
9600 Baud, 8 bits, no–parity, 1 stop
Filename: UCOS_XA.EXE (MS–DOS self–extracting)
Instructions: From the root directory of the drive on which you want µC/OS, type: `UCOS_XA`

R&D Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046–9950
(913) 841–1631
(913) 841–2624 (FAX)
WEB: http://www.rdbooks.com
e–mail: rdorders@rdpub.com

XA bus timings: determining optimum values for BTRH and BTRL

AN712

Author: Mark Hall

CONTENTS

1.0 INTRODUCTION

The Philips Semiconductors XA Microcontroller Family is a new architecture developed to fulfill a growing requirement for higher performance from a 'standard' microcontroller. This new architecture provides this added performance without sacrificing compatibility with current 80C51 designs. The XA-G3 is compatible with the 80C51 at the source code level, and includes every register and operating mode from the 80C51. In addition to the original 80C51 registers, there are several configuration registers that provide great control over the performance enhancements of the XA-G3. This application note will attempt to give the reader a detailed explanation of two of these registers.

The XA-G3 has two Bus Timing Registers that define the timing characteristics of the external bus interface to program memory, data memory and memory-mapped I/O or peripherals. These registers, **BTRH** and **BTRL**, have different settings for controlling the width of code reads, data reads, data writes, data hold time, and ALE. Care should be exercised to avoid the possibility of invalid combinations of settings. The purpose of this application note is to help the designer to determine the optimal settings for **BTRH** and **BTRL** and avoid wasting time with invalid settings. The focus here will be on understanding the waveforms and programmable timing options in terms of XA clock cycles and not on nanosecond level AC timing constraints. Throughout this document the term 'clocks' will be used to refer to actual CPU clock cycles at the given XA-G3 operating frequency.

It should be noted that **BTRH** and **BTRL** have *NO* impact on the access times for internal data or code accesses. *ALL* internal data and code accesses will always occur at the fastest possible timings (2 clock cycles for data memory accesses and 3 clock cycles for code accesses) regardless of the **BTRH/BTRL** settings.

The power-on default values for **BTRH** and **BTRL** are 0FFh and 0EFh (the 0EFh value occurs because of an unused bit in **BTRL** which is always set to zero). These default values result in the slowest possible bus timing values for highest initial reliability (since some devices on the external bus may be slower than others). With these default values, the XA-G3 will operate properly with most types of common memory devices. However, there are some peripheral devices (such as LCD modules) that may require longer bus cycle times than are available via the **BTRH** and **BTRL** controls. These peripherals may require the use of the XA WAIT function, which is discussed in a separate application note.

Let's take a look at the Bus Timing Registers and the actual variables. First we'll take a look at **BTRH**.

2.0 BTRH DESCRIPTION

BTRH (469h)

7	6	5	4	3	2	1	0
DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0

Bus Timing Register – High Byte

BTRH controls the timing parameters for external data write and data read cycles. Each parameter pair allows for up to 4 different settings for data writes and reads both with an ALE (parameter with an 'A'), and without an ALE (parameter without an 'A').

DW1/DW0 controls the width of external data writes without an ALE (hence no 'A' in the parameter name). A data write cycle without an ALE is only possible in a system where the XA is operating in an external 8-bit bus mode. This is true because the first or low data byte write always requires an ALE cycle. The second or high data byte write (when operating in external 8-bit mode) can occur without an ALE since only the A0 signal line must change. A data write cycle in an external 16-bit bus mode system always requires an ALE. **Therefore, the DW1/DW0 setting ONLY affects an XA-G3 operating in external 8-bit mode. Even then it only affects the second or high data byte on a word write.**

DW1	DW0	Data Write without ALE
0	0	Data Write cycle is 2 clocks
0	1	Data Write cycle is 3 clocks
1	0	Data Write cycle is 4 clocks
1	1	Data Write cycle is 5 clocks

DWA1/DWA0 controls the width of external data writes with an ALE (hence the 'A'). **This setting affects BOTH 8-bit mode AND 16-bit mode.**

DWA1	DWA0	Data Write with ALE
0	0	Data Write cycle is 2 clocks
0	1	Data Write cycle is 3 clocks
1	0	Data Write cycle is 4 clocks
1	1	Data Write cycle is 5 clocks

Since ALE typically adds one extra clock to a cycle, you will notice that the code and data read cycles with ALE are one clock cycle longer than the cycles without ALE. The exception to this rule is the Data Write cycle without ALE. Both Data Write with, and Data Write without ALE range from 2 to 5 clocks in duration. This occurs because, even without an ALE, a one clock cycle Data Write is not possible.

DR1/DR0 controls the width of external data reads without an ALE (hence no 'A' in the parameter name). The possible settings range from 1 clock to 4 clocks in duration. A data read cycle without an ALE is only possible in a system where the XA is operating in an external 8-bit bus mode. This is true because the first or low data byte read always requires an ALE cycle. The second or high data byte read (when operating in external 8-bit mode) can occur without an ALE since only the A0 signal line must change. A data read cycle in an external 16-bit bus mode system always requires an ALE. **Therefore, the DR1/DR0 setting ONLY affects an XA-G3 operating in external 8-bit mode. Even then it only affects the second or high data byte on a word read.**

DR1	DR0	Data Read without ALE
0	0	Data Read cycle is 1 clock
0	1	Data Read cycle is 2 clocks
1	0	Data Read cycle is 3 clocks
1	1	Data Read cycle is 4 clocks

DRA1/DRA0 controls the width of external data reads with an ALE (hence the 'A'). The possible settings range from 2 clocks to 5 clocks in duration. **This setting affects BOTH 8-bit mode AND 16-bit mode.**

DRA1	DRA0	Data Read with ALE
0	0	Data Read cycle is 2 clocks
0	1	Data Read cycle is 3 clocks
1	0	Data Read cycle is 4 clocks
1	1	Data Read cycle is 5 clocks

3.0 BTRL DESCRIPTION

BTRL (468h)

7	6	5	4	3	2	1	0
WM1	WM0	ALEW	Reserved	CR1	CR0	CRA1	CRA0

Bus Timing Register – Low Byte

BRTL controls the width and hold time of the write pulse, the ALE pulse width, and the width of external code reads. For external code reads, there are 2 parameter pairs, code reads without ALE (**CR1/CR0**), and code reads with ALE (**CRA1/CRA0**).

WM1 controls the width of the write pulse. The possible settings are **WM1**=0, Write pulse is 1 clock; and **WM1**=1, Write pulse is 2 clocks.

WM1	Write Pulse Width
0	Write pulse width is 1 clock
1	Write pulse width is 2 clocks

WM0 controls the data hold time for write cycles. The possible settings are **WM0**=0, data hold time is minimum or 0 clocks, and **WM0**=1, data hold time is 1 clock. **Since a WM0 value of 0 for the XA-G3 can result in a negative data hold time for external bus write cycles, the WM0 value should always be set to 1.**

WM0	Write Hold Time
0	Write data hold time is 0 clocks **(don't use !)**
1	Write data hold time is 1 clocks

ALEW controls the width of the ALE pulse. The possible settings are; **ALEW**=0, ALE pulse width = 0.5 clocks, and **ALEW**=1, ALE pulse width = 1.5 clocks.

At first glance the default **ALEW** value of 1 would appear to allow for slower external devices, since the ALE pulse width is longer (1.5 clocks vs. 0.5 clocks). However, since the shorter ALE pulse width allows more time for the rest of the cycle to complete, an **ALEW** value of 0 may allow you to use slower, less expensive memories and peripherals. The tradeoff here may be that at higher XA clock frequencies better 373 type latches (such as FAST or ABT) may be required to satisfy the address set-up, hold time and pulse width requirements with the shorter ALE pulse. Overall the price and power difference between an ABT373 and an HCT373 may be much less than the price and power difference between fast and slow speed SRAM's, EPROM's or FLASH devices.

ALEW	ALE Pulse Width
0	ALE pulse width is 0.5 clocks
1	ALE pulse width is 1.5 clocks

"Reserved" is for possible future use. Programs should take care when writing to registers with reserved bits that those bits are always written with a value of '0'.

CR1/CR0 controls the width of the external code read pulse without an ALE (hence no 'A' in the parameter name). The possible settings range from 1 clock to 4 clocks.

Code read pulses without an ALE can occur when the XA-G3 pre-fetch queue does a burst mode code fetch. This burst mode is much faster since it allows up to 16 bytes of external code data to be fetched without the generation of an ALE. This is possible since the lower 4 address lines on the XA-G3 (A0–A3) are always driven and can change without the need for an ALE pulse. The pre-fetch queue can execute a burst mode code fetch anytime the XA needs multiple instruction bytes to fill it's processing queue. **This setting affects BOTH 8-bit mode AND 16-bit mode.**

CR1	CR0	Code Read without ALE
0	0	Code Read cycle is 1 clock
0	1	Code Read cycle is 2 clocks
1	0	Code Read cycle is 3 clocks
1	1	Code Read cycle is 4 clocks

CRA1/CRA0 controls the width of the external code read pulse with an ALE (hence the 'A' in the parameter name. The possible settings range from 2 clocks to 5 clocks. **This setting affects BOTH 8-bit mode AND 16-bit mode.**

CRA1	CRA0	Code Read with ALE
0	0	Code Read cycle is 2 clocks
0	1	Code Read cycle is 3 clocks
1	0	Code Read cycle is 4 clocks
1	1	Code Read cycle is 5 clocks

4.0 EXAMPLE TIMING DIAGRAMS

Now we'll take a look at some examples of the actual timing cycles for each type of external bus access. The following waveforms are actual logic analyzer outputs taken from an operating XA-G3 design. The hardware platform used for these tests is described in the last section of this document.

The timing diagrams will include the following XA-G3 signals:

Signal	Description
CLK	Clock input to the XA-G3, XTAL1
ALE	Address Latch Enable
PSEN–	Program Store Enable
RD–	External Data Memory Read Strobe
WRL–/WRH–	External Data Memory Write Strobe (Low byte/High byte)
Mux Addr	Multiplexed Addresses (A4 to A19)
Umux Addr	Unmultiplexed Addresses (A1 to A3)

(Please consult the XA databook for a detailed explanation of these signals)

The actual XA clock frequency used has been varied where necessary to enhance the readability of the timing diagrams. In some cases the resolution of the Logic Analyzer causes the XA clock to appear non-symmetric, but the XA clock is always a symmetric squarewave.

In the following timing diagrams, the waveforms will be displayed as follows:

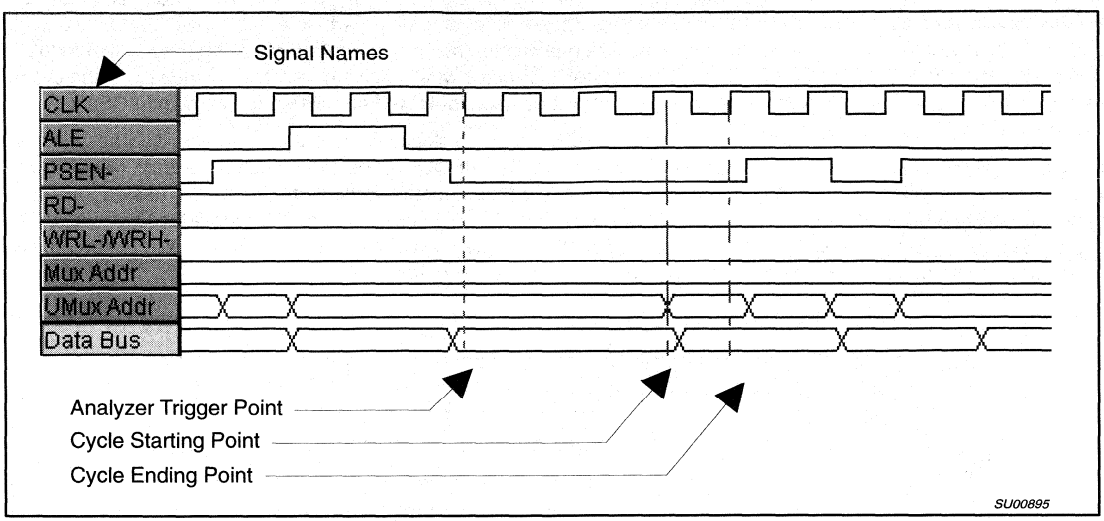


Figure 1. Waveform Display

Signal Names: Names of the signals utilized by the Logic Analyzer. These names correspond to the names utilized in the XA-G3 databook. Some of the signal descriptions had to be brief for display purposes, these are further described here:

> **Mux Addr:** Addresses that are multiplexed address/data signals. Here, they are utilized only for the address portion of the waveforms and hence, are taken from the OUTPUT side of the address latches.
>
> **U Mux Addr:** Addresses that are NOT multiplexed, and are available directly from the XA-G3.

Analyzer Trigger Point: This is the point at which the logic analyzer triggered and centered the waveform about. This trigger point was developed by the author to effectively show the waveform required to display the results needed for the text.

Cycle Starting Point: This is the beginning of the cycle to be discussed by the text of the section.

Cycle Ending Point: This is the end of the cycle to be discussed by the text of the section.

4.1 External Code Memory Read Cycle (with ALE)

Below are examples of external code read cycles showing the timing relationships as programmed in **BTRH** and **BTRL**. Please note that in some cases there may be other register changes made to avoid invalid combinations of settings which would result in incorrect operation. As an example, the setting for **ALEW** has been changed to '0', which results in an ALE width of 0.5 clocks. This shorter ALE pulse width allows more time for the rest of the cycle to complete. This **ALEW** bit change also changes the values used for **BTRH** and **BTRL** to 0FFh and 0CFh.

The slower setting (**ALEW**='1'), which results in an ALE width of 1.5 clocks, is available for slower address latches if required. The address latches utilized in this example DO NOT REQUIRE the slower setting. Please verify your specific requirements by consulting the data sheet of the 373 type address latches utilized in your design. In most cases the normal operating mode will be **ALEW** = 0 if ABT, FAST, FCT or equivalent 373 type latches are used. As a general rule, if LS or HCT speed latches are used at clock frequencies above 16 MHz, an **ALEW** = 1 setting will be required to meet set-up, hold time and pulse width requirements related to ALE.

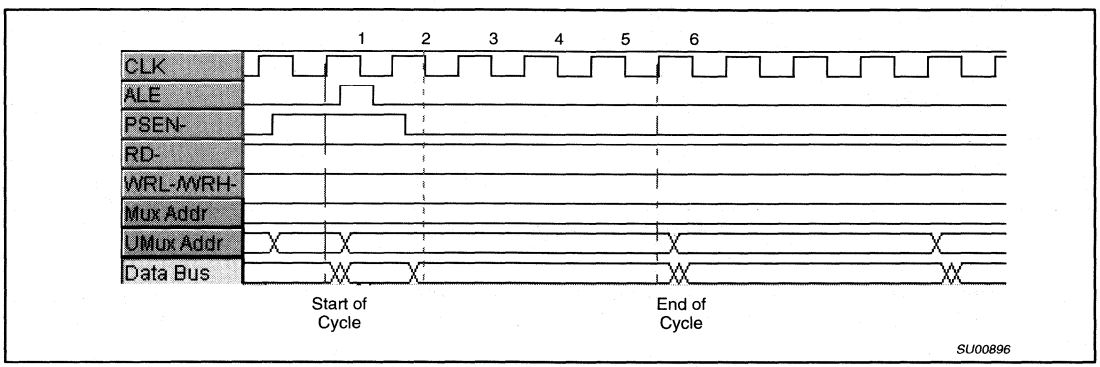


Figure 2. Code Read with ALE — 5 clocks
CRA1/CRA0 = 11, BTRH/BTRL = 0FFCFh

As you can see, this is the slowest external operating mode possible for the XA. The initial code read (from the clock edge before ALE) is approximately 5 clocks and each subsequent read (non-ALE) occurs every 4 clocks.

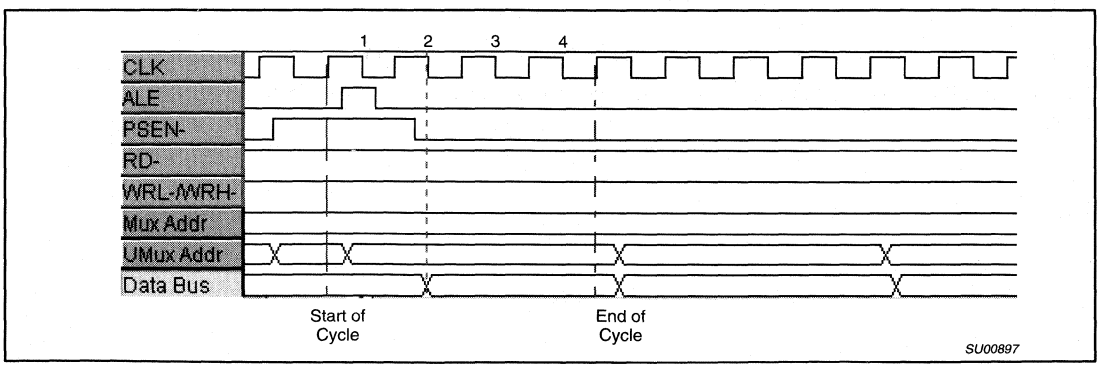


Figure 3. Code Read with ALE — 4 clocks
CRA1/CRA0 = 10, BTRH/BTRL = 0FFCEh

Now the code read (from the clock edge before ALE) is only 4 clocks.

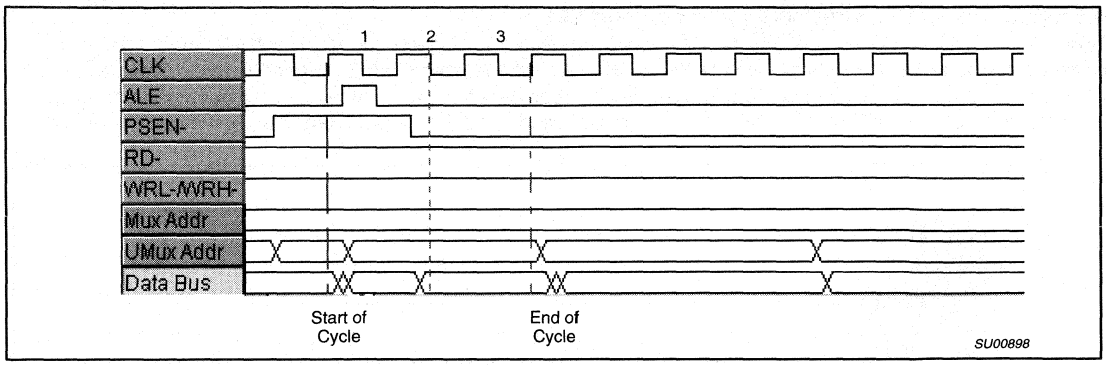


Figure 4. Code Read with ALE — 3 clocks
CRA1/CRA0 = 01, BTRH/BTRL = 0FFCDh

Now the code read (from the clock edge before ALE) is only 3 clocks.

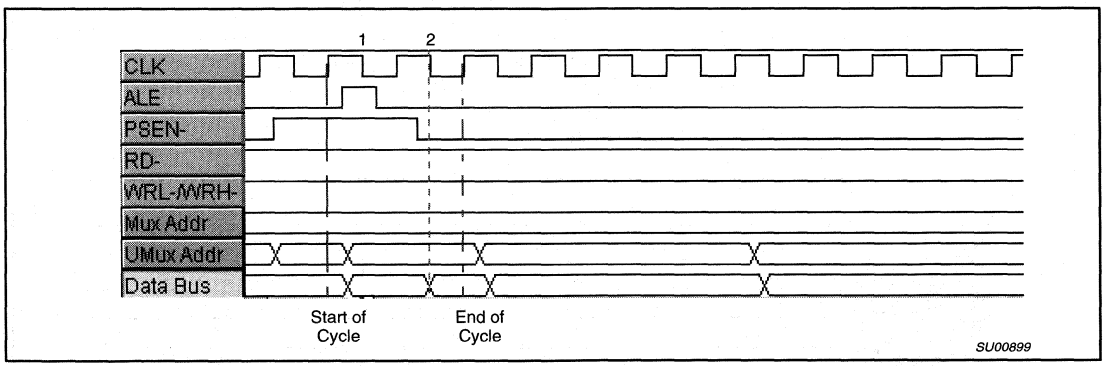


Figure 5. Code Read with ALE — 2 clocks
CRA1/CRA0 = 00, BTRH/BTRL = 0FFCCh

This is the shortest code read cycle at only 2 clocks. As you can see, this setting may not be practical in designs operating at higher frequencies and/or with slower memory devices. At a clock frequency of 30 MHz (33 ns clock cycle) the code device access time (from PSEN– true) required for this setting is around 4ns. Please verify the actual access times from the XA data book.

4.2 External Code Memory Read Cycle (no ALE)

Now let's take a look at the same timing relationships for Non-ALE (burst) code read cycles. Code read cycles without an ALE can ONLY occur when the XA-G3 pre-fetch queue does a burst mode code fetch. This burst mode is much faster since it allows up to 16 bytes of external code data to be fetched without the generation of an ALE. Notice that PSEN– is already true and is thus not re-asserted since this cycle is part of a burst mode sequence.

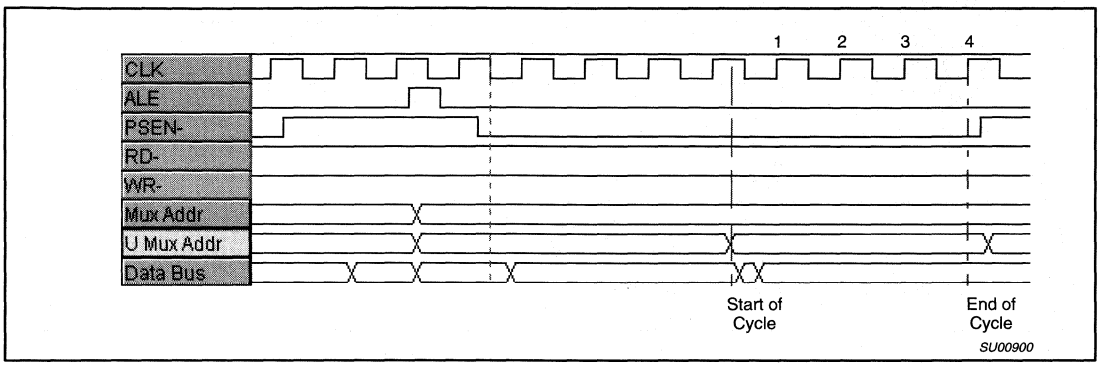


Figure 6. Code Read (no ALE) — 4 clocks
CR1/CR0 = 11, BTRH/BTRL = 0FFCFh

At 4 clock cycles, this is the slowest non-ALE code read cycle.

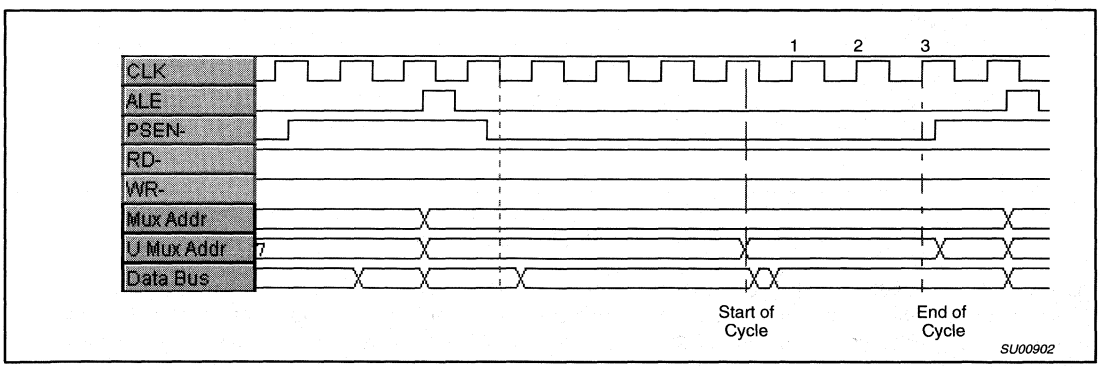


Figure 7. Code Read (no ALE) — 3 clocks
CR1/CR0 = 10, BTRH/BTRL = 0FFCBh

This is the 3 clock non-ALE code read cycle.

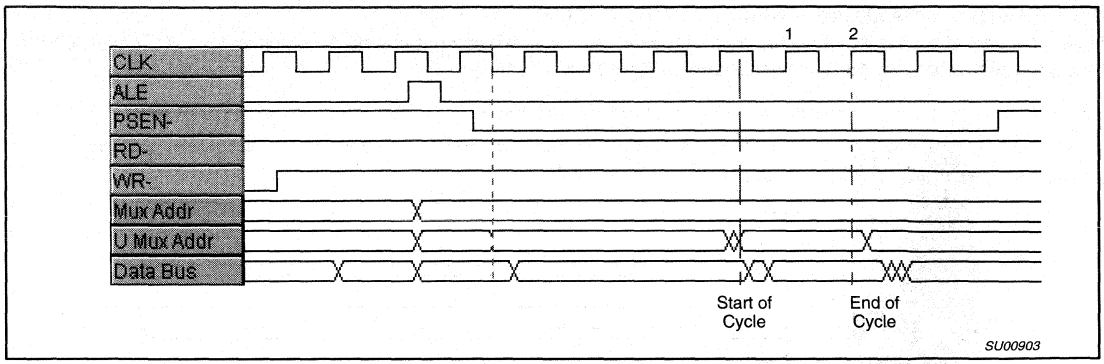


Figure 8. Code Read (no ALE) — 2 clocks
CR1/CR0 = 01, BTRH/BTRL = 0FFC7h

This is the 2 clock non-ALE code read cycle.

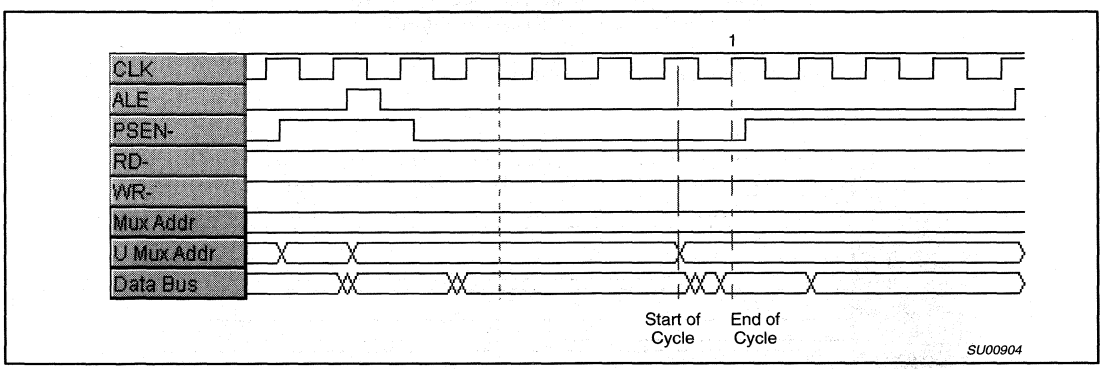


Figure 9. Code Read (no ALE) — 1 clock
CR1/CR0 = 00, BTRH/BTRL = 0FFC3h

This is the fastest non-ALE code read cycle at only 1 clock. This setting may not be practical in designs operating at higher frequencies and/or with slower memory devices. At a clock frequency of 30 MHz (which results in a 33 ns clock cycle) the code device access time required for this setting is around 4ns. Please verify the actual access times from the XA data book.

4.3 External Data Memory Read Cycle (with ALE)

Now let's look at the typical external data memory read cycle which contains an ALE pulse. This is the standard for external data reads in a 16-bit external bus configuration.

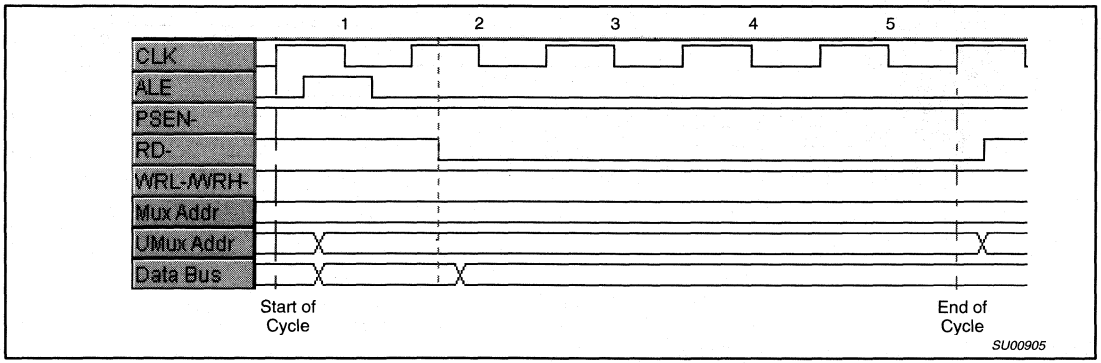


Figure 10. Data Read with ALE — 5 clocks
DRA1/DRA0 = 11, BTRH/BTRL = 0FFCFh

This is the longest external data memory read cycle at 5 clocks.

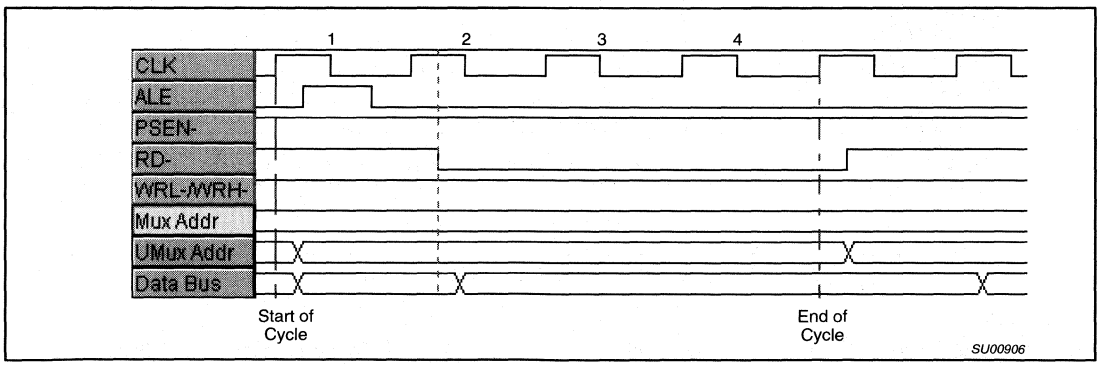


Figure 11. Data Read with ALE — 4 clocks
DRA1/DRA0 = 10, BTRH/BTRL = 0FECFh

This is a 4 clock external data memory read cycle with ALE.

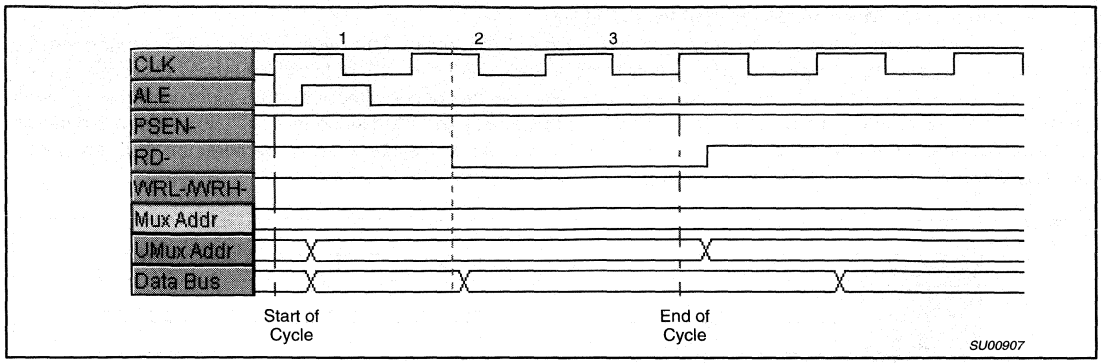


Figure 12. Data Read with ALE — 3 clocks
DRA1/DRA0 = 01, BTRH/BTRL = 0FDCFh

This is a 3 clock external data memory read cycle with ALE.

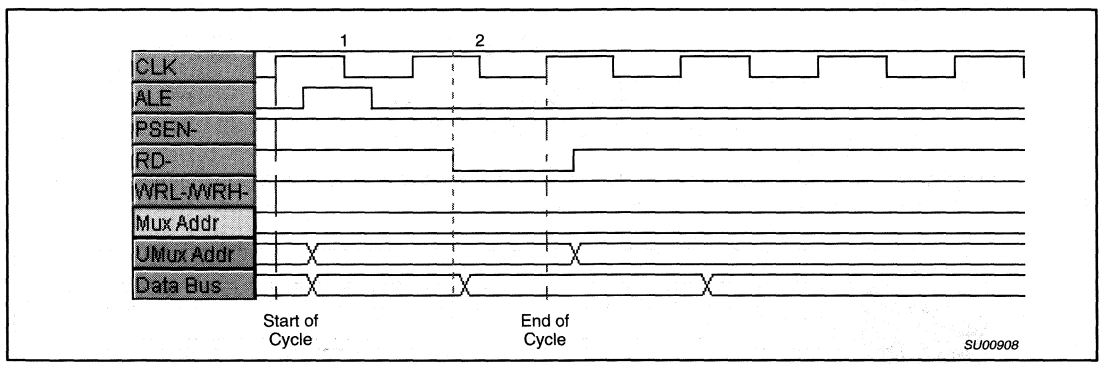


Figure 13. Data Read with ALE — 2 clocks
DRA1/DRA0 = 00, BTRH/BTRL = 0FCCFh

This is the fastest external data memory read cycle with ALE, at only 2 clocks. This setting may not be practical in designs operating at higher frequencies and/or with slower memory devices. At a clock frequency of 30 MHz (33 ns clock cycle) the data device access time required for this setting is around 11ns. Please verify the actual access times from the XA data book.

4.4 External Data Memory Read Cycle (no ALE)

Now let's look at the external data memory read cycle with NO ALE pulse. This cycle is only possible when the XA is operating in an external 8-bit bus mode. This is true because the first or low data byte read always requires an ALE cycle. In an 8-bit system, the second or high data byte read can occur without an ALE since only the A0 signal line must change. Notice that RD– is already true and is thus not re-asserted since this cycle is part of a 2-byte (or word read) sequence. A data read cycle in an external 16-bit bus mode system always requires an ALE, so these waveforms DO NOT APPLY to 16-bit XA systems

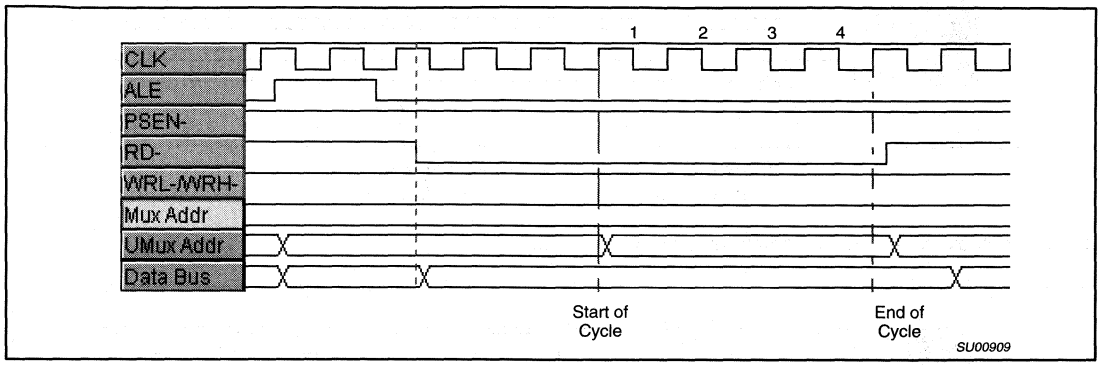


Figure 14. Data Read (no ALE) — 4 clocks
DR1/DR0 = 11, BTRH/BTRL = 0FFEFh

This is the longest external data memory read cycle with no ALE, at 4 clocks. The first byte in the read cycle (with ALE) is also shown for clarity of the full read cycle sequence.

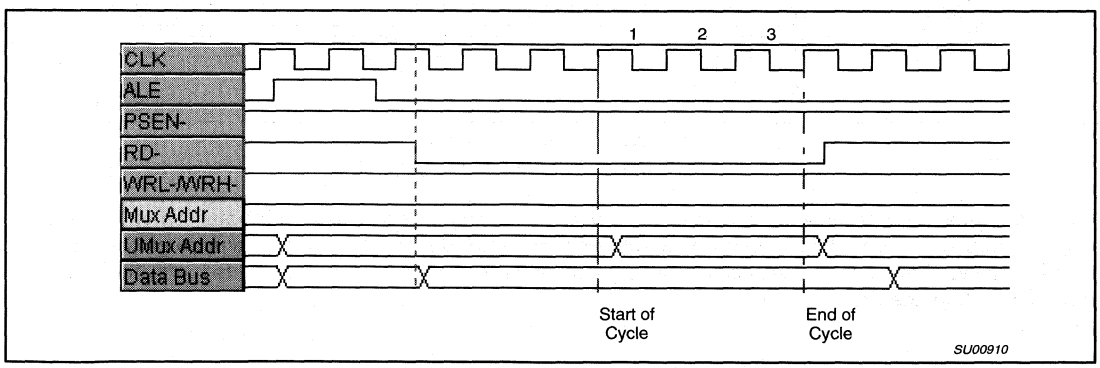


Figure 15. Data Read (no ALE) — 3 clocks
DR1/DR0 = 10, BTRH/BTRL = 0FBEFh

This is the 3 clock external data memory read cycle with no ALE.

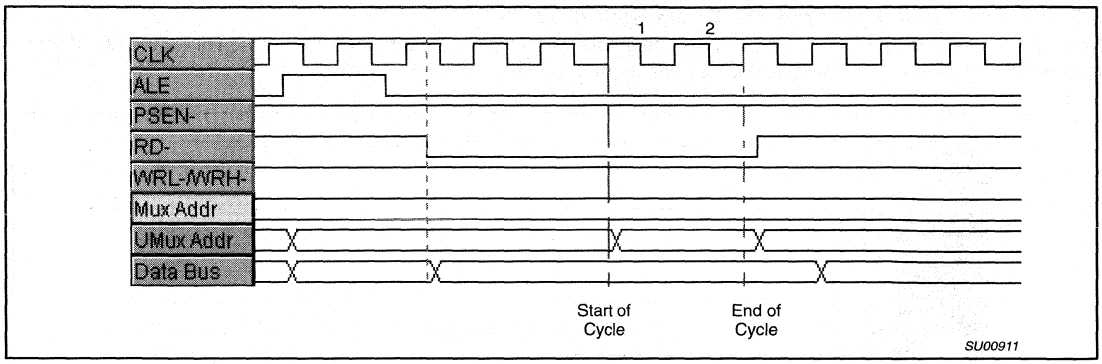


Figure 16. Data Read (no ALE) — 2 clocks
DR1/DR0 = 01, BTRH/BTRL = 0F7EFh

This is the 2 clock external data memory read cycle with no ALE.

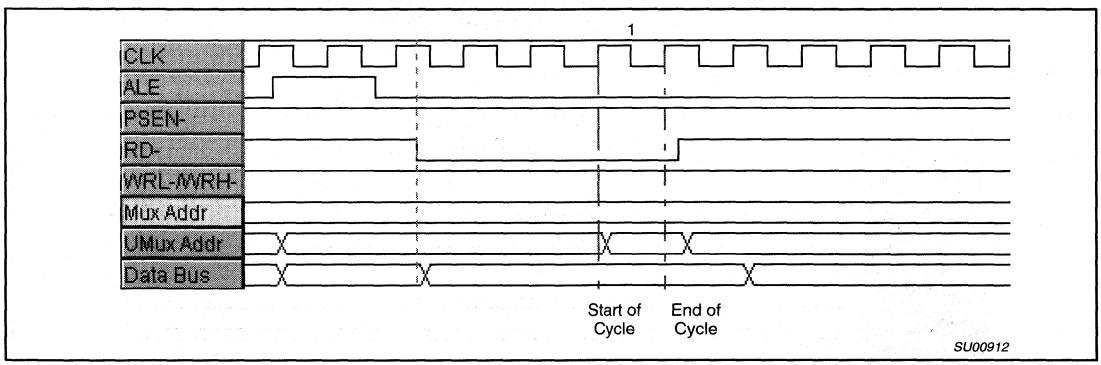


Figure 17. Data Read (no ALE) — 1 clock
DR1/DR0 = 00, BTRH/BTRL = 0F3EFh

This is the fastest external data memory read cycle with no ALE, at only 1 clock. This setting may not be practical in designs operating at higher frequencies and/or with slower memory devices. At a clock frequency of 30 MHz (33 ns clock cycle) the data device access time required for this setting is around 4ns. Please verify the actual access times from the XA data book.

4.5 External Data Memory Write Cycle (with ALE)

Now let's look at the typical external data memory write cycle which contains an ALE pulse. This is the standard for external data writes in a 16-bit external bus configuration.

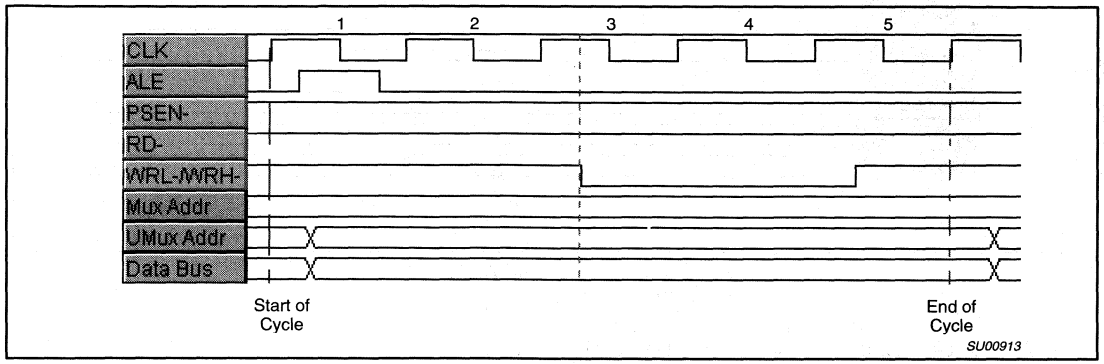


Figure 18. Data Write with ALE — 5 clocks
DWA1/DWA0 = 11, BTRH/BTRL = 0FFCFh

This is the longest external data memory write cycle at 5 clocks.

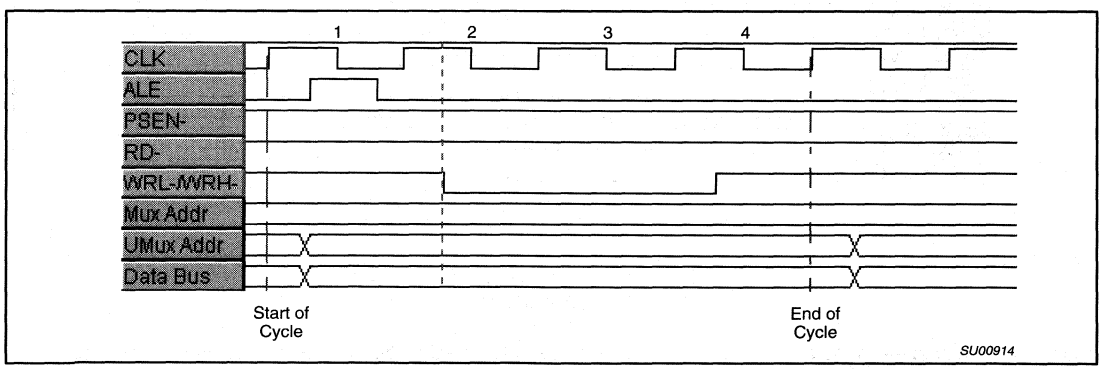


Figure 19. Data Write with ALE — 4 clocks
DWA1/DWA0 = 10, BTRH/BTRL = 0EFEFh

This is a 4 clock external data memory write cycle with ALE.

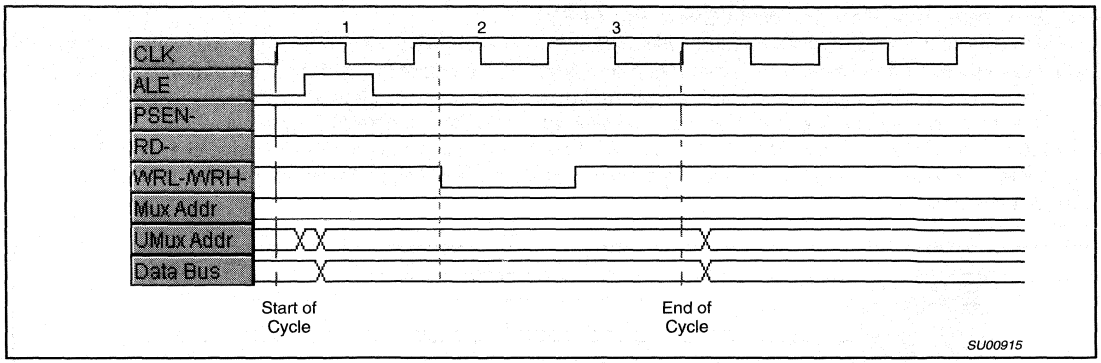


Figure 20. Data Write with ALE — 3 clocks
DWA1/DWA0 = 01, BTRH/BTRL = 0DFEFh

This is a 3 clock external data memory write cycle with ALE.

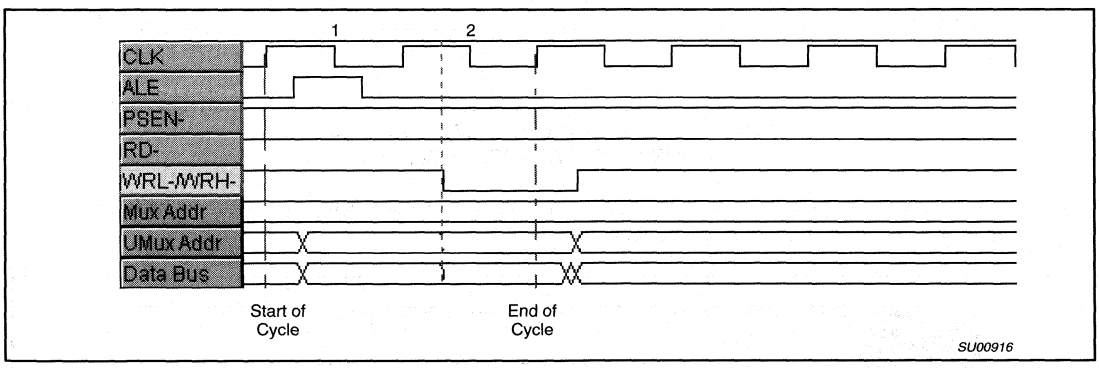


Figure 21. Data Write with ALE — 2 clocks
DWA1/DWA0 = 00, BTRH/BTRL = 0CF0Fh

This is the fastest external data memory write cycle with ALE, at only 2 clocks. This setting may not be practical in designs operating at higher frequencies and/or with slower memory devices.

4.6 External Data Memory Write Cycle (no ALE)

Now let's look at the external data memory write cycle with NO ALE pulse. This cycle is only possible when the XA is operating in an external 8-bit bus mode. This is true because the first or low data byte write always requires an ALE cycle. In an 8-bit system, the second or high data byte write can occur without an ALE since only the A0 signal line must change. Notice that WRL– is first asserted during the low byte ALE write cycle, and then is asserted again for the high byte non-ALE write cycle once the un-multiplexed addresses have changed. A data write cycle in an external 16-bit bus mode system always requires an ALE, so these waveforms DO NOT APPLY to 16-bit XA systems.

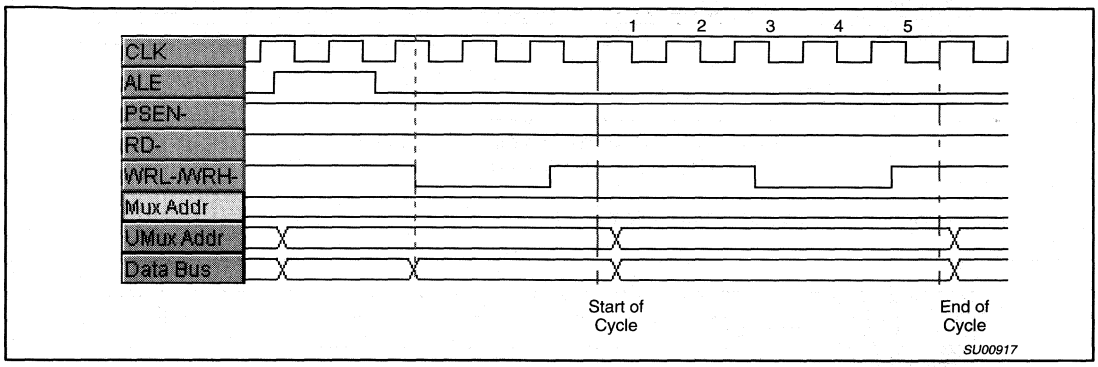


Figure 22. Data Write (no ALE) — 5 clocks
DW1/DW0 = 11, BTRH/BTRL = 0FFEFh

This is the longest external data memory write cycle with no ALE, at 4 clocks. The first byte in the write cycle is also shown for clarity of the full write cycle sequence.

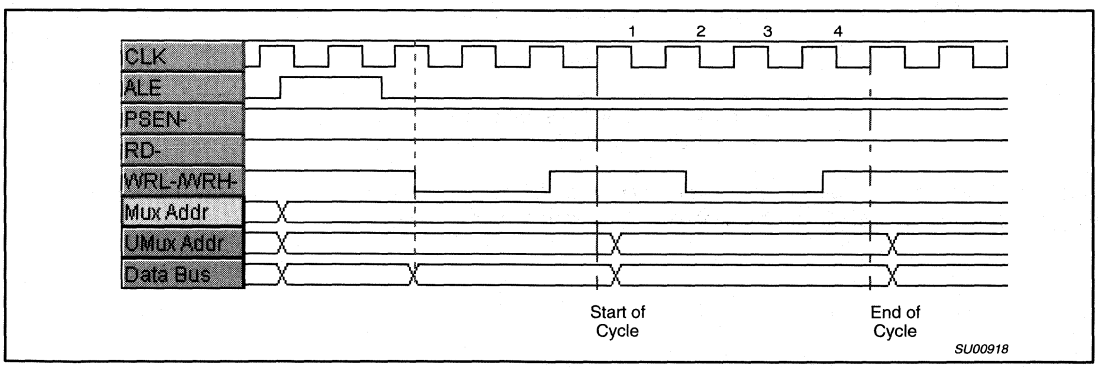


Figure 23. Data Write (no ALE) — 4 clocks
DW1/DW0 = 10, BTRH/BTRL = 0BFEFh

This is the 4 clock external data memory write cycle with no ALE.

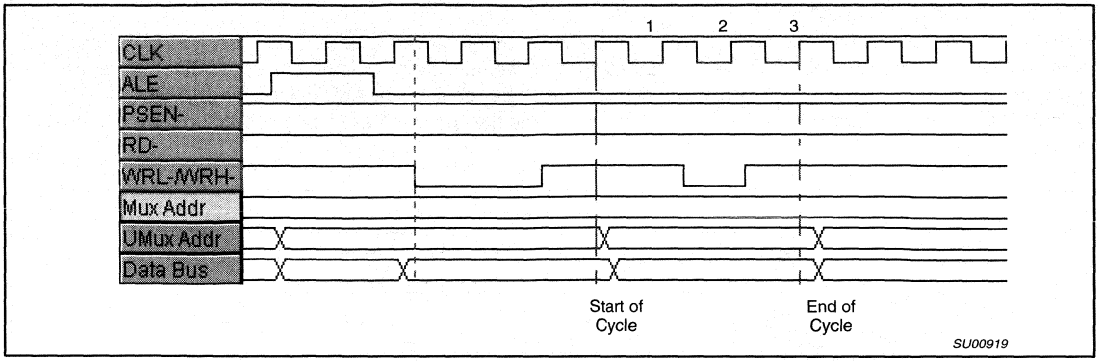


Figure 24. Data Write (no ALE) — 3 clocks
DW1/DW0 = 01, BTRH/BTRL = 07FEFh

This is the 3 clock external data memory write cycle with no ALE.

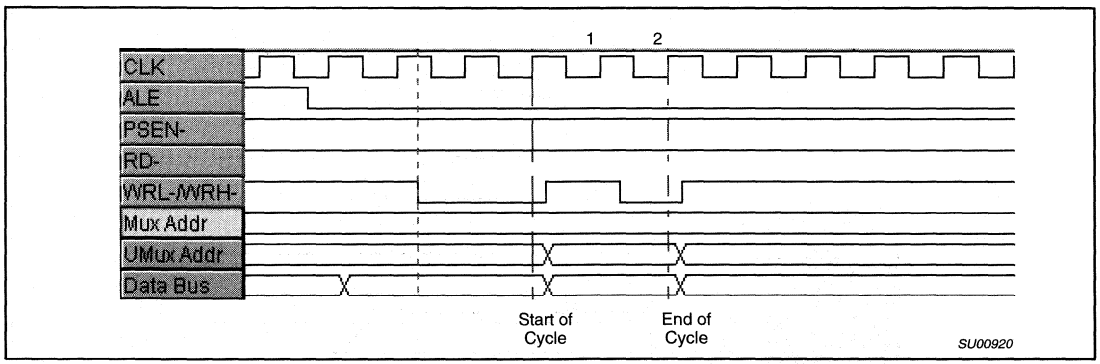


Figure 25. Data Write (no ALE) — 2 clocks
DW1/DW0 = 00, BTRH/BTRL = 03FAFh

This is the 2 clock external data memory write cycle with no ALE. It should be noted that in order for this cycle to be valid, both BTRH *AND* BTRL were changed. The setting for WM0, Write-mode Data Hold Time, was changed from 1 clock to 0 clocks of hold time. By looking at the waveform, it is evident that there is no data hold time provided. If WM0 is left set to 1, there will be NO 2nd write cycle. This setting is NOT a realistic setting and great care should be taken if these values are utilized.

4.7 ALEW Value Examples

ALEW has a significant affect on the first cycle of external memory transfers. As you can see, if ALEW is set to create a longer ALE pulse, then the time is reduced from the actual strobe cycle (in this case, RD–). Note, as shown in Figure 27, the RD– strobe is 1 FULL clock longer, as a result of the ALE pulse being 1 FULL clock shorter. This will have significant impact on designs where the timing values are being optimized for the fastest possible execution.

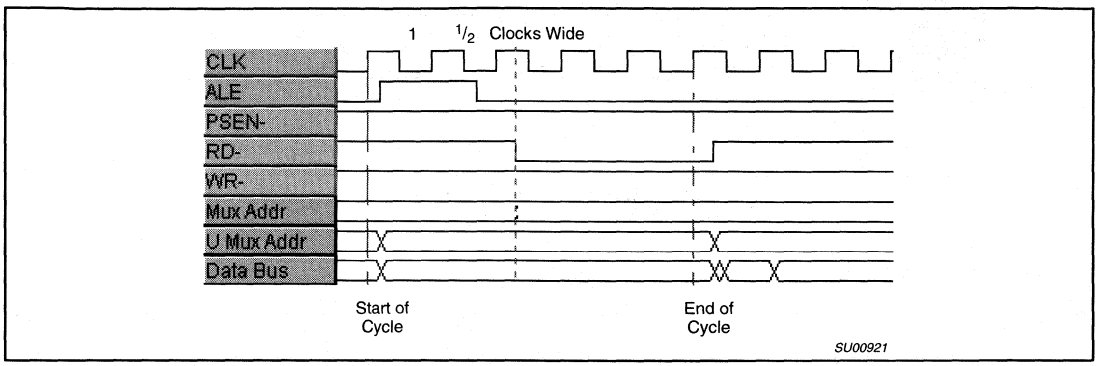


Figure 26. ALEW — 1.5 clocks
ALEW = 1, BTRH/BTRL = 03FEFh

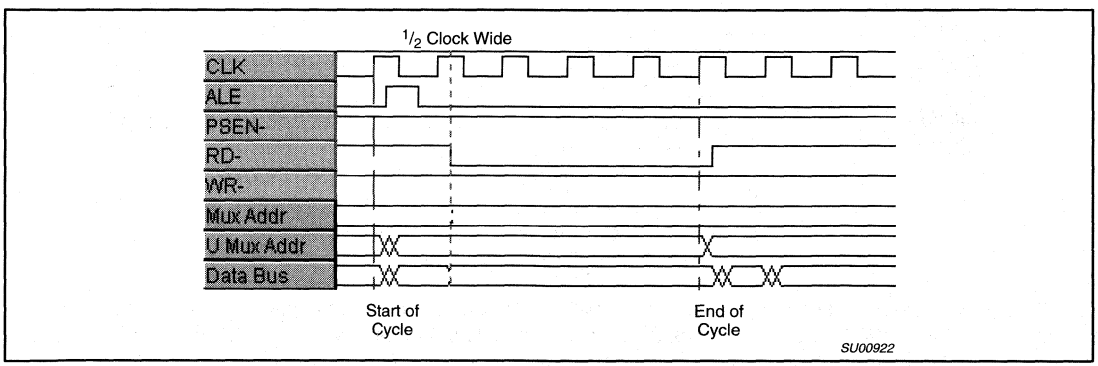


Figure 27. ALEW — 0.5 clocks
ALEW = 0, BTRH/BTRL = 03FCFh

5.0 INVALID REGISTER SETTINGS

There are several settings of **BTRH/BTRL** values that result in invalid bus timings. These settings are typically modes where the programmed cycle time (such as **DWA1/DWA0**) is less than the sum of the required signal lengths (such as **ALEW**, **WM1**, & **WM0**). The XA has no protection against these invalid settings, so it is up to the designer to verify the desired settings before implementation.

The following equations may be utilized as a quick reference to determine the viability of the desired **BTRH/BTRL** settings. All units are in clock cycles.

5.1 Invalid Write Cycles

For data memory write cycles, both non-ALE and ALE writes have constraints, as shown below:

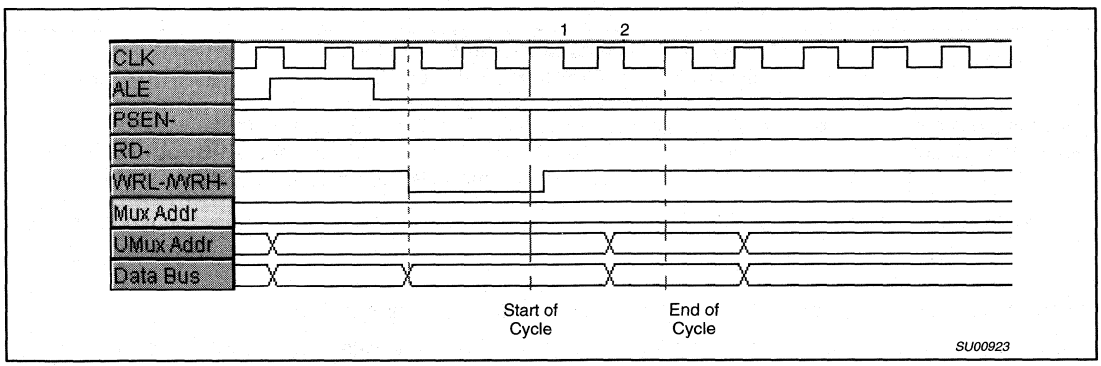


Figure 28. Data Write (no ALE) — 2 clocks
DW1/DW0 = 00, BTRH/BTRL = 03FEFh

As shown in the waveform above, there was ***NO*** second or non-ALE write cycle because of an invalid register setting. The **BTRH/BTRL** setting shown results in a 2 clock non-ALE write cycle with a 2 clock write pulse width and a 1 clock write data hold time (**WM1**=1, **WM0**=1 in **BTRL**). This setting does not satisfy the required equation:

Width of Write Pulse	+	Write Data Hold Time	≤	Width of Non-ALE Write Cycle
2	+	1	≤	2 **(NOT TRUE!)**

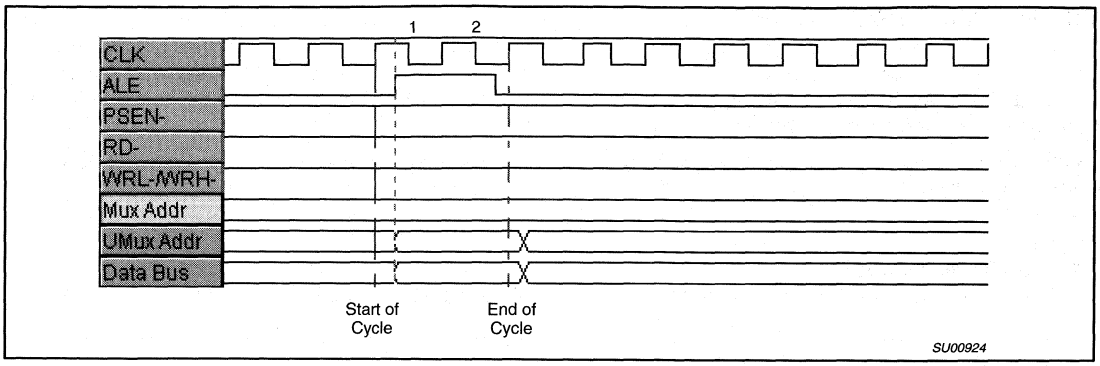


Figure 29. Data Write with ALE — 2 clocks
DWA1/DWA0 = 00, BTRH/BTRL = 0CFEFh

This example is a most severe case of improper register settings. The **BTRH/BTRL** setting shown results in a 2 clock write cycle with ALE, a 2 clock write pulse width, a 1 clock write data hold time, and a 1.5 clock ALE pulse width (**WM1**=1, **WM0**=1, **ALEW**=1 in **BTRL**). This setting does not satisfy the required equation:

Width of ALE Pulse	+	Width of Write Pulse	+	Write Data Hold Time	≤	Width of ALE Write Cycle
1.5	+	2	+	1	≤	2 **(NOT TRUE!)**

As you can see in the figure above, there is no room for the actual write pulse to appear.

Figure 21 shows an example of the correct settings for a 2 clock Data Write with ALE. The value used for **BTRH/BTRL** in this example is 0CF0Fh.

5.2 Invalid Read Cycles

For data or code memory reads, only cycles with ALE have a constraint.

5.2.1 Memory Read with ALE

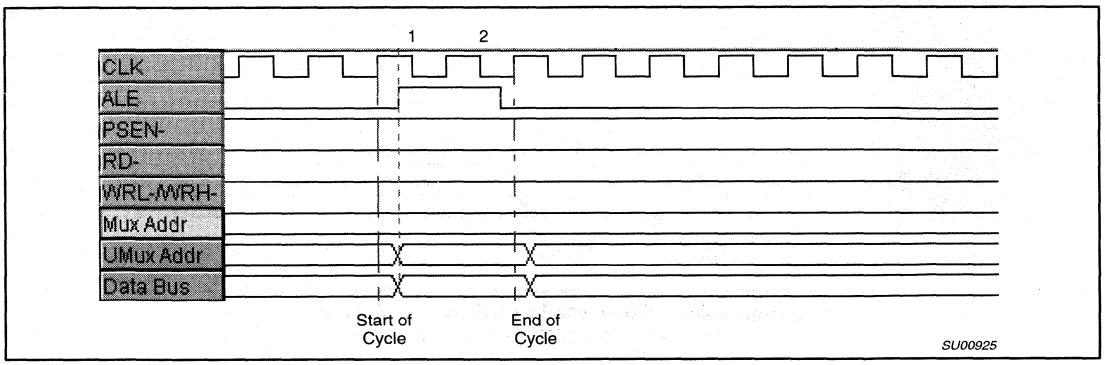


Figure 30. Data Read with ALE — 2 clocks
DRA1/DRA0 = 00, BTRH/BTRL = 0FCEFh

The **BTRH/BTRL** setting shown results in a 2 clock Data Read cycle with ALE, and a 1.5 clock ALE pulse width (**ALEW**=1 in **BTRL**). This setting does not satisfy the required equation:

Width of ALE < Width of ALE Data Read Cycle

Because the setting for **ALEW** is either 0.5 clock or 1.5 clocks, and you must round up to a whole clock cycle value. The following equation can be utilized to verify the values.

ALEW + 1 < DRA + 2
1 + 1 < 0 + 2 **(NOT TRUE!)**

As you can in the figure above, no read pulse will be generated since this setting is invalid.

5.2.2 Code Read with ALE

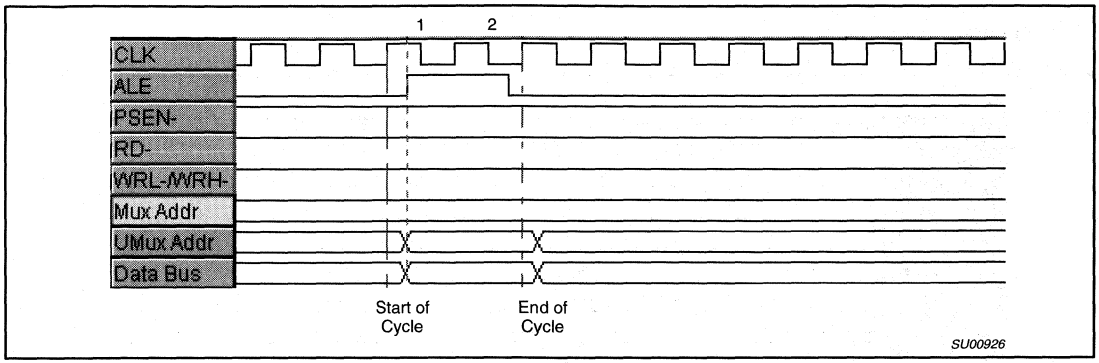


Figure 31. Code Read with ALE — 2 clocks
CRA1/CRA0 = 00, BTRH/BTRL = 0FFECh

The **BTRH/BTRL** setting shown results in a 2 clock Code Read cycle with ALE, and a 1.5 clock ALE pulse width (**ALEW**=1 in **BTRL**). This setting does not satisfy the required equation :

Width of ALE < Width of ALE Code Read Cycle

Because the setting for **ALEW** is either 0.5 clock or 1.5 clocks, and you must round up to a whole clock cycle value. The following equation can be utilized to verify the values.

ALEW + 1 < CRA + 2
1 + 1 < 0 + 2 **(NOT TRUE!)**

As you can in the figure above, no read pulse will be generated since this setting is invalid.

Notice that if the ALE pulse width was set to 0.5 Clocks (**ALEW** = 0) the **BTRH/BTRL** setting would change to 0FFCCh and this would now be a valid configuration. Please see Figure 5 for an example of the waveform associated with this setting.

6.0 EXAMPLE HARDWARE

The example hardware utilized for the development of this application note is the XTEND-G3 Development Board. This board utilizes an XA-G3 operating with external memory devices in 16-bit mode. There is 128KB of FLASH ROM containing the XMON monitor program and 64KB of High-speed SRAM for data. The XTEND-G3 provides a 60-pin expansion header with easy access to every signal used in the example waveforms.

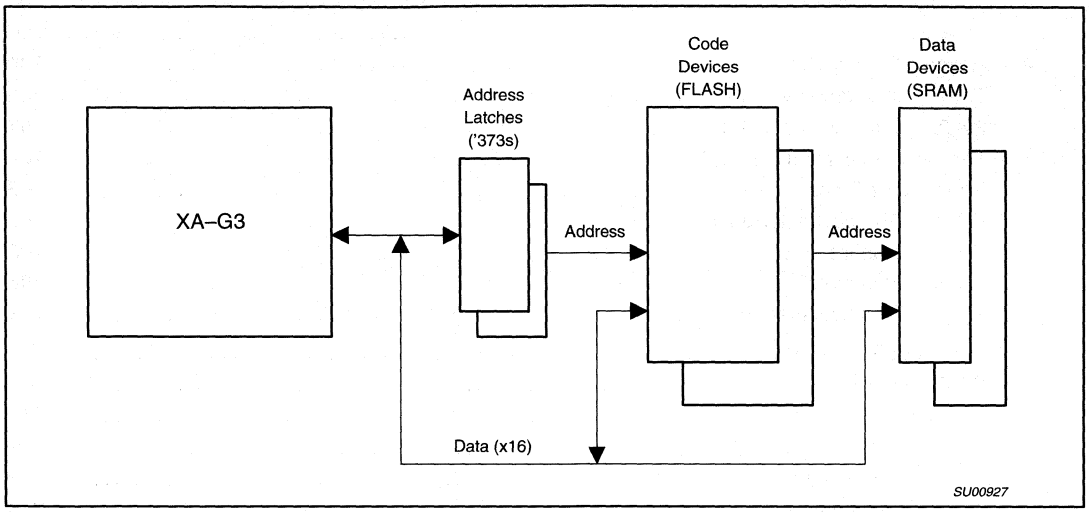


Figure 32. XTEND-G3 Development Board Block Diagram

The complete XTEND-G3 Development Kit is available from any authorized Philips distributor as P/N P51XTEND-SD (12NC: 9352-336-70112) or directly from:

Future Designs, Inc. at (205) 830-4116.

Additional information is available on the Internet at

http://members.aol.com/teamfdi/teamfdi.htm

or via e-mail at

teamfdi@aol.com

XA interrupts AN713

Author: Kent Lowman

CONTENTS

1. INTRODUCTION

This document will discuss the XA Interrupt Structure from two different perspectives. First we will look at the XA Family Interrupt Structure since it is important to have an understanding of all the interrupt options available in the XA Family. This general discussion will introduce us to all available interrupt options in the XA Family. We will cover the details of Exception Interrupts, Trap Interrupts and Software Interrupts since these will generally be standard across the XA Family. However, specific implementations for Event Interrupts will not be covered, since each XA derivative may have a unique subset of Event Interrupts available.

Next we will look in detail at the XA-G3 Interrupt Structure since this is the first available member of the XA derivative family. This discussion will cover the function and detail of all interrupts included on the XA-G3. We will assume an understanding of the general structure and function of XA interrupts as given in the section on XA Family Interrupts. Event Interrupts that are unique to the XA-G3 will be covered in detail.

2. XA FAMILY INTERRUPT STRUCTURE

This section covers all the interrupt options available in the XA Family. It should be used as background material to gain familiarity with the way XA interrupts function. Please refer to the sections on specific XA derivatives for details of their individual interrupt structures.

The XA Family offers a very powerful Interrupt Structure with various levels of user programmable configuration and control. This allows for great flexibility in various applications but does require the user to provide adequate initialization and set-up for interrupts to function as expected. This required initialization is more detailed than that needed for microcontrollers such as the 8051 which have a much simpler interrupt structure.

2.1. XA Family Interrupts

The XA architecture defines four kinds of interrupts. These are listed below in order of intrinsic priority:

- Exception Interrupts
- Trap Interrupts
- Event Interrupts
- Software Interrupts

Exception interrupts reflect system events of overriding importance. Examples are stack overflow, divide-by-zero, and Non-Maskable Interrupt. Exceptions are non maskable and are always processed immediately as they occur, regardless of the Execution Priority of currently executing code.

Trap interrupts are processed as part of the execution of a TRAP instruction. Since the Trap interrupt is non-maskable the interrupt vector is always taken when the TRAP instruction is executed.

Event interrupts reflect less critical hardware events, such as a UART needing service or a timer overflow. These events may be associated with some on-chip device or an external interrupt input. Event interrupts are maskable and are processed only when their priority is higher than that of currently executing code. Event interrupt priorities are software selectable by writing bits in the IPA (Interrupt Priority) register for each interrupt source. In this section we will generically refer to the IPA register but in most XA derivatives this will actually be a group of registers (IPA0–IPAn) based on the number of event interrupts available. Each event interrupt can be set to one of 16 priority levels by writing four bits in the IPA register assigned to the interrupt event. A priority level of zero effectively disables the interrupt since the priority must be greater than the Execution Priority of the code that is currently executing for the interrupt to be serviced.

Software interrupts are an extension of event interrupts, but are caused by software setting a request bit in a Special Function Register or SFR. Software interrupts are also processed only when their priority is higher than that of currently executing code. Software interrupt priorities are fixed at levels from 1 through 7. Thus code with an Execution Priority of 8 or higher can NOT be interrupted by any of the Software Interrupts.

All forms of interrupts trigger the same sequence: First, a stack frame containing the address of the next instruction and then the current value of the PSW (Program Status Word) is pushed on the System Stack. A vector containing a new PSW value and a new execution address is fetched from code memory. The new PSW value entirely replaces the old, and execution continues at the new address, e.g., at the specific interrupt service routine. Since the execution address for the Interrupt Service Routine (ISR) is only 16 bits wide, the ISR for all XA interrupt sources must begin in Page 0 of code memory (the first 64K byte page). This allows a faster interrupt response time than if a full 32 bit ISR address was fetched. Notice that all XA ISR's always begin in Page 0 of code memory independently of whether the XA is operating in Page 0 Mode or not. Page 0 Mode is a special mode where total XA code memory is limited to 64K bytes.

The new PSW value may include a new setting of PSW bit **SM** (System Mode), allowing handler routines to be executed in System or User mode, and a new value of PSW bits **IM3–IM0**, reflecting the Execution Priority of the new task. These capabilities are basic to multi-tasking support on the XA.

Returns from all interrupts should in most cases be accomplished by the RETI instruction, which pops the System Stack and continues execution with the restored PSW context. All interrupt service routines will normally be executed in System Mode. If an RETI instruction is executed from an ISR running in User Mode an exception interrupt will be generated.

The XA architecture contains sophisticated mechanisms for deciding when and if an interrupt sequence actually occurs. As described below, Exception Interrupts are always serviced as soon as they are triggered. Event Interrupts are deferred until their Execution Priority is higher than that of the currently executing code. For both exception and event interrupts, there is a systematic way of handling multiple simultaneous interrupts. Software Interrupts and Trap Interrupts occur only when program instructions generating them are executed, so there is no need for conflict resolution within these two interrupt classes.

2.2. The Interrupt Mask (Execution Priority)

The PSW operating mode flags (shown below) set several aspects of the XA operating mode including the Interrupt Mask or Execution Priority. The terms Interrupt Mask and Execution Priority are two different ways of defining the same thing. Interrupt Mask refers to the fact that all interrupts with a priority equal to or lower than this value are "masked" and are not allowed to occur. Execution Priority refers to the fact that for any interrupt (or task) to be allowed to run, it must have a higher priority than the Execution Priority of the task that is currently running. The four Interrupt Mask bits (**IM3–IM0**) identify the Execution Priority of the code that is currently executing. The XA interrupt controller compares the current setting of the IM bits to the priority of any pending interrupts to decide whether to initiate an interrupt sequence. The value 0 in the IM bits indicates the lowest Execution Priority, or fully interruptable code. The value 15 (or 0F hexadecimal) indicates the highest Execution Priority, which is not interruptable by maskable event interrupts. However, note that an Execution Priority of 15 does not inhibit servicing of Exception Interrupts or Traps since these are non-maskable.

PSWH (401h) – bit addressable

SM	TM	RS1	RS0	IM3	IM2	IM1	IM0

PSW operating mode flags

All of the flags in the upper byte of the PSW (**PSWH**), except the bits **RS1** and **RS0** (Register Bank Select), may be modified only by code running in system mode.

2.3. PSW Initialization

At reset, the initial XA PSW value is loaded from the reset vector located at address 0 in code memory. The initial **PSWH** value sets the stage for system software initialization and its value requires great attention. **PSWL** contains only status flags which do not require initialization. Therefore, the initial value of **PSWL** is generally of no special system-wide importance and may be set to zero or some other value. Philips recommends that the PSW initialization value in the reset vector sets **IM3–IM0** to all 1's so that XA initialization code is set as the highest Execution Priority process (and therefore can not be interrupted by any source other than an exception or trap). It is also recommended that the reset vector set the **SM** bit to 1, so that execution begins in System Mode. This gives an initial PSW value of 8F00H for normal operation. At the conclusion of the user initialization code, the Execution Priority is typically reduced, often to 0, to allow all other maskable interrupt driven tasks to run.

Here's an example set of declarations that create the recommended initial value of **PSWH**:

```
system_mode     equ     8000h
max_priority    equ     0F00h
initial_PSW     equ     system_mode + max_priority
```

2.4. Interrupt Service Data Elements

There are two data elements associated with XA interrupts. The first is the stack frame created when each interrupt is serviced. The second is the interrupt vector table located at the beginning of code memory. Understanding the structure and contents of each is essential to the understanding of how XA interrupts are processed.

2.4.1. Interrupt Stack Frame

A stack frame is generated, always on the System Stack, for each XA interrupt. The stack frame is stored for the duration of interrupt service and used to return to and restore the CPU state of the interrupted code. There is one case where this is not true. The Exception Interrupt triggered by a Reset event re-initializes the stack pointers, so no stack frame is preserved. This makes the Reset Exception Interrupt unique since it is not terminated with an RETI like all other XA interrupts

The stack frame in the native 24-bit XA operating mode is shown in Figure 1. Three words (6 bytes) are stored on the stack in this case. The first word pushed is the low-order 16 bits of the current Program Counter (PC), i.e., the address of the next instruction to be executed. The next word contains the high-order byte of the current PC. A zero byte is used as a pad since the stack must be word aligned. Since a complete 24-bit address is stored in the stack frame a return to any code location in the 16M byte XA address range is possible. The third word in the XA stack frame contains a copy of the PSW at the instant the interrupt was serviced.

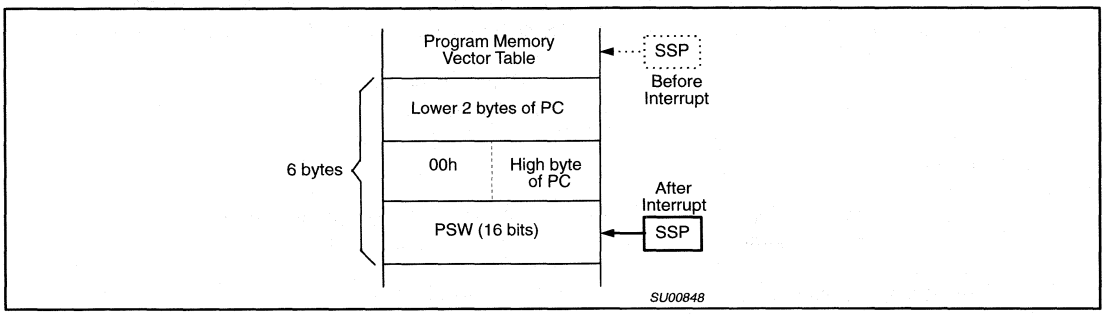


Figure 1. XA Stack Frame – Non Page 0 Mode (24 bit mode)

When the XA is operating in Page 0 Mode (**SCR** bit **PZ** = 1) the stack frame is smaller. In Page 0 Mode, only 16 address bits are used throughout the XA. The stack frame in Page 0 Mode is only four bytes since the High Byte of the PC and the pad byte are not needed. Obviously, it is very important that stack frames of both sizes not be mixed since this would corrupt the return address and therefore the operation of the XA. This is one reason it is recommended that the user set the System Configuration Register (**SCR**) once during XA initialization to select either Page 0 Mode or 24 bit address mode, and leave it unchanged thereafter.

2.4.2. Interrupt Vector Table

The XA uses the first 284 bytes of code memory (addresses 0 – 011B hex) for an interrupt vector table. The table may contain up to 71 double-word entries, each corresponding to a particular interrupt event.

The double-word entries each consist of a 16 bit address of an Interrupt Service Routine (ISR) and a 16 bit PSW replacement value. Because vector addresses are 16-bit, the first instruction of each Interrupt Service Routine must be located in the first 64K bytes of XA memory. The first instruction of all ISR's must also be word-aligned. Note that this is normally handled by the XA assembler, which will insert NOP's automatically to assure word-alignment of ALL labels. The replacement PSW value contains key elements such as the choice of System or User mode for the service routine, the Register Bank selection, and an Interrupt Mask setting.

The first 16 vectors, starting at code memory address 0 are reserved for Exception Interrupt vectors. The second 16 vectors are reserved for Trap Interrupts. The following 32 vectors in the table are reserved for Event Interrupts. The final 7 vectors are used for Software Interrupts. A figure presented later will illustrate the XA vector table and the structure of each component vector. Of the vectors assigned to Exceptions, 7 are assigned to events specific to the XA CPU and 9 are reserved. All 16 Trap Interrupts may be used freely since none are reserved. Assignments of Event Interrupt vectors are derivative dependent and vary with the peripheral device complement and pinout of each XA derivative.

Unused interrupt vector locations should typically be set to point to a "null" service routine (an RETI instruction), rather than be overwritten by executable instructions. This is especially true of the exception interrupts, since these are non-maskable and could conceivably occur in a system where the designer did not expect them. If these vectors are routed to an RETI instruction, the system can essentially ignore the unexpected exception or interrupt condition and continue operation.

Note that when using some hardware development tools it may be preferable not to initialize unused vector locations with a "null handler". This allows the XA development tool to recognize and flag these unexpected interrupt conditions so they can be addressed.

2.5. The Reset Exception Interrupt

Immediately after the –RST line goes high, the XA generates a Reset Exception Interrupt. As a result, the initial PSW and address of the first instruction (the "start-up code") are fetched from the reset vector in code memory at location 0. Here's an example in generalized assembler format of the setup for the Reset Exception Interrupt:

```
                code_seg                ; establish code segment
                org     0h              ; start at address 0

; reset_vector

                dw      initial_PSW     ; PSW reset value – normally 8F00H
                dw      startup_code    ; starting address of code

; the XA Interrupt Vector Table goes from 0 –011Bh in code memory

                org     120h            ; start code at address 120h
                                        ; (above interrupt vector table)

startup_code:   . . .                   ; put user startup code here
                . . .
                . . .
; end user startup code by enabling ALL interrupts

                mov.b   PSWH, #80H      ; PSWH run value to allow ALL interrupts
                mov.b   PSWL, #00H      ; PSWL value is not critical
```

The **PSWH** initialization value given in this example sets System Mode (**SM**), selects register bank 0 (any register bank could be used) and clears **TM** so that Trace Mode is inactive.

The startup_code sequence may be followed directly by user startup code or by a simple branch to any application code. At the end of user initialization code remember to lower the Interrupt Mask value in **PSWH** so maskable event interrupts can occur. Do NOT use an RETI instruction at the conclusion of the startup_code sequence even though this is part of the Reset Exception Interrupt handler. The Reset initializes the Stack Pointer (**SP**) and does

not leave an interrupt stack frame. This makes the Reset Exception Interrupt unique since it is not terminated with an RETI like all other XA interrupts.

Notice that the same Reset Exception Interrupt is generated for any of the three possible XA reset sources:

1. Hardware reset via the –RST pin
2. Software reset via the RESET instruction
3. Watchdog Timer generated reset

2.6. XA Interrupt Types

This section describes the four types of XA interrupts. It addresses interrupts that are available in the XA Family but may or may not be present on any given XA derivative.

2.6.1. Exception Interrupts

Exception interrupts reflect events of overriding importance and are always serviced when they occur. Exceptions currently defined in the XA core include: Reset, Breakpoint, Trace, Divide-by-0, Stack overflow, and Return from Interrupt (RETI) executed in User Mode. Nine additional exception interrupts are reserved.

NMI is listed in the table of exception interrupts below because NMI is handled by the XA core in the same manner as exceptions, and factors into the precedence order of exception processing. However, the vector address reserved for NMI is actually mapped right in the middle of the Event Interrupt vector address space. This should not cause NMI, which is a non-maskable Exception Interrupt, to be confused with the maskable Event Interrupts. Note that NMI is part of the XA Family Interrupt Structure but is not implemented on the first XA derivative (the XA-G3).

Since exception interrupts are by definition not maskable, they must always be serviced immediately regardless of the Execution Priority level of currently executing code (as defined by the IM bits in the PSW). In the unusual case that more than one exception is triggered at the same time, there is a hard-wired service precedence ranking. This ranking determines which exception vector is taken first if multiple exceptions occur. Of course, being non-maskable, any exception occurring during execution of the ISR for another exception will still be serviced immediately. In this case, the exception vector taken last may be considered the highest priority, since its code will execute first. This LIFO (Last-In-First-Out) system means that an Exception Interrupt with a higher service precedence actually has a higher priority. Even though the Exception with the higher service precedence will be taken last, it will still be serviced first.

Programmers should be aware of the following when writing Exception Interrupt handlers:

1. Since another exception could interrupt a stack overflow ISR, care should be taken in all exception handler code to minimize the possibility of a destructive stack overflow. Remember that stack overflow exceptions only occur once as the stack crosses the lower address limit of 0080h.
2. The Breakpoint (caused by execution of the BKPT instruction, or a hardware breakpoint in an emulation system) and Trace exceptions are intended to be mutually exclusive. In both cases, the handler code will want to know the address in user code where the exception occurred. If a breakpoint occurs during trace mode, or if trace mode is activated during execution of the breakpoint handler code, one of the handlers will see a return address on the stack that points within the other handler code.

Exception Interrupts – Non Maskable

Exception Interrupt	Vector Address	Arbitration Ranking	Service Precedence
Breakpoint	0004h–0007h	1	0
Trace	0008h–000Bh	1	1
Stack Overflow	000Ch–000Fh	1	2
Divide-by-zero	0010h–0013h	1	3
User RETI	0014h–0017h	1	4
<reserved1>	0018h–001Bh	—	—
<reserved2>	001Ch–001Fh	—	—
<reserved3>	0020h–0023h	—	—
<reserved4>	0024h–0027h	—	—
<reserved5>	0028h–002Bh	—	—
<reserved6>	002Ch–002Fh	—	—
<reserved7>	0030h–0033h	—	—
<reserved8>	0034h–0037h	—	—
<reserved9>	0038h–003Fh	—	—
NMI	009Ch–009Fh	1	6
Reset	0000h–0003h	0 (High)	7 always serviced immediately aborts other exceptions

2.6.2. Trap Interrupts

Trap Interrupts are intended to support application-specific requirements, as a convenient mechanism to enter globally used routines, and to allow transitions between User Mode and System Mode. TRAP 0 through TRAP 15 are defined and may be used as required by applications. Trap interrupts are generated by the TRAP instruction. A trap interrupt will occur if and only if the instruction is executed, so there is no need for a precedence scheme with respect to simultaneous traps. A trap acts like an immediate non-maskable interrupt, using a vector to call one of several pieces of code that will be executed in System Mode. This may be used to obtain system services for application code, such as altering the Data Segment register for example. Some XA development software and Real Time Operating Systems may reserve certain Trap instructions for specific system functions. An example of this would be the Hitech XA C compilers use of Trap 15 to access system services.

Traps – Non-Maskable

Description	Vector Address	Arbitration Ranking
Trap 0	0040–0043h	1
Trap 1	0044–0047h	1
Trap 2	0048–004Bh	1
Trap 3	004C–004Fh	1
Trap 4	0050–0053h	1
Trap 5	0054–0057h	1
Trap 6	0058–005Bh	1
Trap 7	005C–005Fh	1
Trap 8	0060–0063h	1
Trap 9	0064–0067h	1
Trap 10	0068–006Bh	1
Trap 11	006C–006Fh	1
Trap 12	0070–0073h	1
Trap 13	0074–0077h	1
Trap 14	0078–007Bh	1
Trap 15	007C–007Fh	1

Example of Trap Interrupt:

```
        TRAP    #05             ; generate Trap 5 Interrupt

        ; immediate branch to TRAP05 Interrupt Service Routine (non-maskable)
```

Example of ISR for a Trap Interrupt:

```
TRAP05:     .
            . user code
            .
            RETI
```

Notice that the Execution Priority (**IM3–IM0** value) is not relevant since Traps are non-maskable. When the TRAP instruction is executed the Trap Interrupt will always occur. No user action is required in the ISR to "clear" the Trap before the RETI is executed.

2.6.3. Event Interrupts

On typical XA derivatives, event interrupts will arise from on-chip peripherals and from events detected on external interrupt input pins. Event interrupts may be globally enabled/disabled via the **EA** bit in the Interrupt Enable register (**IE**) and individually masked by specific bits in the **IE** register or registers. When an event interrupt for a peripheral device is disabled but the peripheral is not turned off, the peripheral interrupt flag can still be set by the peripheral. If the peripheral interrupt is re-enabled an interrupt will occur. An event interrupt that is enabled can only be serviced when its Execution Priority is higher than that of the currently executing code. Event Interrupts have 16 priority levels that can be individually set in the Interrupt Priority (IPA) register for the appropriate interrupt source. This allows tight control over the scheduling and occurrence of each maskable XA interrupt source. If more than one event interrupt occurs at the same time, the higher priority setting will determine which one is serviced first. If more than one interrupt is pending at the same priority level, a hardware precedence scheme is used to choose the first to service. Consult the data sheet for a specific XA derivative for details on the hardware precedence scheme or arbitration ranking.

Note that the PSW (including the Interrupt Mask or Execution Priority bits) is loaded from the interrupt vector table when an event interrupt is serviced. Thus, the priority at which the ISR executes could be different from the priority at which the interrupt occurred. Since the occurrence priority is determined by the IPA register setting for that interrupt rather than by the PSW image in the vector table. Normally it is advisable to set the Execution Priority in the interrupt vector to be the same as the IPA register setting that will be used in the code. If the Execution Priority for any ISR is set lower than the Interrupt Priority for that interrupt, then that interrupt will interrupt itself continuously and likely overflow the stack. This can occur since most event interrupts are still pending during part of the ISR.

Event Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable bit	Interrupt Priority	Arbitration Ranking
External interrupt 0	IE0	0080–0083h	EX0	IPA0.3–0	2
Timer 0 interrupt	TF0	0084–0087h	ET0	IPA0.7–4	3
External interrupt 1	IE1	0088–008Bh	EX1	IPA1.3–0	4
Timer 1 interrupt	TF1	008C–008Fh	ET1	IPA1.7–4	5
Timer 2 interrupt	TF2(EXF2)	0090–0093h	ET2	IPA2.3–0	6
<reserved1>		0094–0097h			
<reserved2>		0098–009Bh			
NMI (non-maskable)		009C–009Fh			1
Serial port 0 Rx	RI.0	00A0–00A3h	ERI0	IPA4.3–0	7
Serial port 0 Tx	TI.0	00A4–00A7h	ETI0	IPA4.7–4	8
Serial port 1 Rx	RI.1	00A8–00ABh	ERI1	IPA5.3–0	9
Serial port 1 Tx	TI.1	00AC–00AFh	ETI1	IPA5.7–4	10
<reserved3>		00B0–00B3h			
<reserved4>		00B4–00B7h			
<reserved5>		00B8–00BBh			
<reserved6>		00BC–00BFh			
<reserved7>		00C0–00C3h			
<reserved8>		00C4–00C7h			
<reserved9>		00C8–00CBh			
<reserved10>		00CC–00CFh			
<reserved11>		00D0–00D3h			
<reserved12>		00D4–00D7h			
<reserved13>		00D8–00DBh			
<reserved14>		00DC–00DFh			
<reserved15>		00E0–00E3h			
<reserved16>		00E4–00E7h			
<reserved17>		00E8h–00EBh			
<reserved18>		00EC–00EFh			
<reserved19>		00F0–00F3h			
<reserved20>		00F4–00F7h			
<reserved21>		00F8–00FBh			
<reserved22>		00FC–00FFh			

Notice that the vector address reserved for NMI is mapped into the Event Interrupt vector address space. This should not cause NMI, which is a non-maskable Exception Interrupt, to be confused with the maskable Event Interrupts. The NMI vector address is mapped into this space because NMI shares certain characteristics with the External Interrupts. Both NMI and External Interrupts are generated by a signal on an external XA pin that is then fed into the XA interrupt controller.

2.6.4. Software Interrupts

Software Interrupts act just like event interrupts, except that they are caused by software writing to an interrupt request bit in an SFR. The standard XA implementation of the software interrupt mechanism provides 7 interrupts that are associated with 2 SFRs. One SFR, the Software Interrupt Request register (**SWR**), contains 7 request bits – one for each software interrupt. The second SFR is the Software Interrupt Enable register (**SWE**), containing one enable bit for each software interrupt.

SWR (42Ah) – bit addressable

—	SWR7	SWR6	SWR5	SWR4	SWR3	SWR2	SWR1

Software Interrupt Request

SWE (47Ah) – **NOT** bit addressable

—	SWE7	SWE6	SWE5	SWE4	SWE3	SWE2	SWE1

Software Interrupt Enable

Software interrupts have fixed interrupt priorities, one each at priorities 1– 7. These are shown in the table below. Software interrupts are available in the XA Family Interrupt Structure but may not be present on all XA derivatives. Consult the data sheet for a specific XA derivative for details on the availability of software interrupts.

Software Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable Bit	Interrupt Priority
Software interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0118–011B	SWE7	(fixed at 7)

Example of Software Interrupt:

```
OR.B    SWE, #01    ; enable Software Interrupt 1 indirectly
                    ; since SWE not bit addressable!
SETB    SWR1        ; generate Software Interrupt 1
```

; branch to SWI1 Interrupt Service Routine if and only if the current Execution Priority (**IM3–IM0**) = 0
; if the Execution Priority of the running code is > 0, then SWI1 will NOT occur, but will remain pending

Example of ISR for a Software Interrupt:

```
SWI1:   .
        CLR   SWR1      ; clear Software Interrupt 1
        .
        RETI
```

Notice that Software Interrupt 1 has a fixed priority of 1. This means that the **IM3–IM0** value would need to be 0 (Execution Priority of the current executing code equal 0) for this priority 1 interrupt to occur. Any **IM3–IM0** value > 0 would block the Software Interrupt 1 from occurring. The SWR1 bit must be cleared by the user before exiting the ISR or the Software Interrupt will re-occur.

It is also important to address the Software Interrupt Request bits by their bit addressable names and not by their bit position in **SWR** since they are shifted (i.e., **SWR1** is SWR.0 not SWR.1). Thus it is correct to use these instructions:

```
SETB   SWR1          ; generate Software Interrupt 1
CLR    SWR1          ; clear Software Interrupt 1
```

but incorrect to use these instructions:

```
SETB   SWR.1         ; actually would generate Software Interrupt 2
CLR    SWR.1         ; actually would clear Software Interrupt 2
```

Since the Software Interrupt Enable register is NOT bit addressable it is wise to enable Software Interrupts as shown below (paying close attention to the actual bit position of the desired enable bit):

```
OR.B     SWE, #01H   ; enable Software Interrupt 1 indirectly
OR.B     SWE, #02H   ; enable Software Interrupt 2 indirectly
OR.B     SWE, #04H   ; enable Software Interrupt 3 indirectly
OR.B     SWE, #08H   ; enable Software Interrupt 4 indirectly
OR.B     SWE, #10H   ; enable Software Interrupt 5 indirectly
OR.B     SWE, #20H   ; enable Software Interrupt 6 indirectly
OR.B     SWE, #40H   ; enable Software Interrupt 7 indirectly
```

Using the "OR" instruction allows the individual Software Interrupt to be enabled without affecting the setting of the enable bits for any other Software Interrupts.

The primary purpose of the software interrupt mechanism is to provide an organized way in which portions of the event interrupt routines may be executed at a lower priority level than the one at which the service routine began. An example of this would be an event Interrupt Service Routine that has been given a very high priority in order to respond quickly to some critical external event. This ISR has a relatively small portion of code that must be executed immediately, and a larger portion of follow-up or "clean-up" code that does not need to be completed right away (but does not need to wait until the main software loop). Overall system performance may be improved if the lower priority portion of the ISR is actually executed at a lower priority level, allowing other more important interrupts to be serviced.

If the high priority ISR simply lowers its execution priority at the point where it enters the follow-up code, by writing a lower value to the IM bits in the PSW, a situation called "priority inversion" could occur. Priority inversion describes a case where code at a lower priority is executing while a higher priority routine is kept waiting. An example of how this could occur by writing to the IM bits follows, and is illustrated in Figure 2.

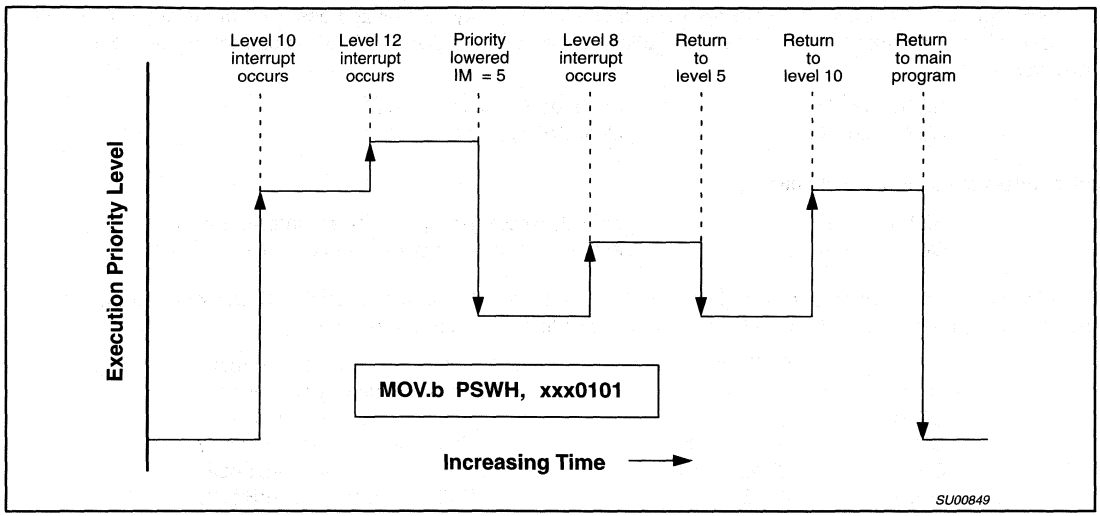


Figure 2. Priority Inversion (No Software Interrupts)

Suppose code is executing at level 0 and is interrupted by an event interrupt that runs at level 10. This is again interrupted by a level 12 interrupt. The level 12 ISR completes a time-critical portion of its code and wants to lower the priority of the remainder of its code (the non-time critical portion) in order to allow more important interrupts to occur. So, it writes to the IM bits, setting the execution priority to 5. The ISR continues executing at level 5 until a level 8 event interrupt occurs. The level 8 ISR runs to completion and returns to the level 5 ISR, which also runs to completion. When the level 5 ISR completes, the previously interrupted level 10 ISR is reactivated and eventually completes.

It can be seen in this example that lower priority ISR code executed and completed while higher priority code was kept waiting on the stack. This is priority inversion.

In those cases where it is desirable to alter the priority level of part of an ISR, a software interrupt may be used to accomplish this without risk of priority inversion. The ISR must first be split into 2 pieces: the high priority portion, and the lower priority portion. The high priority portion remains associated with the original interrupt vector. The lower priority portion is associated with the interrupt vector for a software interrupt, in this case Software Interrupt 5. At the completion of the high priority portion of the ISR, the code sets the request bit for software interrupt 5, and then returns. The remainder of the ISR, now actually the ISR for software interrupt 5, executes when it becomes the highest priority pending interrupt.

The diagram in Figure 3 shows the same sequence of events as in the example of priority inversion, except using software interrupt 5 as just described. Note that the code now executes in the correct order (higher priority first).

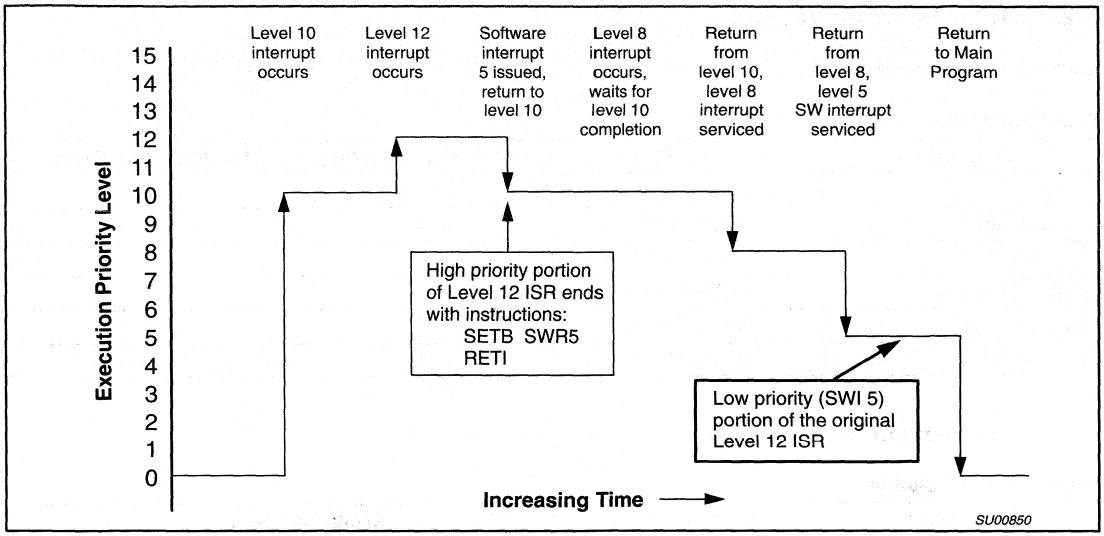


Figure 3. Correct Priority Execution with Software Interrupts

3. XA-G3 INTERRUPT STRUCTURE

This section covers only the interrupts that are implemented on the XA-G3. Recall that not all interrupt options covered in the XA Family Interrupt Structure are available in this first XA derivative product. Our focus here will be on the actual XA-G3 Interrupts and any differences from the XA Family Interrupts. For details on XA-G3 interrupts that are identical to the XA Family Interrupts please refer to the appropriate section in the "XA Family Interrupt Structure".

3.1. XA-G3 Interrupts

The XA-G3 defines four types of interrupts:

- Exception Interrupts – These are system level errors and other very important occurrences that include Stack overflow, Divide by 0, Breakpoint, Trace, User Mode RETI and Reset.
- Trap Interrupts – These are TRAP instructions, generally used to call system services in a multi-tasking system.
- Event Interrupts – These are peripheral interrupts from devices such as UARTs, timers, and external interrupt inputs.
- Software Interrupts – These are equivalent to hardware event interrupts, but are requested only under software control and have fixed priority levels.

Exception interrupts, trap interrupts, and software interrupts are generally standard for XA derivatives and are detailed in the XA Family Interrupt Structure. Event Interrupts tend to be different on various XA derivatives and will be explained in detail for the XA-G3.

The XA-G3 supports 38 vectored interrupt sources. These include 9 maskable Event Interrupts (for the various XA-G3 peripherals), 7 Software Interrupts, 6 Exception Interrupts and 16 Traps.

The complete interrupt vector list for the XA-G3, including all 4 interrupt types, is shown in the following tables. The tables include the address of the vector for each interrupt, the related priority register bits (if any), and the arbitration ranking for that interrupt source. The arbitration ranking determines the order in which interrupts are processed if more than one interrupt of the same priority occurs simultaneously.

3.2. XA-G3 Interrupt Vectors

3.2.1. Exception Interrupts

Exceptions – Non-Maskable

Description	Vector Address	Arbitration Ranking	Service Precedence
Reset	0000–0003h	0 (High)	7
Breakpoint	0004–0007h	1	0
Trace	0008–000Bh	1	1
Stack Overflow	000C–000Fh	1	2
Divide-by-0	0010–0013h	1	3
User RETI	0014–0017h	1	4

3.2.2. Trap Interrupts

Traps – Non-Maskable

Description	Vector Address	Arbitration Ranking
Trap 0	0040–0043h	1
Trap 1	0044–0047h	1
Trap 2	0048–004Bh	1
Trap 3	004C–004Fh	1
Trap 4	0050–0053h	1
Trap 5	0054–0057h	1
Trap 6	0058–005Bh	1
Trap 7	005C–005Fh	1
Trap 8	0060–0063h	1
Trap 9	0064–0067h	1
Trap 10	0068–006Bh	1
Trap 11	006C–006Fh	1
Trap 12	0070–0073h	1
Trap 13	0074–0077h	1
Trap 14	0078–007Bh	1
Trap 15	007C–007Fh	1

3.2.3. Event Interrupts

Event Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable bit	Interrupt Priority	Arbitration Ranking
External interrupt 0	IE0	0080–0083h	EX0	IPA0.2–0	2
Timer 0 interrupt	TF0	0084–0087h	ET0	IPA0.6–4	3
External interrupt 1	IE1	0088–008Bh	EX1	IPA1.2–0	4
Timer 1 interrupt	TF1	008C–008Fh	ET1	IPA1.6–4	5
Timer 2 interrupt	TF2(EXF2)	0090–0093h	ET2	IPA2.2–0	6
Serial port 0 Rx	RI.0	00A0–00A3h	ERI0	IPA4.2–0	7
Serial port 0 Tx	TI.0	00A4–00A7h	ETI0	IPA4.6–4	8
Serial port 1 Rx	RI.1	00A8–00ABh	ERI1	IPA5.2–0	9
Serial port 1 Tx	TI.1	00AC–00AFh	ETI1	IPA5.6–4	10

3.2.4. Software Interrupts

Software Interrupts – Maskable

Description	Flag Bit	Vector Address	Enable Bit	Interrupt Priority
Software interrupt 1	SWR1	0100–0103h	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107h	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010Bh	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010Fh	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113h	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117h	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0118–011Bh	SWE7	(fixed at 7)

Arbitration ranking is only relevant when more than one interrupt (from the same category) is triggered at the same time. For example, 2 exceptions or 2 event interrupts at the same time would use the arbitration ranking to determine which interrupt source was serviced first. The interrupt with the lower arbitration ranking will be serviced first, and thus has a higher priority. Since this simultaneous triggering is not possible for Traps or Software Interrupts, these two interrupt categories do not require an arbitration ranking.

Since Exceptions and Traps are non-maskable they will always occur immediately and therefore do not require an Interrupt Priority. Exceptions and Traps may be considered to have “infinite” priority.

3.3. XA-G3 Event Interrupts

The 9 maskable Event Interrupts on the XA-G3 share a global interrupt enable bit (the **EA** bit in the **IEL** register) and each also has a separate individual interrupt enable bit (in the **IEH** or **IEL** registers). Notice that the power-up reset value for **EA** and each of the separate interrupt enable bits is 0. This effectively disables each of the maskable interrupts in two different places. For a maskable event interrupt to occur the global **EA** bit must be set to "1" and the individual interrupt enable bit in the **IEH/IEL** register must be set for that particular interrupt source. The interrupt enable bits were listed in the previous table of Event Interrupts along with their associated interrupt. As shown below the interrupt enable bits are all bit addressable.

IEH (427h) – bit addressable

—	—	—	—	**ETI1**	**ERI1**	**ETI0**	**ERI0**

Interrupt Enable High Byte

IEL (426h) – bit addressable

EA	—	—	**ET2**	**ET1**	**EX1**	**ET0**	**EX0**

Interrupt Enable Low Byte

In the XA-G3 each event interrupt can be set to occur at 1 of 8 priority levels via bits in the Interrupt Priority (IPA) registers, **IPA0–IPA5**. The value 0 in the IPA field gives the interrupt priority 0, in effect, disabling the interrupt. Since the **IPA0–IPA5** registers all have power-up reset values of 0, each of the event interrupts starts with a priority of 0 and is thus disabled.

The XA-G3 differs slightly from the XA Family Interrupt Structure in the way that interrupt priority levels are set via the IPA registers. Since only 3 of the 4 IPA register bits are implemented in the XA-G3, only 8 of the 16 possible priority levels are available for each of the event interrupts. The value 0000h in one nibble of the **IPA0–IPA5** register gives the interrupt priority 0, a value of 0001h gives the interrupt a priority of 9, the value 0010h gives priority 10, etc. The value 0111h in one nibble of the **IPA0–IPA5** register gives the interrupt priority 15. Since the MSB or 4th bit in each nibble of the **IPA0–IPA5** register is not implemented in the XA-G3, writing the value 0001h or 1001h to the IPA register will yield the same results. The interrupt in question will be set to a priority level of 9 in both cases. However, since the 4th bit in each nibble of the **IPA0–IPA5** register is not implemented it can not be read back if written. If 1001h is written to either nibble of the **IPA0–IPA5** register and then read back, the value returned will be 0001h.

On the XA-G3 the user may want to write any non-zero IPA value with the upper bit always set. This provides both a reminder of the true interrupt priority and software compatibility with future XA derivatives.

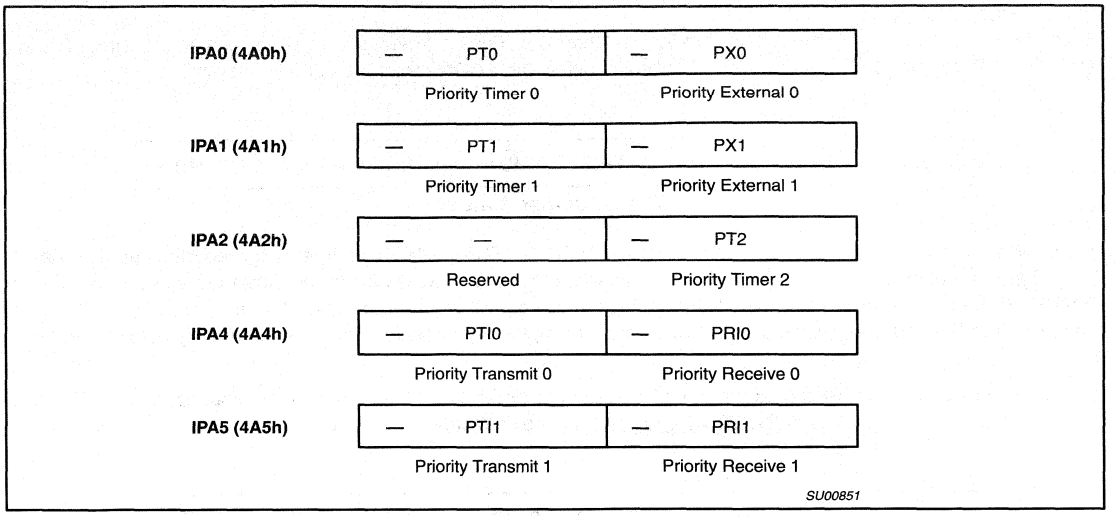


Figure 4. Interrupt Priority Registers IPA0–IPA5

Since event interrupts in the XA-G3 only support 8 of the 16 priority levels available in the XA Family Interrupt Structure, they can only have priorities of either 0 or 9–15. This means that Software Interrupts, with fixed priorities of 1–7, can not be granted higher priority than any of the XA-G3 Event Interrupts.

Event interrupts in the XA-G3 can be grouped into three basic types:

1. External Interrupts
2. Timer Interrupts
3. Serial Port Interrupts

Let's take a detailed look at each type of event interrupt.

3.3.1. External Interrupts

External interrupts available on the XA-G3 are External Interrupt 0 and External Interrupt 1. These external interrupts are controlled by bits in the **TCON** register as shown below:

TCON (410h) – bit addressable

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Timer/Counter Control

External interrupts can be either falling edge triggered or low level triggered. This is controlled by the Interrupt Type Control bits **IT1/IT0**. If **IT1/IT0** is set to "1" then that interrupt will be set for falling edge trigger. If **IT1/IT0** is set to "0" then that interrupt will be set for low level trigger. When an external interrupt is detected it will set the Interrupt Edge Flag **IE1/IE0**. If the external interrupt is enabled the setting of this flag will generate an External Interrupt 1 or External Interrupt 0. The **IE1/IE0** flag will be cleared when the interrupt is processed or it can be cleared by software at any time.

3.3.2. Timer Interrupts

Timer interrupts available on the XA-G3 are Timer 0 interrupt, Timer 1 interrupt and Timer 2 interrupt. Timer 0 and Timer 1 interrupts are identical and are controlled by bits in the **TCON** register as shown below:

TCON (410h) – bit addressable

TF1	**TR1**	**TF0**	**TR0**	**IE1**	**IT1**	**IE0**	**IT0**

Timer/Counter Control

The timer is turned on by setting the Timer Run Control bit **TR1/TR0** to "1". The timer is turned off by setting the Timer Run Control bit **TR1/TR0** to "0". When the timer/counter overflows it will set the Timer Overflow Flag **TF1/TF0**. If the timer interrupt is enabled the setting of this flag will generate a Timer 0 Interrupt or a Timer 1 Interrupt. The **TF1/TF0** flag will be cleared when the interrupt is processed or it can be cleared by software at any time.

Timer 2 on the XA-G3 has additional functional modes over Timer 0 and 1 that will not be discussed here. Timer 2 interrupts are controlled by bits in the **T2CON** register as shown below:

T2CON (418h) – bit addressable

TF2	**EXF2**	**RCLK0**	**TCLK0**	**EXEN2**	**TR2**	**C/T2**	**CP/RL2**

Timer/Counter Control

Timer 2 is turned on by setting the Timer Run Control bit **TR2** to "1". Timer 2 is turned off by setting the Timer Run Control bit **TR2** to "0". When the timer/counter overflows it will set the Timer 2 Overflow Flag **TF2**. If the timer 2 interrupt is enabled, the setting of this flag will generate a Timer 2 Interrupt. The **TF2** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Timer 2 interrupt will reoccur. If **RCLK1/RCLK0** or **TCLK1/TCLK0** are set to "1", then the Timer 2 overflow rate is being used as a baud rate clock source for UART0 or UART1. In this case the **TF2** flag will NOT be set when the timer/counter overflows.

If Timer 2 is enabled in external capture or reload mode, a negative transition on the T2EX pin will set the Timer 2 external flag **EXF2**. If the Timer 2 interrupt is enabled, the setting of the Timer 2 external flag **EXF2** can also generate a Timer 2 Interrupt. The **EXF2** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Timer 2 interrupt will reoccur.

3.3.3. Serial Port Interrupts

The two Serial Ports on the XA-G3 are identical and are called Serial Port 0 and Serial Port 1. Each Serial Port has two interrupts – one for the transmitter and one for the receiver. Notice that this is an enhancement over the Serial Port on the 8051 (which had only a single shared interrupt for both the transmitter and receiver). This gives the XA-G3 a total of four interrupts for the Serial Ports:

1. Serial Port 0 Rx
2. Serial Port 0 Tx
3. Serial Port 1 Rx
4. Serial Port 1 Tx

These Serial Port interrupts are controlled by bits in identical registers called **S0CON** and **S1CON**. To avoid confusion we will look only at **S0CON** as shown below:

S0CON (420h) – bit addressable

SM0_0	**SM1_0**	**SM2_0**	**REN_0**	**TB8_0**	**RB8_0**	**TI_0**	**RI_0**

Serial Port 0 Control

The Serial Port 0 receiver is enabled by setting the Receiver Enable bit **REN_0** to "1". The Serial Port 0 receiver is disabled by setting the Receiver Enable bit **REN_0** to "0". When a character is received by Serial Port 0 the Receive Interrupt Flag **RI_0** will be set. If the Serial Port 0 Rx interrupt is enabled the setting of this flag will generate a Serial Port 0 Rx Interrupt. The **RI_0** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Serial Port 0 Rx interrupt will reoccur.

When a character is transmitted by Serial Port 0 the Transmit Interrupt Flag **TI_0** will be set. If the Serial Port 0 Tx interrupt is enabled the setting of this flag will generate a Serial Port 0 Tx Interrupt. The **TI_0** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Serial Port 0 Tx interrupt will reoccur.

Serial Port 0 also has a Status Interrupt flag **STINT0** that is contained in the Serial Port 0 Extended Status Register (**S0STAT**). If the **STINT0** flag is set to "1" the extended status flags are enabled and any one of them can also generate a Serial Port 0 Rx Interrupt by setting the **RI_0** flag. These extended status flags include Framing Error, Overrun Error and Break Detect. Please refer to the XA-G3 data sheet for more details on these flags. The **RI_0** flag will NOT be cleared when the interrupt is processed so it must be cleared by software or the Serial Port 0 Rx interrupt will reoccur.

As mentioned earlier the function of the Serial Port 1 Interrupts is identical to the Serial Port 0 Interrupts and therefore will not be covered here.

Using the XA EAn/WAIT pin AN96075

Author: Marco Kuystermans, Systems Laboratory Eindhoven, The Netherlands

ABSTRACT

The XA is the high speed 16-bit successor of the 80C51. To be able to use relatively slow (to the XA) 80C51 peripherals, a wait state generator is needed. The wait pin on the XA is multiplexed with the EAn function and therefore some precautions have to be taken. The behavior of this pin is different than on the 80C51, and users must be careful using this pin.

SUMMARY

To limit the number of pins on the XA, some pins have multiple functions. EAn/WAIT is one of them. During RESET, the EAn pin determines the memory configuraiton of the XA. After RESET, the EAn/WAIT pin becomes an active high WAIT pin. To use both functions, extra hardware is needed. This document describes how to construct this hardware.

This document assumes that users are familiar with the XA and its bus interface.

CONTENTS

1 INTRODUCTION

The PHILIPS SEMICONDUCTORS XA is high speed 16 bit 80C51 compatible microcontroller. The XA is developed as growing path for 80C51 users who need multitasking, more power or more addressing space. To minimise the learning curve for 80C51 users, the XA is software and partly hardware compatible with the 80C51. Of course the XA cannot be completely hardware compatible with the 80C51, otherwise the 80C51 limitations would reflect into the XA.

The hardware compatibility contains the following items:

- The embedded peripherals like the UARTs and timers are 80C51 compatible. Because the 80C51 user is already familiar with these peripherals, she/he can start designing immediately. Also existing 80C51 drivers can be ported very easily.
- The XA has been fitted with a 80C51 compatible bus interface to be able to (re) use 80C51 compatible peripherals. This XA bus interface however, is 16 bit wide and faster than the 80C51 bus interface. To be able to adapt the XA bus speed to a slower 80C51 peripheral, the XA includes a WAIT input. This high active signal inserts wait states during a write or read cycle, stretching the corresponding data or code strobe.

Because the XAG3 resides in a PLCC44/QFP44 package, the number of free pins is limited. Due to this limitation, the WAIT pin is being multiplexed with the EAn pin. This EAn pin determines if the XA will boot from internal or external ROM and is only used during XA boot time.

Using the EAn/WAIT pin is not straight forward due to the double function this pin, resulting most of the time in extra hardware. This document provides several tips, design hints and solutions to overcome this problem. Firstly an overview is given in which cases extra hardware is needed. Secondly a number of possible EAn circuits is presented, with their corresponding advantages and disadvantages. Thirdly a collection of wait state generators is given. Finally, because the XAG3 has no CLOCK out, a chapter has been added how to provide a CLOCK to both the XA and the additional hardware.

Any wait state generator described in this document can be combined with any described EAn circuit, so it is up to the user which combination he/she wants to use. As an ultimate solution an implementation in CPLD is included. These sources can be integrated together with user specific equations (e.g. address decoding) to one CPLD.

2 THE XA EAN/WAIT PIN

2.1 General EAn and WAIT aspects

The EAn (External Access) pin is used to configure the XA's memory configuration during reset. After reset this PIN becomes a high active WAIT pin. The XA will run in external memory only mode when EAn is held low DURING reset. When EAn is held high during reset, the XA will run in on-chip or mixed memory mode.
During RESET all I/O pins are pulled high, however for P0 and P2 it depends on the logical level on the EAn pin if it is a weak or a "strong" (low impedance) pull-up. The status of the EAn pin is immediately reflected into the type of pull up, even during reset.

Unlike the 80C51 [2] the XA EAn pin cannot **always** be connected *directly* to Vdd or GND. If EAn is connected to Vdd WAIT states will be inserted as soon as the XA accesses the external bus (but the external bus only, because the WAIT pin only applies to the external bus).
The following table gives an overview of all possible WAIT/EAn combinations.

Table 1, Possible memory / WAIT combinations

	External only	Mixed mode	Internal only
WAIT needed	Connect EAn to WAIT generator	EAn circuit + WAIT generator (2.3,2.4)	NOT APPLICABLE
No WAIT needed	Connect EAn to GND	only EAn circuit needed (2.3)	Connect EAn to Vdd

In the following situations NO extra logic is needed:

- If the XA must run in external mode only and NO wait states need to be generated; the EAn/WAIT pin can be directly connected to ground.
- When in external memory mode wait states are needed, the EAn/WAIT pin can be connected to the WAIT state generator directly without extra logic. During reset ALL data strobes are high and consequently NO wait signal will be generated (be absolutely sure that WAIT cycles are generated only as a result of a data strobe, see 2.4)
- In case the XA will run internal only, EAn can be connected to Vdd directly because no wait states can be generated in this memory mode. Please be absolutely sure the XA is not accessing external memory in this mode. The XA will enter an infinitive wait as soon as external memory is accessed. Setting the WAITD (found in BCR, sfr address 0x46A, no bit address available [1]) bit can prevent this behaviour. This bit disables the WAIT function completely.

In all other cases extra logic is needed described in the following sections.

2.2 EAn/WAIT pin design considerations.

Before the XA WAIT pin can be used, be aware of the following things:
The EAn/WAIT input pin has NO Schmitt-trigger, this means that the rising and falling edges of WAIT must be steep. If NOT, due to oscillation, the XA will generate unpredictable events, for example the XA can generate exceptions. This restriction rules out a wired logic solution, where both signals are connected via an open drain to the EAn/WAIT pin. The slopes of this type of signals is not steep

enough to be used with the XA EAn/WAIT pin.
A WAIT signal needs a holdtime and must be generated within a specified time, see Figure 1.

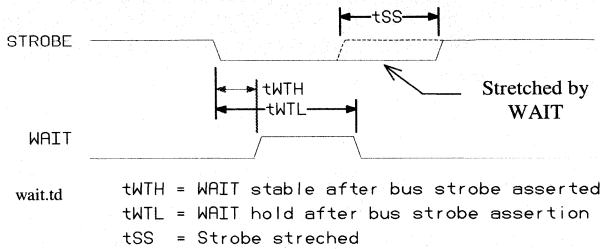


Figure 1, WAIT hold time (2) and WAIT asserted after Strobe asserted (1)

Because XA data strobe (code read and data read/writes) lengths are individually programmable, WAIT must be generated correspondingly [1]. It is impossible to generate WAIT states when the XA bus controller is programmed as 1 clock cycle data reads without ALE. For most XA data cycles WAIT needs to be generated at least 1 clock (plus 34ns) before the end of the strobe (decision point). For code reads and wait cycles with strobes of one clock in length, WAIT may be generated as late as 34ns (does not depend on the crystal frequency) before the end of the strobe (no extra clock).

It is allowed to generate a wait strobe before a data strobe is asserted. The following situation however should be avoided:

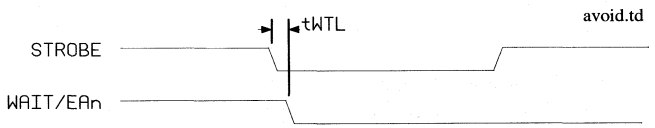


Figure 2, Too short WAIT hold time

Because of the combined WAIT and EAn "feature" this situation can occur when an application initialises (see 2.3), as a result the XA can lock up.

2.3 Generating the EAn signal.

The XA starts sampling the EAn pin at the rising edge of RESETn, however a hold time is needed (tMMH, see Figure 3). This hold time must be at least three clock cycles long. Therefore the EAn

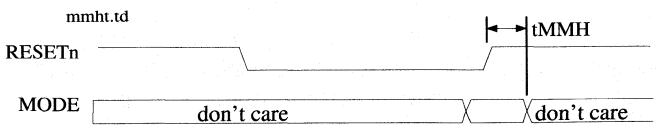


Figure 3, Memory mode hold time

generating logic needs some delay. At 20MHz the hold time is 150ns, so just simply gating this signal with RESETn will not do the job.

Absolute condition for proper operation of all EAn networks is that a "clean" RESETn is supplied to the XA. The XA RESETn input is Schmitt-triggered, so a normal RC network can be used to generate the appropriate reset signal. However, if a RESETn signal is constructed via an RC network and additional logic, it is important to use Schmitt-triggered logic. If not, the additional logic can oscillate at the rising edge of RESETn. The XA does not tolerate this oscillation on the RESETn pin, and can lock-up or initialises in the wrong memory mode.

2.3.1 Generating EAn using an RC network.

The easiest way of generating an appropriate EAn signal is using a simple RC network (Figure 4). The RC loop is triggered at RESETn, holding the EAn signal for a certain period of time determined by the RC time. It is very important to use Schmitt-triggered logic due to the slow slopes.

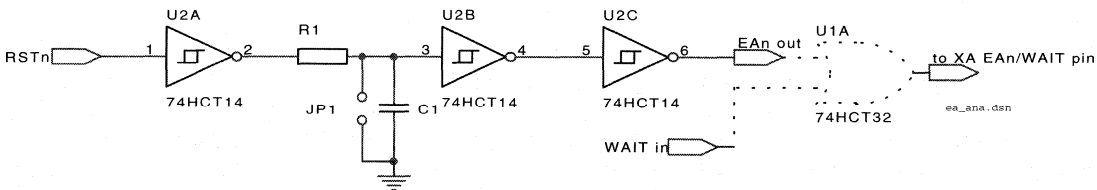


Figure 4, Generating EAn

The RC circuit can be calculated as follows:

$$Uth = Uo \cdot e^{-\frac{t}{RC}} \Rightarrow C = -\frac{t}{R \cdot \ln\left(\frac{Uth}{Uo}\right)}$$

U_{th} = HCT14 Input threshold voltage [3].
U_o = HCT14 output voltage (5V) [3].

When R_{in} >> R (e.g. 1k) and XA running @ 20MHz (t_c = 50ns), it will take a minimum of 7 clock cycles before the first ALE is generated after RESET:

$$C = -\frac{350ns}{10^3 \cdot \ln\left(\frac{0.9}{5}\right)} \Rightarrow C \approx 200pF$$

The minimum of 7 clocks applies when the XA is running external only (EAn = 0). Using the EAn circuit will force the XA to boot internally (EAn = 1). Thus it will take longer before the first external cycle is generated and consequently the value of the capacitor may be higher. Again, as stated in section 2.1, the EAn circuit is not needed if absolutely NO external cycle will be generated (i.e. the XA is executing internal only).

The jumper in Figure 4 has been added to enable the user to choose between the two memory modes. In case WAIT states are needed, the circuit can be combined with a WAIT state generator via the OR gate. If there is no need for generating wait cycles, [EAn out] can be connected directly to the XA EAn/WAIT pin (see Table 1).
The main advantage of this circuit is that it is very simple and only one type of logic is used (74HCT14 [3]). A disadvantage however is that this circuit can never be implemented in programmable logic because a PLD is normally not equipped with Schmitt-triggered inputs [5].

Therefore extra logic is needed something that is prevented in the first place when using programmable logic.
In the next sections however alternative (C)PLD compatible solutions are presented.

2.3.2 Generating EAn using XA reset behaviour

During RESET all XA control, address and data pins are pulled high. Figure 5 shows that after reset it takes some time before ALE will become low for the first time. This behaviour can be used to

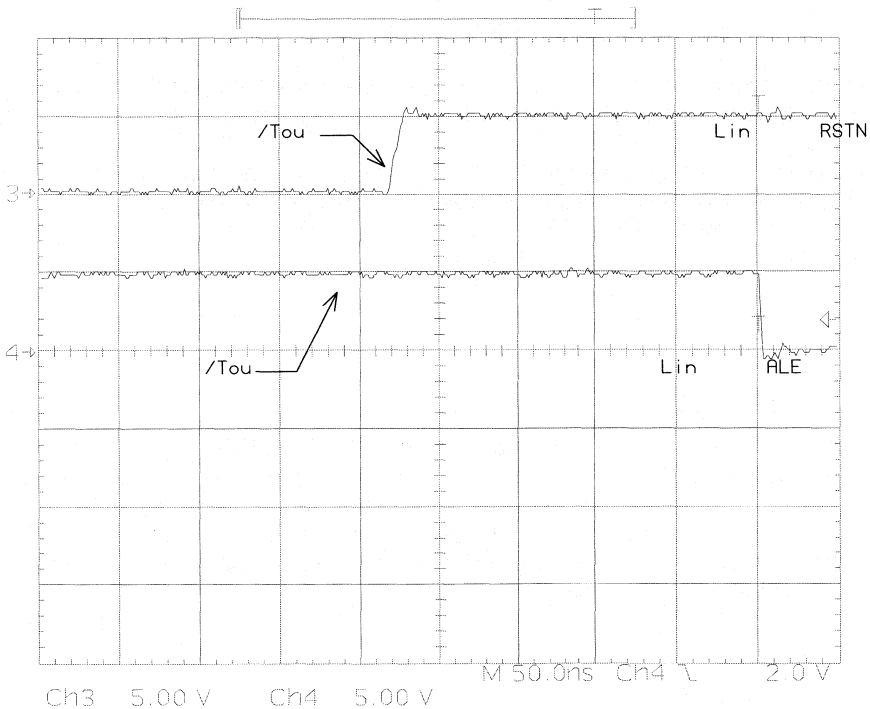


Figure 5, RESETn and XA control line relationship, 3 = RSTn / 4 = ALE

determine whether or not the XA is in reset state or not. This behaviour is based on the fact that an external cycle will be performed, if not the XA runs fully internal and the EAn circuit is not needed. Consequently EAn/WAIT can be connected to Vdd because WAIT only applies for the external bus (see 2.1).
The ALE strobe can be used in combination with for example the PSENn strobe. Besides during reset, ALE and PSENn are both high at the same time during an ALE cycle (i.e. address is available on the address/data bus). Consequently during this period also WAIT will be high. It is however allowed to generate a WAIT strobe outside a data strobe, but in this situation NO wait cycles will be inserted.
To improve EMC behaviour it is desired to suppress these spurious WAIT strobes. This can be achieved by ANDing ALE and PSENn with a couple address lines, e.g. A19 and A18. Of course if the

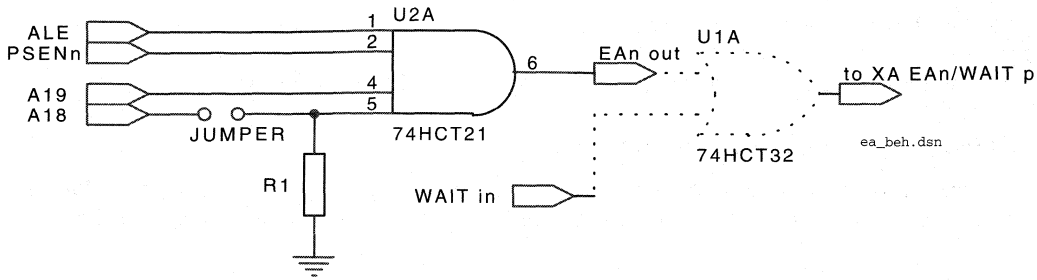


Figure 6, EAn generator using XA RESET state

same address is used to decode chip selects, again this spurious wait will be generated (but now less frequent). Figure 6 shows an implementation, more addresses can be decoded if a larger AND gate is used.
If the Jumper is applied the XA will run on-chip (EAn high during reset), and consequently will cause the I/O pins to be configured as quasi bi-directional. A weak internal pull-up will make the I/O pins high. Therefore R1 in Figure 6 must have a relatively high value, so R1 > 10k.
A CPLD description of this circuit is relatively easy [4]:

```
WAIT = ALE & PSEN & A19 & ... & Ax + EAnIN;
```

2.3.3 Generating EAn using a D flip-flop

The final and most secure option is using a D flip-flop with asynchronous PRESET. During reset (RSTn = 0) the D flip-flop is PRESET, the flip-flop is reset on the rising edge of the first ALE (data = 0 clocked in), ALE is used as clock. The flip-flop will stay set as long as no external cycle is performed. This means if the flip-flop has never been reset, no external cycle has been performed (the XA runs fully internal), consequently the EAn circuit is not needed. In this case EAn/WAIT can be connected to Vdd because WAIT only applies for the external bus (see 2.1).

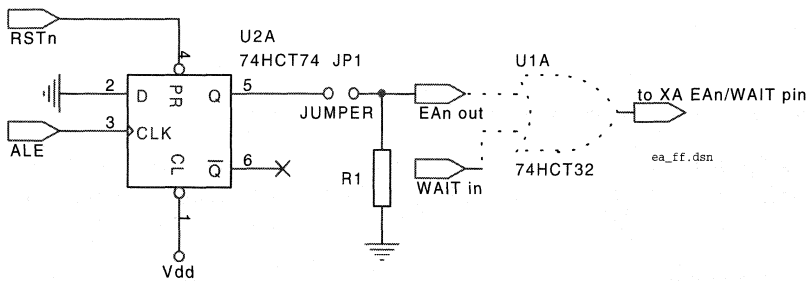


Figure 7, Generating EAn using D-flip-flop

Jumper J1 is used to determine between internal/mixed or external memory mode. When jumper J1 is not applied, resistor R1 will pull down the XA EAn/WAIT pin or, if combined with a wait state generator, one of the OR gate (U1A) inputs. This means that if jumper J1 is applied the XA EAn/WAIT pin will be “0” at RESET, i.e. full external.

Jumper J1 and resistor R1 may both be removed if only on-chip memory mode is needed and no selection between internal/external is required.
As described previously, the above described circuit can be discarded if the XA must run external only. If NO wait states are needed, the EAn pin can be connected directly to GND without extra logic. However if wait states are needed, EAn/WAIT can be connected to the WAIT state generator without extra logic.

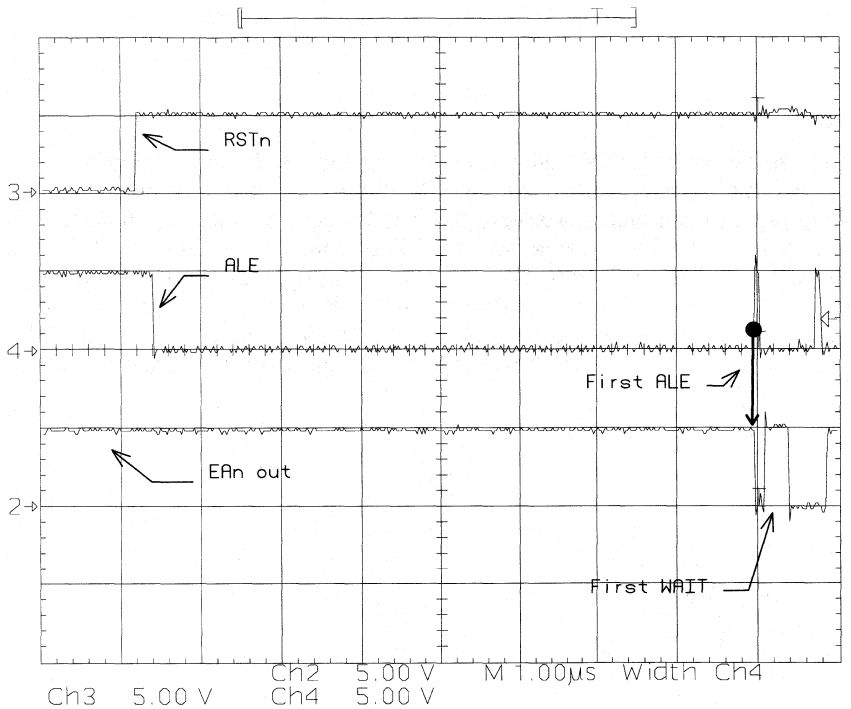


Figure 8, Generating EAn using RSTn and ALE

It is NOT important to buffer the RSTn signal with a Schmitt-trigger because once the flip-flop has been set, it will stay set, up to the rising edge of ALE.
Figure 8 shows how EAn sets when RSTn is low and resets at the ALE rising edge. In this example it takes more than 7 µs before the first ALE strobe (= first external access, XA is running internal because EAn is high) is generated (at 20MHz). The reset falling edge is not displayed, because the reset strobe has a length of several milli seconds. Due to the resolution of the digital storage scope displaying both edges of reset will prevent the sampling of the first ALE and will consequently not be displayed.

Many designs use programmable logic to connect a microcontroller to peripherals and generate chip selects. The schematic shown in Figure 7 can also be implemented in a PLD. The following XPLA

designer equations show it is very easy to implement the above described D flip-flop solution in a (C)PLD:

```
ALE_CLK     pin 4;              /* use clock pin, see PZ5032 datasheet */
ea          node istype 'reg';

equations
ea.pr       = !RSTN;
ea.clk      = ALE_CLK;
ea.d        = 0;
```

2.3.4 EAn/WAIT lock up

As mentioned in section 2.2 some EAn/WAIT situations can lock-up the XA. In this section an example will show how an EAn/WAIT signal can lock-up the XA. Figure 9 shows a problem causing design and should therefore not be used with any XA project. The transparent latch (U2A & U2B) is SET during RESETn and is reset at the first external access (datastrobes WRHn, WRLn, PSENn or RDn). The problem is that EAn/WAIT is still high when the first data strobe is generated. This situation is similar as displayed in Figure 2 (section 2.2); the WAIT hold time is too short and can lock-up the XA. It depends on the propagation delay of the circuit how long it takes before EAn/WAIT is negated after the first data strobe is generated. When using HCT logic the propagation delay for the NAND gate (Figure 9) is 9ns [3].

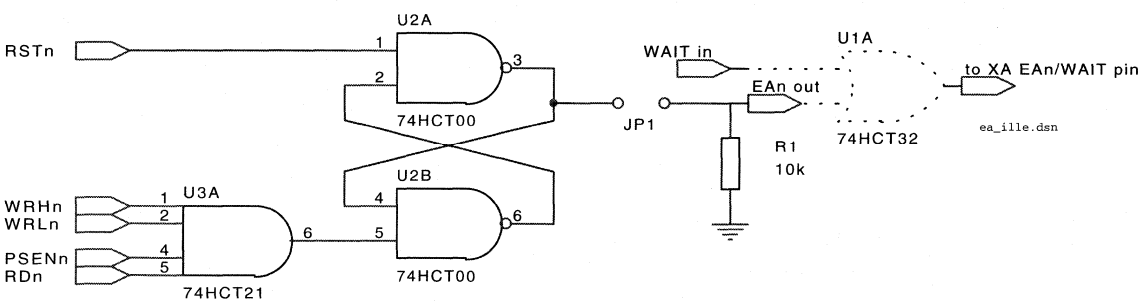


Figure 9, problem generating EAn solution

2.4 Generating WAIT

2.4.1 The XA WAIT pin

With the WAIT pin it is possible to stretch an external XA bus cycle. However the WAIT pin has no influence on code running from on-chip memory. When WAIT is asserted wait states will be inserted as soon as the XA accesses the external bus (that is asserting RDn, PSENn, WRLn and/or WRHn). The WAIT pin is therefore a part of the bus controller and not the core. When wait states are inserted the bus controller will halt the core up to the point WAIT will be negated again. In the mean time also NO internal activity takes place, because WAIT will only be granted in sequence. If not, the following erroneously situation can occur; The XA is executing code internally, at certain point data is needed from the external databus, e.g. to make a go no-go decision. If during this external access a wait signal is generated the external bus will be stretched, thus postponing the data fetch. Wrong decisions can be made if the XA continues to execute the internal code without waiting for the necessary external data.

It is possible to overrule the WAIT pin by setting bit WAITD in the BCR (sfr address 0x46A [1]) register.

The XA WAIT pin is not bi-directional and therefore the XA cannot halt external peripherals.

Although the XA WAIT pin differs from a 68000 DATCKn, it is possible to use a 68000's peripheral with DTACKn output in combination with an XA, please refer to application note AN96098 [7].

2.4.2 Illegal WAIT configuration

The XA itself can never generate a WAIT strobe. When one ore more XA data strobes are connected via an inverter to the XA WAIT input (Figure 10), the XA will latch-up as soon as it tries to access

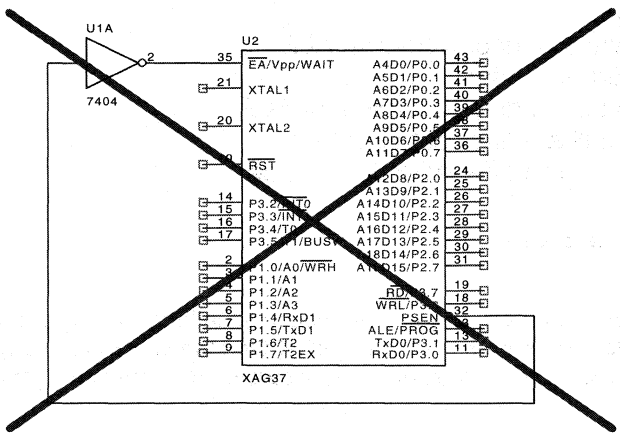


Figure 10, Illegal WAIT construction

external memory. This is caused because when a WAIT is asserted during a data strobe, the strobe will be stretched as long as this WAIT signal is asserted. This mechanism however can never negate

the WAIT signal again, because it is stretched by the same signal. This is the reason why an external WAIT state generator needs to be build.

2.4.3 Burst and non burst wait state generators

XA code reads (PSENn) can be performed during a so called burst mode. In this mode one of the 3 (4 for 8 bit data bus) non-multiplexed LSB address lines can change **without** asserting ALE and **without** negating PSENn (see Figure 11). This burst mode is transparent and cannot be controlled by the user and/or programmer.

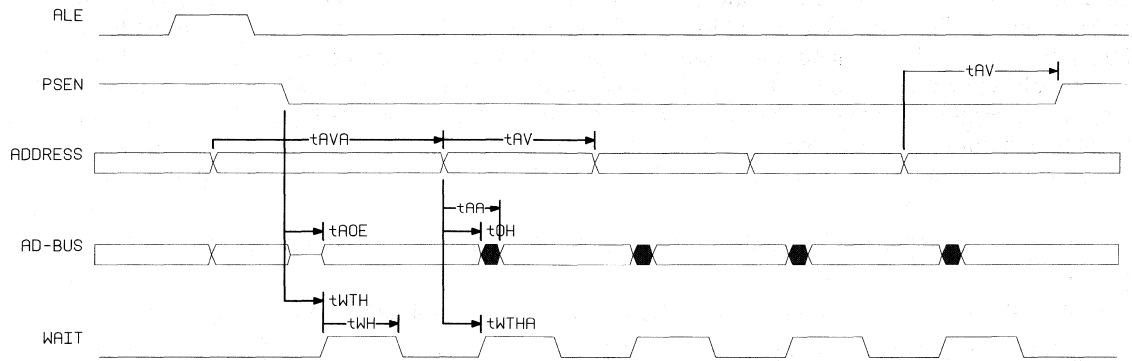


Figure 11, burst (code) read

Row	Name	Comment
1	tOH	Output hold from address change
2	tAOE	Output enable access time
3	tAV	Address valid during cycle without ALE
4	tWH	WAIT high
5	tAVA	Address valid during cycle with ALE
6	tAA	Address access time
7	tAV	Address valid during cycle without ALE
8	tWTH	WAIT generated after data strobe
9	tWTHA	WAIT generated after address change

Table 2, electrical characteristics

Non bust mode wait state generators can therefore only be used in situations where no burst is generated by the XA, i.e. 16 bit data write cycles and read cycles. Or if the connected peripheral is only accessed one byte or word at the time. In all other cases a burst mode wait state generator is needed.

2.4.4 Generating NON burst mode WAIT states

2.4.4.1 Generating non burst WAIT using one shot

Several XA wait state implementations are possible, the easiest solution is using a one shot generator (Figure 12). Looking at the schematic you can see that the WAIT signal is only generated

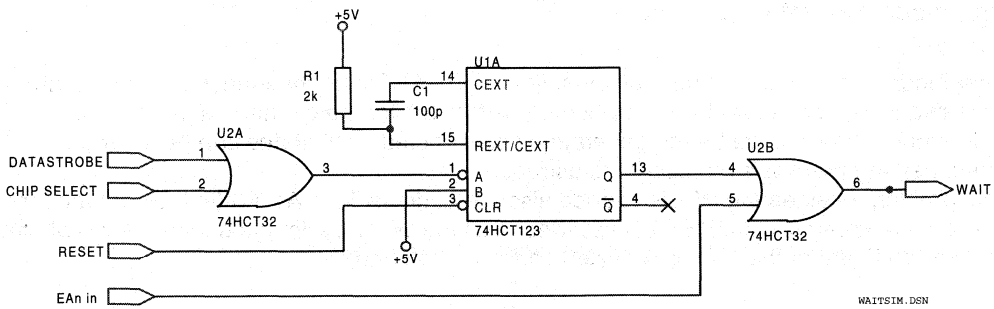


Figure 12: Wait state generator using one shot

AFTER a data strobe (i.e. RDN/WRL/WRH/PSEN) is asserted. The data strobe will also trigger the one shot generator, generating a pulse with a length determined by the RC time. By ORing the strobe with a CSn signal you can select which device needs WAIT states.
The WAIT duration is always a whole number of clock cycles, therefore the RC time is not very critical except at turn over stages. The number of wait states can be made selectable when R1 is replaced by an adjustable resistor.

2.4.4.2 Generating non burst WAIT using shift register

A shift register offers a more reliable way of generating wait states. Figure 13 shows *a* shift register implementation. This circuit only generates a WAIT signal during a datastrobe. The shift register

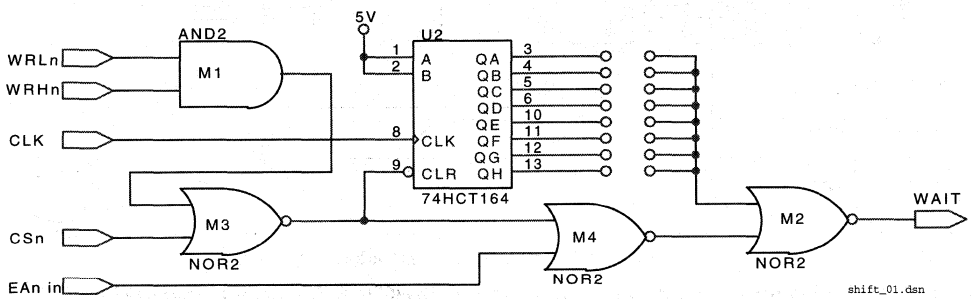


Figure 13: Wait state generator using shift register

master reset is released when one of the data strobes is asserted AND Chip Select is low. The chip select input has been included to make it possible to insert WAIT states for particular devices. The WAIT output will immediately become high and ones (A and B input are one) are shifted into the shift register after a data strobe is asserted.

Counting will continue until the selected (by for example a jumper) shift register output becomes high. Consequently the WAIT signal will be negated and the XA will continue by negating its strobe resulting in clearing the shift register.
To save logic, in contrast with all the other WAIT/EAn designs, in this design for "EAn in" a NOR is used instead of an OR.
The circuit above serves as an example, more solutions are of course available [6].

2.4.5 Burst mode wait state generation

The XA burst mode can cause several problems: firstly NO WAITs are inserted when one of the address lines changes, consequently the burst cycle is too fast for the connected peripheral. Secondly, depending how the wait state generator is constructed, wait states can be inserted endlessly because the hardware expects a datastrobe.
To solve this problem besides the PSENn strobe also the address lines A1-A3 (in 8 bit databus mode A0-A3) must be monitored. So if PSENn is asserted AND the WAIT cycle has ended, new wait states need to be inserted if one of the non multiplexed addresslines change

```
PSENn H->L                      => WAIT = 1
WAIT H->L & PSEN = L            => monitor A0-A3
If A0-A3 H<->L & PSENn = L      => WAIT = 1
```

2.4.5.1 Burst mode wait state generator using standard logic

Figure 14 shows the same shift register used in Figure 13. The magnitude comparator U1 (74HCT85) is the heart of this design. If no data strobe is generated (i.e. RDn = PSENn = WRLn = WRHn = 1)

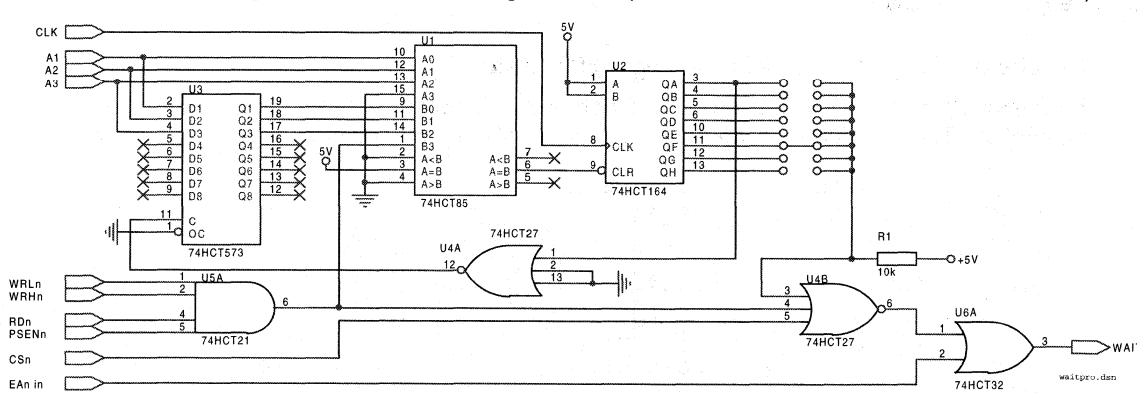


Figure 14, wait state generator with address monitor

the comparator's A=B output will be zero, because the 74HCT85 A3 input is low and the B3 input is high. During this state the shift register is a-synchronously cleared, consequently ALL 74HCT164 outputs (QA to QH) are low. When QA is zero the 74HCT573 latch enable input (C) is high and therefore transparent for address lines A1 to A3. If one or two (both the WRLn and WRHn can be asserted simultaneously) strobes are asserted, the 74HCT85's A=B output will become HIGH because now both A3 and B3 are low. The shift register will start to shift in ones and QA will be high after one clock cycle. When QA is high the 74HCT573 latch enable will be low (C=0) and the current

address is latched in. A wait signal will be generated as soon as **both** CSn and one or more XA data strobes are low up to the point the selected (by jumper) will be become high.
When during a PSENn strobe (no burst is generated in case of a 16bit write cycle or data read cycle = RDn) one or more addresses change (A1..A3), the comparator's A=B output will be low again and resets the shift register. Consequently QA will also be low and the latch (74HCT573) will be transparent again. Immediately the A=B output will become high and the shift register starts to shift. Because ALL shift register outputs are low after the 74HCT164 is cleared, a NEW wait cycle will be generated.

2.4.5.2 Burst mode wait state generator using CPLD

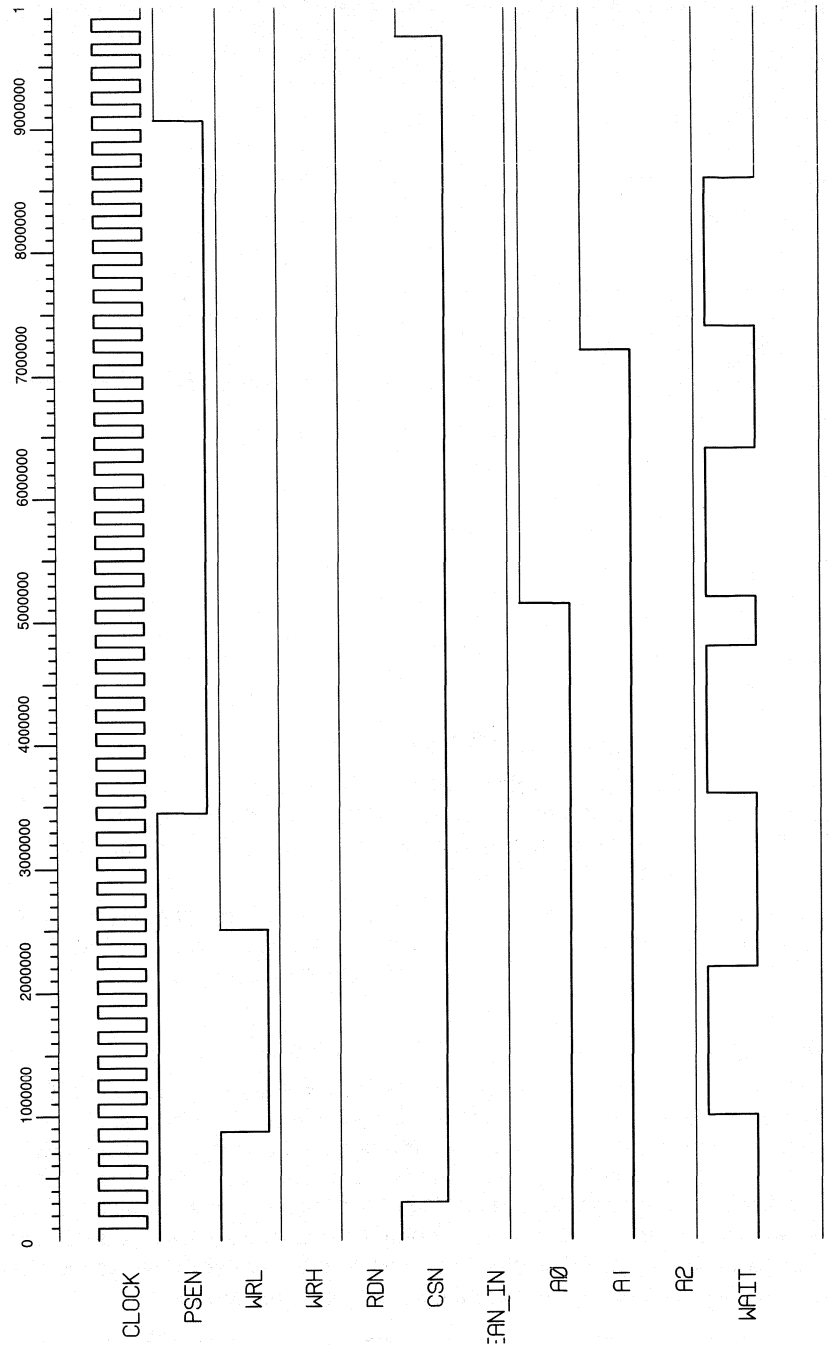
The circuit described in the previous section can also be constructed by using a Philips Semiconductors CPLD (e.g. PZ5032 [5]). The XPLA [4] module displayed below is based on Figure 14, except the shift register has been replaced by a counter. This results in less nodes, three in stead of eight.

```
Module  BURSTMODE_WAITSTATE
/*************************************************************/
/* this waitstate generator has a FIXED number of wait states */
/* Please change {number_of_wait} by required value           */
/*************************************************************/
Title   'Wait state generator for the XA'

CLOCK                   pin;                        /* clock signal from XA oscillator */
RST                     node;                       /* internal counter reset */
PSEN                    pin;                        /* program data strobe input */
WAIT                    pin istype 'reg_d';         /* WAIT signal output */
/* Alternative see section 2.4.6:  WAIT   pin; */
WRL                     pin;                        /* write low strobe input */
WRH                     pin;                        /* write high strobe input */
RDN                     pin;                        /* data read strobe input */
CSN                     pin;                        /* chip select input */
EAN_IN          pin;                                /* XA memory mode pin input */
wait_is_true    node;
/* Alternative see section 2.4.6:  wait_is_true    node istype 'reg_d'; */

bit3..bit0              node istype 'reg';          /* Up counter register bits */
count = [bit3..bit0];                               /* Up counter register */

a2..a0          pin;                                /* address sense pins */
compa = [a2..a0];                                   /* address sense register */

le                      node;                       /* latch enable, used internal only*/
latch3..latch1          node istype 'com';          /* latched in address */
latchinout = [latch3..latch1];                      /* address latch register */
/* if XA is used in 8 bit mode please add latch0 */
/*************************************************************/
equations

le = (count == 0);                                  /* latch is open when count is zero */
latchinout = (le & compa) # (latchinout & !le);     /* latch in current a3-a1*/

RST = (PSEN & RDN & WRL & WRH) # (compa != latchinout);
                                                                        /* do not
count when no strobe is asserted */
count.CLK = CLOCK;
count.AR = RST;
count.d = count.q + (count < 7);                    /* do not count more than 7 */
```

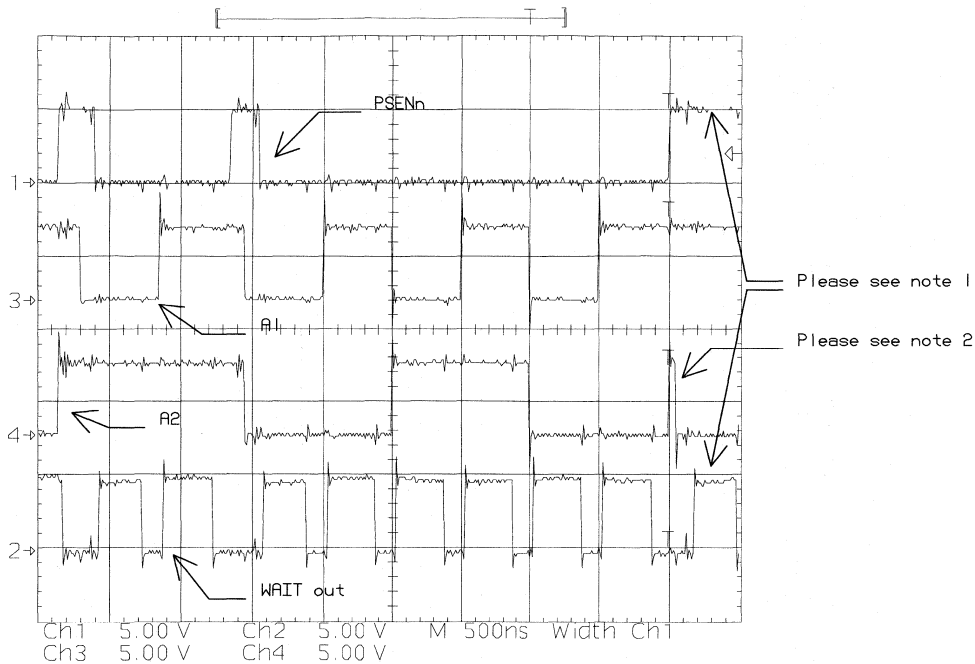
```
/* Alternative, Add this line. see section 2.4.6: wait_is_true.CLK = CLOCK; */
wait_is_true = (count < {number_of_wait}) & (!CSN);

WAIT.CLK = CLOCK;                          /* Alternative see section 2.4.6 : remove this line */
WAIT = ((!PSEN # !WRL # !RDN # !WRH) & wait_is_true) # EAN_IN;
end;
```

==========

On the next page a simulation can be found, the {number_of_wait} is 6.


0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000
9000000
1
CLOCK
PSEN
WRL
WRH
RDN
CSN
EAN_IN
A0
A1
A2
WAIT

The following oscilloscope picture shows that when one or more addresses change, and PSENn is low, a WAIT will be generated. In Figure 16 the following signals are displayed: 1 represents the data



strobe, 3 and 4 are A1 and A2, and 2 is WAIT out.

Note 1: This spike is an abandoned cycle. Abandoned cycles can occur when during a prefetch a JUMP, BRANCH or CALL is executed. If a cycle is abandoned the data/code strobe will be negated again. The WAIT strobe is negated immediately because the WAIT circuit does NOT generate a WAIT when none of the data strobes is asserted.

Note 2 : In this situation WAIT is generated while PSENn is not asserted. However the design also detects data reads (RDn) and writes (WRLn/WRHn), consequently this WAIT strobe is a result of a data stobe in stead of a PSENn strobe (In Figure 16 ONLY PSENn is displayed).

2.4.5.3 Combining one shot generator with burst detector

As an alternative for the burst mode wait state generator with shift register a burst mode wait generator with a one shot generator is displayed. In fact Figure 17 is a combination of Figure 12 and Figure 14 where the shift register is replaced by the one shot generator. This needs some minor changes in the burst mode detector because the one shot can not generate a short pulse to open the input latch. In this design the opening and closing of the latch is achieved by an OR gate which is connected to the comparator's A > B and A < B output. This way an A ≠ B signal is constructed, and is high when no strobe (PSENn, RDn, WRHn or WRLn) is generated because input A3 is low and input B3 is high. When A ≠ B is high the input latch will be transparent.

When a datastrobe is generated both inputs A3 and B3 are low and consequently A = B will become high (and of course A ≠ B low) resulting in closing the address input latch. The one shot generator will be triggered on the positive edge of A = B and thus generating a WAIT.

After WAIT is negated (determined by the one shot generator RC time), the XA will continue running. In a burst mode cycle the data strobe will not be negated, instead one or more of the addresses (A3:1) will alter. The address change will make A ≠ B high again and consequently the latch will be opened triggering the one shot generator. When the latch is transparent the new address will also be available on the comparator's B2:0 input and thus making A ≠ B low, resulting in closing the latch.

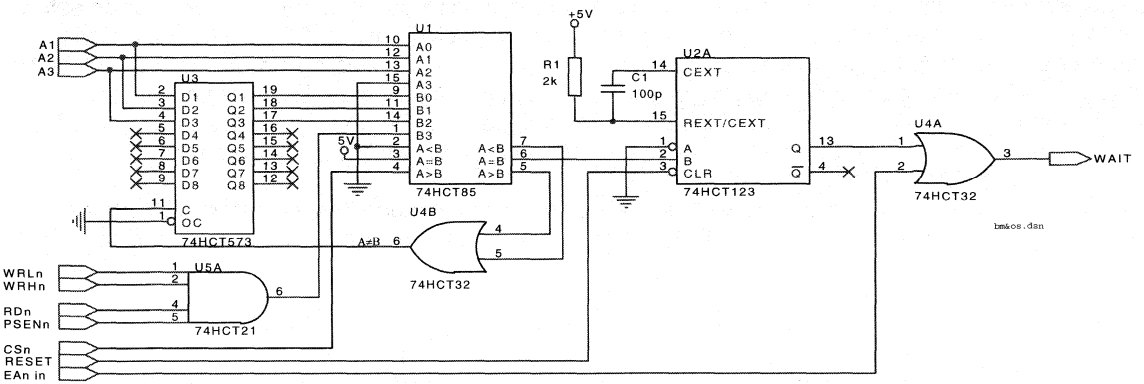


Figure 17, burst mode wait state generator with one shot

At the end of the cycle when the data strobe is negated, A ≠ B will be high and the latch transparent.

In this design the chip select input is connected to the comparator's A > B input. If chip select is not asserted (i.e. CSn = 1), the A = B output will be low and therefore NO wait cycles will be generated.

2.4.6 Known problems and solutions

The CPLD counter used in section 2.4.5.2 has one drawback, it can generate spikes. For example the equation *count < 7* generates a spike at the 4th clock. Going from 3 to 4 :

011(3) -> 111(spike 7) -> 100(4)

To prevent this problem WAIT has been constructed around a D flip-flop (see XPLA listing section 2.4.5.2 [4,5]) . It is however NOT guaranteed that this construction always will function. The XA and CPLD are both clocked on the same source, in some situations it can take up to a clock period before a WAIT is generated. An XA data strobe is generated at the rising edge of CLOCK (although this cannot be guaranteed), just the next rising edge will clock the connected logic resulting in WAIT being asserted. This delay can be a problem, for example when the XA write strobe length has been programmed as one clock cycle. In that case the wait signal must be generated at least at $t_x = 1 * t_c$ - 34ns. Where t_c is the length of the current strobe, this means that wait must be generated 34ns before the end of the strobe.
When the (CPLD [4,5]) logic generates a wait after one clock (minus 17ns, i.e. internal XA delay between rising edge input clock and falling edge strobe) it will be too late and WAIT will not be

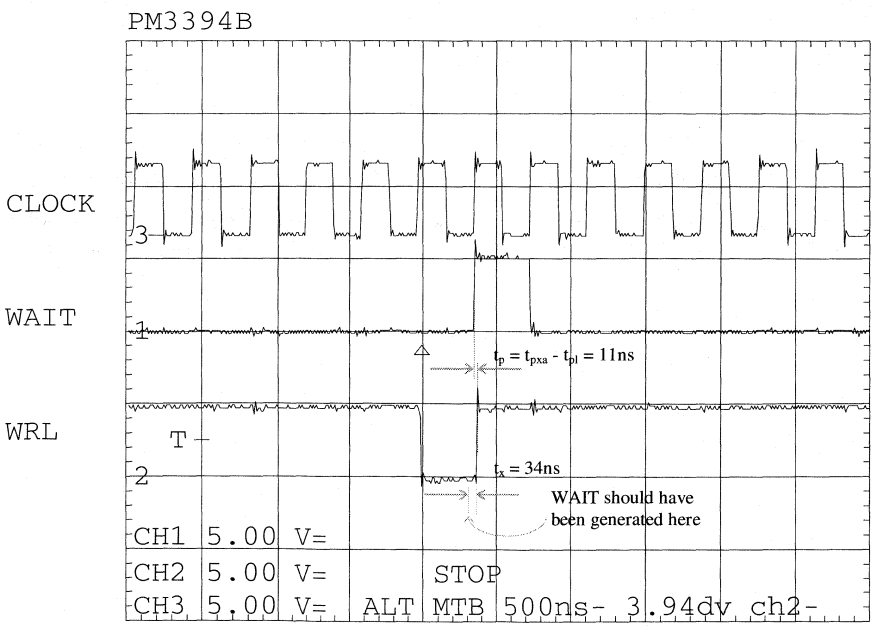


Figure 18, WAIT generated too late

sampled. See Figure 18 for an oscilloscope picture, this picture shows that the wait signal is 23ns late:
$t_l = 34 - (t_{pxa} - t_{pl}) \Rightarrow t_l = 23ns$ (t_l = time late, t_{pxa} propagation delay XA, t_{pl} propagation delay logic) This situation occurs when BTRL (Bus Timing Register Low) contains 0x6F [1].

Please note: although the number_of_wait in this design was 6, WAIT is only high for ONE clock period. This is caused because the counter is cleared in case NO strobe is asserted.

This problem can be solved in two ways, the first method is using !CLOCK in stead of CLOCK. Now WAIT will be asserted at the falling edge in stead of the rising edge, resulting in a delay of half a clock cycle plus logic (inverter) propagation delay.
Also this solution can provoke problems, caused by the internal XA delay between clock and the data strobe falling edge (t_{pxa}). This delay is constant (at least not depending on the clock frequency) and is around 17ns. The wait strobe must be generated after an XA data strobe is asserted, otherwise an extra clock cycle will be inserted before WAIT will be generated. So:

$0.5 * t_c = 17ns - 6ns \implies t_c = 22ns$

Resulting in a maximum frequency of 45MHz, the maximum operating frequency of the XA is currently 30MHz, therefore this solution can be used without restrictions. Please notice, the logic propagation delay can compensate t_{pxa}.

The second solution can be found in the CPLD equations. In stead of using clocked logic, combinatorial logic can be used instead to generate WAIT. The XPLA source need to be adapted

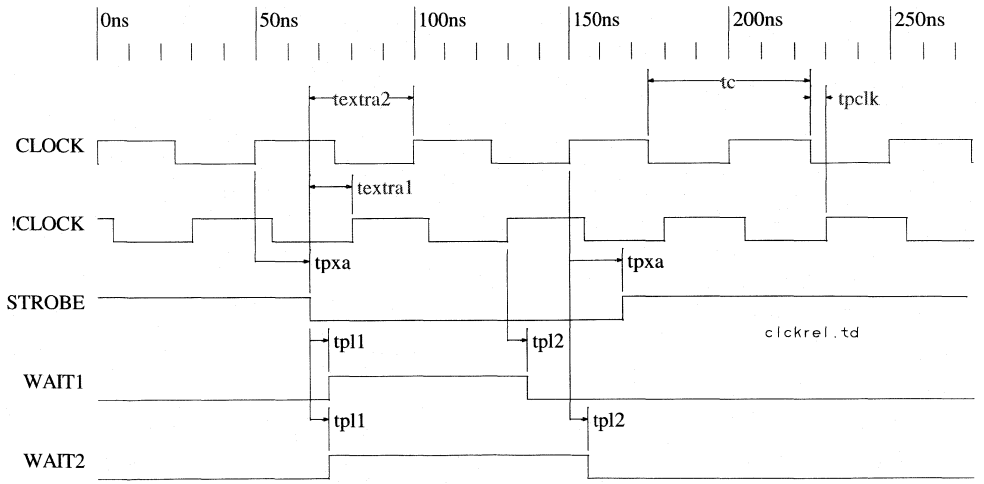


Figure 19, relation between strobe/wait/clock

(see comment in CPLD source, section 2.4.5.2) to realise this implementation.

In the new equations, the WAIT signal is generated immediate (minus propagation delay logic) after a data strobe is asserted. Because the relationship between the supplied clock and the XA core clock will never be 0ns, the length of the WAIT strobe is not a exact number of clocks. In stead it can be up to one clock longer than defined. In case of using CLOCK:

$$t_{extra2} = t_c - t_{pxa}$$

In case of using the inverted CLOCK:

$t_{extra1} = 0.5 * t_c - (t_{pxa} - t_{pclk})$

if $t_c < (t_{pxa} - t_{pclk})$ then:

t_{extra2} $1.5 * t_c - (t_{pxa} - t_{pclk})$

t_{extra} = time making wait strobe longer

t_c = clock period

t_{pxa} = time difference between rising edge clock and falling edge strobe

t_{pclk} = time difference between CLOCK and !CLOCK

t_{pl1} & t_{pl2} = logic propagation delay

Because t_{pl1} and t_{pl2} are both propagation delays caused by the same logic, that cancels both signals in the wait strobe length formula.

t_{pxa} and t_{pclk} are both measured and serve as an indication, the exact value can vary (influenced by temperature and supply voltage).

The next chapter (3) describes how to supply a clock source.

3 CLOCK SOURCE

Some designs in the previous sections need a CLOCK. Unfortunately the XAG3 (future derivatives *can* have a CLOCK out) does not have a CLOCK output, so an absolute core CLOCK reference is not available. Of course the XA's XTAL2 output can be used, but this signal and the core clock are not in phase.

Several ways are available to offer a CLOCK source to external devices. The most common way is using a normal crystal and supplying CLOCK by XTAL2 (see Figure 20). Please be aware that extracting the CLOCK signal from this pin will raise the capacitive load (input capacitance CPLD is 8pF) on the XTAL2 pin, this can be corrected by lowering the capacitor (C2) normally found on the XTAL2 pin.

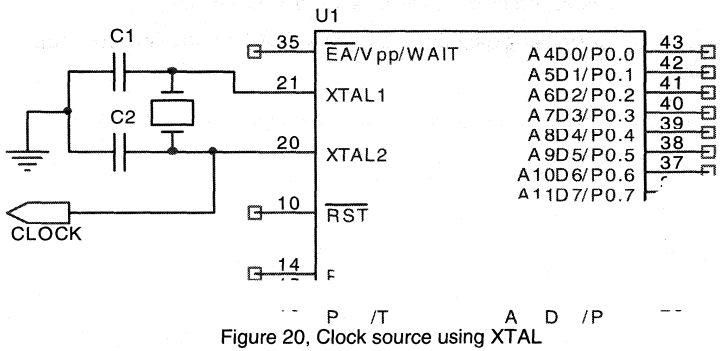


Figure 20, Clock source using XTAL

A better clock is supplied when using an XO (crystal oscillator) device. Advantage of this type of oscillator is the square wave output. An external CLOCK can be supplied in two ways, connecting the XO output to both the XA's XTAL1 input and CPLD or other logic (Figure 21).

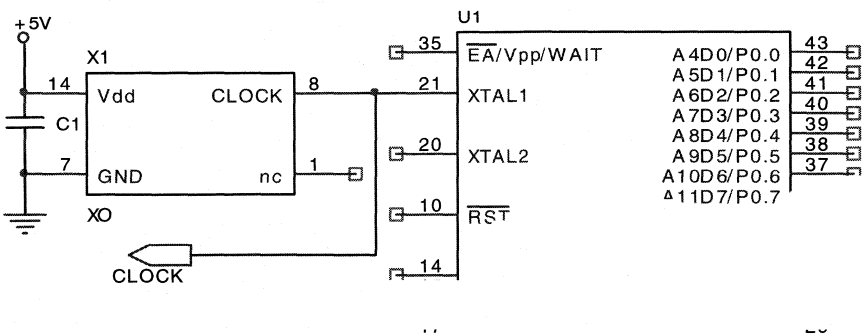


Figure 21, Clock source using XO

The other way is less conventional, the XO output is connected to the XA XTAL1 input, the XA XTAL2 output, normally dangling when using an external clock source, is connected to the CPLD or other logic. For both configurations, crystal or XO, XTAL2 is used to supply CLOCK, so in designs where crystals can be exchanged with XO's and vice-versa no jumpers are needed. Secondly the XTAL2 output is inverted, so the XA itself can provide a !CLOCK source (see section 2.4.6). XTAL2 can be used where a !CLOCK signal is needed, and therefore logic can be saved.

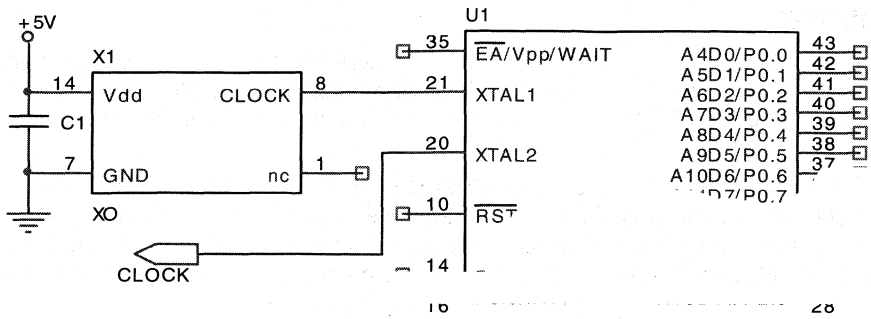


Figure 22, Clock source using XO and XTAL2 output

The CLOCK lines should be as short as possible to improve EMC behaviour and supplying a clean CLOCK to the peripherals.

References

[1] 16-bit 80C51XA Microcontrollers -
Data Handbook IC25
PHILIPS: 9397 750 00733

[2] 80C51 based 8-bit Microcontrollers
Data Handbook IC20
PHILIPS: 9397 750 00013

[3] High speed CMOS Logic Family
Data Handbook IC06
PHILIPS: 9397 750 00454

[4] XPLA Designer version 2.1 User's Manual

[5] DATA SHEET PZ5032 (CPLD)

[6] Electronic Circuits, design and applications
U. Tietsche, Ch. Schenk
Springer-verlag
ISBN 3-540-50608-X

[7] Application note AN96098
Interfacing 68000 family peripherals to the XA
M. Kuystermans, PS-SLE 1996.

Interfacing 68000 family peripherals to the XA AN96098

Author: Marco Kuystermans, Systems Laboratory Eindhoven, The Netherlands

ABSTRACT

The XA is not limited to interface to 80XX compatible peripherals. By using some glue logic, or a PLD if available, the XA can be interfaced to a 68000 compatible peripheral. Using the 68000 DTACKn signal enables the use of slow 68000 peripherals without the need of an external WAIT state generator.

SUMMARY

Many peripherals with a 68000 compatible interface are on the market. This document shows that these 68000 peripherals can be interfaced to the XA very easily. This means that an XA user is not limited to an XA compatible peripheral in his XA design.

Several interfacing aspects are covered: normal interfacing, interrupt interfacing, and bus arbitration.

This document assumes that users are familiar with the XA and its bus interface.

CONTENTS

1. HOW TO INTERFACE 68000 FAMILY PERIPHERALS TO THE XA

1.1 introduction

The Philips Semiconductors XA is a 16 bit high speed microcontroller with an 80XX compatible bus (found on e.g. an 8051 or 8048 with separate read and write strobes). This means that popular periherals with an 80XX compatible bus can be used with the XA.

This documents shows that besides 80XX peripherals also peripherals with a 68000 bus can be used. Extra advantage is that the 68000's DTACKn output is extremely suitable to generate an XA WAIT strobe. The main advantage of using this DTACKn output is that a full handshake is available (synchronous) and consequently NO external wait state generator is needed.

1.1.1 General remarks:

- Because the XA wait mechanisme is rather complex, this document is NOT intended for the starting XA user, instead some fundemental knowledge about the XA is needed.
- In this documed is assumed that the nessecary address latches are provided (2 x 74HCT573), i.e. a demultiplexed bus is used:

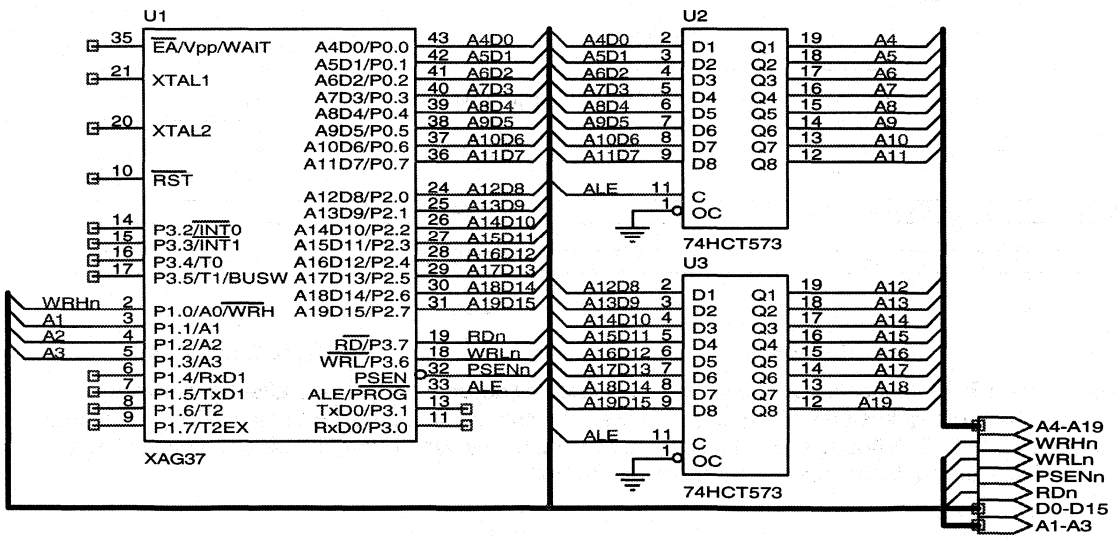


Figure 1, demultiplexed XA bus

1.2 Differences between an 80XX and 68000 bus

A 68000 bus compatible peripheral has a significantly different interface than a 80XX bus type peripheral. 68000 and 80XX bus interfaces consists of several different control signals (see Table 1):

Table 1, Microcontroller control signals

	68k compatible peripheral	*80XX compatible peripheral*	*XA compatible peripheral*
Data strobes	2 + direction: R/Wn *(UDSn + LDSn)	2: WRn + RDn	3: WRLn + WRHn + RDn
Chip select	CSn	CSn	CSn
Hand shake	DTACKn	Not available	~WAIT
Code strobe	Not available	PSENn	PSENn

Both 68000 and 80XX peripherals use data strobes to read or write data. An 80XX peripheral has separate read and write strobes (see Figure 3), a 68000 peripheral uses a data strobe (only indicating a data transfer, not the direction, see Figure 2) AND a R/Wn signal to indicate a READ (R/Wn = 1) or a WRITE (R/Wn = 0) cycle. At both the 68000 and 80XX data strobes' rising edges data will (write) or must (read) be valid.

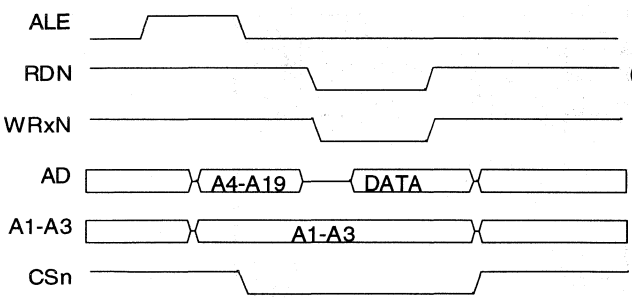


Figure 3, 80XX bus

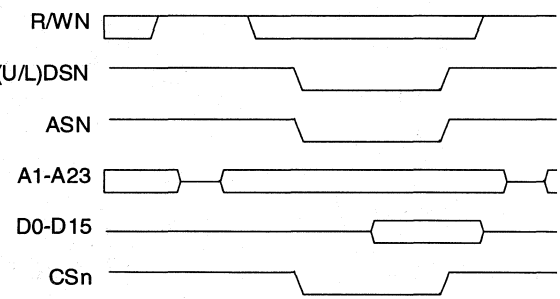


Figure 2, 68000 bus

Normally an 80XX chipselect (not) signal is generated by decoding addresses after the address latches. To avoid spikes on this signal (addresses can change during ALE = 1) the CSn signal should be gated with ALE. A 68000 chipselect can be constructed by decoding addresses and gating it with the Address Strobe (ASn) signal generated by the 68000.

Some 8 bit wide 68000 compatible peripherals do not have separate Data Strobe and Chip Select inputs. On these peripherals the Data strobe and Chip Select are multiplexed to one signal. In this document a multiplexed DSn (data strobe) and CSn (chip select strobe). will be called CSn.

Real 16 bit wide 68000 peripherals have two Data strobes; UDSn (upper data strobe) and LDSn (lower data strobe) and consequently need a Chip Select (not) input.

Both the XA and 68000 are 16 bit microcontrollers. Therefore address line 0 has no meaning and is on the 68000 decoded to UDSn (A0 = 0, little endian!) and LDSn (A0 = 1). On the XA A0 is decoded to WRLn (A0 = 0, big endian!) and WRHn (A0 = 1). On the XA no separate read low and high are available. All read cycles (except if the XA bus is configured as 8bit) are 16 bit, if only 8bits are needed the upper or lower 8 bits are discarded.

The following pictures show how to convert XA strobes to 8 bit or 16 bit 68000 peripherals.

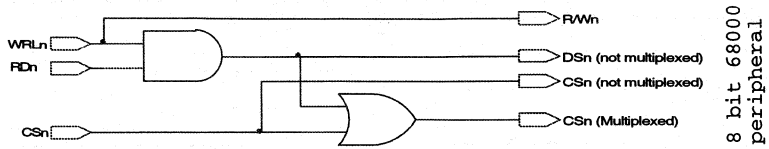


Figure 4, converting XA to 68000 strobes (8 bit)

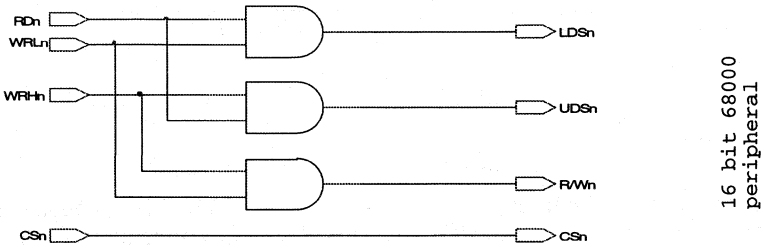


Figure 5, converting XA to 68000 strobes (16 bit)

Please be aware that a 68000 peripheral has no separate data and program memory area, instead a lineair memory space is available with mixed memory. It is up to the user to decide in which XA memory area the 68000 peripheral is located.

Because on the XA it is not possible to write to program memory, writing is always performed via the XA data write strobe (WRHn and/or WRLn). Writing is achieved through the XA's data and extra segment (DS & ES, see XA user's guide [1]). Reads can be executed in both data (RDn strobe via ES & DS) and program memory (PSENn via Program counter or Code segment).

In Figure 4 and Figure 5 RDn can be replaced by PSENn. If PSENn is used to read data from a 68000 peripheral, consequently MOVC must be used.

1.3 Using DTACKn to Generate an XA compatible WAIT signal

1.3.1 Generating 68000 bus signals

Figure 6 shows the signals from both the XA and a 68000 peripheral combined. First of all a 68000 compatible data strobe needs to be constructed composed from XA signals. The easiest way to accomplish this is to combine the RDn or WRLn/WRHn strobes with a chip select (decoded addresses, or the most simple solution only one address line). Figure 6 shows the generated 68000 CSn strobe (or LDSn in case of real 16 bit wide 68000 peripherals because in this example WRLn is used). t_2 in Figure 6 indicates the propagation delay of the decoding circuit.

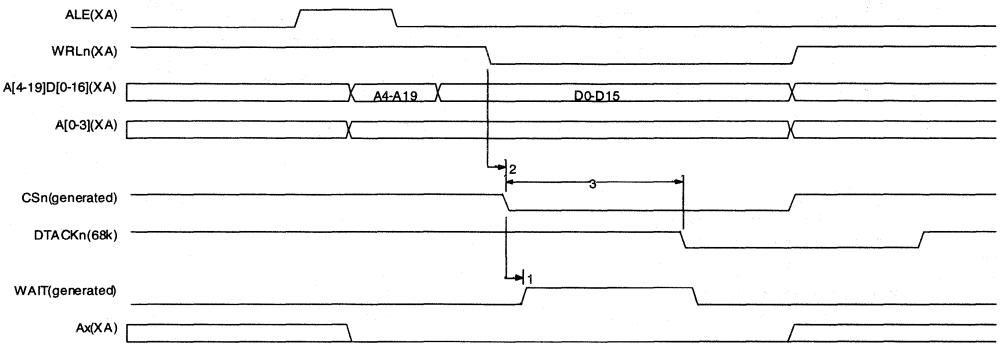


Figure 6, 68000 and XA signals combined

The XA's WRLn/WRHn signal can be used to generated the 68000's R/Wn signal. In case of a 16 bit wide 68000 peripheral BOTH WRLn and WRHn need to be monitored to generate the R/Wn signal. 68000 peripherals with an 8 bit wide data bus can be connected to either D8:15 (using only WRHn) or to D0:7 (using only WRLn).

Please note that some 68000 peripherals do not allow t_2 (R/Wn set-up to CS low) to be 0ns, for example the PCF8584 needs t_2 to be 10ns or higher. You need to use an address line to generate a R/Wn signal (see figure 1, Ax(XA)), if the decoding logic propagation delay is shorter than the required time t_2. A drawback to this solution is reading and writing is not possible on the same address.

1.3.2 68000 DTACKn signal.

A 68000 peripheral generates a DTACKn to provide a real handshake between the microcontroller and its peripheral. DTACKn indicates when the peripheral is ready to receive data (in case of a write cycle), or when the microcontroller can expect valid data on the databus, placed on the bus by a peripheral. This DTACKn can be used to generate an XA compatible WAIT signal

1.3.3 Generating the XA WAIT signal.

An XA WAIT signal must be asserted immediately after (minimal 34ns before end of strobe, i.e. the rising edge) a read (PSENn or RDn) or write (WRLn or WRHn) strobe is asserted. The XA will ONLY insert waitstates after the XA databus is stable up to the point the WAIT signal is de-asserted. A WAIT

signal has no effect if it is asserted outside a data strobe. It is however allowed to generate a WAIT signal before a data strobe is asserted, but will only be active after this strobe is asserted.

After a CSn (LDSn) strobe has been generated the device's DTACKn signal is high until the device is ready to receive. During this high period the XA must generate wait cycles, see t_3 in Figure 6. So the equation for the WAIT signal (t_3) is: /CSn * DTACKn.

There are several ways to construct a XA WAIT signal from a DTACKn signal, discrete or via a PLA. The following schematic (Figure 7) shows *a* discrete implementation for 8 bit wide peripherals.

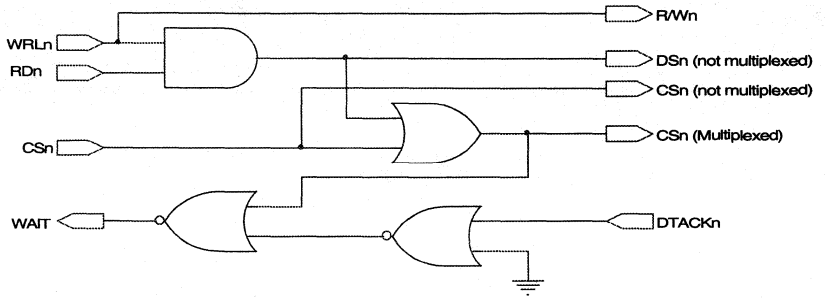


Figure 7, discrete solution (8 bit wide 68000 peripheral)

or if LDSn and UDSn are needed (16 bit peripheral):

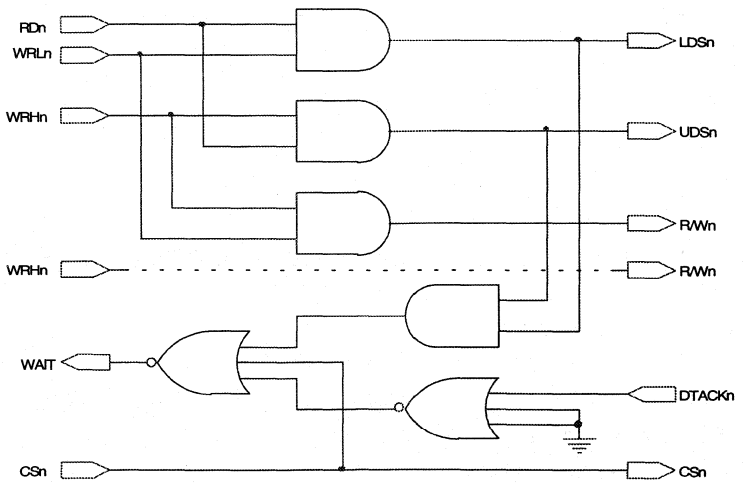


Figure 8, Discrete solution 16 bit peripheral

Table 2, Figure 8 WAIT truth table

U/LDSn	CSn	DTACKn	WAIT
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Table for both UDSn and LDSn

Table 3, Figure 8 data strobe thruth table

WRLn	WRHn	RDn	LDSn	UDSn	R/Wn
0	0	0	x	x	x
0	0	1	0	0	0
0	1	0	x	x	x
0	1	1	0	1	0
1	0	0	x	x	x
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	1	1

x = NOT VALID

You can also use a PLA if one is present (see Figure 9):

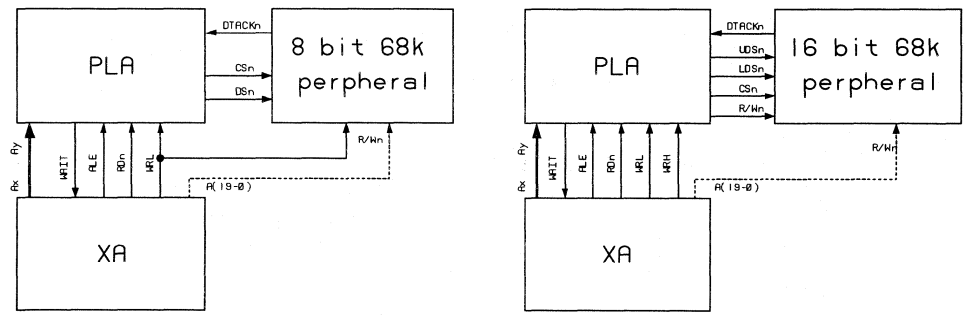


Figure 9, PLA plus XA plus 68000

The PLA must have the following equations (multiplexed DSn and CSn, e.g. SC68C562):

(1a) CSn = /((Ax *..* Ay) * /(WRLn * RDn) * /ALE)

(1b) WAIT = ... + /CSn * DTACKn

(1c) R/Wn = WRLn (R/Wn directly connected to WRLn)

The dots represent other funtions to drive the WAIT pin (see general remarks 1.3.4)

Non multiplexed 8 bit 68k peripheral (e.g. ST 68HC901):

(2a) CSn = /((Ax *..* Ay) * /ALE)

(2b) DSn = /(WRLn * RDn)

(2c) WAIT = ... + /CSn * /DSn * DTACKn

(2d) R/Wn = WRLn (R/Wn directly connected to WRLn)

In case of 16 bit wide 68000 peripherals, where separate UDSn and LDSn strobes are needed (e.g. SCC66470 video and system controller);

(3a) LDSn = /(WRLn * RDn)

(3b) UDSn = /(WRHn * RDn)

(3c) R/Wn = WRHn * WRLn

(3d) CSn = /((Ax *..* Ay) * /ALE)

(3e) WAIT = ... + /(LDSn * UDSn) * DTACKn * /CSn

The (generated) R/Wn, LDSn and UDSn signals are available for ALL 68000 peripherals. The DTACKn signal is generated by a 68000 peripheral. All 68000 DTACKn pins are open drain outputs and connected together as wired OR. The only device specific signal is CSn and therefore ALL CSn signals need to be monitored if a corresponding WAIT signal needs to be generated for that particular device:

(4a) CS1n = /((Ax *..* Ay) * /ALE)

(4b) CS2n = /((Ax *..* Ay) * /ALE)

(4c) WAIT = ... + /(LDSn * UDSn) * DTACKn * (/CS1n + /CS2n)

1.3.4 General remarks:

- The XA WAIT pin is combined with the EAn (external access) function. This pin is sampled at the rising edge of RESET, therefore in functions 1b, 2e and 3c an extra EAn term needs to be added. An example, however beyond the scope of this document, can be: EAn = ALE * WRLn * RDN * EAmode. During RESET ALL XA pins are high, this effect is used to generate EAn or not (depending on EAmode, e.g. a jumper). Of course WAIT strobes will be generated during ALE = 1, but as stated in this paragraph WAIT strobes outside a data strobe will not put the XA in WAIT mode.
- ALL chip select lines are decoded address lines ANDed with /ALE, this prevents generating spikes on the CSn lines while addresses are not stable (ALE = 1).
- Please be sure the WAIT pin is not overridden by the WAIT disable bit.

1.4 68000 Interrupt mechanism

When using a 68000 peripheral, consequently the 68000 interrupt mechanism is used. This means that the XA has to generate an IACKn to this peripheral. It is not allowed to assert the CSn and IACKn strobes at the same time, the IACKn is (so to say) also a Chip Select.

A way of generating a IACKn is decoding an interrupt vector address and combining it with a read strobe (more or less an alternative Chip Select, compare with 1a), e.g. with a PLA:

(5) IACKn = /((Ax * .. * Ay) * /RDn * /ALE)

Ax to Ay must be an other address now than the address in the previous paragraph, so CSn address for normal data accesses IACKn address

So to generate an IACKn (example via Extra Segment = ES):

```
(6a)    MOV.b     ES, #xxh          xx = (addresses A19 - A16)
(6b)    MOV.b     R2, #yyyyh        yyyy = (addresses A15 - A0)
(6c)    OR.b      SSEL, #04h        aligns ES to R2
(6d)    MOV.b     R3, [R2]          R3 holds peripheral`s vector
(6e)    JMP       [R3]
```

Generating an IACKn can also be achieved via a code read if the full code range is not used:

(7) IACKn = /((Ax * .. * Ay) * /PSENn * /ALE)

Using PSENn to generate IACKn enables the use of the same address as the CSn address (with RDn), it is now a code space address. An interrupt acknowledge can have the following construction (example via Code Segment = CS):

```
(8a)    MOV.b     CS, #xxh          xx = (addresses A19 - A16)
(8b)    MOV.b     R2, #yyyyh        yyyy = (addresses A15 - A0)
(8c)    OR.b      SSEL, #04h        aligns CS to R2
(8d)    MOVC.b    R3, [R2]          R3 holds peripheral`s vector
(8e)    JMP       [R3]
```

Notes:

- If XA code is running internal exclusively (within the on board EPROM) and only one peripheral needs an IACKn, NO address decoding is needed and PSENn can be connected to IACKn directly.
- Please be sure xx:yyyy in formula 8a and 8b is above the XA internal memory range, in case of the P51XAG37 xx:yyyy > 0x00:7FFF

1.5 68000 bus arbitration with the XA

Missing on the XA is a bus arbitration scheme. If the XA had an ONCE mode, i.e. a pin that forces all pins to float, it would have been very easy to implement. It is however possible to construct 68000 bus arbitration with discrete components. This XA bus arbitration scheme will also utilise the XA WAIT pin. During a bus arbitration cycle a WAIT signal will halt the XA.

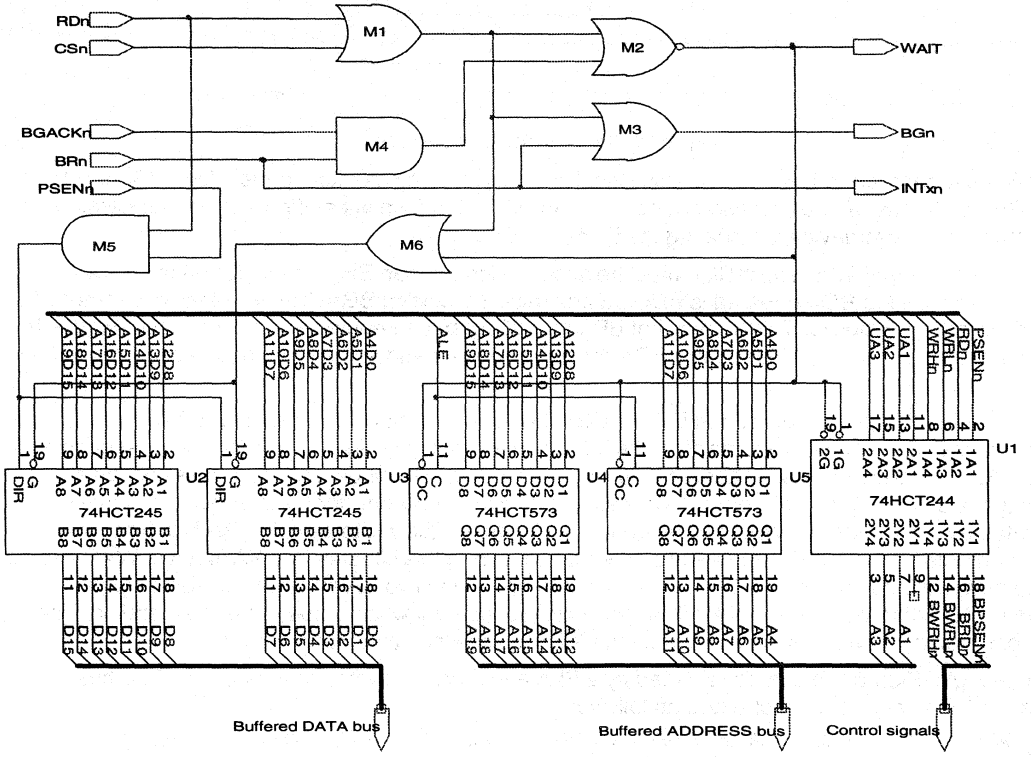


Figure 10, Bus arbitration on XA

A device that requests the bus asserts the BRn (bus request) strobe. The master device must return a BGn (Bus Grand) strobe when the master is ready. After the slave has received the BGn signal it starts the bus arbitration cycle by asserting BGACKn. During this strobe the slave device must have complete access to the (shared) memory, therefore the XA is on hold and its signals must float. As stated NO ONCE pin is available therefore the XA memory bus must be buffered.

Due to the XA multiplexed address/databus structure latches are needed to demultiplex. The latches have an Output Enable (OEn) input, connecting this input to WAIT will float the address bus when the XA is in WAIT.

The only thing missing is the non-floating databus, the not multiplexed address lines A3 to A1 and the control signals RDn, WRLn, WRHn and PSENn. The databus can be made floating by using a bi-

directional buffer. Writing or reading is indicated by the direction (DIR) signal. If DIR=1 a write is indicated else a read. A DIR signal is constructed by ANDing both write strobes (WRLn and WRHn).

A3 to A1 and the control signals only need an output buffer (with Output Enable control). Figure 10 shows a solution with discrete components.

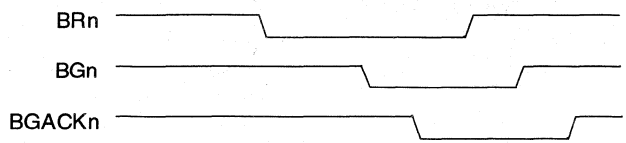


Figure 11, Bus arbitration timing diagram

The XA must generate a BGn signal. This signal is constructed by monitoring the RDn, WRLn, WRHn and PSENn strobes. If one of these strobes is asserted AND a BRn is pending a BGn is generated up to the point the slave device is negating the BRn.

Figure 11 shows that in principle BGn must be asserted longer than BRn. In the 68000 specification BGACKn asserted to BGn negated is min. 1.5 and max. 3.5 clocks. BGACKn asserted to BRn negated min. 20ns max. 1.5 clocks. This shows that BRn and BGn can be negated at the same time. Figure 10 shows that in fact BGn is negated after BRn has been negated and some propagation delay should be considered.

The WAIT signal is generated during BRn or BGACKn asserted, but only if one of the XA control strobes is asserted AND the XA reads at the bus "request enable address". This bus request enable address is a decoded dedicated address. Just using the RDn without decoding address can cause the following problem: If during a "normal" XA read a Bus Request occurs (for the bus request signal is generated aynchronously), WAIT can be generated to late for the XA to sample. The bus however will float when both BRn and RDn strobe are generated. If during this situation the XA is not in WAIT mode, unpredictable things can occur. BRn is connected to one of the two XA external interrupts inputs (INT0n or INT1n) to be sure the XA will access external memory when a bus request is pending. The bus arbitration cycle will only continue after the XA has accessed external memory.

The glue logic (Figure 10) can be replaced by a PLA and combined with other functions. The bus arbitration WAIT signal is constructed as follows:

```
(9)   WAIT = .. + /(RDn * WRLn * WRHn * PSEN) * (Ax * .. * Ay) * /(BRn * BGACKn)
```

The Bus Grand (BGn) is assembled as follows (output to slave);

```
(10)  BGn = /(( RDn * WRLn * WRHn * PSEN) * (Ax * .. * Ay) + BRn)
```

The following instructions must be part of the interrupt routine to generate a BGn and put the XA in wait:

```
(11a)   MOV.b    ES, #xxh          xx = (addresses A19 - A16)
(11b)   MOV.b    R2, #yyyyh        yyyy = (addresses A15 - A0)
(11c)   OR.b     SSEL, #04h        aligns ES to R2
(11d)   MOV.b    R3, [R2]          read to generate BGn
(11e)   RETI                       return from interrupt
```

The XA interrupt latency determines the time it takes for the XA to generate BGn after BRn asserted.

2. EXAMPLE, INTERFACING THE PCF8584 TO THE XA

Interfacing the PCF8584 (See IC12 [2]) to microcontrollers can cause serious problems. Causes for these problems are: The Interface Mode Control, the interface the PCF8584 will run in is determined by the first write cycle to this peripheral. Secondly it is a relatively (to the XA) SLOW device. In contrast this example shows that it is in fact quite easy to interface the PCF8584 to the XA.

2.1.1 PCF8584 to XA implementation

In "80C51" mode the PCF8584 needs write and read strobes longer than 250nS, the XA can generate read strobes with a maximum length of 4 times the oscillator clock period, a XA write strobe is even shorter, its length is max. 2 clock periods. This means when the XA is running at 8MHz the write strobe will match the PCF8584 write strobe specification. Running the XA on higher frequencies than 8MHz will cause the need of an external wait state generator.

It is however possible to use the PCF8584 WITHOUT the use of an external wait state generator on every XA clock (up to the maximum XA clock). Using the PCF8584 in 68000 mode will do the trick. Using the PCF8584 in 68000 mode:

1. R/Wn asserted before CSn (t_2) needs to be 10ns or higher. This time is needed to enable the PCF8584 to decode a 80C51 or 68k bus mode (it is by the way the only 68k peripheral that does **not** allow t_2 to be 0ns). If the decoding logic is too fast an address is needed as R/Wn strobe instead of WRLn or WRHn.
2. The first PCF8584 cycle needs to be a write cycle, this cycle determines the bus mode the PCF8584. In contrast of getting the PCF8584 in 80C51 mode, it is allowed to have write or read cycles to other peripherals first. Having a read cycle from the XA first will put the PCF8584 in 80C51 mode, thus not asserting the DTACKn signal (it is now the RDn input) and therefore causing the XA to be in a endless WAIT.
3. When an address line is used to generate a R/Wn signal, reading and writing is not performed on the same address.
4. Excessive accesses to the PCF8584 will slow down the entire application, so using the PCF8584 in time critical applications in polling mode should be avoided.

The following picture shows an actual hardcopy of all signals used to connect the XA to a PCF8584. Please note that the DTACKn rising edge has the shape of an exponential function:

$$U = \overline{U} \bullet \left(1 - e^{-\frac{t}{RC}}\right)$$

this is caused by the open drain (wired OR) structure of this pin (pull up resistor R and parasitic capacitance C).

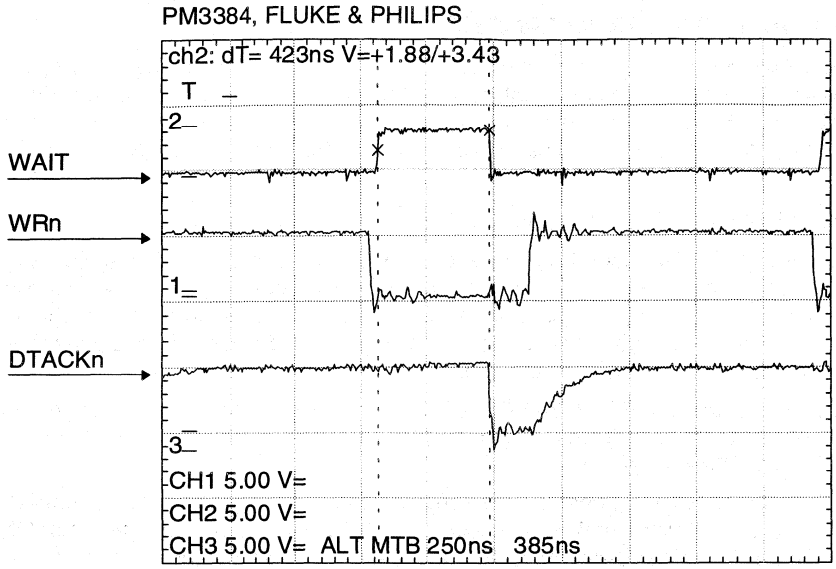


Figure 12, Scope hardcopy

2.1.2 Usage.

Please be sure the first cycle to the PCF8584 is a write cycle.

If using an address line as R/Wn strobe, writing to the PCF8584 must be performed on an other address than reading from the PCF8584 (e.g. if A15 is used):

Reading PCF8584 address 0:

```
(12a)   MOV.b    ES, #0Fh
(12b)   MOV.b    R2, #8000h        address line A15 = 1
(12c)   OR.b     SSEL, #04h        aligns ES to R2
(12d)   MOV.b    R3L, [R2]         Result is stored in R3L
```

Writing to PCF8584 address 0:

```
(13a)   MOV.b    ES, #0Fh
(13b)   MOV.b    R2, #0000h        address line A15 = 0
(13c)   OR.b     SSEL, #04h        aligns ES to R2
(13d)   MOV.b    [R2], #data8      data8 written to PCF8584
```

APPENDIX 1 REFERENCES

[2]	16-bit 80C51XA Microcontrollers -	Data Handbook IC25	PHILIPS: 9397 750 00733
[3]	I2C Peripherals	Data Handbook IC12	PHILIPS: 9397 750 00306
[4]	80C51 based 8-bit Microcontrollers	Data Handbook IC20	PHILIPS: 9397 750 00013
[5]	The 68000 microprocessor	Michalel A. Miller	ISBN: 0675 205220 02

I²C with the XA-G3 AN96119

Author: Paul Seerden, Systems Laboratory Eindhoven, The Netherlands

ABSTRACT

This report describes how to implement I²C functionality (single master), if you're using the Philips XA-Gx microcontroller. Elaborated driver routines (written in C) are given for two alternative solutions:

– Software emulation using two port pins ('bit-banging').

– Using the PCx8584 I²C-bus controller.

SUMMARY

This application note demonstrates the implementation of I²C functionality using the 16-bit XA-G3 microcontroller from Philips Semiconductors.

The note contains two main parts:

– An implementation using the Philips PCx8584 I²C-bus controller (Interrupt driven).

– An implemenation by software emulation of the bus using 2 I/O port pins (polling, 'bit-banging').

Not only the driver software is given. This note also contains a set of (example) interface routines and a small demo application program. All together, it offers the user a quick start in writing a complete I²C system application (single master).

CONTENTS

1. INTRODUCTION

This report describes I²C driver software, in C, for the XA microcontroller. This driver software is the interface between application software and the (hardware) I²C device(s). These devices conform to the serial bus interface protocol specification as described in the I²C reference manual.

The I²C bus consists of two wires carrying information between the devices connected to the bus. Each device has its own adddress. It can act as a master or as a slave during a data transfer. A master is the device that initiates the data transfer and generates the clock signals needed for the transfer. At that time, any addressed device is considered a slave. The I²C bnus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. However, the driver software given in this application note only supports (single) master transfers.

Chapter 2 gives a functional description of the driver program.

Chapter 3 describes the software structure and all driver interface functions ('callable' by the application).

Chapter 4 describes the low level hardware dependent driver software and is split into one general part and two sections. The first section describes a software emulated I²C bus driver ('bit-banging') using two I/O port pins. The other section describes an interrupt driven PCF8584 driver. The PCF8584 is a Philips integrated circuit to be used as a separate I²C bus controller.

Chapter 5 is a short description of the example application program (demo.c and i2cintfc.c).

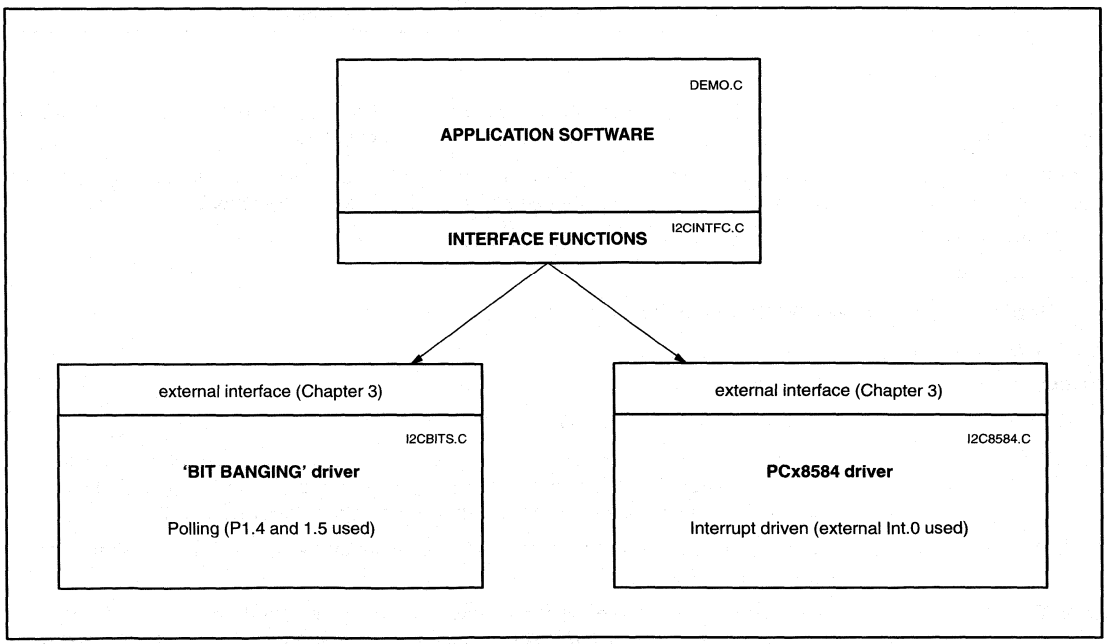


Figure 1. Overview of software layers and modules

1.1 References

Description	Ordering info
Used references:	
The I²C-bus specification	9398 358 10011
The I²C-bus and how to use it	9398 393 40011
Application report PCF8584 I²C-bus controller MAR 93	see BBS / WWW
Specification I²C driver (J. Reitsma)	
C routines for the PCx8584	AN95068
I2CBITS.ASM (by G. Goodhue)	see BBS / WWW
Used development and test tools:	
Hi-Tech C XA complier (version 7.60)	http://www.htsoft.com
Philips Microcore SiXA evaluation board	
FDI XTEND board	XTEND-G3
Philips I²C-bus evaluation board	OM1016
Philips Logic Analyzer with I²C-bus support package PF8681	PM3580/PM3585

1.2 BBS and WWW

This application note (with C source files) is available for downloading from the Philips Bulletin Board Systems and from the world wide web. It is packed in the self extracting PC DOS file: I2CXAG3.EXE.

To better serve our customers, Philips maintains a microcontroller bulletin board. This system is open to all callers, it operates 24 hours a day, and can be accessed with modems up to 28800 bps. The telephone number is:

European Bulletin Board, telephone number: +31 40 272 1102.

Internet access:

Philips Semiconductors WWW: http://www.semiconductors.philips.com

1.3 File overview

the driver package contains the following files:

I2CBITS.C The driver part for 'bit-banging" I²C.

I2C8584.C The PCF8584 drive for master transfers, containing initialization and state handling. This module also contains address definitions of hardware registers of the PCx8584. The user should adapt these definitions to his own system environment (address map).

I2CDRIVR.H This module (include file) contains definitions of local data types and constants, and is used only by the driver package.

I2CINTFC.C This module contains **example** application interface functions to perform a master transfer. In this module, some often-used message protocols are implemented. Furthermore, it shows examples of error handlin, like: time-outs (software loops), retries and error messages. The user must adapt these funtions to his own system needs and environment.

I2CEXPRT.H This module (include file) contains definitions of all 'global' contants, function prototypes, data tyes and structures needed by the user (application). Include this file in the user application source files.

DEMO.C This program uses the driver package to implement a simple application on the Microcore 6 demo / evaluation board. This board contains a PCx8584 I²C-bus controller, a PCF8583 real time clock and a PCF8574 I/O exapander with connections to 4 LEDs. The program runs the LEDs every second.

All driver software programs are tested as thoroughly as time permitted; however, Philips cannot guarantee that they are flawless in all applications.

2. FUNCTIONAL DESCRIPTION

2.1 The I²C bus format

An I²C transfer is initiated with the generation of a start condition. This condition will set the bus busy. After that, a message is transferred that consists of an address and a number of data bytes. This I²C message may be followed either by a stop condition or a repeated start condition. A stop condition will release the bus mastership. A repeated start offers the possibility to send/receive more than one message to/from the same or different devices, while retaining bus mastership. Stop and (repeated) start conditions can only be generated in master mode.

Data and addresses are transferred in eight bit bytes, starting with the most significant bit. During the 9th clock pulse, follwing the data byte, the receiver must send an acknowledge bit to the transmitter. The clock speed is normally 100kHz. Clock pulses may be stretched (for timing causes) by the slave.

A start condition is always followed by a 7-bit slave address and a R/W direction bit.

General format and explanation of an I²C message:

S	SLV_W	**A**	SUB	**A**	**S**	SLV_R	**A**	*D1*	**A**	*D2*	**A**	.....	**A**	*Dn*	**N**	**P**

S : (re)Start condition

A : Acknowledge on last byte

N : No Acknowledge on last byte

P : Stop condition

SLV_W : Slave address and Write bit

SLV_R : Slave address and Read bit

SUB : Sub-address

D1 ... Dn : Block of data bytes

D1.1 ... D1.m : First block of data bytes

Dn.1 ... Dn.m : n_{th} block of data bytes

2.2 Input definition

Inputs (applicaiton's view) to the driver are:

- The number of messages to exchange (transfer)
- The slave address of the I²C device for each message
- The data direction (read/write) for all messages
- The number of bytes in each message
- In case of a write message: The data bytes to be written to the slave.

2.3 Output definition

Outputs (application's view) from the driver are:

- Status information (success or error code)
- Number of messages actually transferred (not the requested number of messages in case of an error)
- For each read message: The data bytes read from the slave.

2.4 Performance

The default maximum speed of the I²C-bus is 100 KHz. With the XA-Gx running at 16 MHz or higher, it's possible to reach this speed using the 'bit-banging' driver. However, it is important to minimize the delay time between successive data bytes, because this delay determines the effective speed of the bus.

The maximum speed of the PCF8584 is limited to 90 KHz.

Software emulation ('bit-banging') of the bus is a heavy load for the XA processor. That's why systems that have to do more time critical tasks better apply the interrupt driven PCF8584 solution.

2.5 Error handling

A transfer 'status' is passed every time the 'transfer ready' function is called by the driver. It's up to the user to handle time outs, retries or all kind of other possible errors. Simple examples of these (no operating system, and no hardware timers) are shown in the file I2CINTFC.C

2.6 Hardware requirements

Bit-bang driver:

The bus requires open-drain device outputs to drive the bus. In fact, all port pins of the XA-Gx are programmable to open-drain outputs. In our example, external memory is connected to port P0 and P2. So, we have chosen to use P1.4 as SDA pin and P1.5 as SCL pin. To change this (for example to P1.6 and P1.7, like at the C51 derivative), adjust the include file I2CDRIVR.H. The code size of the emulation driver in this application note is approximately 800 bytes (Hi-Tech compiler V7.60). The driver is tested and tuned for an XA-G3 running at 20 MHz.

PCx8584 driver:

Selection of either an Intel or Motorola bus interface is achieved by detection of the first WR – CS signal sequence (see data sheet). This driver assumes that previously the right interface is selected (after power-up). The driver uses external interrupt 0 input. To change the base-address (0xF0000) of the PCF8584 edit the file I2C8584.C. In our example, a 3.6864 MHz clock is connected to the PCF8584.

3. EXTERNAL (APPLICATION) INTERFACE

This chapter describes the external interface of the driver towards the application. The C-coded external interface definitions are in the include file I2CEXPRT.H.

The applicaiton's view on the I²C bus is quite simple: The applicaiton can send messages to an I²C device. Also, the applicaiton must be able to exchange a group of messages, optionally addressed to different devices, without losing bus mastership. Retaining the bus is needed to guarantee atomic operations.

3.1 External data interface

All parameters affected by an I²C master transfer are logically grouped within two data structures. The user fills these structures and then calls the interface function to perform a transfer. The data structures are listed below.

```
typedef struct
{
      BYTE            nrMessages;         /* total number of messages                          */
      I2C_MESSAGE     **p_message;        /* ptr to array of ptrs to message parameter blocks  */

} I2C_TRANSFER;
```

The structure I2C_TRANSFER contains the common parameters for an I²C transfer. The driver keeps a local copy of these parameters and leaves the contents of the structure unchanged. So, in many applications the structure only needs to be filled once.

After finishing the actual transfer, a 'transfer ready' function is called. The driver status and the number of messages done, are passed to this function.

The structure contains a pointer (p_message) to an array with pointers to the structure I2C_MESSAGE:

```
typedef struct
{
      BYTE            address;            /* The I2C slave device address                      */
      BYTE            nrBytes;            /* number of bytes to read or write                  */
      BYTE            *buf;               /* pointer to data array                             */
} I2C_MESSAGE;
```

The direction of the transfer (read or write) is determined by the lowest bit of the slave address;

write = 0 and read = 1. This bit must be (re)set by the application.

The array ***buf*** must contain data supplied by the application in case of a write transfer. The user should notice that checking to ensure that the buffer pointed to by ***buf*** is at least nrBytes in length, cannot be done by the driver.

In case of a read transfer, the array is filled by the driver. If you want to use ***buf*** as a string, a terminating NULL should be added at the end. It is the user's responsibility to ensure that the buffer, pointed to by ***buf***, is large enough to receive ***nrBytes*** bytes.

3.2 External function interfaces

This seciton gives a description of the only two 'callable' interface funcitons in the both I²C driver modules.

First, the initialization function (*I2C–Initialize*) is explained. This function directly programs the I²C interface hardware and is part of the low level driver software. It must be called only once after 'reset', but before any transfer function is executed. After that, the interface function used to actually perform a transfer (*I2C_Transfer*) is explained.

void I2C_Initialize(BYTE speed)

Initialize the I²C-bus driver part. Must be called once after RESET.

'Bit Bang': Port pins P1.3 (SCL) and P1.4 (SDA) are programmed to be used as open-drain output pins.
BYTE **speed** Dummy parameter. Not used.

PCx8584: Hardware I²C registers of the PCx8584 interface will be programmed.
Used constants (parameters) are defined in the file I2CDRIVR.H.
BYTE **speed** Contents for clock register S2 (bit rate of I²C-bus).

void I2C_Transfer(I2C_TRANSFER *p, void (*proc)(BYTE status, BYTE msgsDone))

Start a synchronous I²C transfer. When the transfer is completed, with or without an error, call the function ***proc***, passing the transfer status and the number of messages successfully transferred.

I2C_TRANSFER ***p**	A pointer to the structure describing the I²C messages to be transferred.		
void (***proc**(status, msgsDone))	A pointer to the function to be called when the transfer is completed.		
BYTE **msgsDone**	Number of message successfully transferred.		
BYTE **status**	one of:	I2C_OK	Transfer ended No Errors
		I2C_BUSY	I²C busy, so wait
		I2C_ERR	General error
		I2C_NO_DATA	err: No data message block
		I2C_NACK_ON_DATA	err: No ack on data in block
		I2C_NACK_ON_ADDRESS	err: No ack of slave
		I2C_TIME_OUT	err: Time out occurred

4. DRIVER OPERATION

After completing a transfer the function *readyProc* in the application (or interface) is called.

After completing the transmission or reception of each byte (address or data), a state handler is called, either by interrupt (PCx8584) or by software ('bit-banging'). This handler can be in one of the following states:

ST_IDLE	The state handler does not expect any bus activity.
ST_AWAIT_ACK	The driver has sent the slave address and waits for an acknowledge.
ST_RECEIVING	The handler is receiving bytes, and there is still more than one expected.
ST_RECV_LAST	The handler is waiting for the last byte to receive.
ST_SENDING	The handler is busy sending bytes to a device.

Figure 2 shows the state transition diagram. A transition will occur on initiation of a transfer by the application and on each I²C-bus event (state change). The transitions are:

ST_IDLE → ST_SENDING	A transfer is initiated. Send the slave address for the first write message.
ST_IDLE → ST_AWAIT_ACK	A transfer is initiated. A message is to be received from a slave device. The micro transmits the slave address.
ST_SENDING → ST_SENDING	At least one byte to send. Send the next byte. Or no more bytes to send, send repeated start and slave address of next message to write.
ST_SENDING → ST_IDLE	No more bytes to send, no more messages.
ST_SENDING → ST_AWAIT_ACK	No more bytes to send, send repeated start and slave address of next message is to be received.
ST_AWAIT_ACK → ST_RECEIVING	More than 1 byte is to be received. Wait for and acknowledge next byte.
ST_AWAIT_ACK → ST_RECV_LAST	Only one byte to receive, send no acknowledge on last byte.
ST_RECEIVING → ST_RECEIVING	More than one byte to receive. Read received byte.
ST_RECEIVING → ST_RECV_LAST	Only one byte left to receive, send no acknowledge on it.
ST_RECV_LAST → ST_IDLE	Last byte read, send stop. No more messages. Call ReadyProc and give status.
ST_RECV_LAST → ST_SENDING	Last byte read, send repeated start and slave address of next (write) message.
ST_RECV_LAST → ST_AWAIT_ACK	Last byte read, send repeated start and slave address of next (read) message.

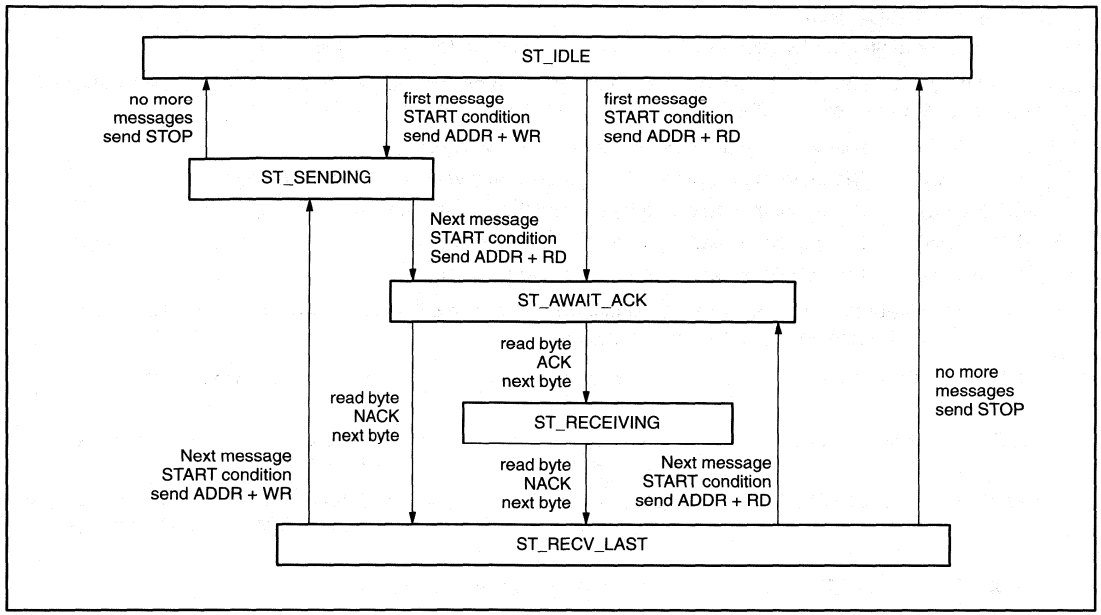


Figure 2. State transition diagram of the master state handler

4.1 Bit-banging driver

The XA-Gx derivative does not incorporate on-chip I²C hardware. However, I²C functionality can be achieved by software emulation. The file I2CBITS.C (Appendix II) performs two main tasks: handling complete transfers that consist of one or more messages (described above, see Figure 2), and the software emulation task. The emulation task consists of: bus monitoring and control, master sending/receiving of bytes conform to the I²C protocol.

The following macro and functions are designed for master I²C-bus control:

delay	Macro for delay loop of about 1 microsecond. Needs to be tuned (in application note done for 20MHz XA). This delay is needed to insure minimum high and low clock times on the bus. Also, the hold and setup times for START and STOP conditions are met with this macro. To optimize the speed, the software (generated by the compiler) delay is measured and included in the total delay times.
SCLHigh()	Function to release (send high) the SCL pin and wait for any clock stretching peripheral devices. At this point, if requested, the user can build in time-outs.
PutByte()	Function that sends one byte of data to a slave device. After that, it checks if slave did acknowledge.
GetByte()	Function to receive one data byte from an addressed slave. and after that it sends (no)acknowledge.
GenerateStart()	Function to generate and I²C (repeated)START condition and send slave address for a message read/write.
GenerateStop()	Function to generate an I²C STOP condition, releasing the bus. It also calls the function readyProc to signal the driver is finished, and pass the status of the transfer.

4.2 PCx8584 driver

The PCx8584 logic provides a serial interface that meets the I²C-bus specification and supports all master transfer modes from and to the bus.

A microcontroller/processor interfaces to the PCx8584 via five hardware registers: S0 (data read/write register), S0' (own address register), S1 (control/status register), S2 (clock register), and S3 (interrupt vector register).

Selection of either an Intel or Motorola bus interface, achieved by detection of the first WR – CS signal sequence is outside the scope of this application note, as well as the insertion of wait states needed to meet the constraints of the XA – PCF8584 bus timing. More information about the hardware interface can be found in the Philips Semiconductors application note, AN96098: *Interfacing 68000 family peripherals to the XA.*

Bus speed

The speed of the I²C-bus is controlled by clock register S2 of the PCx8584. This register provides a prescaler that can be programmed to select one of five different clock rates, externally connected to pin 1 of the PCx8584. Furthermore, it provides a selection of four different I²C-bus SCL frequencies, ranging up to 90 KHz. The value for register S2 is passed as a parameter during initialization of the driver. To select the correct initialization values, refer the the datasheet or the Application Report of the PCx8584.

Interrupt

In this applicaiton note we assume that the interrupt output of the PCF8584 is connected to external interrupt 0 input of the XA. In the initialization function, this interrupt is enabled, its priority is set as well as the general interrupt enable flag. Furthermore, a 'soft' interrupt vector is filled to point to the right interrupt handler. This is only done for debugging purposes. In a 'real' application, this should be replaced by a ROM vector.

After completing the transmission or reception of each byte (address or data), the PIN flag in the control/status register of the PCF8584 is reset to 0. This will send an interrupt to the XA (EX0) and the interrupt service (state) handler will be called (see Figure 2).

If a transfer is started, the driver interface function returns immediately. At the end of the transfer, together with the generation of a STOP condition, the driver calls a function, passing the transfer status. A pointer to this function was given by the applicaiton at the time the transfer was applied for. It is up to the user to write this function and to determine the actions that have to be done (see for example, the function I2cReady in module I2CINTFC.C).

5. DEMO PROGRAM

The modules DEMO.C and I2CINTFC.C use either one of the drivers to implement a simple application on a Microcore 6 demo / evaluation board. They are intended as examples to show how to use the driver routines.

The Microcore 6 board contains a PCx8584 I²C-bus controller, a PCF8583 real time clock and a PCF8574 I/O expander with connections to 4 LEDs. the demo program runs the LEDs every second.

The module I2CINTFC.C gives an example of how to implement a few basic transfer functions (see also previous SLE I²C driver application notes). These functions allow you to communicate with most of the available I²C devices and serve as a **layer** between your application and the driver software. This **layered approach** allows support for new devices (microcontrollers) without re-writing the high-level (device-independent) code. The given examples are:

void I2C_Write(I2C_MESSAGE *msg)

void I2C_WriteRepWrite(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)

void I2C_WriteRepRead(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)

void I2C_Read(I2C_MESSAGE *msg)

void I2C_ReadRepRead(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)

void I2C_ReadRepWrite(I2C_MESSAGE *msg1, I2C–MESSAGE *msg2)

Furthermore, the module I2CINTFC.C contains the functions *StartTransfer*, in which the actual call to the driver program is done, and the function *I2cReady*, which is called by the driver after the completion of a transfer. The flag **drvStatus** is used to test/check the state of a transfer.

In the *StartTransfer* function a software time-out loop is programmed. Inside this time-out loop the *MainStateHandler* is called if the driver is in polling mode and the status register PIN flag is set.

If a transfer has failed (error or time-out) the *StartTransfer* function prints an error message (using standard I/O redirection, like the *printf()* function) and it does a retry of the transfer. However, if the maximum number of retries are reached, and exception interrupt (Trap #14) is generated to give a fatal error message.

APPENDICES

Appendix I I2CINTFC.C

```
/*****************************************************************************/
/*  Name of module : I2CINTFC.C                                              */
/*  Language       : C                                                       */
/*  Name           : P.H. Seerden                                            */
/*  Description    : External interface to the PCx8584 I2C driver            */
/*                   routines. This module contains the **EXAMPLE**          */
/*                   interface functions, used by the application to         */
/*                   do I2C master-mode transfers.                           */
/*                                                                           */
/*        (C) Copyright 1996 Philips Semiconductors B.V.                     */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  History:                                                                 */
/*                                                                           */
/*  96-11-25    P.H. Seerden    Initial version                              */
/*                                                                           */
/*****************************************************************************/

#include "i2cexprt.h"
#include "i2cdrivr.h"


extern void PrintString(code char *s);     /* to send messages out using UART     */


code char retryexp[] = "retry counter expired\n";
code char bufempty[] = "buffer empty\n";
code char nackdata[] = "no ack on data\n";
code char nackaddr[] = "no ack on address\n";
code char timedout[] = "time-out\n";
code char unknowst[] = "unknown status\n";


static BYTE drvStatus;                     /* Status returned by driver           */

static I2C_MESSAGE   *p_iicMsg[2]          /* pointer to an array of (2) I2C mess */
static I2C_TRANSFER  iicTfr;



static void I2cReady(BYTE status, BYTE msgsDone)
/***********************************************
 * Input(s)    : status      Status of the driver at completion time
 *               msgsDone    Number of messages completed by the driver
 * Output(s)   : None.
 * Returns     : None.
 * Description: Signal the completion of an I2C transfer. This function is
 *              passed (as parameter) to the driver and called by the
 *              drivers state handler (!).
 *****************************************************************************/
{
    drvStatus = status;
}
```

```
static void StartTransfer(void)
/*****************************
 * Input(s)    : None.
 * Output(s)   : statusfield of I2C_TRANSFER contains the driver status:
 *                  I2C_OK          Transfer was successful.
 *                  I2C_TIME_OUT    Timeout occurred
 *                  Otherwise       Some error occurred.
 * Returns     : None.
 * Description: Start I2C transfer and wait (with timeout) until the
 *              driver has completed the transfer(s).
 ********************************************************************************************/
{
  LONG timeOut;
  BYTE retries = 0;

    do
    {
        drvStatus = I2C_BUSY;
        I2C_Transfer(&iicTfr, I2cReady);

        timeOut = 0;
        while (drvStatus == I2C_BUSY)
        {
            if (++timeOut > 60000)
                drvStatus = I2C_TIME_OUT;
        }

        if (retries == 6)
        {
            PrintString(retryexp);                  /* fatal error ! So, ..             */
            asm("trap  #14");                       /* escape to debug monitor          */
        }
        else
            retries++;

        switch (drvStatus)
        {
          case I2C_OK                : break
          case I2C_NO_DATA           : PrintString(bufempty);     break;
          case I2C_NACK_ON_DATA      : PrintString(nackdata);     break;
          case I2C_NACK_ON_ADDRESS   : PrintString(nackaddr);     break;
          case I2C_TIME_OUT          : PrintString(timedout);     break;
          default                    : PrintString(unknowst);     break;
        }
    } while (drvStatus != I2C_OK);
}
```

```
void I2C_Write(I2C_MESSAGE *msg)
/*******************************
 * Input(s)    : msg     I2C message
 * Returns     : None.
 * Description: Write a message to a slave device.
 * PROTOCOL    : <S><SlvA><W><A><D1><A> ... <Dnum><N><P>
 ********************************************************************************/
{
    iicTfr.nrMessages = 1;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = msg;

    StartTransfer();
}

void I2C_WriteRepWrite(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)
/*****************************************************
 * Input(s)    : msg1     first I2C message
 *               msg2     second I2C message
 * Returns     : None.
 * Description: Writes two messages to different slave devices separated
 *              by a repeated start condition.
 * PROTOCOL    : <S><Slv1A><W><A><D1><A>...<Dnum><A>
 *                          <S><Slv2A><W><A><D1><A>...<Dnum2><A><P>
 ********************************************************************************/
{
    iicTfr.nrMessages = 2;
    iicTfr.p.message  = p_iicMsg;
    p_iicMsg[0] = msg1;
    p_iicMsg[1] = msg2;

    StartTransfer();
}

void I2C_WriteRepRead(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)
/*****************************************************
 * Input(s)    : msg1     first  I2C message
 *               msg2     second I2C message
 * Returns     : None.
 * Description: A message is sent and received to/from two different
 *              slave devices, separated by a repeat start condition.
 * PROTOCOL     <S><Slv1A><W><A><D1><A>...<Dnum1><A>
                        <S><Slv2A><R><A><D1><A>...<Dnum2><N><P>
 ********************************************************************************/
{
    iicTfr.nrMessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = msg1;
    p_iicMsg[1] = msg2;

    StartTransfer();
}

void I2C_Read(I2C_MESSAGE *msg)
/*******************************
 * Input(s)    : msg     I2C message
 * Returns     : None.
 * Description: Read a message from a slave device.
 * PROTOCOL    : <S><SlvA><R><A><D1><A>...<Dnum><N><P>
 ********************************************************************************/
{
    iicTfr.nrMessages = 1;
    iicTfr.p_message  = p_iicMsg;
    P_iicMsg[0] = msg;

    StartTransfer();
}
```

```
void I2C_ReadRepRead(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)
/*******************************************************
 * Input(s)    : msg1      first I2C message
 *               msg2      second I2C message
 * Returns     : None.
 * Description: Two messages are read from two different slave devices,
 *              separated by a repeated start condition.
 * PROTOCOL    : <S><Slv1A><R><A><D1><A>...<Dnum1><N>
 *                      <S><Slv2A><R><A><D1><A>...<Dnum2><N><P>
 *******************************************************************************/
{
    iicTfr.nrMessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = msg1;
    p_iicMsg[1] = msg2;

    StartTransfer();
}

void I2C_ReadRepWrite(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2)
/*******************************************************
 * Input(s)    : msg1      first I2C message
 *               msg2      second I2C message
 * Returns     : None.
 * Description: A block data is received from a slave device, and also
 *              a(nother) block data is send to another slave device
 *              both blocks are separated by a repeated start.
 * PROTOCOL    : <S><Slv1A><R><A><D1><A>...<Dnum1><N>
 *                      <S><Slv2A><W><A><D1><A>...<Dnum2><A><P>
 *******************************************************************************/
{
    iicTfr.mrNessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = msg1;
    p_iicMsg[1] = msg2;

    StartTransfer();
}
```

Appendix II I2CBITS.C

```
/*****************************************************************************/
/*  Name of module : I2CBITS.C                                               */
/*  Language       : C                                                       */
/*  Name           : P.H. Seerden                                            */
/*  Description    : Driver part for XA-G3 I2C 'bit-bang' code.              */
/*                                                                           */
/*   P1.4 and P1.5 are used for SCL and SDA.                                 */
/*   Everything between one Start and Stop condition is called a TRANSFER.   */
/*   One transfer consists of one or more MESSAGEs.                          */
/*   MESSAGEs are separated by Repeated Staarts.                             */
/*   To start a transfer call function "I2C_Transfer".                       */
/*                                                                           */
/*          (C) Copyright 1996 Philips Semiconductors B.V.                   */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  History:                                                                 */
/*                                                                           */
/*  96-11-25    P.H. Seerden    Initial version                              */
/*                                                                           */
/*****************************************************************************/

#include <xa.h>

#include "i2cexprt.h"
#include "i2cdrivr.h"


static I2C_TRANSFER *tfr;              /* Ptr to active transfer block      */
static I2C-MESSAGE  *msg;              /* ptr to active message block       */

static void (*readyProc) (BYTE,BYTE); /* proc. to call if transfer ended   */
static BYTE mssgCount;                 /* Number of messages sent           */
static BYTE dataCount;                 /* nr of bytes of current message    */
static BYTE state;                     /* state of the I2C driver           */
static bit noAck;



/* about 1 us delay time at 20 MHz.                                          */
/* Used to insure minimum high and low clock times on the I2C bus.           */
/* This macro must be tuned for the actual oscillator frequency used.        */
/* Other parameters involved:   XA bus timing (BTRH and BTRL)                */
/*                              required I2C bus speed (normal or fast)      */
/*                              performance of compiler generated code       */

#define   delay     asm("  nop"); \
                    asm("  nop"); \
                    asm("  nop"); \
                    asm("  nop"); \
                    asm("  nop"); \
                    asm("  nop"); \
                    asm("  nop"); \
                    asm("  nop")
```

```
static void SCLHigh(void)
/************************
 * Input(s)     : none.
 * Returns      : none.
 * Description : Sends SCL pin high and wait for any clock stretching
 *               peripherals.
 ************************************************************************/
{
    SCL = 1;
    while (!SCL)  ;
    delay;
}


static void PutByte(BYTE i)
/*************************
 * Input(s)     : i       byte to be transmitted.
 * Returns      : None.
 * Description : Sends one byte of data to a slave device.
 ************************************************************************/
{
  BYTE n;

    for (n-0x80; n!=0; n=n>>1)
    {
        SDA = (i & n) ? 1 : 0;
        SCLHigh();                  /* make SCL high and check for stretching */
        delay;                      /* extra delay needed                     */
        SCL = 0;
//        delay;                    /* not needed, enough delay in sw loop    */
    }
    delay;                          /* extra delay needed (1 us)              */
    SDA = 1;                        /* release data line for acknowledge      */
    SCLHigh();                      /* make SCL high and check for stretching */
    noAck = SDA;                    /* check acknowledge                      */
    SCL = 0;
}


static BYTE GetByte(void)
/************************
 * Input(s)     : None.
 * Returns      : received byte.
 * Description : Receive one byte of an addressed slave.
 ************************************************************************/
{
  BYTE n,i;

    for (n=0; n<8; n++)                       /* read 8 bits              */
    {
        SCLHigh();                  /* make SCL high and check for stretching */
        i = i | SDA;
        SCL = 0;
        delay;
        i = i<<1;
    }
    SDA = noAck;
    SCLHigh();                      /* make SCL high and check for stretching */
    SCL = 0;
    SDA = 1;
    return i;
}
```

```
static void GenerateStart(void)
/*******************************
 * Input(s)     : None.
 * Returns      : None.
 * Description : Generate a start condition, and send slave address.
 ************************************************************************/
{
    SCL = 1;                            /* needed for repeated start          */
    SDA = 1;
    noAck = FALSE;                          /* clear no ack status flag       */

    if (SCL && SDA)                         /* both lines high ??             */
    {
        SDA = 0;
        delay;
        delay;                      /* hold time start condition min. 4 us    */
        delay;
        SCL = 0;
        PutByte(msg->address);
    }
    else
        readyProc(I2C_ERR, mssgCount);       /* Signal driver is finished  */
}

static void GenerateStop(BYTE status)
/*************************************
 * Input(s)     : status       status of the driver.
 * Returns      : None.
 * Description : Generate a stop condition, releasing the bus.
 ************************************************************************/
{
    SDA = 0;
    SCLHigh();                  /* make SCL high and check for stretching */
    SDA = 1;                    /* stop condition setup time min. 4 us    */
    state = ST_IDLE;
    readyProc(status, mssgCount;          /* Signal driver is finished    */
}
```

```
void StateHandler(void)
/*********************
 * Input(s)     : None.
 * Returns      : None.
 * Description : Master mode state handler for I2C bus.
 ************************************************************************/
{
    switch (state)
    {
     case ST_SENDING :
        if (noAck)
            GenerateStop(I2C_NACK_ON_DATA);
        else
            if (dataCount < msg->nrBytes)
                PutByte(msg->buf[dataCount++]);        /* sent next byte   */
            else
            {
                if (msgCount < tfr->nrMessages)
                {
                    dataCount = 0;
                    msg = tfr->p_message[mssgCount++];
                    state = (msg->address & 1) ? ST_AWAIT_ACK : ST_SENDING;
                    GenerateStart();
                }
                else
                    GenerateStop(I2C_OK);              /* transfer ready   */
            }
            break;
         case ST_AWAIT_ACK :
            if (noAck)
                GenerateStop(I2C_NACK_ON_ADDRESS);
            else
                if (msg->nrBytes == 1)
                {
                    noAck = TRUE;                      /* clear ACK        */
                    state = ST_RECV_LAST;
                }
                else
                    state = ST_RECEIVING;
            break;
         case ST_REEIVING :
          msg->buf[dataCount++] = GetByte();
            if (dataCount + 1 == msg->nrBytes)
            {
                noAck = TRUE;                         /* clear ACK         */
                state = ST_RECV_LAST;
            }
            break;
         case ST_RECV_LAST :
            msg->buf[dataCount] = GetByte();
            if (mssgCount<tfr->nrMessages)
            {
                dataCount = 0;
                msg = tfr->p_message[mssgCount++];
                state = (msg->address & 1) ? ST_AWAIT_ACK : ST_SENDING;
                GenerateStart();
            }
            else
                GenerateStop(I2C_OK);                 /* transfer ready     */
            break;
         case ST_IDLE :
            break;
         default :                                    /* impossible          */
            GenerateStop(I2C_ERR);                    /* just to be sure     */
            break;
        }
}
```

```
void I2C_Transfer(I2C_TRANSFER *p, void (*proc)(BYTE, BYTE))
/*************************************************************
 * Input(s)    : p             address of I²C transfer parameter block.
 *               proc          procedure to call when trnasfer completed,
 *                             with the driver status passed as parameter.
 * Output(s)   : None.
 * Returns     : None.
 * Description: Start an I²C transfer, containing 1 or more messages. The
 *              application must leave the transfer parameter block
 *              untouched until the ready procedure is called.
 ************************************************************************/
{
    tfr = p;
    readyProc = proc;
    mssgCount = 1;
    dataCount = 0;
    msg = tfr->p_message[0];                    /* first message           */

    state = (msg->address & 10 ? ST_AWAIT_ACK : ST_SENDING;

    GenerateStart();

    while (state != ST_IDLE)
        StateHandler();
}


void I2C_Initialize(BYTE dum)
/****************************
{
    state = ST_IDLE;

    P1CFGA = P1CFGA & 0xcf;         /*P1.4 and P1.5 as open drain ports     */
    P1CFGB = P1CFGB & 0xcf;
}
```

Appendix III I2C8584.C

```
/*****************************************************************************/
/*  Name of module : I2C8584.C                                               */
/*  Language       : C                                                       */
/*  Name           : P.H. Seerden                                            */
/*  Description    : Interrupt driven driver for the XA-Gx and the           */
/*                   PCx8584 I2C bus controller.                             */
/*                                                                           */
/*   Uses external interrupt 0 of the XA.                                    */
/*   Everything between one Start and Stop condition is called a TRANSFER.   */
/*   One transfer consists of the one or more MESSAGEs.                      */
/*   To start a transfer call function "I2C_Transfer".                       */
/*                                                                           */
/*        (C) Copyright 1996 Philips Semiconductors B.V.                     */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  History:                                                                 */
/*                                                                           */
/*  96-11-25    P.H. Seerden    Initial version                              */
/*                                                                           */
/*****************************************************************************/

#include <xa.h>

#include "i2cexprt.h"
#include "i2cdrivr.h"


/******************CHANGE ADDRESSES FOR OTHER APPLICATIONS******************/

#define BYTE_AT(x)    (*(far unsigned char*)x)
                                        /*                      ES0 ES1 ES2   */
#define AR_8584     BYTE_AT(0xF0000)    /* Address Register     0   0   0     */
#define VR_8584     BYTE_AT(0xF0000)    /* Vector Register      0   0   1     */
#define CL_8584     BYTE_AT(0xF0000)    /* Clock Register       0   1   0     */
#define DR_8584     BYTE_AT(0xF0000)    /* Data Register        1   0   0     */

#define CR_8584     BYTE_AT(0xF0002)    /* Control Register     0   x   x     */
#define CS_8584     BYTE_AT(0xF0002)    /* Cntrl/Status Reg     1   x   x     */

/*****************************************************************************/


static I2C_TRANSFER *tfr;               /* Ptr to active transfer block       */
static I2C_MESSAGE  *msg;               /* ptr to active message block        */

static void (*readyProc)(BYTE,BYTE);    /* proc. to call if transfer ended    */
static BYTE mssgCount;                  /* Number of messages sent            */
static BYTE dataCount;                  /* nr of bytes of current message     */
static BYTE state;                      /* state of the I2C driver            */


static void GenerateStop(BYTE status)
/*************************************
 * Input(s)     : status       status of the driver.
 * Output(s)    : driver status to the upper layer.
 * Returns      : none.
 * Description : Generate a stop condition.
 *****************************************************************************/
{
    CR_8584 = PIN_MASK | ESO_MASK | STO_MASK | ACK_MASK;
    state   = ST_IDLE;

    readyProc(status, mssgCount);          /* Signal driver is finished     */
}
```

```
interrupt void I2C_Interrupt(void)
/*********************************
 * Input(s)     : none.
 * Output(s)    : none.
 * Returns      : none.
 * Description : Interrupt handler for PCF8584 int. at external int 0 pin.
 ********************************************************************************/
{
    switch (state)
    {
     case ST_SENDING :
        if (CS_8584 & LRB_MASK)
            GenerateStop(I2C_NACK_ON_DATA);
        else
            if (dataCount < msg->nrBytes)
                DR_8584 = msg->buf[dataCount++];      /* sent next byte     */
            else
            {
                if (mssgCount < tfr->nrMessages)
                {
                    dataCount = 0;
                    msg = tfr->p_message[msgCount++];
                    state = (msg->address & 1) ? ST_AWAIT_ACK : ST_SENDING;
                    CS_8584 = ESO_MASK | STA_MASK | ACK_MASK;
                    DR_8584 = msg->address;
                }
                else
                    GenerateStop(I2C_OK);              /* transfer ready   */
                }
        break;
     case ST_AWAIT_ACK :
        if (CS_8584 & LRB_MASK)
            GenerateStop(I2C_NACK_ON_ADDRESS);
        else
        {
          BYTE dummy;
            if (msg->nrBytes == 1)
            {
                CS_8584 = ESO_MASK;                    /* clear ACK        */
                state = ST_RECV_LAST;
            }
            else
                state = ST_RECEIVING;
            dummy = DR_8584;     /* start generation of clock pulses
                                    for the first byte to read           */
        }
        break;
     case ST_RECEIVING :
        if (dataCount + 2 == msg->nrBytes)
        {
            CS_8584 = ESO_MASK;                        /* clear ACK        */
            state = ST_RECV_LAST;
        }
        msg->buf[dataCount++] = DR_8584;
        break;
     case ST_RECV_LAST :
        if (mssgCount < tfr->nrMessages)
        {
            msg->buf[dataCount] = DR_8584;
            dataCount = 0;
            msg = tfr->p_message[mssgCount++];
            state = (msg->address & 1) ? ST_AWAIT_ACK : ST_SENDING;
            CS_8584 = ESO_MASK | STA_MASK | ACK_MASK;
            DR_8584 = msg->address;
        }
        else
        {
            GenerateStop(I2C_OK);                     /* transfer ready   */
            msg->buf[dataCount] = DR_8584;
        }
        break;
     default :                                  /* impossible               */
        GenerateStop(I2C_ERR);                  /* just to be sure          */
        break;
    }
}
```

```
void I2C_Transfer(I2C_TRANSFER *p, void (*proc)(BYTE, BYTE))
/*************************************************************
 * Input(s)    : p             address of I2C transfer parameter block.
 *               proc          procedure to call when transfer completed.
 *                             with the driver status passed as parameter.
 * Output(s)   : None.
 * Returns     : None.
 * Description: Start an I2C transfer, containing 1 or more messages. The
 *               application must leave the transfer parameter block
 *               untouched until the ready procedure is called.
 *************************************************************************/
{
    tfr = p;
    readyProc = proc;
    mssgCount = 0;
    dataCount = 0;
    msg = tfr->p_message[mssgCount++];

    state   = (msg->address & 1) ? ST_AWAIT_ACK : ST_SENDING;
    CS_8584 = ESO_MASK | STA_MASK | ACK_MASK;            /* generate start */
    DR_8584 = msg >address;
}


void I2C_Initialize(BYTE speed)
/*****************************
 * Input(s)    : speed      clock register value for bus speed.
 * Output(s)   : None.
 * Returns     : None.
 * Description: Initialize the PCF8584.
 *************************************************************************/
{
    state     = ST_IDLE;
    readyProc = NULL;

    AR_8584 = 0x26;                        /* dummy own slave address      */
    CR_8584 = 0x20;                        /* write clock register         */
    CL_8584 = speed;

/* for Microcore 6 and XTEND,                                              */
/* now fill the secondary vector table with the right interrupt vector     */
#asm
    mov.w  r2,#680h
    mov.w  r0,#_I2C_Interrupt&(0+65535)
    mov.w  r1,#seg _I2C_Interrupt
    mov.w  [r2+],r0
    mov.w  [r2],r1
#endasm

    CR_8584 = ESO_MASK;                    /* set serial interface ON      */

    IPA0 = IPA0 | 7;
    EX0 - 1;
    EA = 1;                              /* General interrupt enable       */
}
```

Appendix IV I2CDEMO.C

```
/*****************************************************************************/
/*  Name of module     : DEMO.C                                              */
/*  Program language   : C                                                   */
/*  Name               : P.H. Seerden                                        */
/*  Description        : XA-Gx I2C driver test (PCF8584 + bit bang)          */
/*                       Runs on MICROCORE 6                                 */
/*                       Read time from the real time clock chip PCF8583.    */
/*                       run leds connected to PCF8574 every second.         */
/*                                                                           */
/*         (C) Copyright 1996 Philips Semiconductors B.V.                    */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  History:                                                                 */
/*                                                                           */
/*  96-11-25    P.H. Seerden    Initial version                              */
/*                                                                           */
/*****************************************************************************/

#include "i2cexprt.h


#define PCF8574_WR   0x40                   /* i2c address I/O poort write */
#define PCF8574_RD   0x41                   /* i2c address I/O poort read  */
#define PCF8583_WR   0xA0                   /* i2c address Clock           */
#define PCF8583_RD   0xA1                   /* i2c address Clock           */

static BYTE  rtcBuf[1];
static BYTE  iopBuf[1];

static I2C_MESSAGE  rtcMsg1;
static I2C_MESSAGE  rtcMsg2;
static I2C_MESSAGE  iopMsg;


static void Init(void)
{
    I2C_Initialize(0x10);       /* for PCF8584, 4.43MHz and SCL = 90KHz    */

    rtcMsg1.address = PCF8583_WR;
    rtcMsg1.buf     = rtcBuf;
    rtcMsg1.nrBytes = 1;
    rtcMsg2.address = PCF8583_RD;
    rtcMsg2.buf     = rtcBuf;
    rtcMsg2.nrBytes = 1;

    iopMsg.address = PCF8574_WR;
    iopMsg.buf     = iopBuf;
    iopMsg.nrBytes = 1;
    iopBuf[0] = 0xff;
    I2C_Write(&iopMsg);
}
```

```
void main(void)
{
  BYTE  oldseconds,port;

    Init();

    oldseconds = 0;
    port = 0xf7;
    while (1)
    {
        rtcBuf[0] = 2;                                  /* read seconds            */
        I2C_WriteRepRead(&rtcMsg1, &rtcMsg2);

        if (rtcBus[0] != oldseconds)                    /* one second passesed ?   */
        {
            oldseconds = rtcBuf[0];

            switch (port)
            {
              case 0xf7: port = 0xfe;   break;
              case 0xfb: port = 0xf7;   break;
              case 0xfd: port = 0xfb;   break;
              case 0xfe: port = 0xfd;   break;
              default:    break;
             }
             iopBuf[0] = port;
             I2C_Write(&iopMsg);
        }
    }
}
```

Appendix V I2CEXPRT.H

```
/*************************************************************************/
/*  Name of module : I2CEXPRT.H                                          */
/*  Language       : C                                                   */
/*  Name           : P.H. Seerden                                        */
/*  Description    : This module consists of a number of exported        */
/*                   declarations of the I2C driver package. Include     */
/*                   this module in your source file if you want to      */
/*                   make use of one of the interface functions of the   */
/*                   package.                                            */
/*                                                                       */
/*        (C) Copyright 1996 Philips Semiconductors B.V.                 */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*  History:                                                             */
/*                                                                       */
/*  96-11-25    P.H. Seerden    Initial version                          */
/*                                                                       */
/*************************************************************************/

typedef unsigned char    BYTE;
typedef unsigned short   WORD;
typedef unsigned long    LONG;

typedef struct
{
    BYTE   address;           /* slave address to sent/receive message     */
    BYTE   nrBytes;           /* number of bytes in message buffer         */
    BYTE   *buf;              /* pointer to application message buffer     */
} I2C_MESSAGE;

typedef struct
{
    BYTE        nrMessages;           /* number of message in one transfer */
    I2C_MESSAGE **p_message;          /* pointer to pointer to message     */
} I2C_TRANSFER;

/*************************************************************************/
/*        E X P O R T E D   D A T A   D E C L A R A T I O N S            */
/*************************************************************************/

#define FALSE       0
#define TRUE        1

#define I2C_WR      0
#define I2C_RD      1

/**** Status Errors ****/

#define I2C_OK                  0       /* transfer ended No Errors      */
#define I2C_BUSY                1       /* transfer busy                 */
#define I2C_ERR                 2       /* err: general error            */
#define I2C_NO_DATA             3       /* err: No data in block         */
#define I2C_NACK_ON_DATA        4       /* err: No ack on data           */
#define I2C_NACK_ON_ADDRESS     5       /* err: No ack on address        */
#define I2C_TIME_OUT            6       /* err: Time out occurred        */


/*************************************************************************/
/*       I N T E R F A C E   F U N C T I O N   P R O T O T Y P E S       */
/*************************************************************************/

extern void I2C_Initialixe(BYTE speed);

extern void I2C_Write(I2C_MESSAGE *msg);
extern void I2C_WriteRepWrite(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2);
extern void I2C_WriteRepRead(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2);
extern void I2C_Read(I2C_MESSAGE *msg);
extern void I2C_ReadRepRead(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2);
extern void I2C_ReadRepWrite(I2C_MESSAGE *msg1, I2C_MESSAGE *msg2);
```

Appendix VI I2CDRIVR.H

```
/*****************************************************************************/
/*  Name of module : I2CDRIVR.H                                              */
/*  Language       : C                                                       */
/*  Name           : P.H. Seerden                                            */
/*  Description    : This module contains a number of 'local'                */
/*                   declarations for the XA-Gx I2C driver package.          */
/*                                                                           */
/*        (C) Copyright 1996 Philips Semiconductors B.V.                     */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  History:                                                                 */
/*                                                                           */
/*  96-11-25    P.H. Seerden    Initial version                              */
/*                                                                           */
/*****************************************************************************/

static bit          SDA  @ 0x38C          /* port P1.4                       */
static bit          SCL  @ 0x28D;         /* port P1.5                       */


#define ST_IDLE             0
#define ST_SENDING          1
#define ST_AWAIT_ACK        2
#define ST_RECEIVING        3
#define ST_RECV_LAST        4

#define ACK_MASK            0x01
#define STO_MASK            0x02
#define STA_MASK            0x04
#define ESO_MASK            0x48            /* also interrupt enable     */

#define BB_MASK             0x01
#define LAB_MASK            0x02
#define AAS_MASK            0x04
#define LRB_MASK            0x08
#define BER_MASK            0x10
#define STS_MASK            0x20
#define PIN_MASK            0x80

extern void I2C_Transfer(I2C_TRANSFER *p, void (*proc)(BYTE, BYTE));
```

Using Flash memory with the XA AN97019

Author: Paul Seerden, Systems Laboratory Eindhoven, The Netherlands

ABSTRACT

This application note examines the various options involved in connecting Flash memories to the XA. Special attention is paid to the possibility of dynamic software updating (re-loading the Flash).

SUMMARY

This report is a guide for designers who consider an XA design with external Flash memory. Most of the problems/aspects that they meet are brought to light. Pro's and con's of different design choices are dealt with. Hardware interface examples, as well as software routines are illustrated. The focus is on applications with the possibility of dynamically updating the software (re-loading the Flash).

CONTENTS

1. INTRODUCTION

The Philips Semiconductors "P51XA" (Extended Architecture) is a 16-bit microcontroller family designed for applications needing high performance and high integration. A typical XA system can be built using external Flash memory for program storage. Using Flash memory provides an advantage over traditional non-volatile memories. Unlike EPROMs, Flash devices can be programmed in-system ! However, connection of a Flash device to a microcontroller like the XA is not straightforward. This application note will discuss how to achieve an efficient XA - Flash memory interface.

Chapter 2 describes hardware interfacing examples of XA systems without external data memory (so, they use only the internal XA on-chip data memory).

Chapter 3 describes an application example of an XA system with additional external data memory.

Chapter 4 describes a software driver to (re)load the Flash memory contents. The driver is written in C and for loading the Flash memory the Intel-hex file format is used. Serial downloading of the hex file is done using UART0 of the XA-G3.

1.1 What is Flash memory again ?

If the write-enable input of Flash parts is left out of consideration, then this type of memory is in fact the same as a normal EPROM. The real difference is the simplicity of data storage and erasure of the memory. Unlike EPROM's , Flash parts can be erased and programmed in-system. Flash devices are able to receive commands, like chip erase and program, by an exact defined sequence of instructions. That makes accidentally erasing the Flash almost impossible. The different instruction command codes are shown below in table 1.

TABLE 1 Flash instructions (AMD 29F010)

Command	First WR Cycle		Second WR cycle		Third WR cycle		Fourth RD/ WR cycle		Fifth WR cycle		Sixth WR cycle	
	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read / Reset	5555	AA	2AAA	55	5555	F0	RA	RD				
Autoselect	5555	AA	2AAA	55	5555	90						
Byte Program	5555	AA	2AAA	55	5555	A0	PA	PD				
Chip Erase	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	5555	10
Sector Erase	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	SA	30

Notes:

Addressbits A15...A18 are don't care for all address commands except for the Program Address (PA) and Sector Address (SA)

RA = Address of the memory location to read

PA = Address of the memory location to program (at falling edge on WE input)

SA = Address of the sector to erase

RD = Contents of memory location RA

PD = Contents of memory location PA (at falling edge on WE input)

Read / Reset : This command puts the Flash in the read/reset state, and enables the Flash for data reads.

Autoselect : Used to get the device type and manufacturer code.

Byte Program : Used to write bytes to the flash.

Chip Erase : Will program and verify the entire memory for an all zero pattern.

Sector Erase : Flash memory is divided into sectors. This command erases multiple sectors concurrently.

1.2 To take into consideration

- Please realize that it is impossible to execute code from a normal Flash memory part while programming it. So, the program code that reloads the Flash needs to be located in a separate memory area/component.
- Flash interface timing versus XA frequency. In the design examples of chapters 2 and 3 a possible need for wait states is not taken into account. The software example is tested on a 20Mhz XA-G3 system (slowest WR cycle is 100ns) with the use of 90ns Flash parts.
- The software (chapter 4) to serially download (UART0 used) the Intel-hex file does NOT use flowcontrol (like RTS/CTS or xon/xoff). So, the host (sending the Intel-hex file) needs to send the data at a rate the XA software can handle.
- All given design examples in this report are with an XA-G3 8-bit wide databus mode. Please realize that for 16-bit systems the A0 line becomes the WRH signal. This means that the addresses, used by the driver software, to which the commands are written to, must be changed (see chapter 4). Also, in this case, words in stead of bytes must be written to both (or all) the Flash parts.
- If the Flash memory is mapped at address 0 in data memory space (to enable writing it) realize that the instruction MOVX is needed to write (and read) the lower 512 bytes of the flash. If MOVX is not used the internal RAM of the XA is addressed.
- All the examples use 128 kBytes Flash parts (29F010), so only address lines 0 to 16 are connected. If a system needs to be prepared for larger flash memory parts then address lines A17 and A18 can be connected to pin 1 and pin 30 of the 29F010 part.

This application note (with C source files) is available for downloading from the Philips Bulletin Board Systems and from the world wide web. It is packed in the self extracting PC DOS file: XAFLASH.EXE.

- North American Bulletin Board, telephone number: (800) 451 6644 (in the US) or (408) 991 2406
- European Bulletin Board, telephone number: +31 40 272 1102
- Philips Semiconductors WWW: http://www.semiconductors.philips.com

2. SYSTEM WITHOUT XRAM

Applications not requiring additional external data memory (only use the internal on-chip RAM) have three options for locating the Flash loader software part. These three options are explained in detail in the next paragraphs.

- Loader on-chip (EPROM or OTP XA) and boot from on-chip ROM
- Loader on-chip (EPROM or OTP XA), but boot from external Flash memory
- Romless XA, loader in separate external ROM

2.1 Boot from on-chip ROM

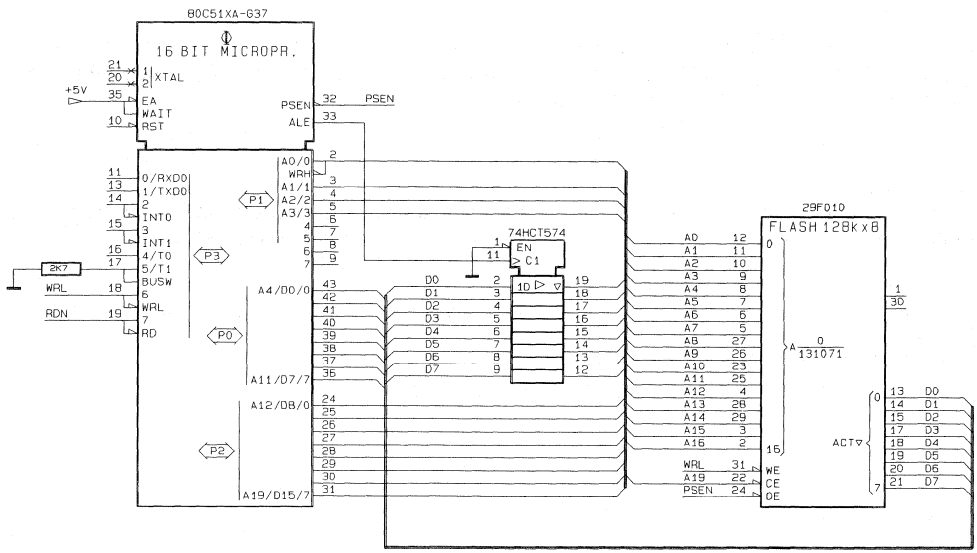


Figure 1. Boot from on-chip XA ROM

The system shown in figure 1 always boots from internal XA ROM (EA pin high). The internal XA ROM contains the start-up code, the vector table and the Flash loader program.

After start-up initialization the application program inside the external Flash memory is called. However, before the XA executes an external bus cycle the WAITDisable bit in the BCR register must be set. This causes the XA to ignore the status of WAIT input pin and allows tying the WAIT input high for applications that use internal code and do not need the WAIT function.

Besides start-up / initialization code the internal ROM of the XA also contains the Flash (re)loader program. After receiving a request to reload the Flash (for example via the UART) the application calls the loader software inside the internal ROM (see chapter 4). After updating the Flash memory contents the 'new' application program is called (or a software reset is generated).

The XA interrupt vectors are located at fixed addresses from address 0 to 120h and thus also part of the on-chip ROM. Interrupt vectors are pointers to interrupt routines. These routines are part of the application program and located inside the Flash memory. Therefore, the vectors should in some way be re-directed to fixed address pointers inside the Flash memory, or all interrupt routines must be at fixed addresses.

Advantages:

- Simple / cheap design (no extra external hardware needed)
- Start-up with empty Flash is easy to test. This is convenient for production and diagnostic purposes.

Disadvantages:

- Not flexible (fixed addresses of routines and vectors and pointers)
- Start-address of application must be fixed
- Price of OTP XA against Romless XA
- Need of a kind of 'secondary vector table' with fixed addresses
- When mapped at address 0 the first 32 Kbyte of the Flash memory is useless, because of overlap with on-chip ROM
- No easy software emulation, because it is not located at address 0

2.2 Boot from external Flash

The external Flash memory of the system shown in figure 2 contains the start-up code, the vector table and the application program. The XA on-chip ROM contains the Flash reloader software.

The intention of the system is to boot / power-up always from code inside external Flash memory. This means that the EA pin needs to be 'low' during power-up, when it is sampled by the XA. To take care of this the output of the flip-flop is put in 'reset' state (by the resistor and capacitor).

However, the Flash reloader software is inside the on-chip XA ROM. If a Flash reload is requested a re-start, but now from internal ROM (EA high), is needed. To achieve this, an external data memory write cycle to the flip-flop (at address $80000) with D0 (databus line 0) high is performed. After that, the system is re-started using an external reset circuit (MAX311). Now the internal on-chip code is running and the Flash memory can be re-loaded.

Again, if running internally (EA/WAIT pin high), before an external bus cycle is performed the WAITDisable bit must be set in the BCR register (see previous paragraph).

The address ($80000) used to set the flip-flop must be outside the Flash memory range, because otherwise a write operation into the Flash is executed. To generate an external system reset a '0' is written to port pin P3.3 (manual reset input of the MAX811).

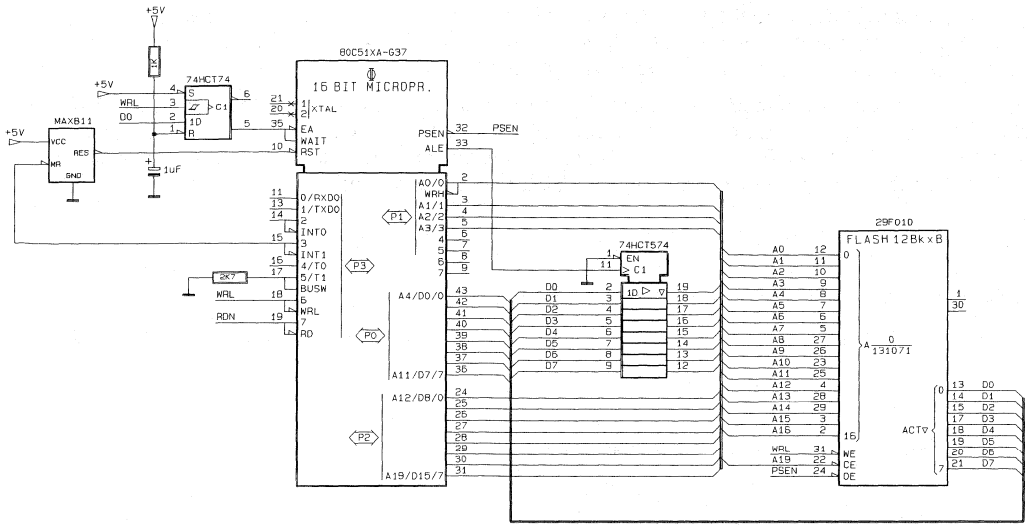


Figure 2. XA with on-chip code, but always boot from external flash

After updating the Flash contents again the system needs to be re-started, but now from the newly loaded code in external Flash memory. This means that now a 'zero' needs to be written to the flip-flop (EA low), and then again an external system 'reset' must be performed (write '0' to P3.3).

The external reset circuitry is used because the XA doesn't (re)sample the EA and BUSW pins after an internal reset (software RESET instruction or watchdog). If the use of port pin P3.3 (or other) is a problem, one could consider to connect the RDN line to the manual reset (MR) input of the MAX811. In that case, to generate a hardware reset, an external data memory read needs to be performed.

Advantages:

- Flexible design
- Interrupt vector table inside Flash
- Easy emulation of code, because Flash is mapped at address 0
- Linking / loading at address 0

Disadvantages:

- Extra glue logic needed (reset circuitry, flip-flop)
- Price of OTP XA against Romless XA
- Extra measures to take at first time power-up with empty flash (force EA high !)

2.3 Boot from external ROM

The system in figure 3 shows a Romless XA system with external Flash memory, containing the application program, at address $0 and a separate Flash loader PROM mapped above the Flash memory space ($20000). At power-up or after reset the system boots the application from the Flash memory (EA pin is low).

If a request to reload the Flash is received (for example via the UART) the application 'jumps' to the reload routine (see chapter 4) in external PROM. After reloading the Flash a software reset must be generated to restart the application.

For the example in figure 3 a 4 Kbyte PROM is selected. This is because of chip size and price reasons. Of course it is also possible to use a 'normal' EPROM type 27C...

When more Flash memory is needed, both chip enable inputs of Flash and PROM can be connected to address line A19 instead of A17.

Writing into the Flash at address $0 up to the first 512 bytes (1Kbyte for the XA-S3) is only possible using the MOVX instruction.

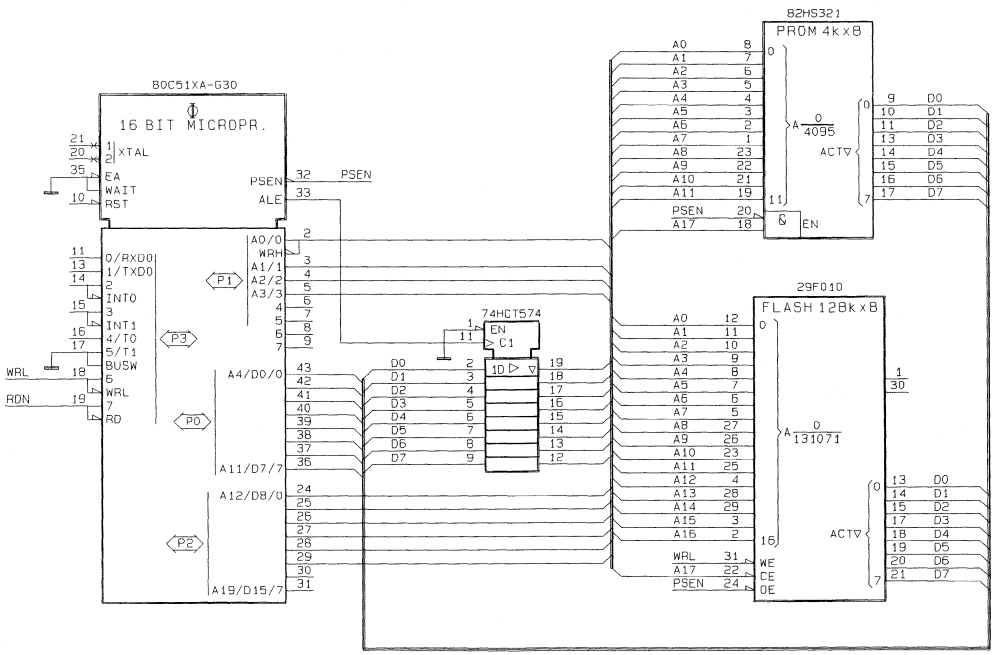


Figure 3. Romless XA, external ROM and Flash

Advantages:

- Flexible design
- Interrupt vector table inside Flash
- Easy emulation of code
- Linking / loading at address 0
- ROMless XA + external PROM is cheaper than on-chip ROM/OTP XA

Disadvantages:

- Extra external (E)PROM needed
- More PCB space needed
- Extra measures to take at first time power-up with empty flash (force boot from PROM)

3. SYSTEM WITH XRAM

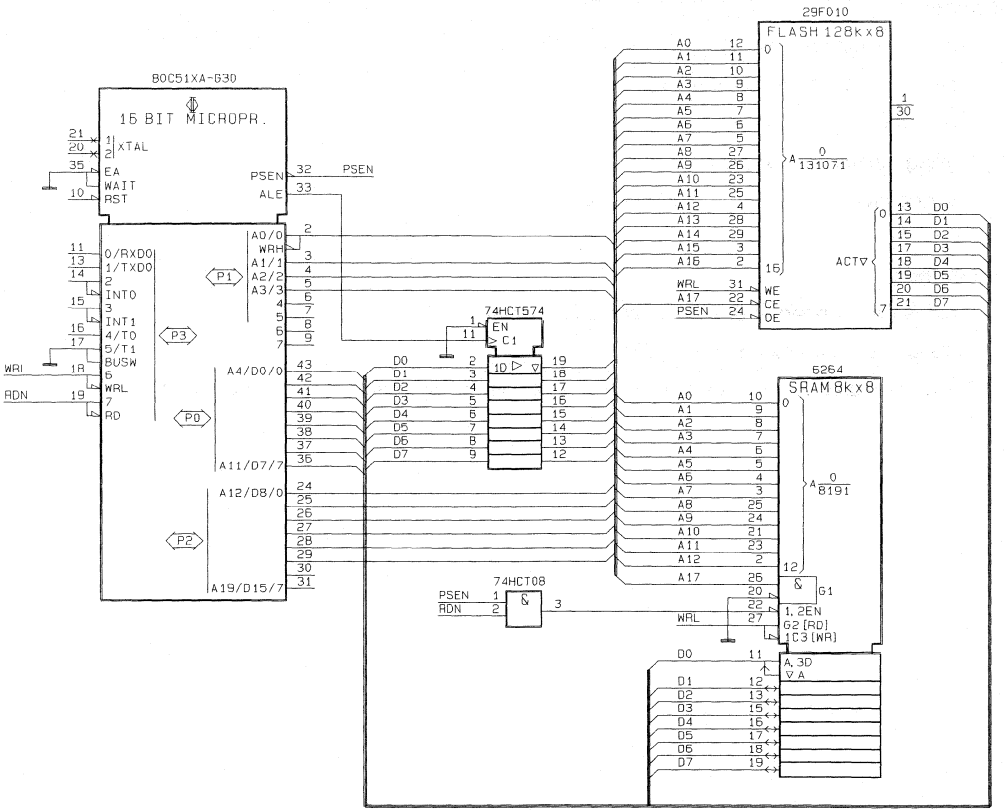


Figure 4. Romless XA, external SRAM and Flash

The system shown in figure 4 has 8 Kbytes (or if necessary more) of external data memory available. In this case an OTP/Eprom XA is superfluous, therefore a ROMless XA is selected.

At power-up the system always boots from code inside external Flash memory (EA is low). The Flash device also contains the reloader software. The SRAM device is mapped in both data and program memory space at address $20000. If a Flash reload is requested the re-loader software part is copied from flash into the static RAM and then executed.

After updating the Flash contents the system needs to re-boot, but now from the new re-loaded code in external Flash. This is simply done by executing a RESET instruction.

If a larger amount of Flash memory is needed, both chip enable inputs of Flash and SRAM can be connected to address line A19 instead of A17 (SRAM space is moved to $80000).

Writing into the Flash at address $0 up to the first 512 bytes (1Kbyte for the XA-S3) is only possible when using the MOVX instruction.

Advantages:

- Simple design
- Interrupt vector table inside Flash
- Easy emulation of code
- Linking / loading at address 0
- Lower cost (Romless XA, but SRAM device needed)

Disadvantages:

- RAM is not mapped at 0, so not usable for system stack
- PCB space (extra external SRAM)
- Extra measures to take at first time power-up with empty flash (insert a programmed Flash !)

4. FLASH LOADER SOFTWARE

This chapter describes tested driver software for an 8-bit XA-G3 system with an external AMD 29F010 (128 Kbytes) Flash part, but without external data memory (SRAM). The driver software provides the source code necessary to erase and re-program the 29F010 Flash part. The code (contents) for Flash memory re-loading is obtained from an Intel-hex file that is serially downloaded using UART0.

In the example application the base address of the Flash memory is mapped at $40000h in data memory space. To change this adjust the include file flash.h and recompile all modules.

The compiler used: XAC compiler V1.1 from TASKING.

Overview of all modules:

BOOT.C	Contains the main routine called after power-up initialization. First the flash memory is erased. Next, an Intel-hex file is uploaded and the flash is re-programmed. And, finally a RESET instruction is executed to re-boot the system.
LOADER.C	Handles reading the Intel-hex file. Checksum of each record is checked and if needed an error message is generated. Data bytes are passed to the embedded program function to write into the flash device.
FLASH.C	Contains the 29F010 Flash embedded program and chip-erase functions: *EmbeddedProgram()* Performs the actual byte program on the Flash device. After variable setup, the unlock sequences are sent, with the final unlock cycle being the byte to program at the appropriate address. *ChipErase()* Performs a chip erase on the Flash device, clearing all data and returning all locations to a value of FF hexadecimal. Protected sectors are not checked. On return a status is flagged (un / successful chip erase).
SERIAL.C	Handles the serial communication part using UART0. Contains hardware initialization, send byte, receive byte and send ASCII string routines.
FLASH.H	Include file that contains local definitions for Flash driver software package.

Note

For systems using 16-bit databus width the module flash.c needs to be adjusted. All addresses used in the unlock sequences need to be incremented by 1. This is because in a 16-bit system address line A0 is used as the WRH signal and A1 to A16 is connected to the Flash device.

5. REFERENCES AND TOOLS

Used references and development tools:	**how to get**
• AMD Flash development kit	see http://www.amd.com
• AMD Flash memory products 1994/1995 data book	see http://www.amd.com
• MAXIM Integrated products data book	see http://www.maxim.com
• HI-Tech XA C compiler (version 7.60)	see http://www.htsoft.com
• Tasking XA C compiler (version 1.1)	see http://www.tasking.com
• FDI XTEND board (rev. A1)	XTEND-G3
• Philips 16-bit 80C51XA data handbook IC25	9397 750 00733

APPENDIX 1. BOOT.C

```
/*****************************************************************************/
/*  Name of module     : BOOT.C                                                */
/*  Creation date      : 1997-04-21                                            */
/*  Program language   : C                                                     */
/*  Name               : P.H. Seerden                                          */
/*                                                                             */
/*          (C) Copyright 1997 Philips Semiconductors B.V.                     */
/*                                                                             */
/*****************************************************************************/
/*                                                                             */
/*  History:                                                                   */
/*                                                                             */
/*  97-04-21    P.H. Seerden    Initial version                                */
/*                                                                             */
/*****************************************************************************/

#include "flash.h"


void main(void)
{
    ua_init();          /* initialise UART0 for console-in/console-out     */

    PrintString("\n\n\n29F010 Flash loader Program V1.0\n\n"
                "Step 1 - Erase Flash memory\n"
                "Step 2 - Upload Intel-hex file\n"
                "Step 3 - Execute software RESET\n\n"
                "Hit any key to continue !!");

    ci();                           /* wait for any character              */

    PrintString("\nErasing chip .");
    if (ChipErase())
        PrintString("\nChip erase error !!\n");
    else
        PrintString("\nSuccessfully erased 29F010 !\n");

    PrintString("\n * downloading *\n");
    switch (Download())
    {
        case 0 : PrintString("\nready and ok\n");          break;
        case 1 : PrintString("\nChecksum Error\n");        break;
        case 2 : PrintString("\nBad record type\n");       break;
        case 3 : PrintString("\nFlash program Error\n");   break;
        default: PrintString("\nUnknown Error\n");         break;
    }

#pragma asm
    RESET               ; execute software reset instruction
#pragma endasm
}
```

APPENDIX 2. LOADER.C

```
/***************************************************************************/
/*  Name of module     : LOADER.C                                          */
/*  Creation date      : 1997-04-21                                        */
/*  Program language   : C                                                 */
/*  Name               : P.H. Seerden                                      */
/*  Description        : Handles downloading Intel Hex-32 files.           */
/*                                                                         */
/*          (C) Copyright 1997 Philips Semiconductors B.V.                 */
/*                                                                         */
/***************************************************************************/

#include "flash.h"


_rom char ascii[] = "0123456789ABCDEF";
static BYTE checkSum;


/***************************************************************************/
/*  Load2Hex()                                                             */
/*                                                                         */
/*  purpose:   This routine reads two hex bytes and returns their          */
/*             binary byte value. To read the two hex bytes the routine    */
/*             ci() (console in) is called.                                */
/*                                                                         */
/***************************************************************************/
static BYTE Load2Hex(void)
{
  BYTE  t,i,s;

    i = ci();                       /* read one hex byte from the UART       */
    for (t = 0; t < 16; t++)        /* convert hex byte to binary low nibble */
        if (ascii[t] == i)
            break;

    i = ci();                       /* read second hex byte                  */
    for (s = 0; s < 16; s++)        /* convert to binary high nibble         */
        if (ascii[s] == i)
            break;

    s = (t<<4) + s;                 /* merge low and high nibble             */
    checkSum = checkSum + s;        /* update checksum of this record        */
    return s;
}


/***************************************************************************/
/*  Load4Hex()                                                             */
/*                                                                         */
/*  purpose:    This routine reads four hex bytes and returns their        */
/*              binary word value.                                         */
/*                                                                         */
/***************************************************************************/
static WORD Load4Hex(void)
{
  WORD t;

    t = Load2Hex();                 /* first do the low byte                 */
    return (t<<8) + Load2Hex();     /* now the high byte and merge           */
}
```

```
/*****************************************************************************/
/*  Download()                                                               */
/*                                                                           */
/*  purpose:    Download Intel-hex file. Use EmbeddedProgram() to write      */
/*              bytes to flash.                                              */
/*                                                                           */
/*****************************************************************************/
BYTE Download(void)
{
  BYTE length;
  LONG addr,segm,address;

    segm = 0;
    while (1)                          /* start downloading Intel Hex file    */
    {
        while (ci() != ':')            /* wait for : (start record)           */
            ;

        checkSum = 0;
        length   = Load2Hex();                          /* read record length  */
        addr     = Load4Hex();                          /* read address        */

        switch (Load2Hex())                             /* read record type    */
        {
          case 0:                                       /* read data record    */
                address = base + segm + addr;
                break;
          case 1:                                       /* read end record     */
                Load2Hex();                             /* fetch last checksum */
                if (checkSum != 0)
                    return 1;                           /* checksum error      */
                return 0;                               /* download OK         */
          case 2:                                       /* read esar record    */
                segm = Load4Hex();
                segm = segm<<4;
                length -= 2;
                break;
          case 3:                                       /* read start address  */
                break;
          case 4:                                       /* read elar record    */
                segm = Load4Hex();
                segm = segm<<16;
                length -= 2;
                break;
          default:
                return 2;                               /* bad record type     */
        }
        while (length != 0)                             /* read data bytes     */
        {
            if (EmbeddedProgram(address,Load2Hex()))
                return 3;                               /* byte program error  */
            address ++;
            length --;
        }
        Load2Hex();                                     /* fetch checksum      */
        if (checkSum != 0)
            return 1;                                   /* checksum error      */
    }
}
```

APPENDIX 3. FLASH.C

```
/*****************************************************************************/
/*  Name of module     : FLASH.C                                             */
/*  Creation date      : 1997-04-21                                          */
/*  Program language   : C                                                   */
/*  Name               : P.H. Seerden                                        */
/*                                                                           */
/*          (C) Copyright 1997 Philips Semiconductors B.V.                   */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  Description:                                                             */
/*                                                                           */
/*   Source code for programming 29F010 Flash components.                    */
/*                                                                           */
/*****************************************************************************/

#include "flash.h"


/*****************************************************************************/
/*  EmbeddedProgram()                                                        */
/*                                                                           */
/*  purpose:     Performs the AMD Embedded Programming algorithm.            */
/*               The device is a 29F010, it is not checked for protected     */
/*               sectors. A polling address is setup, the address is         */
/*               started, a 4-cycle unlock sequence is initiated the         */
/*               device is polled using dq7 poll for programming.            */
/*                                                                           */
/*****************************************************************************/
/*  parameters:      LONG address = address to program                       */
/*                   BYTE val     = byte to program                          */
/*****************************************************************************/
/*  return value:    0 : successful                                          */
/*                   1 : unsuccessful                                        */
/*****************************************************************************/
BYTE EmbeddedProgram(LONG address, BYTE val)
{
  _far BYTE *p;
       BYTE i;

    p  = (_far BYTE *)(base + 0x5555);
    *p = 0xAA;                              /* first WR cycle               */

    p  = (_far BYTE *)(base + 0x2AAA);
    *p = 0x55;                              /* second WR cycle              */

    p  = (_far BYTE *)(base + 0x5555);
    *p = 0xA0;                              /* third WR cycle               */

    p  = (_far BYTE *)address;
    *p = val;                               /* output the byte to the device */

    for (;;)
    {
        i = *p;                                 /* read data                */
        if ((val & 0x80) == (i & 0x80))         /* DQ7 = true data bit ?    */
            return 0;                           /* Embedded program ok !    */
```

```
        if (i & 0x20)                       /* is DQ5 = 1 ?                */
        {
            i = *(_far BYTE *)address;       /* DQ6 + DQ5 both changed ?    */
            if ((i & 0x80) == (val & 0x80))
                return 0;                   /* device passed DATA POLL     */
            else
                return 1;                   /* DQ7 not true, while DQ5=1   */
        }
    }
}


/*****************************************************************************/
/*  ChipErase()                                                              */
/*                                                                           */
/*  purpose:    Performs the AMD Embedded Erase algorithm.                   */
/*              The device is a 29F010 5v device, it is not checked for      */
/*              protected sectors. The 6-cycle unlock sequence is            */
/*              initiated, and the device is polled using dq7 poll.          */
/*                                                                           */
/*****************************************************************************/
/* return value:   0 : successful                                            */
/*                 1 : unsuccessful                                          */
/*****************************************************************************/
BYTE ChipErase(void)
{
  _far BYTE *address;
       BYTE temp;

    address = (_far BYTE *)(base + 0x5555);
    *address = 0xAA;                        /* first WR cycle              */

    address = (_far BYTE *)(base + 0x2AAA);
    *address = 0x55;                        /* second WR cycle             */

    address = (_far BYTE *)(base + 0x5555);
    *address = 0x80;                        /* third WR cycle              */

    address = (_far BYTE *)(base + 0x5555);
    *address = 0xAA;                        /* fourth WR cycle             */

    address = (_far BYTE *)(base + 0x2AAA);
    *address = 0x55;                        /* fifth WR cycle              */

    address = (_far BYTE *)(base + 0x5555);
    *address = 0x10;                        /* sixth WR cycle              */

    for (;;)                        /* polling algorithm                   */
    {
        temp = *address;            /* read data from unprotected address  */
        if (temp & 0x80)            /* is DQ7 = 1 ?                        */
            return 0;               /* We're done, DATA POLL ok            */

        if (temp & 0x20)            /* is DQ5 = 1 ?                        */
        {
            temp = *address;        /* DQ6 + DQ5 changed simultaneously ?  */
            if (temp & 0x80)
                return 0;           /* device passed DATA POLL algorithm   */
            else
                return 1;           /* DQ7 not true, even after DQ5 = 1    */
        }
    }
}
```

APPENDIX 4. SERIAL.C

```
/*****************************************************************************/
/*  Name of module     : SERIAL.C                                            */
/*  Creation date      : 1997-04-21                                          */
/*  Program language   : C                                                   */
/*  Name               : P.H. Seerden                                        */
/*  Description        : Console I/O for P51XA UART 0.                       */
/*                                                                           */
/*          (C) Copyright 1997 Philips Semiconductors B.V.                   */
/*                                                                           */
/*****************************************************************************/

#include <regxag3.sfr>

#define BAUD_RATE       19200
#define OSC             20000000L               /* Xtal frequency           */
#define DIVIDER         (OSC/(64L*BAUD_RATE))

void ua_init(void)
{
    TL1 = RTL1 = -DIVIDER;
    TH1 = RTH1 = -DIVIDER >> 8;
    TR1 = 1;                                /* enable timer 1               */
    S0CON = 0x52;                           /* mode 1, receiver enable      */
}

void co(char c)
{
    if (c == '\n')
    {
        while (!TI_0)   ;
        S0BUF = '\r';
        TI_0 = 0;
    }
    while (!TI_0)   ;
    S0BUF = c;
    TI_0 = 0;
}

char ci(void)
{
  char    c;

    while (!RI_0)   ;
    c = S0BUF & 0x7F;
    RI_0 = 0;
    return c;
}

void PrintString(const char *p)
{
  _rom char *s;

    s = (_rom char *)p;
    while (*s != '\0')
    {
        if (*s == '\n')
            co('\r');                       /* output a '\r' first          */
        co(*s);
        s++;
    }
}
```

APPENDIX 5. FLASH.H

```
/*****************************************************************************/
/*  Name of module     : FLASH.H                                             */
/*  Creation date      : 1997-04-21                                          */
/*  Program language   : C                                                   */
/*  Name               : P.H. Seerden                                        */
/*                                                                           */
/*          (C) Copyright 1997 Philips Semiconductors B.V.                   */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  Description:                                                             */
/*                                                                           */
/*   Local declarations for the XA flash driver package.                     */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  History:                                                                 */
/*                                                                           */
/*  97-04-21    P.H. Seerden    Initial version                              */
/*                                                                           */
/*****************************************************************************/

typedef unsigned char    BYTE;
typedef unsigned short   WORD;
typedef unsigned long    LONG;


#define base      0x40000           /* base address of flash memory        */


extern void co(char ch);
extern char ci(void);
extern void ua_init(void);
extern void PrintString(const char *s);
extern BYTE Download(void);
extern BYTE EmbeddedProgram(LONG address, BYTE val);
extern BYTE ChipErase(void);
```

Section 8
Third Party Development Tools and Demo Software

CONTENTS

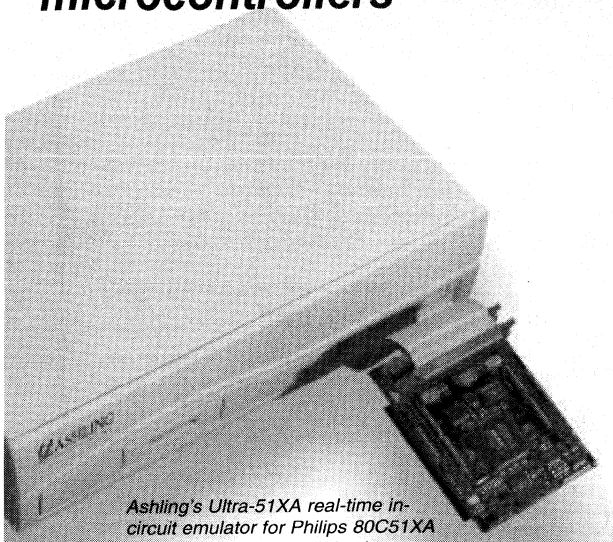
XA microcontroller development tools

COMPANY	WORLD WIDE WEB	TELEPHONE	FAX
Software: Compilers, Assemblers, Simulators			
Avocet Systems, Inc.	www.midcoast.com/~avocet/	1.207.236.9055	1.207.236.6713
Ceibo/Pantasoft	www.ceibo.com	1.314.830.4084 +972.9.9555387	1.314.830.4083 +972.9.9553297
CMX	www.cmx.com	1.508.872.7675	1.508.620.6828
Philips Semiconductors*	www.semiconductors.philips.com	1.408.991.51XA	1.408.991.3773
Tasking	www.tasking.nl	1.781.320.9400	1.781.320.9212
Emulators (including Debuggers)			
Ashling Microsystems	www.ashling.com	1.408.747.0440 +353.61.334466	1.408.747.0688 +353.61.334477
Ceibo	www.ceibo.com	1.314.830.4084 +972.9.9555387	1.314.830.4083 +972.9.9553297
Nohau Corp.	www.nohau.com/nohau/	1.408.866.1820 +46.40.592200	1.408.378.7869 +46.40.592229
Programmers			
Advin Systems	www.wco.com/~advin/	1.408.243.7000	1.408.736.2503
BP Microsystems	www.bpmicro.com	1.713.688.2675	1.713.688.0920
Ceibo	www.ceibo.com	1.314.830.4084 +972.9.9555387	1.314.830.4083 +972.9.9553297
Data I/O Corp.	sirius.data-io.com	1.425.881.6444	1.425.882.1043
Philips NZ	www.he.net/~pds/	1.408.991.51XA	1.408.991.3773
Programming Adapters			
EDI Corp.	–	1.702.735.4997	1.702.735.8339
Logical Systems	www.logicalsys.com	1.315.478.0722	1.315.479.6753
Translators			
Philips Semiconductors*	www.semiconductors.philips.com	1.408.991.51XA	1.408.991.3773
Real-Time Operating Systems			
CMX	www.cmx.com	1.508.872.7675	1.508.620.6828
Embedded Systems Products	www.esphou.com	1.281.561.9990	1.281.561.9980
R&D Books	www.rdbooks.com	1.785.841.1631	1.408.848.5784
Development Boards			
Future Designs, Inc.	members.aol.com/ teamfdi/teamfdi.htm	1.205.830.4116	1.205.830.9421
CMX	www.cmx.com	1.508.872.7675	1.508.620.6828
Philips NZ	www.he.net/~pds/	1.408.991.51XA	1.408.991.3773

* The cross assembler, simulator, and translator are available on the Philips Microcontroller Support File site at www.philipsmcu.com/. The file name is XA-TOOLS.ZIP

Ultra-51XA

Real-time in-circuit emulator for Philips 80C51XA microcontrollers



Ashling's Ultra-51XA real-time in-circuit emulator for Philips 80C51XA microcontrollers and derivatives.

Real-time in-circuit emulator

Ashling's in-circuit emulators provide accurate, real-time emulation of the target microcontroller. Your application-program is monitored and traced while it executes at full clock speed, without wait-states, and using no target resources. Ashling's emulators for the 80C51XA family provide full real-time in-circuit emulation, source-level debugging, performance analysis and software quality assurance tools for rapid and reliable development of 80C51XA applications.

Modular stand-alone solution

New embedded applications often require a new processor family or a new derivative. Ashling's emulator architecture provides support for a wide range of derivatives; a simple board swap will support a new microcontroller. A single probe supports both single-chip and expanded modes. Ashling's in-circuit emulators operate in stand-alone mode and are supplied with their own power unit.

Key features

- Real-time in-circuit emulator for Philips 80C51XA microcontrollers and derivatives
- *PathFinder for Windows* source-level debugger and user interface
- Supports program development in C and assembler
- On-the-fly setup and display
- *STARS* real-time, non-statistical Performance Analyser
- *CodeScan* Code Coverage measurement system
- 32K frame trace expandable to 512K; time-stamp on every traced frame
- Modular stand-alone system with power supply
- Interchangeable probe cards for all derivatives and packages
- Full hardware and software development environment for Philips 80C51XA
- Developed in co-operation with Philips Semiconductors

Source-level debugger

Ashling's *PathFinder for Windows* software provides the source-level debugger and user-interface for the *Ultra-51XA* emulator. Controlled by button-bar, menus, mouse or automated command files, *PathFinder* supports all 80C51XA compilers and assemblers including Hi-Tech, Tasking and Hiware. *PathFinder* provides Special Function Registers (SFR), internal RAM, memory, variables, user stack, system stack and register windows.

Performance Analysis and Code Coverage options

Ashling's *STARS* Performance Analyser (Software Test, Analysis and Reporting System) is a built-in real-time, high-speed measurement instrument for embedded development. The *STARS* system allows the user to measure the minimum, maximum or total execution time of a function, module or program in an embedded application. *CodeScan* is an integrated Code Coverage Measurement option within the *PathFinder* debugger.

ASHLING
THE DEVELOPMENT SYSTEMS COMPANY

Ultra-51XA

System Specification

Clock

- Up to 40MHz

Emulator memory

- 256KB code memory, expandable to 16MB
- 256KB data memory, expandable to 16MB
- Symbolic disassembly supported
- "On-the-fly" monitoring, using dual-port RAM

Mapping to target

- 128 byte resolution

Triggers

- 6 multiple trigger event recognisers
- Up to 10 addresses and/or data ranges per event
- Symbolic, binary or hex entry
- Boolean combinations of events for start/stop triggers
- Pre/post/centre triggering

Trace

- 32K frames by 96 bits (expandable to 512K)
- 40 bit time-stamp on all frames
- Display trace in C source, assembly or cycle-by-cycle
- All on-chip ports are traced
- 8-bit external trace port

Breakpoints

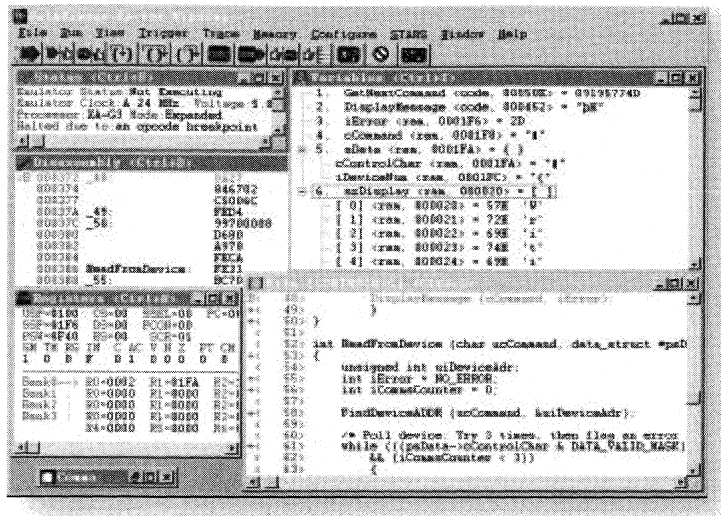
- Break-before-execute breakpoints (no "skid")
- 256K instruction execution breakpoints
- 256K data read and write breakpoints
- Break on frames after stop trigger
- Break on trace buffer full
- Execution timer, external break input

Single stepping

- Single step
- Multiple instruction stepping
- Step over, into or out of a function

Power supply; PC interface

- Supplied with 100V - 230V 50/60Hz universal power unit
- Standard RS232C or optional high-speed serial link to PC



PathFinder for Windows provides real-time in-circuit emulation and source-level debugging for the 80C51XA family, with mouse, command-line, accelerator-key or button-bar controls.

ROM	ROMless	EPROM
P51XAG13	X	P51XAG17
P51XAG23	X	P51XAG27
P51XAG33	P51XAG30	P51XAG37
P51XAS33	P51XAS30	P51XAS37
SAA5800	X	X
SMART$_{XA}$	*(Smart Card, in planning)*	

Upgrade Path

All *Ultra-51XA* systems can be easily field-upgraded to a different processor type. Ashling's Development Support Co-operation Agreement with Philips ensures that a full range of development-support tools is provided for each new XA or 80C51 derivative introduced by Philips.

ISO 9001 Certification

Ashling Microsystems is an ISO 9001 certified company.

Ashling Microsystems Ltd
National Technological Park
Limerick
Ireland
Tel: +353-61-334466
Fax: +353-61-334477
Email: ashling@iol.ie

Ashling Microsystèmes
2, rue Alexis de Tocqueville
Parc de Haute Technologie
92183 Antony Cedex, France
Tel: 01.46.66.27.50
Fax: 01.46.74.99.88
Email: ash-fr@world-net.sct.fr

Ashling Microsystems Ltd
Intec 2
Wade Road, Basingstoke
Hants. RG24 8NE, U.K.
Tel: (01256) 811998
Fax: (01256) 811761
Email: sales@ash-uk.demon.co.uk

Ashling Mikrosysteme
Brunnenweg 4
86415 Mering
Germany
Tel: 08233-32681
Fax: 08233-32682
Email: ashling.ger@t-online.de

Ashling Sales and Support Center
Orion Instruments, Inc.
1376 Borregas Avenue
Sunnyvale, CA 94089-1004, USA
Tel: (800) 729 7700
Fax: (408) 747 0688
Email: sales@ashling.com

Distributed by:

DISTRIBUTORS IN:
Australia, Austria, Belgium, France, Germany, Greece, Hungary, Ireland, Israel, Italy, Korea, Malaysia, Netherlands, Singapore, Spain, Sweden, Switzerland, Taiwan, Turkey, United Kingdom and USA.

ASHLING
THE DEVELOPMENT SYSTEMS COMPANY

http://www.ashling.com

ASHLING
THE DEVELOPMENT SYSTEMS COMPANY

CodeScan Code Coverage Measurement System for Embedded Microcontroller Development

Specification

Systematic and rigorous testing of newly-developed or newly-modified code is a vital part of any Software Quality Assurance or Software Lifecycle Management Procedure. Ashling's **CodeScan** Code Coverage measurement system automates and records real-time Code Coverage measurement for designers using the 8051, C500, 80C51XA, 68HC12, MELPS 7700 and TriCore microcontroller families for embedded systems development.

- **CodeScan** is installed as an optional sub-system in Ashling's Ultra-series Emulators.
- Provides a record of all code addresses accessed by your program and displays it in Ashling's **PathFinder for Windows** source-debugger display.
- Code not executed or tested can be located at-a-glance.
- Code Coverage data can be saved and re-loaded.
- Code Coverage data can be accumulated over several test sessions.
- Code blocks which have been executed are displayed in both the Source and Disassembled Code Windows.
- Code Coverage results can be logged and included in a printed report.
- No intrusion on or modification to your code
- Hosted under Windows™, Windows®95 or Windows™NT.

Benefits

- Helps ensure that all code has been executed during testing.
- Unused or un-reachable code blocks or functions can be isolated easily and quickly, thus improving code memory utilisation and efficiency.
- Using the CodeScan report generator, Code Coverage and test information can be included in product completion reports, to document the extent of system verification and validation.
- Non-intrusive, non-instrumented system; measures production code in real-time.

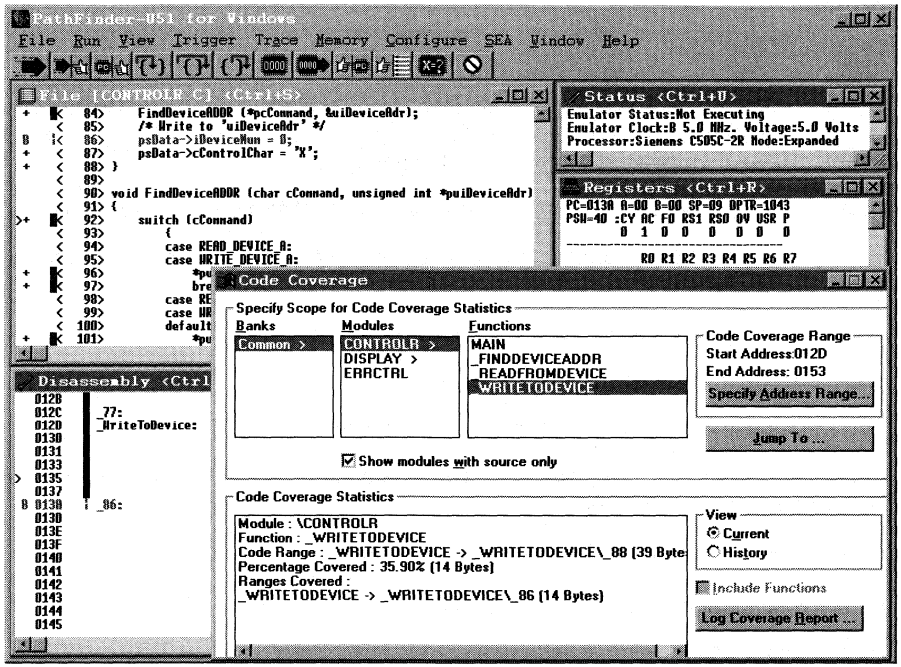
Brief Description

CodeScan is a real-time code execution analyser for 8-bit, 16-bit and 32-bit microcontroller systems. It is fully integrated into Ashling's **PathFinder for Windows** Source Level Debugger software.

Using a simple menu option, Code Coverage can be set up and analysed. By choosing to tag the source and disassembly windows with coverage data, you can determine what code has been executed by simply scrolling through either the source code or disassembly windows.
The View Code Coverage menu option enables you to analyse your complete embedded program at a glance through a flexible interactive window interface. The system's Code Coverage display window shows you all of the functions and code blocks which have not been executed.

Report Generation

The CodeScan Code Coverage report generation facility enables you to easily integrate your Code-Coverage and test results with any project report. This can be used to verify that your product has been tested to the required depth.



***CodeScan** Code Coverage is a real-time code execution analyser available in Ashling's STARS Performance Analyser, which is integrated in Ashling's Ultra-Series in-circuit emulators. The solid ■ marker indicates executed (tested) code in both source code and assembly -instruction representations.*

Accumulating Code Coverage Results

CodeScan can load, save and merge code coverage data over several test sessions; you can continue testing over a period of time, thereby building systematically towards a level of 100% test verification and confirmation.

Ashling Microsystems Ltd. is certified to I.S. EN ISO 9001, NSAI Registration No. M619

Ashling Microsystems Ltd
National Technological Park
Limerick
Ireland
Tel: +353-61-334466
Fax: +353-61-334477
Email:ashling@iol.ie

Ashling Microsystèmes
2, rue Alexis de Tocqueville
Parc de Haute Technologie
92183 Antony Cedex, France
Tel: 01.46.66.27.50
Fax: 01.46.74.99.88
ash-fr@world-net.sct.fr

Ashling Microsystems Ltd
Intec 2
Wade Road, Basingstoke
Hants. RG24 8NE, U.K.
Tel: (01256) 811998
Fax: (01256) 811761
sales@ash-uk.demon.co.uk

Ashling Mikrosysteme
Brunnenweg 4
86415 Mering
Germany
Tel: 08233-32681
Fax: 08233-32682
ashling.ger@t-online.de

Ashling Sales and Support Center
Orion Instruments, Inc.
1376 Borregas Avenue
Sunnyvale, CA 94089-1004, USA
Tel: (800) 729 7700
Fax: (408) 747 0688
sales@ashling.com

Distributors in Australia, Austria, Belgium, France, Germany, Greece, Hungary, Ireland, Israel, Italy, Korea, Malaysia, Netherlands, Singapore, Spain, Sweden, Switzerland, Taiwan, Turkey, United Kingdom, USA.

http://www.ashling.com

Doc: Ds183.doc Mar 98 Rev 1.1

ASHLING
THE DEVELOPMENT SYSTEMS COMPANY

ASHLING
THE DEVELOPMENT SYSTEMS COMPANY

STARS Performance Analyser for Embedded Microcontroller Development

Ashling's STARS Performance Analyser (Software Test, Analysis and Reporting System) is a high speed execution analyser for embedded systems using the 80C51, C500, 80C51XA, 68HC12, MELPS 7700 and TriCore microcontroller families. STARS is non-intrusive, non-statistical and measures all events accurately and securely.

STARS Features

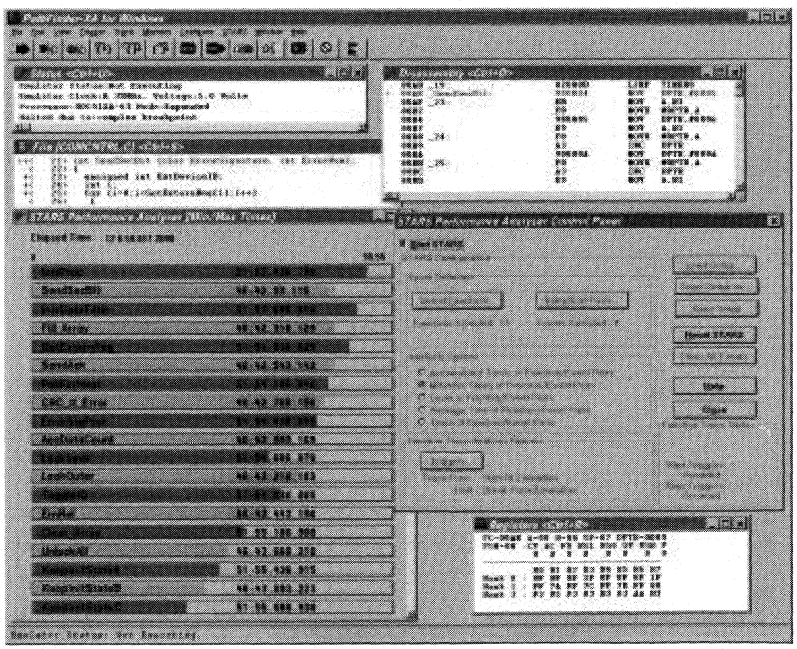
Function Selection: Functions whose performance you wish to analyse are selected from a list of the C functions in your program. The entry and exit points of each function are automatically selected (multiple exits are supported). Alternatively, you can select code sections for analysis by specifying symbolic or absolute entry and exit addresses.

Performance Analysis: A number of performance measurements can be derived from the functions selected. The total actual execution time, and the percentage of overall time used by each function is measured and displayed as a bar-graph with numerical results. The number of occurrences of each function can also be recorded. Exact minimum and maximum times spent in each code section are measured and used to highlight any out-of-specification performance.

Function Trace with Time-stamp: The STARS Performance Analyser records and time-stamps program execution flow from function to function. The calling sequence of functions and the absolute or relative times of all entries and exits are displayed, showing the run-time nesting of calls. Start and stop triggers can be used to selectively qualify the data collected.

Operation: Data is read without code instrumentation, without halting, slowing or in any way affecting program execution.

Logging of results and report generation: STARS results can be logged to disk and a report of system performance results can be prepared.



The STARS Performance Analyser is integrated into Ashling's Ultra-series In-Circuit Emulator and PathFinder for Windows source-level debugger

Summary Description

When using the STARS Performance Analyser in its Min/Max mode, the worse case execution time of individual events is displayed.

A Function Trace provides a time-stamped view of code execution, including call sequence and nesting. It can be used to see if a sequence of functions was performed in the correct order and in a particular time. Function Trace also increases productivity in debugging by displaying a top-level view of the code design. Function Trace also displays task execution and allocation in a Real-Time Operating System (RTOS). Report generators provide documentary evidence of system analysis, as required by many Software Quality Assurance procedures.

Specification

Hardware

The STARS Performance Analyser is installed as a hardware and software option for **Ashling's Ultra-Series** Microprocessor Development Systems.

Software

The STARS Performance Analyser software is integrated in Ashling's **PathFinder for Windows** source-level debugger.

Measurement Options

Measures continuously, in real-time, the following parameters for **up to 8192 functions** in the program-under-test:

Performance analysis measurement

- Accumulated time
- Number of times entered and exited
- Average time
- Minimum and maximum times
- For the time measurements, the user may choose to include or exclude time spent in called functions and interrupt routines

Function Trace provides user-selectable triggering

- Start trigger
  - Entry into emulation
  - Execution address
  - On a start trigger from the emulator trigger system (cross triggering)
- Stop trigger
  - On exit from emulation
  - Execution address
  - On a stop trigger from emulator trigger system (cross-triggering)
  - When trace buffer is full

User interface

PathFinder for windows Source Level Debugger: user-controller multi-window emulation control system. Up to 19 active windows. User control of size, position and on-screen combination. Mouse, menu-bar, and function key control. On-line context-sensitive hypertext help.

Host

PC with Windows™, windows®95 or Windows™NT, 4MB RAM.

Ashling Microsystems Ltd. is certified to I.S. EN ISO 9001, NSAI Registration No. M619

Ashling Microsystems Ltd
National Technological Park
Limerick
Ireland
Tel: +353-61-334466
Fax: +353-61-334477
Emails:ashling@iol.ie

Ashling Microsystèmes
2, rue Alexis de Tocqueville
Parc de Haute Technologie
92183 Antony Cedex, France
Tel: 01.46.66.27.50
Fax: 01 46.74.99.88
ash-fr@world-net.sct.fr

Ashling Microsystems Ltd
Intec 2,
Wade Road, Basingstoke
Hants. RG24 8NE, U.K.
Tel: (01256) 811998
Fax: (01256) 811761
sales@ash-uk.demon.co.uk

Ashling Mikrosysteme
Brunnenweg 4
86415 Mering
Germany
Tel: 08233-32681
Fax: 08233-32682
ashling.ger@t-online.de

Ashling Sales and Support Center
Orion Instruments, Inc.
1376 Borregas Avenue
Sunnyvale, CA 94089-1004, USA
Tel: (800) 729 7700
Fax: (408) 747 0688
sales@ashling.com

Distributors in Australia, Austria, Belgium, France, Germany, Hungary, Ireland, Israel, Italy, Korea, Malaysia, Netherlands, Singapore, Spain, Sweden, Switzerland, Taiwan, Turkey, United Kingdom, USA.

http://www.ashling.com

doc:DS184.doc Mar 98 ver 1.1 DS/184

ASHLING
THE DEVELOPMENT SYSTEMS COMPANY

ASHLING WORLDWIDE
International Product Managers

;TRALIA
ıam Sargent
'omatics Pty. Ltd.
3ox 1258, Eagle Farm
ɔane, Queensland 4009
Tel: 073868-4255
Fax: 073868-4147
Email: gsargent@metromatics.com.au

;TRIA
fgang Seeger
lling Mikrosysteme
ınenweg 4
15 Mering, Germany
Tel: 0049-8233/32681
Fax: 0049-8233/32682
Email: ashling.ger@t-online.de

.GIUM
ırice van Thiel
Parts International B.V.
nue Huart-Hamoirlaan 1
:34
030 Bruxelles - Brussels
Tel: 02 241 64 60
Fax: 02 241 81 30
Email: thielm@pdt.air-parts.com

ANCE
sín O'Keeffe
ıling Microsystèmes
ue Alexis de Tocqueville
c de Haute Technologie
83 Antony cedex
Tel: 01.46.66.27.50
Fax: 01.46.74.99.88
Email: ash-fr@world-net.sct.fr

RMANY
olfgang Seeger
hling Mikrosysteme
innenweg 4
415 Mering
Tel: 08233/32681
Fax: 08233/32682
Email: ashling.ger@t-online.de

IEECE
nagiotis Tsolakidis
ntech
Tsimiski Str.
C. 54623
essaloniki
Tel: 031-278692
Fax: 031-284691
Email: pantech@mob.forthnet.gr

JNGARY
eorge Bisztray
inGuard Kft.
1537 Budapest
O.B. 414
Tel: 30-335985
Fax: 1-3953661

ELAND
ary Hynes
shling Microsystems Ltd
ational Technological Park
nerick
Tel: 061-334466
Fax: 061-334477
Email: ashling@iol.ie

RAEL
vi Rosenwieg
DT Equipment and Systems Ltd.
tidim" Technological Park
O.Box 58072,
el Aviv 61580
Tel: 03-6450783
Fax: 03-6478908
Email: ravi@ankor.co.il

'ALY
ntonio Boldrin
II-Data srl
ia Volontari del Sangue 11
0092 Cinisello Balsamo (MI)
Tel: 02-66015566
Fax: 02-66015577
Email: 75143.1055@compuserve.com

.OREA
ilbert Ko
laSan Technology
35-26, Haesung B/D, 3F
amsung-Dong, Kangnam-Gu
eoul
Tel: (02) 511 9846
Fax: (02) 511 9845
Email: dasan@soback.kornet.nm.kr

MALAYSIA
Ooi Chin Keong
Flash Technology Sdn. Bhd.
70-1, Jalan 8/62A
Bandar Sri Menjalara
52200 Kuala Lumpur
Tel: 603-6367924
Fax: 603-6324046
Email: ooick@pc.jaring.my

NETHERLANDS
Maurice van Thiel
Air Parts B.V.
PO Box 255
2400 AG Alphen-aan-den-Rijn
Tel: 0172-422455
Fax: 0172-421022
Email: thielm@pdt..air-parts.com

SINGAPORE
Teo Kian Hock
Flash Technology Pte. Ltd.
Block 3006 #04-386
Ubi Road 1
Singapore 408700
Tel: 749 6168
Fax: 749 6138
Email: flashsgp@pacific.net.sg

SPAIN
John Sessler
Sistemas Jasper s.l.
Foresta 17, 3G
28760 Tres Cantos (Madrid)
Tel: (91) 803 8526
Fax: (91) 804 1623
Email: 100136.3561@compuserve.com

SWEDEN
Jonny Svensson
Ferner Elektronic AB
Box 600
175 26 Jarfalla
Tel: 08-76083 60
Fax: 08-76083 41
Email: jonny@ferner.se

SWITZERLAND
Roland Lips
Litronic AG
Fiechthagstrasse. 19
CH-4103 Bottmingen
Tel: 061 426 90 90
Fax: 061 426 90 99
Email: sales@litronic.ch

TAIWAN
Manfred Lai
Chinatech Corporation
5F, No.10, Alley 6, Lane 45,
Pao Hsing Rd., Hsin Tien City
Taipei Hsien
Tel: (02) 9160977
Fax: (02) 9126641
Email: chntech@ms2.hinet.net

TURKEY
Yildirim Acar
OTES Elektronik Ltd
Inonu Cad. Turaboglu Sk SUMKO Sit
A3-Blok D
1 Kozyatagi-Istanbul
Tel: 0216-463 76 29
Fax: 0216-463 76 30
Email: otes@usa.net

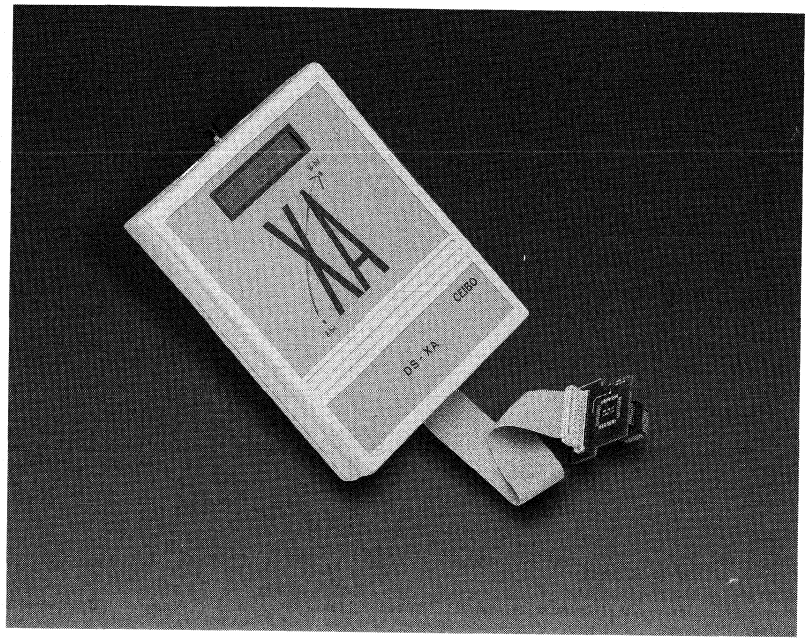
UNITED KINGDOM
Tony Parker
Ashling Microsystems Ltd
Intec 2
Wade Road, Basingstoke
Hants. RG24 8NE
Tel: (01256) 811998
Fax: (01256) 811761
Email: sales@ash-uk.demon.co.uk

USA
Carmel McGroarty
Ashling Sales and Support Center
Orion Instruments, Inc.
1376 Borregas Avenue
Sunnyvale, CA 94089-1004
Tel: (800) 729 7700
Tel: (408) 747 0440
Fax: (408) 747 0688
Email: sales@ashling.com

All other countries, please contact:
Ashling Microsystems Ltd
National Technological Park
Limerick
Ireland
Tel: +353-61-334466
Fax: +353-61-334477
Email: ashling@iol.ie

CEIBO

DS-XA In-Circuit Emulator



In-Circuit Emulator for Philips XA Microcontrollers

FEATURES

- Real-Time and Transparent In-Circuit Emulator
- 3.3V and 5V Microcontroller Emulation
- Maximum Frequency of 30MHz
- 64K to 2M of Internal Memory
- 32K to 512K Trace Memory and Logic Analyzer "on the Fly"
- External and Internal Trace Triggers
- Hardware and Conditional Breakpoints
- Source-Level Debugger for Assembler and C
- On-line Assembler and Disassembler
- MS-Windows Debugger
- Performance Analyzer

Serially linked to IBM PC at 115 KBaud

CEIBO
Development Tools of Choice

www.ceibo.com

DESCRIPTION

Ceibo DS-XA is a real-time in-circuit emulator dedicated to the Philips XA family of microcontrollers. It is serially linked to a PC or compatible host at 115 KBaud and carries out a transparent emulation on the target microcontroller. DS-XA supports the low-voltage and 5 Volt XA derivatives. The emulator uses Philips' bond-out emulation technology. Its software includes Source-Level Debugger for Assembler and C, Performance Analyzer, On-line Assembler and Disassembler, Conditional Breakpoints, a Software Simulator and many other features. The Debugger runs under MS-Windows and supports full C and Assembler expression evaluation. DS-XA accepts files generated by PantaSoft C-Compiler and Assembler and other available compilers with IEEE-695 format. From your Assembler or C Source Code Screen you can run the program in real-time, specify a breakpoint, redefine the Program Counter, execute a line step or an assembly instruction, open a watch window to display variables, display the trace memory, registers and data. The number of windows that may be displayed is not limited. Systems are supplied with 64K to 2 MBytes of Internal Memory with mapping capabilities, Hardware Breakpoints, 32K to 512K Real-Time Trace Memory and Logic Analyzer with external test points. DS-XA emulates every XA derivative in the complete voltage and frequency range specified by the microcontroller. The minimum frequency is determined by the emulated chip characteristics, while maximum frequency is up to 30MHz.

SPECIFICATIONS

EMULATION MEMORY

DS-XA provides 64 KBytes to 2 MByte of internal memory with software mapping capabilities. The map resolution is 2K. Every 2K memory block may be defined as code or data and also as belonging to the emulator or target circuit.

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.

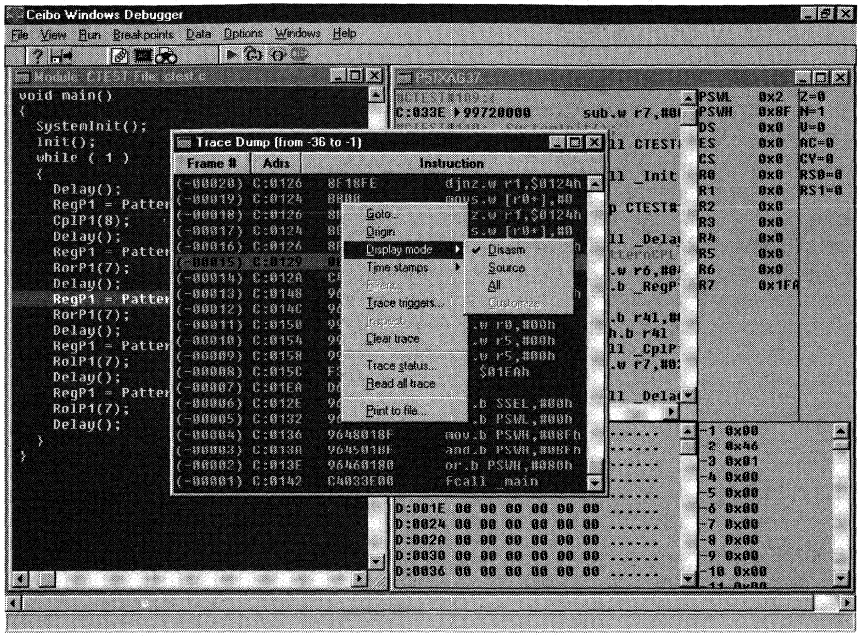
LANGUAGES AND FILE FORMATS

DS-XA accepts files with IEEE-695 and Intel Hex format. Assemblers and high-level languages such as PantaSoft ASM-XA and C-XA are fully supported.

SOURCE-LEVEL DEBUGGER

Ceibo Debug-XA Windows Debugger is a complete Source-Level Debugger for MS-Windows that 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.



DS-XA Software

PERSONALITY PROBES

DS-XA uses Philips bond-out microcontrollers for hardware and software emulation. The selection of a different XA derivative is made by replacing the probe or changing the software setup. 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 30MHz.

The available personality probes and supporting devices are:

Personality Probe	Supported Microcontroller
P-G3	XA-G3
P-S3	XA-S3
P-SCC	XA-SCC

As the list of supported devices and available probes is continuously evolving, call Ceibo to receive the latest update.

TRACE AND LOGIC ANALYZER
The 32 KByte to 512 KByte frames of 104 bits belonging to the Trace Memory are 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 without stopping the emulation ("on the fly").

PERFORMANCE ANALYZER
This useful function checks the trace buffer and provides time statistics on modules and procedures as a percentage of the total execution time.

HOST CHARACTERISTICS
IBM PC or compatible systems with 2 MBytes of RAM and one RS-232 Port.

INPUT POWER
5VDC/1.5A.

MECHANICAL DIMENSIONS
26mm x 151mm x 195mm (1" x 6" x 7").

ITEMS SUPPLIED AS STANDARD
In-Circuit Emulator with 64 KBytes of Internal Memory. Personality Probe for P51XAG3. Ceibo Debug-XA for MS-Windows. User's Manual. RS-232 Cable. Power Supply.

OPTIONS
1 or 2 MBytes of Internal Memory. 128K or 512K Trace Memory. Personality Probes and adapters for different microcontroller derivatives and packages.

WARRANTY
Two years limited warranty, parts and labor.

ADDRESSES

For more information contact us today:
Toll Free - U.S.A and Canada : 1-800-833-4084
Email : info@ceibo.com
Web : www.ceibo.com

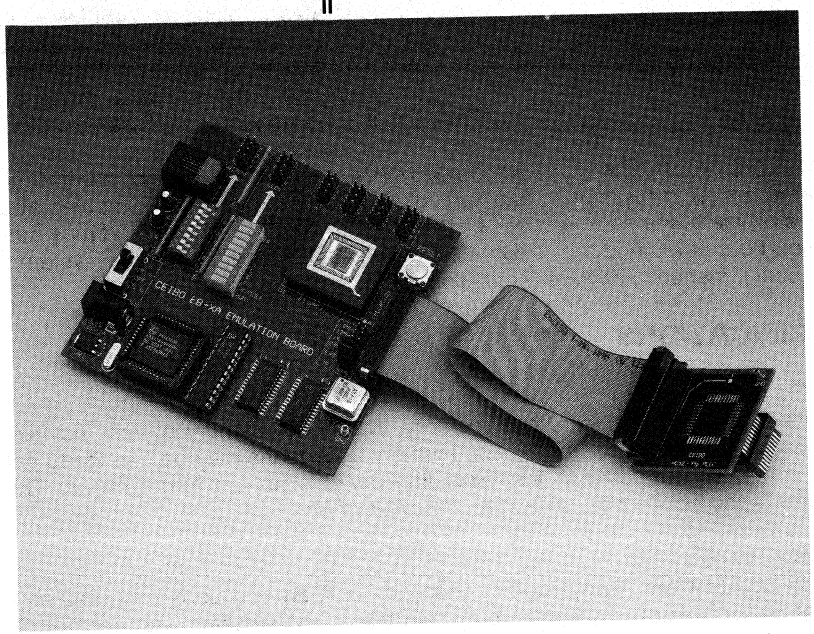
CEIBO USA
TEL: 314-830-4084
FAX: 314-830-4083
7 Edgestone Court
Florissant MO 63033
Email : usa@ceibo.com

CEIBO ISRAEL
TEL:+972-99-555387
FAX:+972-99-553297
32 Maskit St.
46120 Herzelia
Email : israel@ceibo.com

CEIBO EUROPE
Am Wehrhahn 86
D40211 Duesseldorf
Germany
TEL: +49-211-35 26 92
FAX: +49-211-16 24 28
Rufen Sie uns gebuehrenfrei an (Toll Free) : 800-3526-9300
Email : europe@ceibo.com

CEIBO

EB-XA Emulation Board



Development Tool for 80C51XA/G3 Microcontrollers

FEATURES

- Emulates 80C51XA/G3 Microcontrollers and Derivatives
- Real-Time Operation up to 30 MHz
- 3.3V or 5V Voltage Operation
- Source-Level Debugger for C and Assembler
- MS-Windows Debugger Software
- Reduced Set of C-Compiler and Assembler
- Support for ROMless and ROMed Microcontrollers
- 64K of Code Memory
- Performance Analyzer
- Real-Time and Conditional Breakpoints
- 44-pin PLCC Emulation Header and Signal Testpoints
- Serially Linked to IBM PC at 115 KBaud

CEIBO
Development Tools of Choice

www.ceibo.com

DESCRIPTION

EB-XA is an emulation board dedicated to all Philips 80C51XA/G3 microcontroller derivatives. It is serially linked to a PC 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, 14.7456MHz, 12MHz and 6MHz. The system emulates the microcontroller in ROMed mode while supporting split code mapping.. A special Philips bond-out chip is used to emulate the microcontroller, releasing all the microcontroller resources to the user. The software includes a Source-Level Debugger for C and Assembler, On-line Assembler and Disassembler, Software Trace, Conditional Breakpoints and many other features. The system includes a Debugger for MS-Windows. The code memory permits downloading and modifying of user's programs. 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-XA provides 64K of user code memory. This RAM memory permits downloading and modifying of user programs. The code memory boundaries may be defined to partially map the memory as belonging to the emulation board or to the target circuit. The software control sets the boundaries to 4K, 8K, 16K, 32K or 64K.

BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address. A breakpoint may be set to any address of the system code memory. Breakpoints on user target code addresses are possible if this memory can be written by the microcontroller.

USER SOFTWARE

The Windows Debugger runs under MS-Windows 3.1 or MS-Windows 95.

SYMBOLIC DEBUGGER

EB-XA allows symbolic debugging of assembler or high-level languages. The symbolic debugger uses symbols contained in the absolute file generated by the most commonly used Assemblers and high-level language Compilers.

SOURCE-LEVEL DEBUGGER

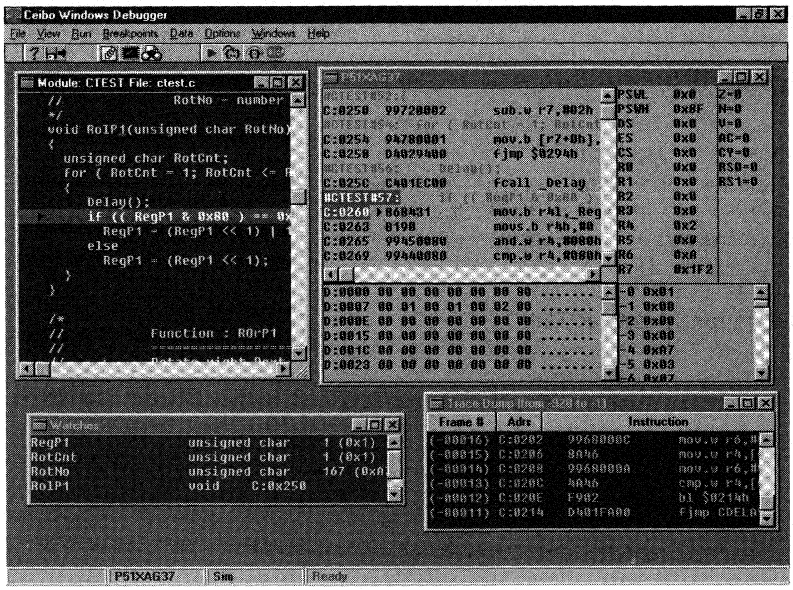
The EB-XA software includes a source-level debugger for Assembler and high-level languages (C and others) with the capability of executing lines of the program while displaying the state of any variable.

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. This function is enabled in simulation or in-circuit simulation modes.

IN-CIRCUIT SIMULATION MODE

The in-circuit simulation mode allows the software to be tested without any hardware. All the emulation functions are supported by this powerful simulation debugger.



EB-XA Windows Debugger

SUPPORTED MICROCONTROLLERS

The supported microcontrollers are all the Philips XA derivatives functional compatible with the current bond-out device and with up to 64K internal code memory.

MICROCONTROLLER SELECTION

EB-XA uses a Philips bond-out chip for hardware and software emulation. The selection of a supported microcontroller is done by means of software. The debugger menu is used to choose the desired emulated derivative. The minimum and maximum frequencies are determined by the bond-out chip characteristics, while the emulator maximum frequency is 30MHz.

FREQUENCY

The system includes a crystal oscillator able to supply clock frequencies of 6MHz, 12MHz, 24MHz and a fixed frequency of 14.7456MHz. Additionally, the user may select any other frequency by connecting an external clock source through the application hardware. The crystal oscillator itself is mounted on a socket and may be replaced by another oscillator with different frequency value. Frequency selection is done by means of jumpers.

EMULATION VOLTAGE

EB-XA emulates the microcontrollers at 3.3V or 5V. The selection of the voltage is defined by the position of the slide switch mounted on the emulator board.

HOST CHARACTERISTICS
PC or compatible systems with 2 MByte of RAM, one RS-232 interface card for the PC, MS-Windows 3.1 or later.

INPUT POWER
5V, 1.5A power supply supplied.

MECHANICAL DIMENSIONS
10cm x 10cm.

ITEMS SUPPLIED AS STANDARD
Development tool with 64 KByte Memory, 44-pin PLCC Emulation Header, Windows software including Source-Level Debugger, On-Line Assembler and Disassembler, User's Manual, RS-232 Cable and Power Supply.

WARRANTY
Two year limited warranty, parts and labor.

ADDRESSES

For more information contact us today:
Toll Free - U.S.A and Canada : 1-800-833-4084
Email : info@ceibo.com
Web : www.ceibo.com

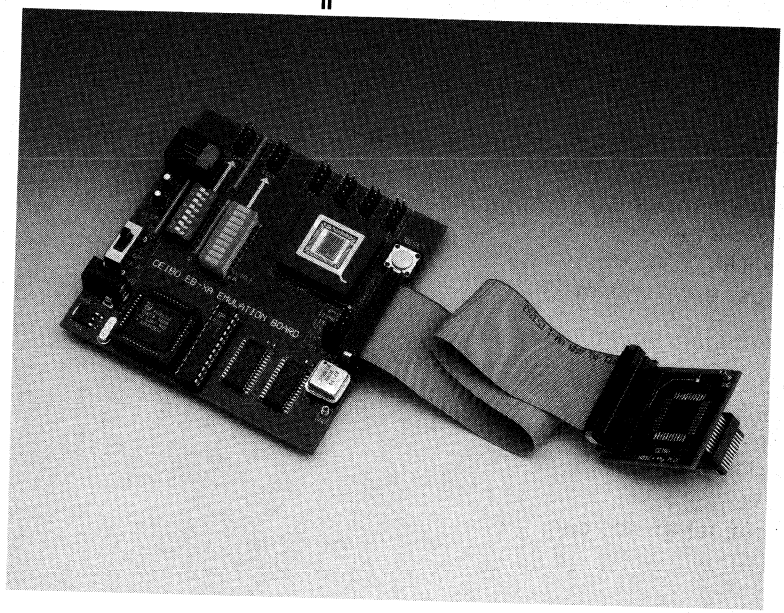
CEIBO USA
TEL: 314-830-4084
FAX: 314-830-4083
7 Edgestone Court
Florissant MO 63033
Email : usa@ceibo.com

CEIBO ISRAEL
TEL:+972-99-555387
FAX:+972-99-553297
32 Maskit St.
46120 Herzelia
Email : israel@ceibo.com

CEIBO EUROPE
Am Wehrhahn 86
D40211 Duesseldorf
Germany
TEL: +49-211-35 26 92
FAX: +49-211-16 24 28
Rufen Sie uns gebuehrenfrei an (Toll Free) : 800-3526-9300
Email : europe@ceibo.com

CEIBO

EB-XAS3 Emulation Board



Development Tool for 80C51XA/S3 Microcontrollers

FEATURES

- Emulates 80C51XA/S3 Microcontrollers and Derivatives
- Real-Time Operation up to 30 MHz
- 3.3V or 5V Voltage Operation
- Source-Level Debugger for C and Assembler
- MS-Windows Debugger Software
- Reduced Set of C-Compiler and Assembler
- Support for ROMless and ROMed Microcontrollers
- 64K of Code Memory
- Performance Analyzer
- Real-Time and Conditional Breakpoints
- 68-pin PLCC Emulation Header and Signal Testpoints
- Serially Linked to IBM PC at 115 KBaud

www.ceibo.com

DESCRIPTION

EB-XAS3 is an emulation board dedicated to all Philips 80C51XA/S3 microcontroller derivatives. It is serially linked to a PC 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, 14.7456MHz, 12MHz and 6MHz. The system emulates the microcontroller in ROMed mode while supporting split code mapping.. A special Philips bond-out chip is used to emulate the microcontroller, releasing all the microcontroller resources to the user. The software includes a Source-Level Debugger for C and Assembler, On-line Assembler and Disassembler, Software Trace, Conditional Breakpoints and many other features. The system includes a Debugger for MS-Windows. The code memory permits downloading and modifying of user's programs. 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-XAS3 provides 64K of user code memory. This RAM memory permits downloading and modifying of user programs. The code memory boundaries may be defined to partially map the memory as belonging to the emulation board or to the target circuit. The software control sets the boundaries to 4K, 8K, 16K, 32K or 64K.

BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address. A breakpoint may be set to any address of the system code memory. Breakpoints on user target code addresses are possible if this memory can be written by the microcontroller.

USER SOFTWARE

The Windows Debugger runs under MS-Windows 3.1 or MS-Windows 95.

SYMBOLIC DEBUGGER

EB-XAS3 allows symbolic debugging of assembler or high-level languages. The symbolic debugger uses symbols contained in the absolute file generated by the most commonly used Assemblers and high-level language Compilers.

SOURCE-LEVEL DEBUGGER

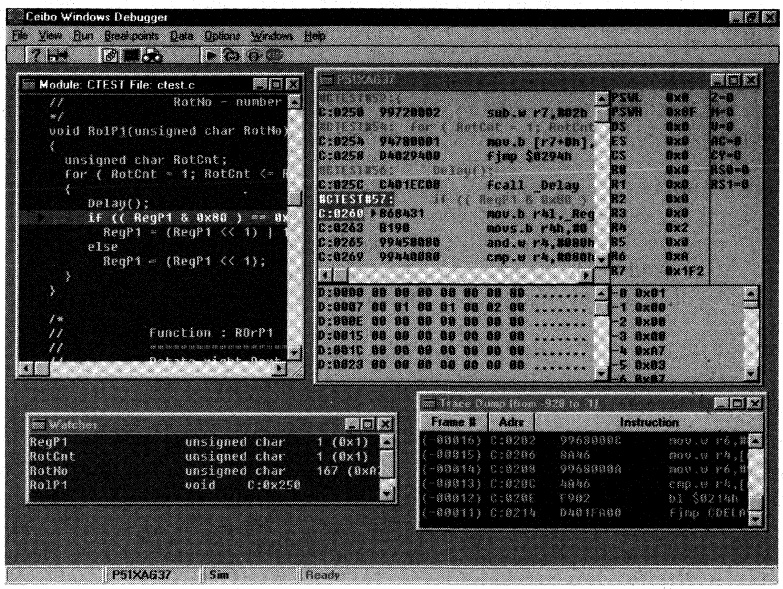
The EB-XAS3 software includes a source-level debugger for Assembler and high-level languages (C and others) with the capability of executing lines of the program while displaying the state of any variable.

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. This function is enabled in simulation or in-circuit simulation modes.

IN-CIRCUIT SIMULATION MODE

The in-circuit simulation mode allows the software to be tested without any hardware. All the emulation functions are supported by this powerful simulation debugger.



EB-XAS3 Windows Debugger

SUPPORTED MICROCONTROLLERS

The supported microcontrollers are all the Philips XAS3 derivatives functional compatible with the current bond-out device and with up to 64K internal code memory.

MICROCONTROLLER SELECTION

EB-XAS3 uses a Philips bond-out chip for hardware and software emulation. The selection of a supported microcontroller is done by means of software. The debugger menu is used to choose the desired emulated derivative. The minimum and maximum frequencies are determined by the bond-out chip characteristics, while the emulator maximum frequency is 30MHz.

FREQUENCY

The system includes a crystal oscillator able to supply clock frequencies of 6MHz, 12MHz, 24MHz and a fixed frequency of 14.7456MHz. Additionally, the user may select any other frequency by connecting an external clock source through the application hardware. The crystal oscillator itself is mounted on a socket and may be replaced by another oscillator with different frequency value. Frequency selection is done by means of jumpers.

EMULATION VOLTAGE

EB-XAS3 emulates the microcontrollers at 3.3V or 5V. The selection of the voltage is defined by the position of the slide switch mounted on the emulator board.

HOST CHARACTERISTICS
PC or compatible systems with 2 MByte of RAM, one RS-232 interface card for the PC, MS-Windows 3.1 or later.

INPUT POWER
5V, 1.5A power supply supplied.

MECHANICAL DIMENSIONS
10cm x 10cm.

ITEMS SUPPLIED AS STANDARD
Development tool with 64 KByte Memory, 68-pin PLCC Emulation Header, Windows software including Source-Level Debugger, On-Line Assembler and Disassembler, User's Manual, RS-232 Cable and Power Supply.

WARRANTY
Two year limited warranty, parts and labor.

ADDRESSES

For more information contact us today:
Toll Free - U.S.A and Canada : 1-800-833-4084
Email : info@ceibo.com
Web : www.ceibo.com

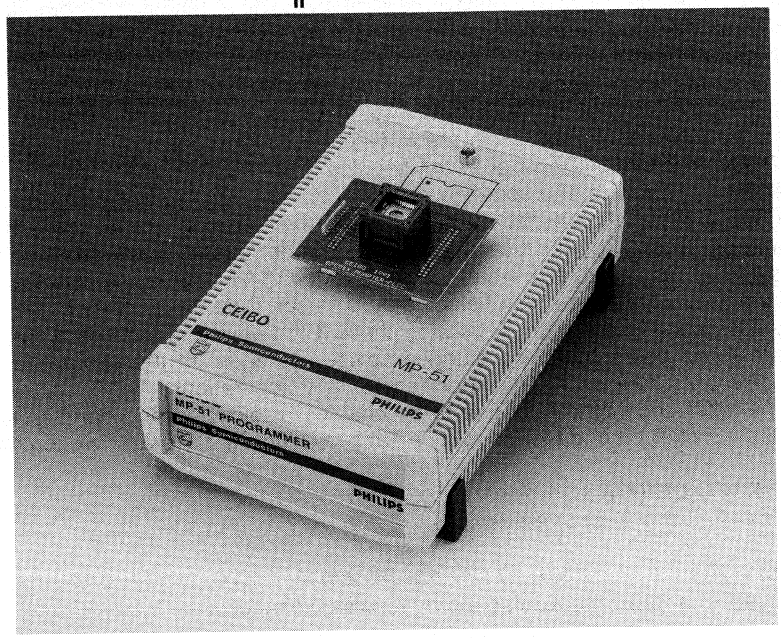
CEIBO USA
TEL: 314-830-4084
FAX: 314-830-4083
7 Edgestone Court
Florissant MO 63033
Email : usa@ceibo.com

CEIBO ISRAEL
TEL:+972-99-555387
FAX:+972-99-553297
32 Maskit St.
46120 Herzelia
Email : israel@ceibo.com

CEIBO EUROPE
Am Wehrhahn 86
D40211 Duesseldorf
Germany
TEL: +49-211-35 26 92
FAX: +49-211-16 24 28
Rufen Sie uns gebuehrenfrei an (Toll Free) : 800-3526-9300
Email : europe@ceibo.com

CEIBO

MP-51 Programmer



EPROM, Flash, PLD and Microcontroller Programmer

www.ceibo.com

FEATURES

- EPROM, Flash, PLD and Microcontroller Programmer
- Serially Linked to PC or Compatible Host
- Programs all the 8051 and XA Microcontrollers
- Programs 16-bit Microcontrollers
- Supports 24 to 32-pin EPROMs
- Programs 32-pin Flash Memories
- Programs PLD and PSD Devices
- Macros and Easy to Follow Menus
- Handles Hex, Binary, Object and JEDEC Files
- Programs DIP, QFP, LCC and PLCC Devices
- Supports Lock Bits, Encryption Tables and Security Bits
- Includes Format Converters

DESCRIPTION

Ceibo 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 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 for 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.

SUPPORTED DEVICES

Following is a 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.

8/16 BIT MICROCONTROLLERS: PCD3745A, PCD3755A, PCD3756A, 8751H, 8751BH, 87C51, 87C51FA, 87C51FB, 87C51FC, 87L51FA, 87L51FB, 87C51GB, P51XAG17, P51XAG27, P51XAG37, P51XAS3, 8752BH, 87C52, 87C54, 87C58, 87CL134, 87C251SA, 87C251SB, 87C251SP, 87C251SQ, 87C451, 87C453, 87C504, 87C508, 87C520, 87C524, 87C528, 87C530, 87C550, 87C552, 87C560, 87C575, 87C576, 87C592, 87C598, 87C652, 87C654, 87C748, 87C749, 87C750, 87C751, 87C752, 87C754, 87C766, 87C769, 87CL880, 87C054, 87C055, 89C51, 89C52, 89C55, 89C558, 89C1051, 89C2051, PCA5097.

PLDs: AT22V10, ATV750, ATV2500, ATV5000, ATV5100.

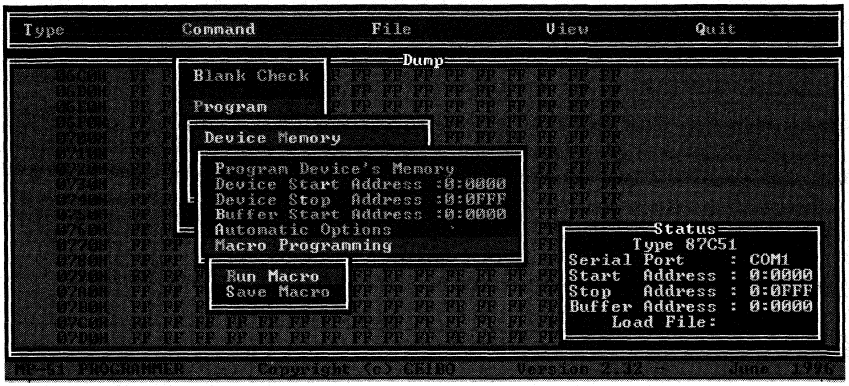
PSDs: PSD301, PSD301L, PSD302, PSD302L, PSD303, PSD303L, PSD311, PSD311L, PSD312, PSD312L, PSD313, PSD313L.

FILE FORMATS

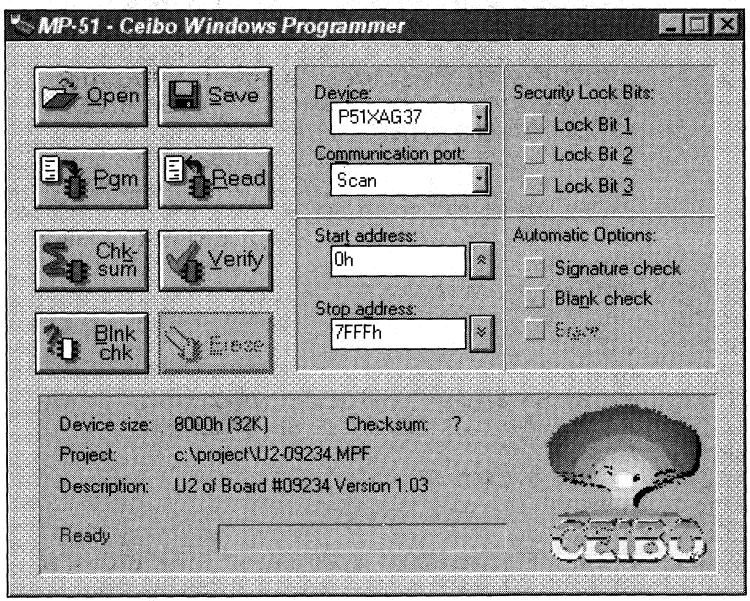
MP-51 loads different file formats: Intel Hex and Motorola S-records, Binary files, Object files and JEDEC files. It saves portions of memory in Intel, Motorola, Binary and JEDEC formats.

USER SOFTWARE

The MP-51 comes with Dos and MS-Windows software. The commands are powerful yet extremely user friendly.



MP-51 Dos Software



MP-51 Windows Software

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.

HOST CHARACTERISTICS

IBM PC or compatible systems with 512 KBytes of RAM, one floppy disk drive, one RS-232 interface card for the PC, PC-DOS 2.0 or later.

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.

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:	
27040D-32 PIN DIP	27010 to 27040, 28F256, 28F512, 28F010, 28F020.
27040P-32 PIN PLCC	27010 to 27040, 28F256, 28F512, 28F010, 28F020.
27512D-28 PIN DIP	2716 to 27512 NMOS and CMOS.
27512P-40 PIN DIP	2716 to 27512 NMOS and CMOS.
8/16 BIT Microcontrollers:	
C51D-40 PIN DIP	8751H, 8751BH, 87C51, 87C51FA, 87C51FB, 87C51FC, 87L51FA, 87L51FB, 87C52BH, 87C52, 87C54, 87C58, 87C504, 87C508, 87C520, 87C524, 87C528, 87C530, 87C550, 87C652, 87C654, 89C51, 89C52, 89C55.
C51P-44 PIN PLCC	8751H, 8751BH, 87C51,87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52BH, 87C52, 87C54, 87C58, 87C504, 87C508, 87C520, 87C524, 87C528, 87C530, 87C550, 87C652, 87C654, 89C51, 89C52, 89C55.
C51Q-44 PIN QFP	87C51, 87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52, 87C54, 87C58, 87C504, 87C508, 87C520, 87C524, 87C528, 87C530, 87C652, 87C654, 89C51, 89C52, 89C55
87C51GBP-68 PIN PLCC	87C51GB
P51XAG3P-44 PIN PLCC	P51XAG17, P51XAG27, P51XAG37
P51XAS3P-68 PIN PLCC	P51XAS3
87CL134D-42 PIN SDIP	87CL134
87CL134Q-44 PIN QFP	87CL134
87C251D-40 PIN DIP	87C251SA, 87C251SB, 87C251SP, 87C251SQ
87C251P-44 PIN PLCC	87C251SA, 87C251SB, 87C251SP, 87C251SQ
87C451D-64 PIN DIP	87C451
87C451P-68 PIN PLCC	87C451, 87C453
87C550P-44 PIN PLCC	87C550
87C552P-68 PIN PLCC	87C552
C575D-40 PIN DIP	87C575, 87C576, 87C51, 87C51FA, 87L51FA, 87C51FB, 87L51FB, 87C51FC, 87C52, 87C54, 87C58, 87C504, 87C508, 87C524, 87C528, 87C652, 87C654.
87C592P-68 PIN PLCC	87C592
87C598Q-80 PIN QFP	87C598, 89CE558, 89CE560
87C751D-24 PIN DIP	87C748, 87C750, 87C751
87C751P-28 PIN PLCC	87C748, 87C750, 87C751
87C752D-28 PIN DIP	87C749, 87C752
87C752P-28 PIN PLCC	87C749, 87C752
87C754D-28 PIN DIP	87C754
87C754S-28 PIN SSOP	87C754
87C766-42 PIN SDIP	87C766
87C769-52 PIN SDIP	87C769
87CL880Q-64 PIN QFP	87CL880
87C054D-42 PIN SDIP	87C054, 87C055
89C1051D-20 PIN DIP	89C1051, 89C2051

3755AD-28 PIN DIP	PCD3745A, PCD3755A, PCD3756A
3755AS-28 PIN SO	PCD3745A, PCD3755A, PCD3756A
3755AQ-32 PIN QFP	PCD3745A, PCD3755A, PCD3756A
5097Q-100 PIN QFP	PCA5097

PLDs:

ATV-24D-24 PIN DIP	AT22V10, ATV750
ATV-40D-40 PIN DIP	ATV2500
ATV-68P-68 PIN PLCC	ATV5000, ATV5100

PSDs:

301P-44 PIN PLCC	PSD301, PSD301L, PSD302, PSD302L, PSD303, PSD303L, PSD311, PSD311L, PSD312, PSD312L, PSD313, PSD313L

As the list of supported devices and available adapters is continuously evolving, call Ceibo to receive the latest update.

ITEMS SUPPLIED AS STANDARD

MP-51 Programmer, user software, user's manual, RS-232 cable and any two adapters. Power cord not included.

WARRANTY

Two years limited warranty, parts and labor.

ADDRESSES

For more information contact us today:
Toll Free - U.S.A and Canada : 1-800-833-4084
Email : info@ceibo.com
Web : www.ceibo.com

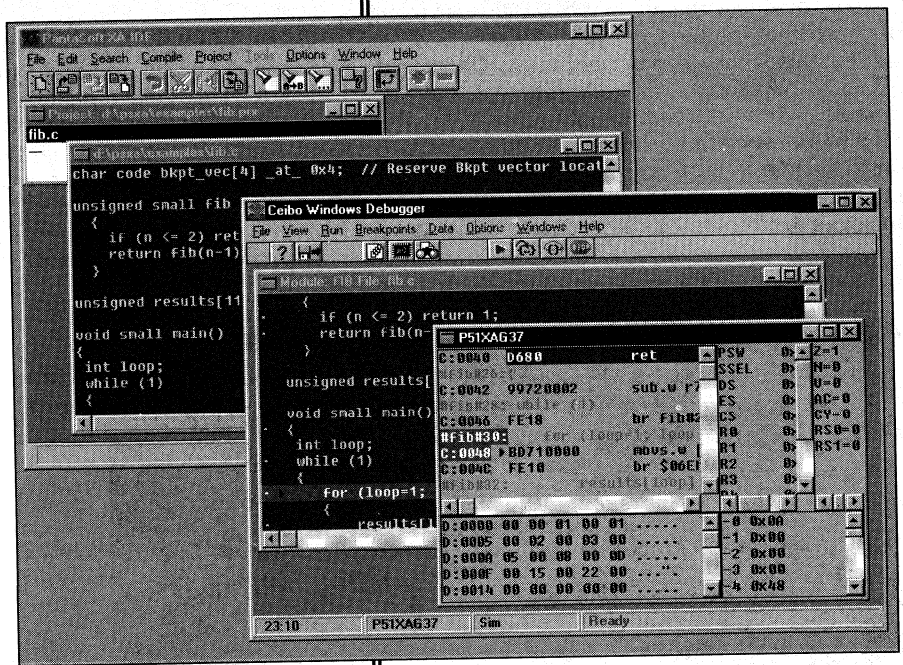
CEIBO USA
TEL: 314-830-4084
FAX: 314-830-4083
7 Edgestone Court
Florissant MO 63033
Email : usa@ceibo.com

CEIBO ISRAEL
TEL:+972-99-555387
FAX:+972-99-553297
32 Maskit St.
46120 Herzelia
Email : israel@ceibo.com

CEIBO EUROPE
Am Wehrhahn 86
D40211 Duesseldorf
Germany
TEL: +49-211-35 26 92
FAX: +49-211-16 24 28
Rufen Sie uns gebuehrenfrei an (Toll Free) : 800-3526-9300
Email : europe@ceibo.com

CEIBO

PantaSoft-XA Software Tools



Software Tools for Philips XA Microcontrollers

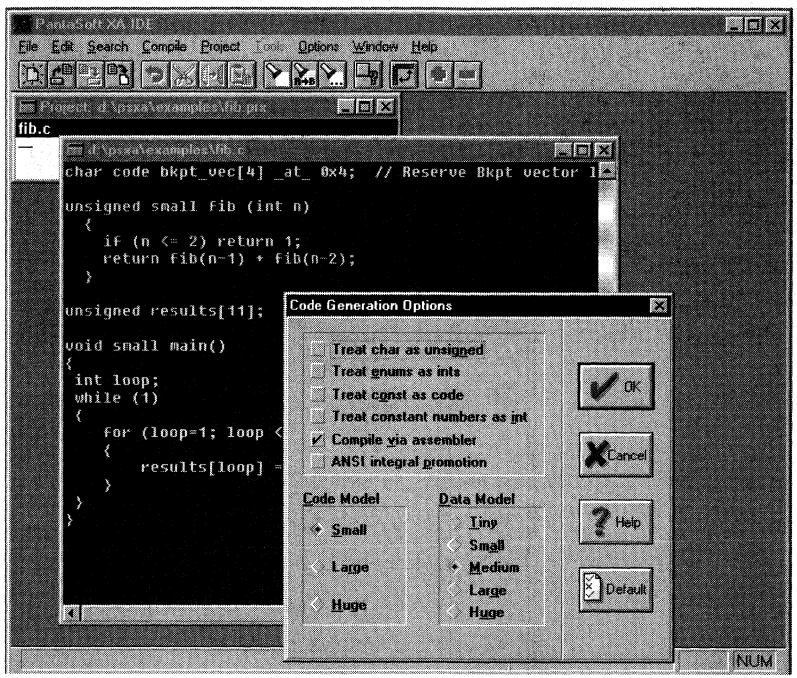
FEATURES

-
- ANSI Standard C Cross Compiler
- XA Relocatable Assembler
- XA Linker and Locator
- C Source-Level Simulator/Debugger
- Full MS-Windows Interface
- Extended Keywords Specific to XA
- Full Floating Point Support
- Direct C Interrupt Handling
- Built-In Optimizer
- Direct Interface to In-Circuit Emulators
- Full Embedded Run-Time Libraries
- Complete XA Architecture Support up to 16 MByte

www.ceibo.com

C-XA COMPILER

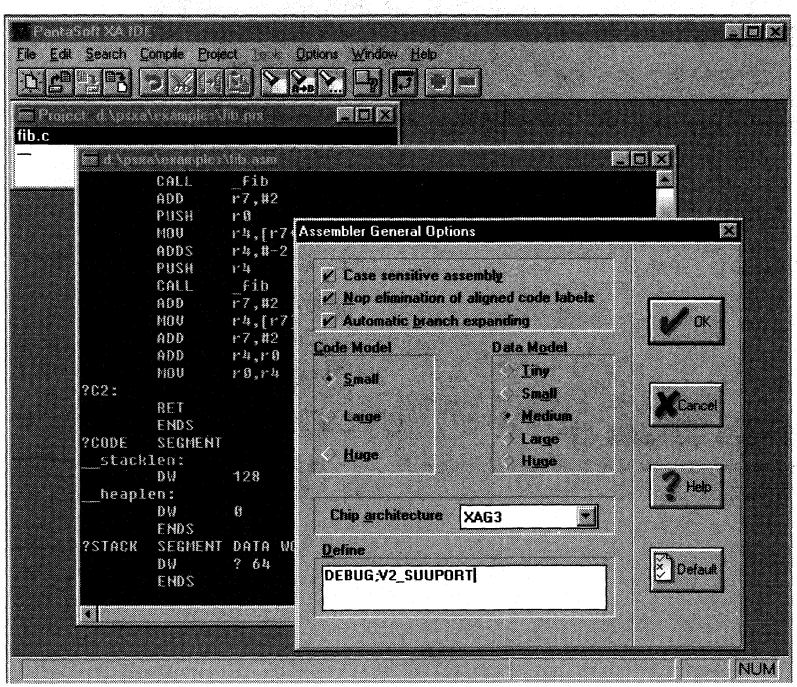
C-XA is an ANSI C compiler with extensions designed to support XA special features. The compiler is compatible with other ANSI C compilers. C-XA Compiler is designed to make the code faster and more compact by using the special chip features. A complete code and data memory models configuration is available to fit any application.
C-XA special features permit the user direct access from C source to the XA chip. The features also include direct access to the interrupt mechanism, as well as access to the XA SFRs. C-XA supports 9 different data types including floating point variables and bit variables. The C Compiler comes with a variety of the most frequently used C Library functions associated with embedded systems. Some of the low-level functions, like those which handle I/O, are provided with source code. The Compiler may be used with in-line assembler instructions for direct access to XA resources. High performance optimizer achieves the best results by reducing the code size.



C-XA

ASM-XA RELOCATABLE ASSEMBLER

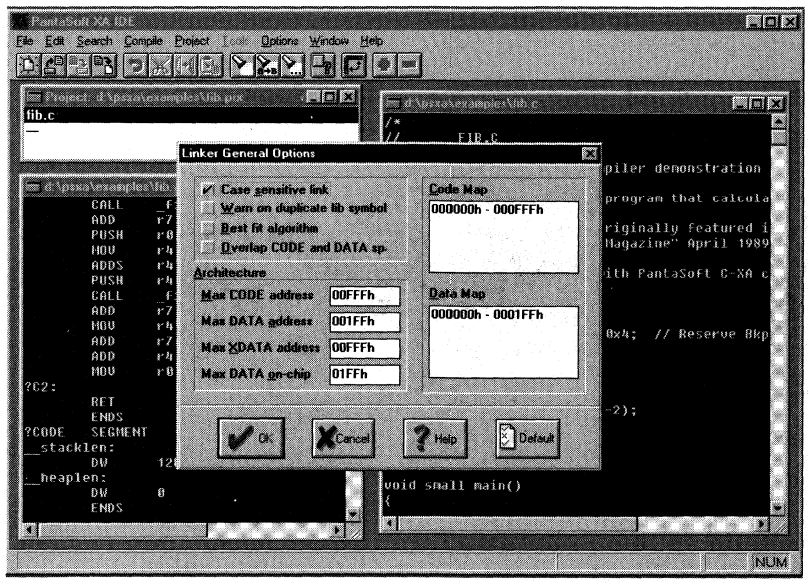
The ASM-XA Relocatable Assembler translates an XA Relocatable Assembly Language program into relocatable object code. The Relocatable Assembler includes commands and directives specially designed to fit the XA architecture. ASM-XA processes the input file and executes assembly directives and commands. The conditional compilation support may be used to include or exclude sections of your source while assembling the project. Special NOP optimization is provided to align code labels without affecting the speed performance.
The object code generated by the Assembler includes information about the symbols and lines used in the source file. It has been specially designed to facilitate easy translation of code from the 8051 Assembler. ASM-XA is integrated into a modern graphic interface, which paves the way to invoke tasks wanted for a complete development environment.



ASM-XA

LINK-XA LINKER AND LOCATOR

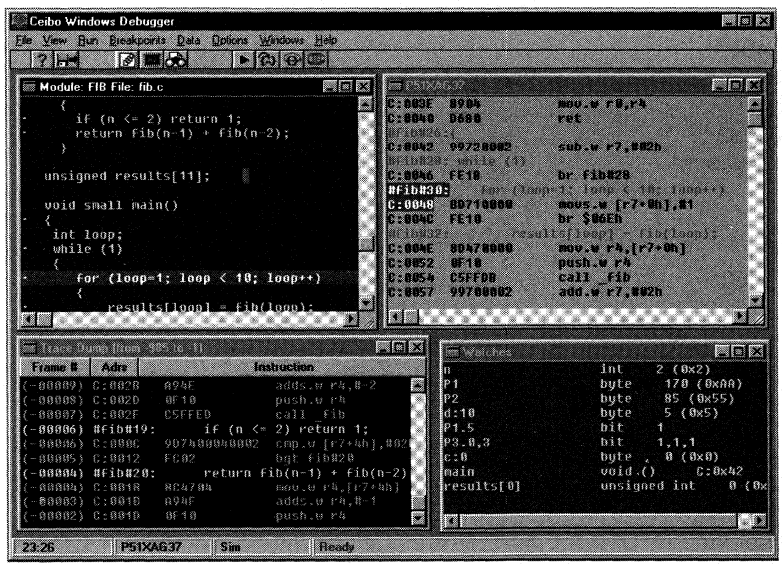
LINK-XA supports complete linkage, relocation and format generation for producing absolute object code. LINK-XA accesses only the requested modules in the library and combines them in the absolute object code. A complete Librarian utility maintains the libraries. The linker can combine object files created by the C-XA Compiler and ASM-XA Relocatable Assembler into one absolute file, as well as find the necessary objects from libraries created by the Librarian. LINK-XA can create the absolute file in several output formats, including Intel HEX and IEEE-695. The Linker can also create a detailed map file including information about the location of segments and symbols.



LINK-XA

DEBUG-XA DEBUGGER/SIMULATOR

DEBUG-XA is a Source-level Debugger/Simulator for the XA architecture. The program enables fast and reliable program debugging at source-level for C-XA and ASM-XA.
The Simulator/Debugger for the PantaSoft C-XA is fully source-level and controls the program flow in HLL or Assembler. The debugger operates with or without an in-circuit emulator. Without the emulator, all the functions are simulated in the PC. DEBUG XA can be used with Ceibo EB-XA Emulation Board and Ceibo DS-XA In-Circuit Emulator for complete real-time operation. The user can inspect variables using the watch window and set breakpoints in the source code. DEBUG-XA also includes an on-line assembler which the user can invoke to change the executable code during the debug session. DEBUG-XA is based on a Windows platform, so it may be used in a multi-tasking and multi-window application.



DEBUG-XA

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
Email : info@ceibo.com
Web : www.ceibo.com

CEIBO USA
TEL: 314-830-4084
FAX: 314-830-4083
7 Edgestone Court
Florissant MO 63033
Email : usa@ceibo.com

CEIBO ISRAEL
TEL:+972-99-555387
FAX:+972-99-553297
32 Maskit St.
46120 Herzelia
Email : israel@ceibo.com

CEIBO EUROPE
Am Wehrhahn 86
D40211 Duesseldorf
Germany
TEL: +49-211-35 26 92
FAX: +49-211-16 24 28
Rufen Sie uns gebuehrenfrei an (Toll Free) : 800-3526-9300
Email : europe@ceibo.com

These are just a few of the obstacles that a programmer encounters every day when writing code for a processor.

In some cases, well structured "linear" programming will be sufficient for a product. In most cases, though, a programmer would appreciate not having to worry about how to structure a "linear" program to perform all the tasks necessary in a timely manner.

This is where a Real-Time Operating System (RTOS) such as CMX's comes into play. An RTOS allows tasks, which are really just pieces of code that do specific duties, to run "quasi-concurrently". This means that tasks will seem to run all at the same time, doing many specific jobs simultaneously.

All operating systems are not created equal, however. Many OS's offer only co-operative scheduling, which means that the running task has to call the scheduler to perform a task switch. This takes an unnecessarily large amount of thought by the programmer to decide when a task should cause a task switch.
(How does the task decide?)

Other OS's offer time slicing, where each task runs for a certain period of time, at which point a task switch takes place no matter what.
(What happens if a task needs to complete something prior to the task switch? What about a high priority task needing service?)

Still other OS vendors CLAIM to be fully preemptive, yet they do not let ANY interrupt cause a preemption. They may just set a flag that indicates that a rescheduling must occur when the next scheduling tick occurs. Depending upon the scheduling rate and the correlation of when the flag was set, this could be a long time.
(Is this acceptable?)

A preemptive OS allows a task of higher priority that is ABLE to run, whether from the beginning of its code or resuming its code from where it was executing last, to preempt the lower priority running task. This will cause the scheduler to save the context of the running (lower priority) task, and restore the context of the higher priority task, so that the higher priority task is now the running task. A truly preemptive OS also allows interrupts cause an immediate task switch. What this means is that the interrupts now have the added ability of using the OS's functions.

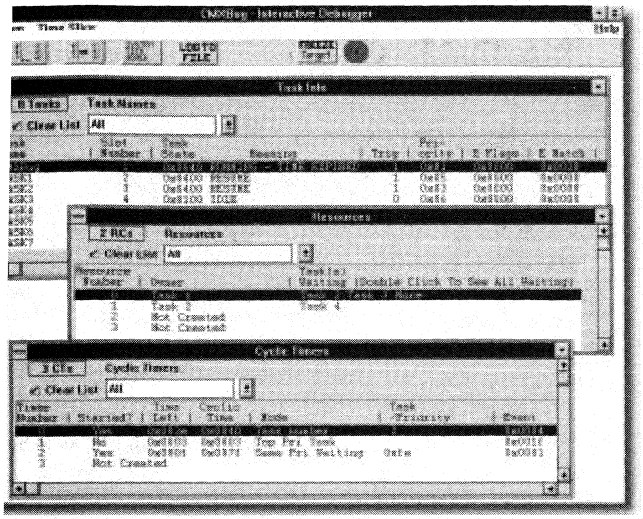
CMX Company offers such a true preemptive multitasking operating system to the programmer, giving the ability to work with interrupts and tasks in numerous ways. Allowing both tasks AND interrupts to utilize the powerful CMX functions, along with true pre-emption at the heart of the scheduler, and optional co-operative and time-slicing scheduling that can be turned on with merely the flick of a bit, means that the CMX-RTOS is a true winner in the field of embedded development not only in terms of functionality, but also speed and efficiency.

CMX tech support is infamous throughout the world as well. It's no wonder why, when taken into consideration the fact that over 95% of tech support calls are resolved over the phone. If the question can't be answered over the course of a phone call, as is the case with the remaining tech support calls, the answer is usually found and relayed to the customer within hours.
(When was the last time you got such quick tech support?)

The CMX philosophy also includes giving the engineer all of the source code to the OS, exclusive of the CMX-Tiny RTOS. This is, we feel, an invaluable tool, as it lets the user turn on or off any and all compiling/linking switches they feel are necessary and making it obsolete for an engineer to come back once a compiler company comes out with a new version of their tools. Not only does giving the source code to the user create an invaluable debugging tool, but it knocks down the usual steep learning curve normally associated with an RTOS. This also allows for smaller code size because only functions that are used are linked in to the final output module.

The CMX products are completely scalable. Start with the basic RTOS package, whether it is the full-blown CMX-RTX package or the CMX-Tiny, CMX-Tiny+, etc., and choose any one of the CMX-AIM (CMX-Add In Module) products for whatever your needs are. This could be the CMXBug debugger, the CMXTracker real-time task flow analyzer, the CMX-CAN control area network add-in, CMX-TCP/IP network module, or any of the other CMX modules. Being fully scalable also means that you as the engineer are able to pick and choose any number of combinations of these CMX-AIM products.

When it comes to the CMX RTOS, speed of integration and user happiness are our bottom line.



CMXBug™

Interactive Debugger
CMX-AIM Product

The CMX CMXBug™ debugger provides the user the ability to view and modify different aspects of the CMX multi-tasking operating system environment, while the user's application code is running. CMXBug allows the user to "single step" one system TICK, thus allowing normal activity to occur for one system TICK, with CMXBug resuming after this "single step". Also the user can set the number of system TICKS that CMXBug will wait, allowing normal activity, before it again resumes. This is a very powerful and helpful feature.

CMXBug allows the user access to most of the CMX OS features, such as: Tasks, Cyclic timers, Resources, Mailboxes, Queues, Stacks, the system TICK and IESLICE scales, etc. CMXBug also shows the user information pertaining to each task percentage of RUNNING time to the total. ; enables the user to obtain an accurate picture of each task in relation to all tasks. It also shows the amount of time that the proces- is "IDLE" with no task running. This allows the user a very powerful insight as to how the processor time is being spent.

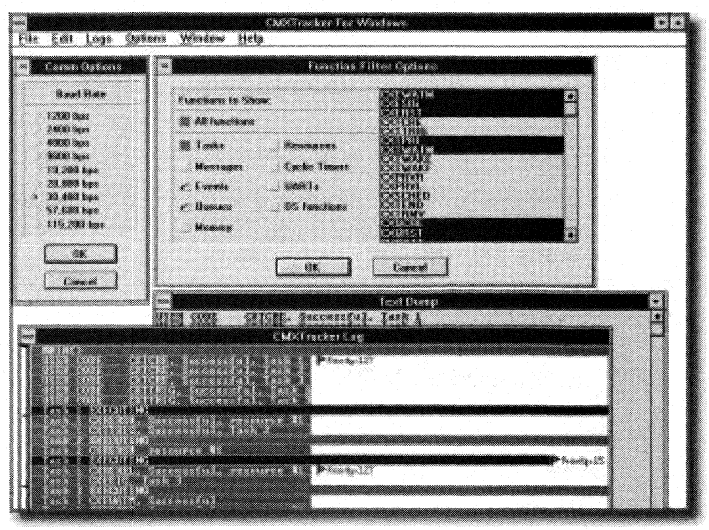
CMXTracker™

Real-Time Flow Analyzer
CMX-AIM Product

The CMX CMXTracker™ provides the user the ability to log chronologically in real-time, the tasks' execution flow, the CMX functions called and their parameters, interrupts using CMX functions and the CMX system TICK.

The user may view the log with different degrees of log resolution with the CMX system TICK being a "timeline" stamp. These are: coarse; which shows just the system ticks and tasks' executing. Mid; which shows all of Coarse plus the CMX functions called. Fine; which shows all activity, including what CMX functions were called with their parameters and results returned (such as the message sent or received, event bits set, timed out, etc.), interrupts and more.

The user may reset the log, resume running of application code and possibly change some aspects of the log, such as "autowake" CMXTracker after a certain number of entries. CMXTracker allows the user to "single step" one system TICK, thus allowing normal activity to occur for one system TICK, with CMXTracker resuming after this "single step". The user can also set the desired number of system TICKS that CMXTracker will wait, allowing normal activity, before it again resumes. This is a very powerful and helpful feature.



Powerful CMX-RTX™ Functions:

(Just A Partial Listing!)

TASK MANAGEMENT
Create a task
Remove a task
Start a task
Suspend a task, with time-out provision
Wake a suspended task
Forcefully wake a task
Change a task's priority
Terminate a task early
Do a co-operative rescheduling
Disable task scheduling
Enable task scheduling

TIMER MANAGEMENT
Create a cyclic timer
Change cyclic timer event parameters
Start a cyclic timer
Restart a cyclic timer
Restart a cyclic timer, with new initial time period and/or new cyclic time period
Stop a cyclic timer

MEMORY MANAGEMENT
Create a fixed block pool
Request free block from pool
Release block back to pool

RESOURCE MANAGEMENT
Get a resource
Reserve a resource, with time-out provision
Release a resource
Automatic priority inversion

EVENT MANAGEMENT
Wait on event(s), with time-out provision
Set an event
Clear an event

SYSTEM MANAGEMENT
Initialize CMX
Enter CMX
Enter an interrupt
Exit an interrupt
Enter power-down mode (automatically done by the scheduler)

MESSAGE MANAGEMENT
Get a message
Wait for a message, with time-out provision
Send a message
Send a message, wait for reply
Wake task that sent message

QUEUE MANAGEMENT
Create a circular queue
Reset a queue to empty
Add to top of queue
Add to bottom of queue
Remove from top of queue
Remove from bottom of queue

SEMAPHORE MANAGEMENT
Pend on semaphore, with time-out provision
Post to a semaphore
Reset semaphore count

Some Points You Should Know About The CMX Real-Time Multi-Tasking Operating Syste

- Fully Supports Nested Interrupts
- ALL Source Code Supplied (except CMX-Tiny)
- Extremely FAST Context Switch Times
- Very Low Interrupt Latency Times
- Automatic Power-down Capabilities
- Several C Vendors Supported
- Online Help and User Manual
- Scheduler and Interrupt Handler Written in Assembly for Speed and Optimization
- All CMX Functions Contained in Library
- Very Compact CODE, Yet Robust
- Easily Interfaces to Assembly Language
- ROMable
- User Configurable
- Interrupt Callable Functions such as: Send a Message, Set an Event, Post to a Semaphore, Manipulate Cyclic Timers, Wake a Task, Start a Task, etc.

OCESSORS SUPPORTED (INCLUDES SUPPORT FOR ALL DERIVATIVES)

: THIS IS JUST A PARTIAL LISTING

8051	MIPS	7700	SH
80x86	80251	ST10	ARM
H8/300	80C16x	M16C	78K3
68HC11	68HC16	SM6000	78K4
68HC12	8051-XA	H8/300H	MN10200
PowerPC	80196 & 296	H8S/2000	Many DSPs
ST9 & ST9+	68K & 683xx	TLCS-900	Z80/64180/Z180

ny C vendors are supported, such as:

nimedes, ARM, Avocet Systems, Borland, Ceibo, Cosmic Software, Diab Data, Franklin, GNU, Green Hills, Hitachi, Hi-n, IAR Systems, Intermetrics, Introl, KEIL, Microsoft, Microtec Reasearch, Mitsubishi, NEC, Panasonic, PLC, sonance, SGS-Thomson, Sierra Systems, Software Develpoment Systems, Tasking, Texas Instruments, Toshiba, com, Whitesmiths & more.

IF THERE IS A PROCESSOR OR COMPILER THAT YOU WANT TO USE, BUT DON'T SEE IT HERE, CALL US, AS THERE ARE MORE THAT ARE SUPPORTED ALL THE TIME.

CP/IP : CMX TCP/IP is a portable high performance TCP/IP implementation for embedded systems. Memory usage is localized and deterministic. It uses RTOS signaling mechanisms to provide a true multitasking reentrant stack. When RTOS is available, CMX TCP/IP provides a straight forward single threaded stack which supports multiple sockets via the select mul-exer.

X TCP/IP can be used for anything from set top boxes that connect a customer's home up to the internet using their TV all the way emote management that allows a person to check the status of an alarm halfway across the world. This is possible through the won- of the internet, with which CMX TCP/IP is connectable. It is sophisticated enough to handle the toughest wide area networking s, but is still compact and economical enough to be used in a simple LAN application.

OS FILESYSTEM : The CMX DOS compatible file system allows all implementations of a file system. This includes large databases as well as small configuration files. Partitions of over 32 gabytes are supported.

CMCIA : One of the greatest features of PCMCIA is insertion of devices; anything from disk controllers to networking cards, while the processor is running. Memory mapping, one of the hardest features of PCMCIA, is provided hin the function calls.

CProto-RTX : The CMX PCProto-RTX™ Real-Time Multi-Tasking Operating System is the CMX-RTX™ RTOS ported to work specifically on the user's PC and it's environment. This allows and 80x86 PC to be :d as a development platform, regardless of the target processor.

:Proto-RTX allows the user to write, develop, and test their application code using the sophisticated tools available for the PC Many ›grammers are familiar with the PC, therefore this allows them to get their application code up and running faster, possibly then work- ; with the cross compiler and target processor.

l features that are supplied with the CMX-RTX are included within the PCProto-RTX such as provisions for nested interrupts, inter-›t callable functions, true preemptive scheduling, along with available cooperative and timeslicing.

MX-CAN™ : CMX provides a sophisticated CAN interface software package for application programs as an extension to the CMX Real Time Operating System. The CAN system is sold integrated with the MX RTOS, as well as a separate stand alone product.

ıe CMX-CAN software sets up a CAN supervisor task running under CMX to manage the CAN controller, and the deliv- y of messages between application tasks.

A few users of CMX products:

AMD	Ford	TRW
Philips	AMP	Enraf
TV/COM	U.S. Navy	Xerox
Analog Devices	U.S. Robotics	Siemens
Fujitsu Telecom	AT&T Wireless	Rockwell
Ericsson Mobile	Hewlett Packard	Kenwood
Nokia Telecomm	Hughes Network	Benefon OY
Temic Telefunken	Bose Corporation	Oak Technologies
Sony Trans Comm	Lucent Technologies	Honeywell GmbH

CMX also distributes compiler packages.
Through this arrangement, we can offer a total package discount when you buy both an RTOS and the compiler for it from us. Just ask for the CMX Total Embedded Solution™ package for your processor.

Some Other Kernel Products

CMX-Tiny™ The CMX-Tiny Real-Time Multi-Tasking Operating System is a "Tiny" kernel for those processors that have a very small amount of RAM embedded on the processor's silicon (about 128 bytes). This allows the user to develop their application code and have it run under an RTOS, using only the onboard RAM that the processor provides. The CMX-Tiny does NOT need any external RAM, regardless of whether the processor can support the use of external RAM or not. The code size of the CMX-Tiny is also extremely small, thus allowing the processor's on-board ROM to support both the user application code and the CMX-Tiny code in most cases. This kernel has a lot of functionality, as well as the more frequently used functions, of the full-blown CMX-RTX™. True preemptive scheduling is provided, with cooperative scheduling available if needed.

CMX-Tiny+™ The CMX-Tiny+ Real-Time Multi-Tasking Operating System is a "Tiny Plus" kernel for those processors that have a small amount of RAM embedded on the processor's silicon (minimum of 512 bytes and up). This allows the user to develop their application code and have it run under an RTOS, using only the onboard RAM that the processor provides. The CMX-Tiny+ does NOT need any external RAM, regardless of whether the processor can support the use of external RAM or not. The code size of the CMX-Tiny+ is also very small, allowing the processor's on-board ROM to support both the user application code and the CMX-Tiny+ code in most cases. This kernel, based on a scaled down version of the CMX-RTX™, retains most of the functionality, as well as the more frequently used functions. True preemptive scheduling is provided, with cooperative scheduling available if needed.

A Few Products In The CMX-AIM Line-Up

Some CMX-AIM products do not need the CMX RTOS in order to be used.

Kernel Awareness for the technologies of today and tomorrow

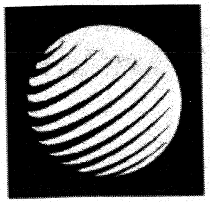
JAVA
More being added everyday...

Founded in 1989, CMX is the favorite choice of embedded developers. Working closely with most major silicon and compiler vendors, we can offer any engineer a well designed and thoroughly tested product for many of the embedded microprocessors and microcontrollers on the market today.

Your Distributor Is

CMX has distributors worldwide.
Contact CMX or visit our web page for the distributor near you.

CMX COMPANY
680 Worcester Road
Framingham, MA 01702
Phone:(508) 872-7675
e-mail:cmx@cmx.com
WWW: http://www.cmx.com
Fax:(508)620-6828



Embedded
System
Products

Embedded System Products, Inc.
10450 Stancliff, Suite 110
Houston, Texas 77099.4336
TEL: 281.561.9990
FAX: 281.561.9980

RTXC™ Real-Time Kernel

What Is RTXC?

RTXC is a flexible, field-proven, multitasking real-time kernel for use in a wide variety of embedded applications on a broad range of microprocessors, microcontrollers, and DSP processors. Embedded System Products, Inc. (ESP) distributes the RTXC source code royalty-free. RTXC is written primarily in C and features a single Application Programming Interface (API) for all supported processors. As a result, RTXC has a configurable, powerful multitasking architecture that helps you get the job done and preserves your software investment.

What's In RTXC?

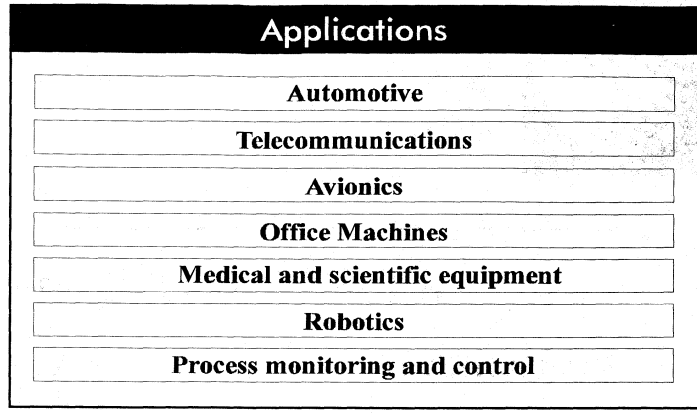
RTXC, like all multitasking real-time kernels, manages tasks and time, synchronizes tasks with events, and transfers data between tasks. But RTXC goes beyond these basic requirements with its extensive set of understandable kernel services, each operating on one of seven classes of kernel objects. RTXC also contains kernel services for RAM management and exclusive access to any entity.

RTXC Features

- Multitasking with preemptive, round robin, and time-sliced task scheduling
- Support for static and dynamically created tasks
- Fixed or dynamically changeable task priorities
- Intertask communication and synchronization through semaphores, messages, and queues
- Efficient timer management for one-shot and periodic timers, delays, and time-outs
- RAM management through static and dynamic memory partitions
- Resource management for exclusive access control
- Fast context switching and short interrupt latency
- Small RAM and ROM requirements
- Standard programming interface on all processors
- System configuration utility that permits flexible, easy customization and reduces maintenance costs
- Well-indexed 650+ page user's manual
- Royalty free source code included
- Quality technical support

Tasks

In RTXC, tasks are the main operational elements of the application. Because different tasks have different relative importance, RTXC supports a task prioritization design permitting both static and dynamically variable values. Tasks may be predefined during system generation or dynamically defined at runtime.



RTXC supports a scheduling policy that permits tasks to gain control of the CPU in different manners. Preemptive scheduling ensures that the highest priority task in a ready state is in control of the CPU. If several tasks share the same priority, they may be scheduled in a round-robin fashion. Time-sliced scheduling is also available within a given priority and each task may be given its own time quantum.

Semaphores

Semaphores are event synchronization objects in RTXC. Semaphores support both internal and external events through a unified event processing design. Tasks may synchronize with an event by waiting on a semaphore, consuming no CPU time while doing so. A waiting task will resume when the event occurs and the associated semaphore is signaled. RTXC kernel services provide for single and multiple event handling.

FIFO Queues

FIFO Queues permit tasks to pass data from one to another in chronological order. All Queues are defined globally and may have multiple producers and multiple consumers. Various queue conditions such as *Full, Empty* and *Not_Empty* can be associated with semaphores to provide rapid and deterministic handling for multiple queues without polling.

Messages and Mailboxes

Messages and Mailboxes give RTXC users a prioritized means of passing data between tasks. Mailboxes are globally declared and a task may use none, one, or many. Any task may send Messages to any Mailbox, synchronously or asynchronously, and any task may receive Messages from any Mailbox. Semaphores may be coupled with Mailboxes for use in serving multiple Mailboxes without the need for periodic polling.

Timers

RTXC's Timer kernel object class manages one-shot and periodic time durations. Expiration of a timed period is an event associated with a semaphore. RTXC design for managing timer updates is very efficient requiring a fixed overhead regardless of the number of active timers.

Memory Partitions

RTXC manages RAM through a partitioning design using Memory Partition kernel objects. Memory Partitions prevent RAM fragmentation that can occur when multiple tasks randomly allocate and free blocks of RAM from the system heap. Memory Partitions may be statically defined during system generation or created dynamically at runtime.

Resources

The need to ensure exclusive access to some entity in a real-time system gives rise to the RTXC Resource kernel object. A task can gain ownership of an entity, real or logical, and use it exclusively. When the task completes its use of the entity, it relinquishes ownership to permit other tasks to gain control of the entity. RTXC options provide a convenient way to handle priority inversion in which a low priority task owns a Resource needed by a high priority task.

Interrupt Management

RTXC provides a generalized design for servicing interrupts in an efficient yet flexible manner achieving minimum interrupt latency and maximum responsiveness. Special kernel services may be called from an interrupt service routine to perform such functions as event signaling, buffer allocation, and ISR exit.

System Configuration Utility

RTXCgen is an interactive program used to define the
kernel objects needed for the application. Objects are defined in a simple interactive dialog with the program. When all definitions are complete, RTXCgen produces C source code of the kernel objects. The result is error free system generation. RTXCgen is a standard part of the RTXC distribution.

System Level Debug Task

During the debugging phase, RTXCbug may be employed as
a task to examine the interaction between the application tasks and RTXC. RTXCbug remains blocked and consumes no CPU time until invoked by the process or by operator intervention. When in use, RTXCbug displays coherent snapshots of the various classes of kernel objects showing current states and relationships with other objects. There is also support for some direct operator intervention. RTXCbug is provided with all RTXC distributions.

Portability

Because RTXC is written in C, the API is processor independent and, therefore, highly portable. You can move application code written for RTXC easily from one type of processor to another. RTXC may be successfully employed in applications using 8-bit and 16-bit microcontrollers, 16-bit and 32-bit microprocessors, and DSP processors.

Warranty and Support

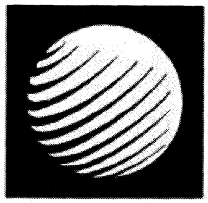
RTXC carries a standard 90-day initial warranty and support period. If you have a question, you are backed up by a highly acclaimed user's manual of over 650 pages and knowledgeable telephone support by the engineers who wrote RTXC. If you want to get the job done right and get it done quickly, RTXC is the tool for you.

Training

Embedded System Products, Inc. periodically conducts RTXC training classes in our Houston offices. These classes feature a combination of lecture and hands-on exercises. We can provide training classes at the customer's site by special arrangement. Call for a course schedule, details, and prices.

Visit Our Web Site

For the latest information about our products and services, including currently supported processors and training class schedules, visit our RTXC web site at www.rtxc.com



Embedded
System
Products

Embedded System Products, Inc.
10450 Stancliff, Suite 110
Houston, Texas 77099.4336
TEL: 281.561.9990
FAX: 281.561.9980

RTXCnet™ Networking Stack

RTXCnet Features

- Simple and efficient networking
- Full networking capacity from application to physical layer
- Integrated with the high-performance RTXC kernel
- Well-indexed user's manual
- Royalty-free source code included
- Quality technical support

What is RTXCnet?

A DREAM COME TRUE!

TCPIPDHCPSNMPUDPFTPHTTP!!?!?! Alphabet soup or embedded networking? If communications is a headache, RTXCnet is the aspirin! One initialization call and the stack is up and running. One function call from a task and your application can transmit around the globe. Fully integrated with the RTXC kernel, RTXCnet not only provides a full networking stack but also a seamless interface to the RTXC kernel for networking applications. Transfer files, commands, and data across the room or across the world with RTXCnet in a fraction of the development time traditionally required. This kind of integration was once only a dream. **But Embedded System Products and RTXCnet make the dream come true.**

What's in RTXCnet?

SIMPLY POWER!

Power to interface your application with the World Wide Web or your own specialized client programs. Power to develop your application on time, every time. Yet, simplicity is accomplished using the RTXCnet manager, a transparent component of RTXCnet. It arbitrates synchronization and interactions among the application, the stack, and the kernel. What makes the operation so easy is that the RTXCnet manager is already equipped to use the kernel resources it needs. There is no debugging and no searching through source code to connect the stack to the kernel. Simply put, RTXCnet is SIMPLY POWER.

Why choose RTXCnet?

QUICK DEVELOPMENT/LOW COST!

Ease of Use: While RTXCnet is a highly specialized sub-system, it is not complicated to use. After making one initialization call and one call from the task, RTXCnet is ready to carry your information from the application layer through the stack and out to the world. RTXCnet allocates all the needed resources and initializes the network stack including all network drivers.

Continuity: RTXC and RTXCnet are consistent in their written styles. The products fit together like hand and glove eliminating the need for time spent on integration.

Code Size: RTXCnet is highly optimized and requires little memory space. Now you have more room with which to work for your application code and data.

Cost: You can incorporate RTXCnet into your product royalty-free and the time you save will be well worth the price.

Company: Embedded System Products, Inc. has built products for the real-time, embedded market since 1978 with an engineering staff ready to answer your every question concerning our products.

RTXCip

RTXCip is one of the most fundamental protocols. Below this layer is the actual hardware where communication is done only node to node. RTXCip handles all the routing issues of a network including gateways. It includes sources for IP, UDP, ARP, DNS, DHCP Client, and ICMP, and full technical documentation. RTXCip can be configured as a standard client machine, an IP router, or a multi-homed server.

IP Features:

- Support for NAT Routing
- IP Routing
- Multi-homed
- Small memory requirement

RTXCtcp

RTXCtcp sits on top of the RTXCip stack. While RTXCip handles the routing of information it has no error checking and has a very primitive interface. RTXCtcp provides a simple interface (Berkeley-like sockets) for applications and performs error checking. It includes source code, Sockets API, and full technical documentation.

TCP Features:

- Nagle Algorithm (Slow Start)
- Sockets Interface
- VJ Smoothed Round Trip Timing
- Delayed ACKs

RTXCtcp/ip

As a combination of RTXCip and RTXCtcp, RTXCtcp/ip is a complete software development package for applications requiring networking capabilities. RTXCtcp/ip requires very little memory and supports two-way tasking. Take advantage of your multitasking environment and gain event-driven performance by eliminating wasteful polling. RTXCtcp/ip works with Ethernet, SLIP, and PPP link layers.

RTXCppp

RTXCppp provides an interface usually associated with modems, though it can run over just a simple serial line. RTXCppp includes source code for LCP, IPCP, CHAP, and PAP, and full technical documentation. RTXCppp can function as a client or a server, and supports all standard PPP authentication protocols and DHCP over PPP. RTXCppp has a small memory requirement, which is ideal for all embedded applications.

Features:

- VJ Header Compression
- PAP and CHAP security
- Hayes Dialing code compatible
- Supports Multiple Simultaneous Links
- Supports DHCP

RTXCdhcp-s

RTXCdhcp-s (DHCP server, DHCP client is provided as part of the RTXCip package) is a complete drop-in software module that allows an embedded system to assign temporary or permanent IP addresses to clients. Compatible with Windows 95 and NT and backward compatible with BOOTP clients, the DHCP Server supports its database on disk or in flash. The DHCP Server is the standard way to assign IP addresses to probes or to slave devices that do not need permanent IP addresses. The Server can also assign private IP addresses for Local Area Network devices that are connected to a NAT Router to gain access to the Internet.

Features:

- Compatible with Windows 95 and NT
- Backward compatible with BOOTP
- Supports two methods of data base storage
- Eliminates need to purchase individual IP addresses for each device
- Makes managing of IP addresses easier and less time consuming

RTXCsnmp

RTXCsnmp - Simple Network Management Protocol provides information and a means to send commands around a network. It includes source code for an SNMP agent, a MIB Compiler to convert MIBs into C code, and full technical documentation. It has a small footprint, and requires no multitasking features. All code is implemented as an event driven state machine. Special time-based operations help DHCP leases and NAT table entries to expire. RTXCsnmp can be easily added to RTXCip.

Features:

- Very small code size
- Compatible with TCP APIs, including sockets & Winsock
- Requires no memory management
- Event driven- does not require a separate task
- Sample agents for DOS and Windows 95 (Winsock) included
- Supports SETs, GETs, NEXTs, and TRAPs.

RTXChttp

Add web access to you embedded application using RTXChttp. Now you can retrieve the status of your application and send commands from any standard web browser. RTXChttp allows you to pass HTML pages, use forms and dynamic pages. This is the perfect solution for remote access and control when Internet connectivity is desired.

RTXCftp

RTXCftp adds sophisticated file service capabilities to embedded systems. The RTXCftp server can even initiate file requests. FTP is one of the most reliable and popular methods of file transfer, which allows the end-user to upgrade firmware and copy large data blocks from the embedded system over a network efficiently, easily, and with standard tools. RTXCftp uses a native file system on the target processor or a virtual file system (VFS).

Features:

- Sockets interface makes porting quick and easy
- Supports passive mode
- Multi-user and multi-session
- Two-way tasking

Ethernet Drivers

Ethernet drivers are available for a wide variety of chip sets. Now you can use "drop-in" software from the physical layer up to your application to save you time and money.

Warranty and Maintenance

RTXCnet carries a 90-day initial warranty period during which you get free maintenance and updates to your RTXCnet license. You can extend your maintenance period through an optional agreement and receive continued maintenance, including updates and upgrades of your RTXCnet license, in additional twelve-month increments.

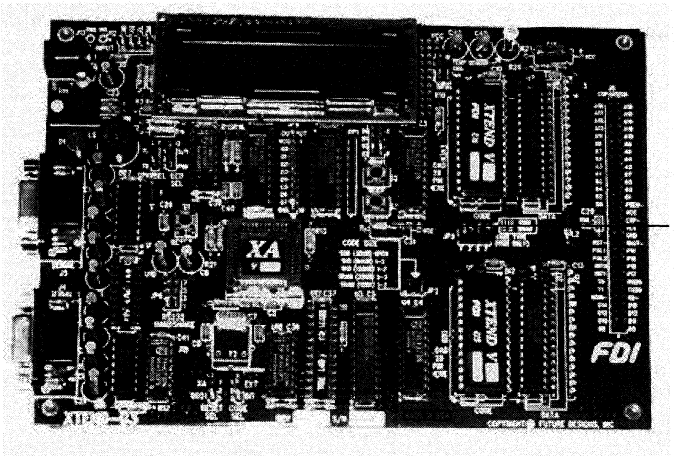
Documentation and Support

A thorough user's manual accompanies comprehensive technical support. When covered by initial warranty or an extended maintenance agreement, you have access to the very engineers who wrote the RTXCnet code.

Visit Our Web Site

For the latest information about our products and services, including currently supported processors and training class schedules, visit our web site at www.rtxc.com.

FDI Future Designs, Inc.
Your Development Partner



XTEND

XA
Trainer &
Expandable
Narrative
Design

The *XTEND*, XA Trainer and Expandable Narrative Design, is designed to provide the user with a stable hardware and software platform for application development with the XA-G3. In many cases the *XTEND* may serve as a quick prototype for the actual user application. With the on-board prototype area or optional expansion boards, the user can quickly and easily get a new customized design running with the XA-G3.

Features

1. Philips XA-G3 Microcontroller Socket for 44-pin PLCC supporting both internal and external code execution

2. 128KB Standard FLASH ROM Code Space, expandable to 256KB (dual sockets for 16-bit access)

3. Code Space supports EPROM, FLASH (5V), NVSRAM, or SRAM

4. 64KB Standard High-Speed Data Space SRAM, expandable to 256KB (dual sockets for 16-bit access)

5. Two DB-9 RS232 Serial Communications Ports with Optional Hardware Handshake & RS232 Cable and DB25 Adapter Included

6. On-board speaker for tone generation

7. Interface for character type LCD modules with 16 character LCD included

8. 60-pin Expansion Header for Full Expansion Capability

9. 9 VDC Input with on-board 5V regulator, UL Approved Power Supply included

10. 5.25" x 7.25" 2-layer PCB with Full Silkscreen Information

11. 2.5" x 1" wire-wrap area on-board

12. Four TTL User Inputs via double-row header

13. Two user input pushbuttons

14. Eight TTL User Outputs via single-row header

15. Users manual and schematics included

16. Optional I2C Interface/Monitor

17. Planned Future Expansion Boards Include:

- I2C Expansion Board with LCD, Keypad, and Real Time Clock
- Prototype Board with 'Pad-per-hole' Area
- Virtually any type may be custom designed

18. Example Routines for Applications
- XA-G3 Initialization and Setup Routines
- Serial Port Drivers
- Timer/Counter Drivers
- Watchdog Timer Routines
- Interrupt Routines
- 8051 to XA Translation Examples

19. Internal XA-G3 Monitor Supporting
- Serial Host Communication
- Register Dump/modify
- External Data memory dump/Modify
- External Code Memory Load (from RS232)
- Code disassembly
- Execute Code with up to 4 user defined breakpoints
- Single-step through user code

20. CMX Evaluation Package
- CMX RTOS XA Demo for XTEND
- CMX RTOS Demo for the PC
- HiTech XA C Compiler Demo

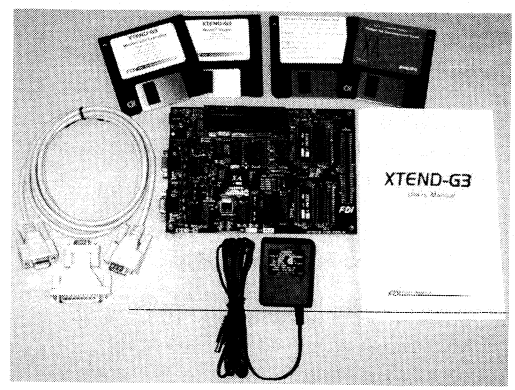
Ordering Information

Part Number: XTEND-G3
Price: $249.00 (USD) complete
Warranty: 30-day money back guarantee
Availability: Stock

(205) 830-4116 Information
(800) 278-0293 Sales

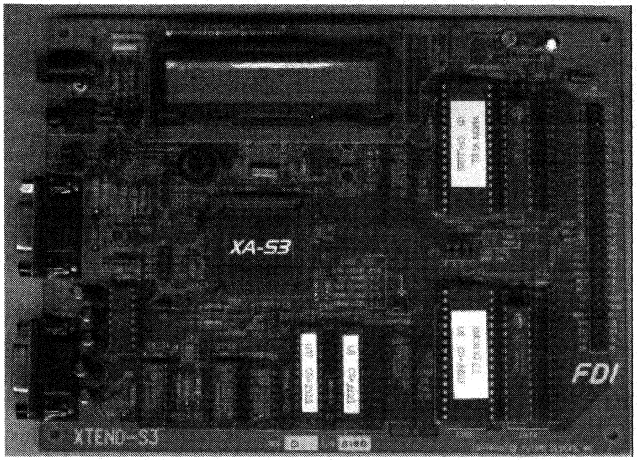
(205) 830-9421 FAX
e-mail teamfdi@aol.com

Future Designs, Inc.
P.O. Box 7362
Huntsville, AL 35807



Kit Contents

FDI Future Designs, Inc.
Your Development Partner



XTEND

XA
Trainer &
Expandable
Narrative
Design

The *XTEND*, XA Trainer and Expandable Narrative Design, is designed to provide the user with a stable hardware and software platform for application development with the XA-S3. In many cases the *XTEND* may serve as a quick prototype for the actual user application. With the on-board prototype area or optional expansion boards, the user can quickly and easily get a new customized design running with the XA-S3.

Features

1. Philips XA-S3 Microcontroller Socket for 68-pin PLCC supporting both internal and external code execution

2. 128KB Standard FLASH ROM Code Space, expandable to 256KB (dual sockets for 16-bit access)

3. Code Space supports EPROM, FLASH (5V), NVSRAM, or SRAM

4. 64KB Standard High-Speed Data Space SRAM, expandable to 256KB (dual sockets for 16-bit access)

5. Two DB-9 RS232 Serial Communications Ports & RS232 Cable and DB25 Adapter Included

6. On-board speaker for tone generation

7. Interface for character type LCD modules with 16 character LCD included

8. 60-pin Expansion Header for Full Expansion Capability (16MB memory and I/O supported)

9. 9 VDC Input with on-board 5V regulator, UL Approved Power Supply included

10. 5.25" x 7.25" 4-layer PCB with Full Silkscreen Information

11. 2.5" x 1" wire-wrap area on-board

12. Users manual and schematics included

FDI Future Designs, Inc.
Your Development Partner

13. A/D Converter support includes on-board 3V analog supply, precision reference, 10K POT, and thermister

14. PCA Support includes I/O access to the speaker, LED, and a dedicated 4-pin header

15. Two user input pushbuttons

16. Optional I2C Interface/Monitor

17. Planned Future Expansion Boards May Include:

- Prototype Board with 'Pad-per-hole' Area
- Other custom designs

18. Example Routines for Applications
- XA-S3 Initialization Routines
- Serial Port Drivers
- Timer/Counter Drivers
- Watchdog Timer Setup
- PCA Drivers
- ADC Drivers
- Interrupt Routines

19. External XA-S3 XMON Monitor Supporting
- Serial Host Communication
- Register Dump/Modify
- External Data memory Dump/Modify
- External Code Memory Load (from RS232)
- Code disassembly
- Execute Code with up to 4 user defined breakpoints
- Single-step through User Code

20. Tasking XA Toolchain Demo for Windows

- Tasking Embedded Development Enviroment
- XA C-Compiler Demo
- Crossview Debugger Demo with ROM Monitor pre-programmed into internal XA-S3

21. CMX RTOS Information Package

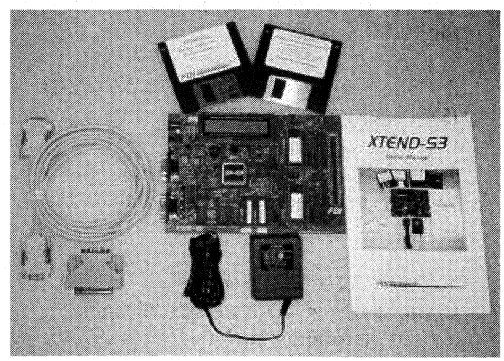
22. Philips XA Assembler, Simulator and 8051 to XA Translator

Ordering Information

Part Number: XTEND-S3
Price: $249.00 (USD) complete
Warranty: 30-day money back guarantee
Availability: Stock
Philips 12NC: 935261800112

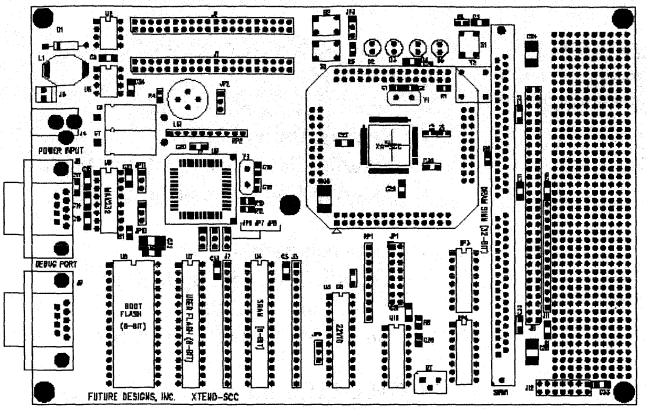
(256) 830-4116 Information
(800) 278-0293 Sales
(256) 830-9421 FAX
e-mail teamfdi@aol.com

Future Designs, Inc.
P.O. Box 7362
Huntsville, AL 35807



Kit Contents

FDI Future Designs, Inc.
Your Development Partner



XTEND

XA
Trainer &
Expandable
Narrative
Design

The *XTEND*, XA Trainer and Expandable Narrative Design, is designed to provide the user with a stable hardware and software platform for application development with the XA-SCC. In many cases the *XTEND* may serve as a quick prototype for the actual user application. With the on-board prototype area or optional communications modules, the user can quickly and easily get a new customized design running with the XA-SCC.

Features

1. Philips XA-SCC Microcontroller utilizing 100-pin LQFP package

2. 64KB Standard FLASH ROM Code Space, expandable to 1MB

3. Code Space supports EPROM, FLASH (5V), NVSRAM, or SRAM

4. 32KB Standard High-Speed Data Space SRAM, expandable to 256KB

5. 72-pin DRAM SIMM Socket supporting 1MB to 16MB of EDO or FPM DRAM

6. On-board speaker for tone generation

7. Interface for character type LCD modules with 16 character LCD included

8. 60-pin Expansion Header for Full Expansion Capability (16MB memory and I/O supported)

9. 9VDC Input with on-board 5V power supply, UL Approved Wall Supply included

10. 5.0" x 7.5" 4-layer PCB with Full Silkscreen Information

11. 1.25" x 4" wire-wrap area on-board

12. Users manual and schematics included

PRELIMINARY

13. On-board DUART provides two DB-9 RS232 Serial Communications Ports, RS232 Cable and DB25 Adapter Included

14. Two user input pushbuttons

15. Four user controlled LEDs

16. Example Routines for Applications
- XA-SCC Initialization Routines
- Serial Port Drivers
- Timer/Counter Drivers
- Watchdog Timer Setup
- Interrupt Routines

17. Tasking XA Toolchain Demo for Windows

- Tasking Embedded Development Enviroment
- XA C-Compiler Demo
- Crossview Debugger Demo with ROM Monitor

18. External XA-SCC XMON Monitor Supporting:
- Serial Host Communication
- Register Dump/Modify
- External Data memory Dump/Modify
- External Code Memory Load (from RS232)
- Code disassembly
- Execute Code with up to 4 user defined breakpoints
- Single-step through User Code

19. CMX RTOS Information Package

20. Philips XA Assembler, Simulator and 8051 to XA Translator

21. Planned Communications Modules Include:

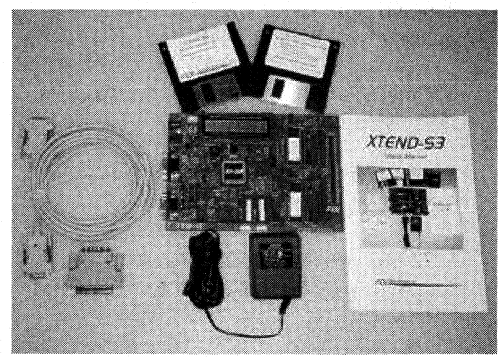
- Quad Multi-mode Async/Sync Serial Module
- ISDN Terminal Adapter

Ordering Information

Part Number: XTEND-SCC
Mainboard Price: $449.00 (USD) complete
Warranty: 30-day money back guarantee
Availability: 3Q'98
Philips 12NC: TBD

(256) 830-4116 Information
(800) 278-0293 Sales
(256) 830-9421 FAX
e-mail teamfdi@aol.com

Future Designs, Inc.
P.O. Box 7362
Huntsville, AL 35807



Kit Contents
(XTEND-S3 Kit shown for example only)

PRELIMINARY

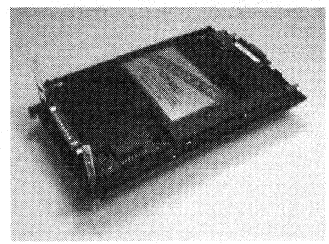
nohau
CORPORATION

Web: www.nohau.com
E-mail: sales@nohau.com

EMUL51XA-PC

In-Circuit Emulator for the P51XA Family

Nohau's EMUL51XA-PC supports the Philips' P51XA family of microcontrollers. The EMUL51XA-PC's user-friendly Microsoft Windows-based interface includes dynamically changing menus, moveable and scrollable "child" windows, function key shortcuts and context sensitive help. These features, together with the "bond-out" chip technology used, result in accurate, in-depth emulation to effectively minimize debugging time.



System Specifications

Supported P51XA Derivatives

- P51XAG3, P51XAS3

Hosts

- 386 or better PC
- Windows 3.1x/'95/NT

High-LevelDebugging

- Window for source-level debugging.
- Single step or line step with breakpoints marked directly in the code.
- Full support of local and global variables.

Symbolic Support

- Full symbolic debugging with type checking.
- Same symbols can be used for different modules.
- All special function registers supported.

Single Stepping

- Single or multiple instruction stepping.
- Step over calls and interrupts.
- Line stepping in high-level languages.

Program Performance Analyzer

- Histogram and statistical information of program execution in real time. Exact timing information on execution time in each program routine.

Real-Time Emulation

- Full speed emulation.
- 128K data memory and 128K code memory, or
  512K & 1M data memory and 512K & 1M code memory, or
  256K, 1M or 2M code/data (½ code and ½ data or all code).

Memory Mapping

- Mappable down to 16 bytes (G3).
- Mappable down to 128 bytes (S3).

Real Time Trace

The trace can be operated "on-the-fly" which means thatit can be viewed, programmed, and retriggered without disturbing program execution.

Real Time Trace (Con't)

Two tracing modes are provided. The normal mode allows you to specify up to three trigger events. Each event can consist of virtually any number of addresses or address ranges. The second mode, window mode, uses one trigger to turn on recording and one to turn off recording.

- Up to 512K deep, 128 bits wide (frame) with timestamp.
- Three trigger levels: programmable pre/post trigger location, external address & data.
- Instruction queue decoding prevents false triggers.

Trace record includes:

- internal address (16 bits)
- internal data (16 bits)
- external address (24 bits)
- external data (16 bits)
- user defined data (16 bits)
- cycle based timestamp (40 bits)

- User can specify the addresses that are captured.
- Ability to trace 8 user defined signals from the pod.
- The last trigger event has a 8 bit counter that inhibits the event until the counter expires.
- Trace buffer available in 128K and 512K depths.
- A trigger event can consist of virtually any number of addresses in combination with data where each can be qualified as data read or data write.
- A centering counter allows positioning of the trigger point anywhere in the trace buffer to trace before, after or around the trigger.
- The code coverage feature can monitor up to 1M of address. The specified addresses do not have to be continuous. The fetches are captured.

Breakpoints

- Hardware and software breakpoints.
- Unlimited program breakpoints.
- Break on external address
- with the trace board option, break on any address, data or external signal

For more information, visit our web site at: http://www.nohau.com

51 E. Campbell Avenue, Campbell, CA 95008 PH: (408) 866-1820 Fax: (408) 378-7869

nohau CORPORATION

EMUL51XA™-PC
U. S. Parts List

April 15, 1998

URL: http://www.nohau.com

Order Number **Description**

In-Circuit Emulators for P51XA

SYSTEM CONFIGURATIONS

A minimum system consists of a communication interface plus a POD. A POD requires an adapter to connect to a target system. Trace buffer boards are optional. The EMUL51XA-PC debugger software is included with the pod. This software interface offers these features for any supported C Compiler: Global and local (stack) variable support, source level debugging, C structures, arrays and C expressions.

COMMUNICATION INTERFACES

These communication interfaces must be connected to a POD board to operate (order separately). The communication interface includes a cable that connects to the POD board.

EMUL-PC / LC-B ..... This communication interface is an ISA plug-in board (EMUL-LC/ISA) which communicates with the POD board. Includes CBL-B-LC25/25.

EMUL-PC / EPC ..... This high speed interface device communicates with the POD through the standard PC parallel port.

PODS

General Features

The communication interface connects to the POD board through the cable supplied with the communication interface. The POD board usually requires an adapter, listed under the Adapters section. The POD board has emulation memory, and an on-board oscillator. Jumpers select POD or user target crystal or oscillator. The POD and communication interface can run stand alone without being connected to any user target board. EMUL51XA-PC debugging software is included with the purchase of a pod, and supports Microsoft® Windows™ 3.1x / '95 / NT.

These PODs support both Internal Mode (when the chip is started with /EA high) and External Mode (when the chip is started with /EA low). Hardware breakpoints can be set on addresses of external data read/write with word resolution and can be set either on "internal" code addresses or "external" code addresses. All pods include a power supply.

P51XAG3:

POD-51XAG3-256 / IE-16 ..... 16 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.

POD-51XAG3-256 / IE-20 ..... 20 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.

*Changes since December 30, 1997

Order Number	Description

PODs (Con't)

P51XAG3 (Con't):

POD-51XAG3-256 / IE-25	25 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.
POD-51XAG3-256 / IE-30	30 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.
POD-51XAG3-1M / IE-30	30 MHz POD board for P51XAG3. 1M emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.
POD-51XAG3-2M / IE-30	30 MHz POD board for P51XAG3. 2M emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.

P51XAS3:

***POD-51XAS3-256 / IE-16**	16 MHz POD board for P51XAS3. 256 kB emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.
***POD-51XAS3-256 / IE-30**	30 MHz POD board for P51XAS3. 256 kB emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.
***POD-51XAS3-1M / 1M-30**	30 MHz POD board for P51XAS3. 1M emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.
***POD-51XAS3-2M / IE-30**	30 MHz POD board for P51XAS3. 2M emulation RAM. Requires separate adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.

*Changes since December 30, 1997

Order Number **Description**

POD-51XAG3/IE Speed Limitations:

1 MHz to 25 MHz in 16-bit mode: WM0 must be equal to 1 in BTRL (see chart titled "External Bus Signal Timing Configuration" shown below).

External Bus Signal Timing Configuration

	CR1,CR0	CRA1, CRA0	DW1,DW0	DWA1, DWA0	DR1,DR0	DRA1, DRA0
00	Supported	Supported	N/A	Not supported	N/A	Supported
01	Supported	Supported	N/A	Supported	N/A	Supported
10	Supported	Supported	N/A	Supported	N/A	Supported
11	Supported	Supported	N/A	Supported	N/A	Supported

25 MHz to 30 MHz in 16-bit mode: WM0 must be equal to 1 in BTRL (see chart titled "External Bus Signal Timing Configuration" shown below).

External Bus Signal Timing Configuration

	CR1,CR0	CRA1, CRA0	DW1,DW0	DWA1, DWA0	DR1,DR0	DRA1, DRA0
00	Not supported	Not supported	N/A	Not supported	N/A	Not supported
01	Supported	Supported	N/A	Supported	N/A	Supported
10	Supported	Supported	N/A	Supported	N/A	Supported
11	Supported	Supported	N/A	Supported	N/A	Supported

1 MHz to 20 MHz in 8-bit mode: WM0 must be equal to 1 in BTRL (see chart titled "External Bus Signal Timing Configuration" shown below).

External Bus Signal Timing Configuration

	CR1,CR0	CRA1, CRA0	DW1,DW0	DWA1, DWA0	DR1,DR0	DRA1, DRA0
00	Supported	Supported	Not supported	Not supported	Supported	Supported
01	Supported	Supported	Supported	Supported	Supported	Supported
10	Supported	Supported	Supported	Supported	Supported	Supported
11	Supported	Supported	Supported	Supported	Supported	Supported

Order Number **Description**

POD-51XAG3/IE Speed Limitations (Con't)

20 MHz to 30 MHz in 8-bit mode: WM0 must be equal to 1 in BTRL (see chart titled "External Bus Signal Timing Configuration" shown below).

External Bus Signal Timing Configuration

	CR1,CR0	CRA1, CRA0	DW1,DW0	DWA1, DWA0	DR1,DR0	DRA1, DRA0
00	Not Supported	Not Supported	Not supported	Not supported	Not Supported	Not Supported
01	Supported	Supported	Supported	Supported	Supported	Supported
10	Supported	Supported	Supported	Supported	Supported	Supported
11	Supported	Supported	Supported	Supported	Supported	Supported

POD-51XAS3/IE Speed Limitations:

1 MHz to 20 MHz 16-Bit Mode: WM0 must equal 1 in BTRL. Table 1 shows the external bus signal timing configurations.

Table 1. Configurations for 1 MHz to 20 MHz 16-Bit Mode

	CR1, CR0	CRA1, CRA0	DW1, DW0	DWA1, DWA0	DR1, DR0	DRA1, DRA0
00	Supported	Supported	N/A	Not supported	N/A	Supported
01	Supported	Supported	N/A	Supported	N/A	Supported
10	Supported	Supported	N/A	Supported	N/A	Supported
11	Supported	Supported	N/A	Supported	N/A	Supported

20 MHz to 30 MHz in 16-Bit Mode: WM0 must equal 1 in BTRL. Table 2 shows the external bus signal timing configurations.

Table 2. Configurations for 20 MHz to 30 MHz 16-Bit Mode

	CR1, CR0	CRA1, CRA0	DW1, DW0	DWA1, DWA0	DR1, DR0	DRA1, DRA0
00	Not supported	Not supported	N/A	Not supported	N/A	Not supported
01	Supported	Supported	N/A	Supported	N/A	Supported
10	Supported	Supported	N/A	Supported	N/A	Supported
11	Supported	Supported	N/A	Supported	N/A	Supported

Order Number **Description**

POD-51XAS3/IE Speed Limitations (Con't):

1 MHz to 20 MHz in 8-Bit Mode: WM0 must equal 1 in BTRL. Table 3 shows the external bus signal timing configurations.

Table 3. Configurations for 1 MHz to 20 MHz 8-Bit Mode

	CR1, CR0	CRA1, CRA0	DW1, DW0	DWA1, DWA0	DR1, DR0	DRA1, DRA0
00	Supported	Supported	Not supported	Not supported	Supported	Supported
01	Supported	Supported	Supported	Supported	Supported	Supported
10	Supported	Supported	Supported	Supported	Supported	Supported
11	Supported	Supported	Supported	Supported	Supported	Supported

20 MHz to 30 MHz in 8-Bit Mode: WM0 must equal 1 in BTRL. Table 4 shows the external bus signal timing configurations.

Table 4. Configurations for 20 MHz to 30 MHz 8-Bit Mode

	CR1, CR0	CRA1, CRA0	DW1, DW0	DWA1, DWA0	DR1, DR0	DRA1, DRA0
00	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported
01	Supported	Supported	Supported	Supported	Supported	Supported
10	Supported	Supported	Supported	Supported	Supported	Supported
11	Supported	Supported	Supported	Supported	Supported	Supported

POD UPGRADES

Customers who previously purchased Internal or External pod boards may upgrade to any configuration (buffer size and speed) of the above pod boards and receive a 50% credit for their old Internal or External pod. Offer expires 6/30/98. This service is available only if the pod board to be upgraded is a working unit in good condition, as judged by Nohau Corporation. Upgrade warranty period is three months or until the expiration of the original warranty period, whichever is longer.

nohau CORPORATION

EMUL51XA™-PC
U. S. Parts List

April 15, 1998

URL: http://www.nohau.com

Order Number **Description**

TRACE OPTIONS

Internal / External Data Trace Options

These trace boards connect directly to the pod and do not reside inside the PC. They record both the internal and external address and data busses at their full width. You can record per frame or per clock. They can trigger and filter on both internal and external address and data bus. You cannot use the filter / trigger function at the same time as the Code Coverage feature since the same memory is used for both functions. (For a version of the trace with full 1 Mb trigger/filter, Shadow RAM and Code Coverage, see EMUL51XA-PC/IETR512-30).

128k:

The trigger and filter on the external address busses can take place within a 256kB area. This area is mappable throughout the 4Mb address space in one of sixteen 256kB blocks. A 256kB Shadow RAM is also mappable throughout the 4 Mb address space in one of sixteen 256kB blocks. The code coverage feature is also mappable 256kB in sixteen blocks.

EMUL51XA-PC / IETR128-16 ..... 16 MHz trace board with 128k deep buffer.

EMUL51XA-PC / IETR128-20 ..... 20 MHz trace board with 128k deep buffer.

EMUL51XA-PC / IETR128-25 ..... 25 MHz trace board with 128k deep buffer.

EMUL51XA-PC / IETR128-30 ..... 30 MHz trace board with 128k deep buffer.

512k:

The trigger and filter on the external address busses can take place within a 1 Mb area. This area is mappable throughout the 16Mb address space in one of sixteen 1 Mb blocks. A 1 Mb Shadow RAM is also mappable throughout the 16 Mb address space in one of sixteen 1 Mb blocks. The code coverage feature is also mappable 1 Mb in sixteen blocks.

EMUL51XA-PC / IETR512-30 ..... 30 MHz trace board with 512k deep buffer. This trace board has a 512k frame (or clock) depth and covers the entire 1 Mb address space for triggers, filters, Shadow RAM and Code Coverage.

TRACE UPGRADES

Customers who previously purchased Standard Trace or Data Trace (in PC) boards may upgrade to any configuration (buffer size and speed) of the above Internal / External Data Trace boards and receive a 50% credit for their old trace board. (Offer expires 6/30/98.) This service is available only if the trace board to be upgraded is a working unit in good condition, as judged by Nohau Corporation. Upgrade warranty period is three months or until the expiration of the original warranty period, whichever is longer.

LanICE LOCAL AREA NETWORK OPTION
for XWindows (Sun, HP, and other Workstations)

LanICE ..... Discontinued

*Changes since December 30, 1997

nohau CORPORATION

EMUL51XA™-PC
U. S. Parts List

April 15, 1998

URL: http://www.nohau.com

Order Number	Description

ADAPTERS AND ACCESSORIES

EDI / 44PG/PL-L	Adapter to plug 44 pin POD into 44-pin PLCC socket.
EDI/ 44PG/LC-SD	Adapter assembly, 44 pin PGA socket to 44 pin PLCC, to solder to user target board. Includes one top and one EDI/44LC-SD base.
EDI/ 44LC-SD	Additional base only. 44 pin PLCC solder down base for EDI/44PG/LC-SD.
***ET / AP4-68-SUB1**	Adapter for 68-pin PLCC socket.
PWRSUP6	Additional or replacement 6 amp power supply. This power supply is included with all EMUL51XA-PC pods and is not normally intended to be sold separately.

Adapter Disclaimer

Depending on your target, you may purchase an adapter for use with Nohau's emulator system.

BEFORE YOU ADAPT NOHAU'S POD BOARD TO YOUR TARGET:

Please be advised that some difficulties have been experienced when adapting Nohau's pod board to a target due to improper placement or positioning of the adapter, especially with plcc and clip-over adapters. Often this is because it is extremely easy to damage the adapter, the socket, or create shorts by offsetting the pins when positioning the adapter. Clip-over adapters for 132+ SQFP pin packages and low profile plcc sockets have the highest potential for failure due to bad connections, and only get worse with wear. Solder down, direct pod header connections, DIP and PGA adapters have proved to be the most reliable. Another potential problem lies in working with full height versus low profile sockets.
Some examples of common problems are:

(1) Not realizing that the orientation of the adapter is not usually the same as the special emulation chip (refer to the manual or software);
(2) Understanding that the adapter is usually connected to the bottom of the pod -- the side opposite the special emulation chip.

THEREFORE, PLEASE NOTE THAT IT IS THE USER'S RESPONSIBILITY TO ENSURE THAT THE ADAPTER CONNECTION IS SOLID. WITHOUT DOING SO, NOHAU'S EMULATORS WILL NOT FUNCTION PROPERLY AND MAY EVEN BE DAMAGED.

Other than using extra care when adapting Nohau's pod board to your target, one solution is for customers to design-in pin headers according to the dimension diagrams provided by Nohau for various chips (or solder-in connections in the case of clip-over adapters). Another suggestion is to ohm out a couple of lines from the pod to the target BEFORE turning on the power. If a few address lines or port pins ohm out properly, it is likely that the adapter orientation is correct.

Removing an Adapter

If it is necessary to remove an adapter once it is installed, please use **extreme caution** to avoid damaging the PC board. DO NOT USE ANY TYPE OF SHARP INSTRUMENT (like a screwdriver). Instead, we recommend using a plastic or rubber coated instrument, such as the handle of rubber coated pliers. Then, gently raise the adapter up from the board by inserting the coated instrument at each corner between the pins and carefully pushing up on the adapter (do not let the instrument touch the board itself). This may result in minor damage to the adapter, but will ensure that no damage is done to the board.

*Changes since December 30, 1997

Order Number **Description**

Adapter Disclaimer (Con't)

We hope that by calling your attention to these important matters, you will achieve prompt operational success of your Nohau emulator system.

Please call your nearest Nohau representative or contact Nohau Technical Support directly for more information.

**Soldering/Desoldering Adapter Services*

Unless you are highly skilled in soldering fine-pitch IC's, it may be a good idea to use a professional soldering company to do this for you. This procedure involves both removing IC's and soldering in the adapter. It is very easy to damage both your PC board and your adapter if you don't have the right tools and the proper skills required to do this properly. Nohau recommends the following two companies who can reliably perform these services, which may save you both time and money.

Best Electronics Soldering Technologies, Inc. (BEST, INC.) .....(847) 699-1025
Emulation Technology .....(800) 232-7837

SOFTWARE SUPPORT PACKAGES

Avocet Systems, Inc.
C Compiler

AVOCET / HTXC51XA..... Discontinued — Replaced by Hi-Tech HTXC51XA family C Compiler.

Avocet is a registered trademark of Avocet Systems, Inc.

Archimedes Software, Inc.
C Compiler / Assembler / Simulator

ARCHIMEDES / IDE-8051XA..... Includes an ANSI C compiler, linker, librarian, simulator / C debugger.

Archimedes is a trademark of Archimedes Software, Inc.

Hi-Tech Software
C Compiler

***HI-TECH / HTXC51XA**..... Hi-Tech HTXC51XA family C Compiler.

HI-TECH is a trademark of Hi-Tech Software.

TASKING, Inc.
C Compiler / Assembler

***TASKING/TK012-002**..... C Compiler/Assembler/Linker/Simulator and EDE Package.
***TASKING/TK012-000**..... Assembler/Linker Package.

TASKING is a registered trademark of Tasking, Inc.

nohau CORPORATION

EMUL51XA™-PC
U. S. Parts List

April 15, 1998

URL: http://www.nohau.com

Order Number **Description**

HARDWARE UPGRADE PRICES

This service is available only if the unit to be upgraded is a working unit in good condition, as judged by Nohau Corporation. Trace upgrades available: speed and buffer size. POD upgrades available: speed, emulation RAM size. Upgrade warranty period is three months or until the expiration of the original warranty period, whichever is longer.

Trace board upgrades..... Price difference plus .
Example: Upgrade to IETR128-25 from IETR128-20: - + =
POD board upgrades..... Price difference plus .
Example: Upgrade to IE-30 from IE-16: - + =

EXTENDED HARDWARE WARRANTIES

IT IS THE CUSTOMER'S RESPONSIBILITY TO RENEW HARDWARE WARRANTIES. NO WARRANTY EXPIRATION REMINDER NOTICES WILL BE SENT TO CUSTOMERS BY NOHAU.

Purchase of each major EMUL51XA-PC item is covered by a one-year warranty as described elsewhere in this list. At the end of the first year, an additional year of hardware service coverage is available. Coverage must be continuous and is not available if coverage has been allowed to lapse. An additional year of coverage may also be purchased each year at the time an additional paid year's coverage ends.

Communication interface extended warranty coverage, 1 yr. For Communication interface.
Trace extended warranty coverage, 1 yr. ..... For Trace.
POD extended warranty coverage, 1 yr. ..... For POD. Bondout PODs are warranted for one replacement if Nohau determines the failure was not due to damage caused by the user's action.

NON-WARRANTY REPAIR

Repair service for units beyond an applicable initial one-year warranty period, repairs not covered by that warranty, or for customers who have elected to not carry an extended hardware warranty.

Hourly rate .....
Minimum charge .....
Maximum charge ..... One half the purchase price.

Bondout POD repair,
chip-replacement charge ..... $

Prices are subject to change without notice. The EMUL51XA™-PC Communication interface board, Trace board, Communication interface Cable, and POD (excluding the bondout processor) are sold with a one-year warranty starting from the date of purchase. The bondout processor on the POD is warranted for one replacement if Nohau determines the failure was not due to damage caused by the user's action. Each optional adapter, cable, and extender is sold with a 90-day warranty, except that it may be subject to repair charges if damage was caused by the user's actions. The EMUL51XA™-PC Emulation software is sold with no warranty. 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. Third-party software sold by Nohau carries the manufacturer's warranty. EMUL51XA-PC is a trademark of Nohau Corporation. P51XA is a trademark of Philips Semiconductors. Windows is a registered trademark of Microsoft Corp. Technical support to be provided by local area representative.

*Changes since December 30, 1997

Order Number **Description**

In-Circuit Emulators for the P51XA

The following information is a cumulative addendum to the EMUL51XA-PC price list dated April 15, 1998.

PODS

General Features

The communication interface connects to the POD board through the cable supplied with the communication interface. The POD board usually requires an adapter, listed under the Adapters section. The POD board has emulation memory, and an on-board oscillator. Jumpers select POD or user target crystal or oscillator. The POD and communication interface can run stand alone without being connected to any user target board. EMUL51XA-PC debugging software is included with the purchase of a pod, and supports Microsoft® Windows™ 3.1x / '95 / NT.

These PODs support both Internal Mode (when the chip is started with /EA high) and External Mode (when the chip is started with /EA low). Hardware breakpoints can be set on addresses of external data read/write with word resolution and can be set either on "internal" code addresses or "external" code addresses. All pods include a power supply.

Note regarding Data / Address Bus Configurations: The configurations of a 8-bit data bus and a 12-bit address bus in external mode are not supported.

P51XAG3:

POD-51XAG3-256 / IE-16 ..... 16 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate 44 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.

POD-51XAG3-256 / IE-20 ..... 20 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate 44 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.

POD-51XAG3-256 / IE-25 ..... 25 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate 44 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.

POD-51XAG3-256 / IE-30 ..... 30 MHz POD board for P51XAG3. 256 kB emulation RAM. Requires separate 44 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.

POD-51XAG3-1M / IE-30 ..... 30 MHz POD board for P51XAG3. 1M emulation RAM. Requires separate 44 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.

POD-51XAG3-2M / IE-30 ..... 30 MHz POD board for P51XAG3. 2M emulation RAM. Requires separate 44 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.

nohau CORPORATION

EMUL51XA™-PC U. S. Parts List Addendum

May 18, 1998

URL: http://www.nohau.com

Order Number	Description
PODs (Con't)	
P51XAS3:	
POD-51XAS3-256 / IE-16	16 MHz POD board for P51XAS3. 256 kB emulation RAM. Requires separate 68 or 80 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.
POD-51XAS3-256 / IE-30	30 MHz POD board for P51XAS3. 256 kB emulation RAM. Requires separate 68 or 80 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide.
POD-51XAS3-1M / 1M-30	Part name correction; replaced by POD-51XAS3-IM / IE-30.
POD-51XAS3-1M / 1E-30	30 MHz POD board for P51XAS3. 1M emulation RAM. Requires separate 68 or 80 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.
POD-51XAS3-2M / IE-30	30 MHz POD board for P51XAS3. 2M emulation RAM. Requires separate 68 or 80 pin adapter to connect to target (see "ADAPTERS"). Includes power supply (PWRSUP6), EMUL51XA-PC debugger software, and EMUL51XA-PC/MANUAL User's Guide. Call Nohau for availability.

ADAPTERS AND ACCESSORIES

44 pin Adapters for G3 pods

EDI / 44PG/PL-L	Adapter to plug 44 pin POD into 44-pin PLCC socket.
EDI/ 44PG/LC-SD	Adapter assembly, 44 pin PGA socket to 44 pin LPFQ, to solder to user target board. Includes one top and one EDI/44LC-SD base.
EDI/ 44LC-SD	Additional base only. 44 pin LPFQ solder down base for EDI/44PG/LC-SD.

68 pin Adapters for S3 pods

ET / AP4-68-SUB1	Adapter for 68-pin PLCC socket.

80 pin Adapters for S3 pods

ES / 110-7393-80	Adapter for 80 pin LPFQ. Includes one top and one ES/000-4532 base.
ES / 000-4532	Additional base only. 80 pin LPFQ solder down base for ES / 110-7393-80.

Other Accessories

PWRSUP6	Additional or replacement 6 amp power supply. This power supply is included with all EMUL51XA-PC pods and is not normally intended to be sold separately.

Prices are subject to change without notice. Depending on stock availability, orders placed before 12 noon Pacific Time according to Nohau's terms and conditions are shipped the same day. Orders placed after noon are shipped the following business day. The EMUL51XA™-PC Communication interface board, Trace board, Communication interface Cable, and POD (excluding the bondout processor) are sold with a one-year warranty starting from the date of purchase. The bondout processor on the POD is warranted for one replacement if Nohau determines the failure was not due to damage caused by the user's action. Each optional adapter, cable, and extender is sold with a 90-day warranty, except that it may be subject to repair charges if damage was caused by the user's actions. The EMUL51XA™-PC Emulation software is sold with no warranty. 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. Third-party software sold by Nohau carries the manufacturer's warranty. EMUL51XA-PC is a trademark of Nohau Corporation. P51XA is a trademark of Philips Semiconductors. Windows is a registered trademark of Microsoft Corp. Technical support to be provided by local area representative.

NOHAU Sales Offices, Reps and Distributors

INTERNATIONAL

ARGENTINA
I. T. D. Ingeniería S. A.
25 de Mayo 611 - 7o 4
1002 Buenos Aires
Argentina
Tel : +54 1 312-1079/9103
Fax: +54 1 315-3431
E-Mail itding@datamarkets.com.ar

AUSTRALIA
Electro Optic s Pty. Ltd.
Level 1, 1 Nelson Street
Kenthurst
New South Wales 2156
Australia
Loca ltel : (02) 9654 1873
Int. Tel : 61 - 2 9654 1873
Fax: 61 - 2 9654 1539
E-Mail sales@electro.com.au
URL: http://www.electro.com.au

AUSTRIA
Ing. M. Schwar z Ges.m.b.H.
Badstrasse 6
A-234 0Moedling
Austria
Local Tel : 02236/42 7 64-0
Int. Tel: +43 2236 42764-0
Fax: +43 2236 42 7 64-11
E-Mail ms@m-schwarz.com
URL: http://www.m-schwarz.com

BELGIUM
See Benelux

BENELUX
(Belgium, Netherlands,
Luxembourg)
TRITEC Benelux B. V.
P. O. Box 212
3340 A EHendri kd oAmbacht
The Netherlands
Local Tel : (078) 681 61 33
Int. Tel : +31-78-681 61 33
Fax: +31-78-682 00 30
E-Mail development tools@tritec.nl

BRAZIL
Anacom Software
Rua Conceição, 627
São Caetano d Sul
09530-06 0São Paulo
Brazil
Local Tel : (011) 453-5588
Int. Tel : +55 - 11 - 453-5588
Fax: +55 - 11 - 441-5177
E-Mail vendas@anacom.com.br
URL:
http://www.anacom.com.br/empresas/nohau/nohau.htm

CANADA
Techmatron Instruments Inc.
130 Principale
Laval Prov. o Québec H7W 3S6
Canada
Tel : 1 - 514 689-5889
Fax: 1 - 514 689-0868
CompuServe:
techmatronmtl@compuserve.com

NOHAU Sales Offices, Reps and Distributors

INTERNATIONAL

CHINA
Nohau Technical Support Center
Microcontroller Beijing Open Laboratory (BOL)
Room 401, P. O. Box 9716
Beijing 100101
P. R. China
Local Tel : (010) 6205 9495
Int. Tel: +86-10-6205 9495
Fax: +86-10-6803 6796
E-mail bol@eastnet.com.cn

DENMARK
Nohau Danmark A/S
Naverland 2
DK 260 0Glostrup
Denmark
Tel : 45 43 44 60 10
Fax: 45 43 44 60 20
E-Mail salg@nohau.dk
URL: http://www.nohau.dk

EGYPT
Telecomp International
29 Anas Ib nMale kstr.,
Mohandessin Guiza 12411
Egypt
Loca ltel: 3488841
Tel : +20 2 3488841
Fax: +20 2 3489812

FINLAND
Fintronic/Computer 2000 Finland Oy
P.O. Box 44
(Pyyntitie 3)
FIN-0223 Espoo
Finland
Local Tel: 09 887 33 330
Int. Tel : +358 9 887 33 330
Fax: +358 9 887 33 289
E-Mail Fintronic@c2000.fi

FINLAND (Con't)
Arrow-Finland Oy
Työpajak 5
PL25
00581 Helsinki
Finland
Int. Tel : +358-476660
Fax: +358-47666356
E-mail kari.keskikastari@arrow-finland.fi
URL: http://www.arrowfi.com

FRANCE
EMULATIONS S.A.R.L.
A13 -Burospace
Chemin d eGizy
91572 BIEVRES CEDEX
France
Loca lTél : 01 69 41 28 01
Int Tél : 33 - 1 69 41 28 01
Local Fax : 01 60 19 29 50
Int. Fax: 33 - 1 60 19 29 50
E-Mail servcomm@emulations.fr

GERMANY
Nohau Elektronik GmbH
Goethestr. 4
7543 3Maulbronn
Germany
Local Tel : 07043/92470
Int. Tel : 49 - 7043-92470
Fax: 49 - 7043-924718
Sales: 49 - 7043-924725
Support: 49 - 7043-924726
E-Mail sales@nohau.de
URL: http://www.nohau.de

NOHAU Sales Offices, Reps and Distributors

INTERNATIONAL

GREAT BRITAIN
Nohau UK Ltd.
The Station Mill Alresford
Hampshire S024 9JG
England
Local Tel : 01962-733 140
Int. Tel : 44 - 1962-733 140
Fax: 44 - 1962-735 408
E-Mail sales@nohau.co.uk
URL: http://www.nohau.co.uk

GREECE
Pouliadis Associates Corp.
Aristotelous St. 3/Sygrou Avenue 150
Athens 17671
Greece
Local Tel : (01) 924 20 72
Int. Tel : +30 - 1 - 924 20 72
Fax: +30 - 1 - 924 10 66
E-Mail espathas@pouliadis.gr

INDIA
Microcomputer Solutions Pvt. Ltd.
22/ 6Premnagar
behin dPushp a Mangal Karyalaya
Bibwewadi
Pune 411 037
India
Local Tel : 0212 - 512164 0212-
510615
Int. Tel : 91 - 212 - 512164
Fax: 91 - 212 - 516798
E-Mail:
MCS.PUNE@INTELMAC.sprintrpg.ems
.vsnl.net.in

INDONESIA
See Singapore

ISRAEL
ITEC Ltd.
P. O. Box 10002
Tel Aviv 61100
Israel
Local Tel : 03 - 6491202
Int. Tel : 972 - 3 6491202
Fax: 972 - 3 6497661
E-Mail itec@netvision.nel.il

ITALY
MICROTASK Embedded S.r.l.
Piazz aVirgilio 3
201 2 3Milano
Italy
Local Tel : (02) 49 82 051
Int. Tel : +39 2 49 82 051
Fax: +39 2 49 86 949
E-Mail microtask@cdc.it

JAPAN
Sophia Systems Co., Ltd.
6-2 ,Minamikurokawa, Asao ku,-
Kawasaki-city, Kanagawa 215
Japan
Local Tel : 044-989-7239
Int. Tel : +81-44-989-7239
Fax: +81-44-989-7005
E-Mail TSC@sophia-systems.co.jp
URL: http://www.sophia-systems.co.jp

KOREA
Zeus emTEK Co. Ltd.
RM 202 Woo-Young Building,
#242 Poi-Dong,
Kangnam-Gu, Seoul, Korea
Local Tel : (02) 3463-7841
Int. Tel : 82 - 2 3463-7841
Fax: 82 - 2 3463-7844
E-Mail: zeuscom@nuri.net
URL: http://www.emtek.co.kr

NOHAU Sales Offices, Reps and Distributors

INTERNATIONAL

LUXEMBOURG
See Benelux

MALAYSIA
See Singapore

THE NETHERLANDS
See Benelux

NEW ZEALAND
Nilsen Technologies (NZ) Limited
Unit 4, Ambury Court
1 Porters Avenue, Eden Terrace
Auckland
New Zealand
Local Tel: 09-3092464
Int. Tel: 64-9-3092464
Fax: 64-9-3092968
E-Mail: sales@nilsentech.co.nz

NORWAY
Nortelco AS
Ryensvingen 3
Postboks 116, Manglerud
N-0612 Oslo
Norway
Tel: (+47) 22 67 40 20
Fax: (+47) 22 67 40 30

PORTUGAL
FATRONICA, S. A.
Taguspark
Edif. Tecnologia I, No 18
2780 Oeiras
Portugal
Local Tel: 01 - 4213141
Int. Tel: 351 - 1- 4213141
Fax: 351 - 1- 4214747
E-Mail: fatron@mail.telepac.pt

RUSSIA
EFO Ltd.
Politechnicheskaja Str. 21
St. Petersburg
194021 RUSSIA
Local Tel: (812) 247-8900
Int. Tel: +7 812 247-8900
Fax: +7 812 247-5340
E-Mail: zav@efo.spb.su

SINGAPORE
EMULTEQ PTE LTD
8 Kaki Bukit Road 2
#03-27 Ruby Warehouse Complex
Singapore 417841
Local Tel: 749-0870
Int. Tel: +65 749-0870
Fax: +65 749-0332
E-Mail: gde9000@singnet.com.sg

SOUTH AFRICA
Eagle Technology
31-35 Hout St.
P. O. Box 4376
Cape Town 8000
South Africa
Local Tel: (021) 234 943
Int. Tel: 27 - 21-234 943
Fax: 27 - 21-244 637
E-mail: asher@eagle.co.za
URL: http://www.eagle.co.za

SPAIN
Captura Electronica sccl
C/. Albert Einstein, S/N
Edificio Forum de la Tecnologia
E-08042 Barcelona
Spain
Local Tel: (93) 291 76 33
Int. Tel: 34 - 93 291 76 33
Fax: 34 - 93 291 76 35
E-Mail: capel01@ibm.net

NOHAU Sales Offices, Reps and Distributors

INTERNATIONAL

SWEDEN
Nohau Elektronik AB
Derbyvägen 4
S-212 35 Malmö
Sweden
Local Tel: 040 - 59 22 00
Int. Tel: 46 - 40 59 22 00
Fax: 46 - 40 59 22 29
E-Mail (Info): info@nohau.se
E-Mail: support@nohau.se
URL: http://www.nohau.se

SWITZERLAND
Anatec AG
Sumpfstrasse 7
CH-6300 Zug - Switzerland
Local Tel: 041 748 32 32
Int. Tel: +41 41 748 32 32
Fax: +41 41 748 32 31
E-Mail: anatec_ag@compuserve.com

TAIWAN
Hi-Lo System Research Co., Ltd.
4F, No. 2, Sec. 5
Ming Shen E. Rd.
Taipei, Taiwan, R. O. C.
Local Tel: 02 27640215
Int. Tel: 886-2-27640215
Fax: 886-2-27566403
E-Mail (tech):
support@hilosystems.com.tw
E-Mail (sales): sales@hilosystems.com.tw

THAILAND
Complex Technology Co., Ltd.
159/2 Ranong 1 Rd.
Dusit, Bangkok 10300
Thailand
Local Tel: (02) 668-5080
Int. Tel: 66-2-668-5080
Fax: 66-2-668-5081
E-Mail:
complex@asiaaccess.net.th

U. K.
See Great Britain

NOHAU Sales Offices, Reps and Distributors

INTERNATIONAL

Look up Zip Code to find sales office:

00000 - 01199	Contact Nohau Corp.
01200 - 02799	MA: AD Electronics
02800 - 02999	RI: AD Electronics
03000 - 03899	NH: AD Electronics
03900 - 04999	ME: AD Electronics
05000 - 05999	VT: AD Electronics
06000 - 06999	CT: AD Electronics
07000 - 07999	NJ: Rical Assoc.
08000 - 08799	NJ: AUSTEK, Inc.
08800 - 08999	NJ: Rical Assoc.
09000 - 09999	Contact Nohau Corp.
10000 - 11999	NY: Rical Assoc.
12000 - 12399	Contact Nohau Corp.
12400 - 12799	NY: Rical Assoc.
12800 - 14999	Contact Nohau Corp.
15000 - 19199	PA: AUSTEK, Inc.
19300 - 19699	PA: AUSTEK, Inc.
19700 - 19999	DE: AUSTEK, Inc.
20000 - 20099	DC: Embedded Technology
20100 - 20199	VA: Embedded Technology
20200 - 20599	DC: Embedded Technology
20600 - 21999	MD: Embedded Technology
22000 - 24699	VA: Embedded Technology
24700 - 26999	Contact Nohau Corp.
27000 - 28999	NC: ETA
29000 - 29999	SC: ETA
30000 - 31999	GA: ETA
32000 - 34999	FL: ETA
35000 - 36999	AL: ETA
37000 - 38599	TN: ETA
38600 - 39799	MS: ETA
39800 - 45999	Contact Nohau Corp.
46000 - 46299	IN: The Dearborn Group
46300 - 46699	IN: Silicon Engines, Ltd.
46700 - 47899	IN: The Dearborn Group
47900 - 47999	IN: Silicon Engines, Ltd.
48000 - 48599	MI: The Dearborn Group
48600 - 48799	Contact Nohau Corp.
48800 - 49299	MI: The Dearborn Group
49300 - 51999	Contact Nohau Corp.
52000 - 52099	IA: Silicon Engines, Ltd.
52100 - 52699	Contact Nohau Corp.
52700 - 52899	IA: Silicon Engines, Ltd.
52900 - 52999	Contact Nohau Corp.
53000 - 53599	WI: Silicon Engines, Ltd.
53600 - 53699	Contact Nohau Corp.
53700 - 53799	WI: Silcon Engines, Ltd.
53800 - 54999	Contact Nohau Corp.
55000 - 56799	MN: Claassen & Associates
56800 - 59999	Contact Nohau Corp.
60000 - 61999	IL: Silicon Engines, Ltd.
62000 - 62299	Contact Nohau Corp.
62300 - 62399	IL: Silicon Engines, Ltd.
62400 - 62499	Contact Nohau Corp.
62500 - 62799	IL: Silicon Engines, Ltd.
62800 - 69999	Contact Nohau Corp.
70000 - 71499	LA: Columbus Technology
71600 - 72999	AR: Columbus Technology
73000 - 74999	OK: Columbus Technology
75000 - 79799	TX: Columbus Technology
79800 - 79999	Contact Nohau Corp.
80000 - 81699	CO: Harco Company
81700 - 84999	Contact Nohau Corp.
85000 - 86599	AZ: MicroWare Technology
86600 - 89999	Contact Nohau Corp.
90000 - 93599	CA: MicroWare Technology
93600 - 96199	CA: Enable Engineering Co.
96200 - 99999	Contact Nohau Corp.

NOHAU Sales Offices, Reps and Distributors
UNITED STATES

AD Electronics
10 Pepperidge Trail
Old Saybrook, CT 06475-1055
Tel : 1 - 860 388-2204
Fax: 1 - 860 388-9607
E-mail gary.j.mcneil@snet.com

AusTek, Inc.
P. O. Box 2156
Jenkintown, PA 19046-0756
Tel : 1 - 215 924-6228
Fax: 1 - 215 924-4448
CompuServe:
austek@compuserve.com

Claassen & Associates
P.O. Box 50634
Minneapolis, MN 55405
Tel : 1 - 612 470-2100
Fax: 1 - 612 430-3898
E-Mail claassen@winternet.com

Columbus Technology
9003 Oakwin dCt. #104
Dallas, TX 75243-6351
Tel 1 - 214 342-1492
Fax: 1 - 214 343-8011
E-Mail johnm@clmbs.com
URL: http://www.clmbs.com

Dearborn Group, Inc.
27007 Hills Tech Dr.
Farmington Hills, MI 48331-5727
Tel : 1 - 248 488-2080
Fax: 1 - 248 488-2082
E-Mail dg@dgtech.com
URL: http://www.dgtech.com

Embedded Technology Corp.
690 Pine St.
Herndon, VA 20170-4600
(Mail Address: P. O. Box 1065
Herndon, VA 20172-1065)
Tel : 1 - 703 742-0040
Fax: 1 - 703 318-9390
E-Mail embedded@erols.com
URL: http://www.erols.com/embedded

Enable Engineering Co. (EECo)
430 Peninsula Ave. #4
San Mateo, CA 94401-1653
Tel : 1 - 650 375-0409
N.Cal :1-800 68-NOHAU
(1 - 800 686-6428)
Fax: 1 - 650 375-8666
E-mail: sales@eecosales.com
URL: http://www.eecosales.com

ETA
4680 Lipscomb St. N. E., #10A
Palm Bay, FL 32905
Tel 1 - 407 676-9090
Toll Free: 1 - 888 965-9090
Fax: 1 - 407 676-9059
E-mai lchoward@iu.net

Harco Engineering, Inc.
2300 Emerald Road
Boulder, CO 80304-0914
Tel : 1 - 303 447-9611
Fax: 1 - 303 449-5469
E-Mail tomharr@harco.com
URL: http://www.harco.com

NOHAU Sales Offices, Reps and Distributors

UNITED STATES

MicroWare Technology
9761 Caminit Suelto
San Diego, CA 92131-2115
Tel : 1 - 619 693-4280
Fax: 1 - 619 693-1438
E-Mail gperkins@microware-tech.com

Nohau Corporation
51 E. Campbell Ave.
Campbell, CA 95008-2053
Tel : 1 - 408 866-1820
Fax: 1 - 408 378-7869
E-Mail sales@nohau.com
support@nohau.com
URL: http://www.nohau.com

Rical Associates
43 Fulton Street
Newark, NJ 07102-4506
Tel 1 - 973 643-0880
Fax: 1 - 973 643-4409
E-Mail ttease1@aol.com

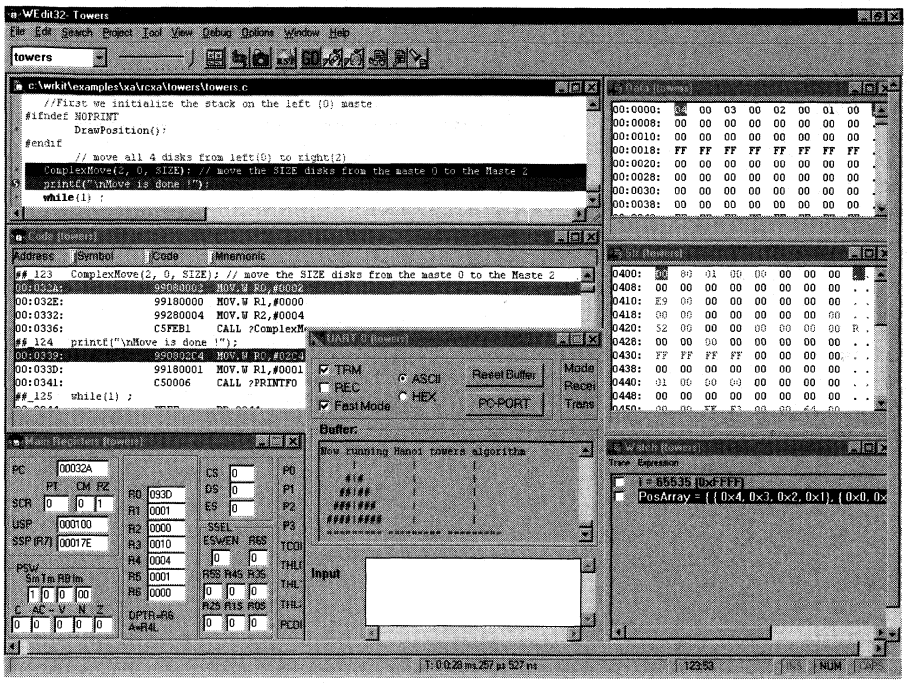
Silicon Engines, Ltd.
2101 Oxford Road
Des Plaines, IL 60018-1919
Tel 1 - 847 803-6860
Fax 1 - 847 803-6870

Rentals:
Technilease
43 Fulton Street
Newark, NJ 07102-4506
Tel : 1 - 800 451-RENT
(1-800 451-7368)
1 - 973 621-7368
Fax: 1 - 973 643-4409
E-mail ttease1@aol.com

Rkit-XA

RAISONANCE

Rkit-XA is a complete set of software tools for developing applications based upon the Philips 80C51XA. Rkit-XA features an optimizing ANSI compliant C language Compiler, an ASM-51 like Macro-Assembler, and a powerful integrated Simulator. Rkit-XA provides a clean simple solution for migrating from the classic 8051 to the extended architecture of the XA.



Integrated into WEdit

Rkit-XA is delivered with WEdit, a Microsoft Windows® based fully integrated development environment (IDE). WEdit features a color syntax highlighting editor, a project manager, and full on-line help as well as providing all the facilities to call the different Rkit-XA tools.

WEdit is a powerful environment, but simple and easy to use. It is fully compliant to the Microsoft specifications for Windows 95 and Windows NT. WEdit permits the simultaneous management of both 8051 and XA projects, by selecting the appropriate tools.

DOS and 16-bit Versions

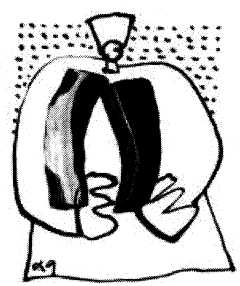
WEdit also provides convenient access to 16-bit DOS and Windows 3.1 versions of the 51 tool set. As an additional advantage, the DOS tools run in a DPMI mode and are not memory limited.

Flexible Tools

Each of the Rkit-XA tools are tightly integrated into WEdit, offering the professional developer a complete and cleanly integrated tool kit. In addition, each tool: assembler, compiler, linker, simulator, and so on, can be run, "stand-alone", as an individual self-supported tool.

Easy migration from the 8051

The Rkit-XA provides an upward compatibility path for converting your 8051 based applications. Each tool accepts existing 8051 directives and syntax. This permits the new XA core to be treated like a superset of the 8051. An 8051 based C language file is directly compiled. An 8051 based assembler file is easily translated by the integrated translator, and then assembled in the usual manner.



Optimizing ANSI-C Compiler

A super-set of ANSI-C

RC-XA implements the continuously evolving ANSI standards for he C language, expanded with XA specific keywords and powerful new directives. These keywords include 8051 specific keywords:

bit	generic	code	sbit	sfr
asm	at	_at_	near	far
interrupt	exception	trap	system	user
task	priority	group	swinterrupt	using
reentrant	stack	istack		

Memory Models

The memory model specifies the configuration and the size of the application:

Model	Memory type	Notes
TINY	Code+constant< 64 KB data < 32 KB	Mode: PAGE 0 Compact generic pointers
SMALL	Code+constant< 64 KB data < 64 KB	Mode: PAGE 0
COMPACT	Code < 16 MB constant < 64 KB data < 64 KB (16MB far)	Mode: standard near pointers by default (except for ptr to functions)
LARGE	Code+constant < 16 MB data < 64 KB (16MB far)	Mode: standard far pointers by default
*HUGE**	Code+constant < 16 MB data < 16 MB	Mode: standard far pointers by default

**huge is not available on versions < V3.0*

Base Types

- Integer Types: 8 bits : "signed char" and "unsigned char",
  16 bits : "signed int" and "unsigned int",
  32 bits : "signed long" and "unsigned long",
  Real Types: 32 bits : "float" (IEEE754 single precision),

Pointers Types

Two types of pointers are always available, generic pointers and space qualified pointers. Moreover, the distinction between near and far pointers allows to adapt as well as possible the language to the application constraints.

Code Optimizations

RC-XA optimizes the code to be both compact and fast. Many global optimizations have been inherited from our highly optimizing RC-51 compiler, others have been introduced specifically to suit the XA core. RC-XA allows to reach the following Benchmark results:

Program (model)	Code Size* (module / total)	Speed** (@ 12 MHz)
Sieve (Small)	156 / 1098	2.41 s
Dhrystone (Small)	1628 / 3304	194 ms
Whetstone (Small)	2624 / 7682	1169 ms

* Optimized Size ** Optimized Speed

Parameter Passing

One of the benefits of the XA core is the availability of many more registers. RC-XA places the first parameters (up to 8 bytes) into registers, then pushes the others onto the processor's stack. This method permits recursion and makes the code naturally compact.

Libraries

RC-XA is supplied with ANSI C standard libraries as described in: math.h, stdio.h, string.h, ctype.h, and stdlib.h standard header files.

RAISONANCE
755, avenue Ambroise Croizat
38920 CROLLES FRANCE
Tel. : (+33) 4 76 08 18 16 Fax : (+33) 4 76 08 09 97
email: info@raisonance.fr
www: http://www.raisonance.fr

MA-XA Macro-Assembler

The MA-XA syntax is directive and macro compliant with MA-51 (or ASM-51).
The instruction syntax implemented is compliant with the rules specified by Philips. In addition the old 8051 notations are accepted too. For example, both "@Ri" and "[Ri]" can be used to specify indirection.

ASM-51 to XA Translator

To ease the migration of existing 8051 assembly code to the XA, Rkit-XA includes an assembler instructions translator. This tool emits an AXA source file from an A51 source file by replacing 8051 instructions by their equivalent XA instructions.

RL-XA Linker/Relocator

RL-XA is based on existing RL-51 concepts, and is expanded by new directives. New concepts, like segment grouping, permit better CPU control and more efficient mapping of user memory.
RL-XA produces a symbolic output format (OMF-XA), a HEX output file, and a complete listing file.

KR-XA Real Time OS

KR-XA offers a variety of services, and is fully compliant with KR-51 providing time and memory management, semaphores, synchronizing events, inter-task communications, and more sophisticated functions such as a perpetual calendar. Kernel uses the multitasking features of the XA core to permit fast task switching, full preemption, and a time-slicing mechanism.
KR-XA is fully integrated into Rkit-XA. Task declaration is done at the C-source level, while configuration of the kernel is made at the project level. The SIMICE-XA simulator provides a variety of information on the status of the Kernel and each task.

SIMICE-XA Simulator/Debugger

The simulator is tightly integrated into WEdit and uses the information generated by RC-XA or MA-XA tools to provide fully symbolic high level debugging.
SIMICE-XA takes into account the characteristics of the selected device and simulates —in considerable detail— all internal peripherals. Associated with WEdit, SIMICE-XA permits simulation of multi-processor applications, and offers various solutions that can simulate external inputs and outputs. SIMICE-XA provides various emulator-like functions such as trace management or complex breakpoint control. SIMICE-XA can also be run as a stand alone simulator.

MONITOR-XA ROM Monitor

Rkit-XA is supplied with a ROM Monitor, directly driven by WEdit. The ROM Monitor permits simple commands like "step by step execution" and variable inspection among others. If the target provides a "write CODE" capability, advanced features such as "execution line by line" or dynamic code breakpoint are then possible too.

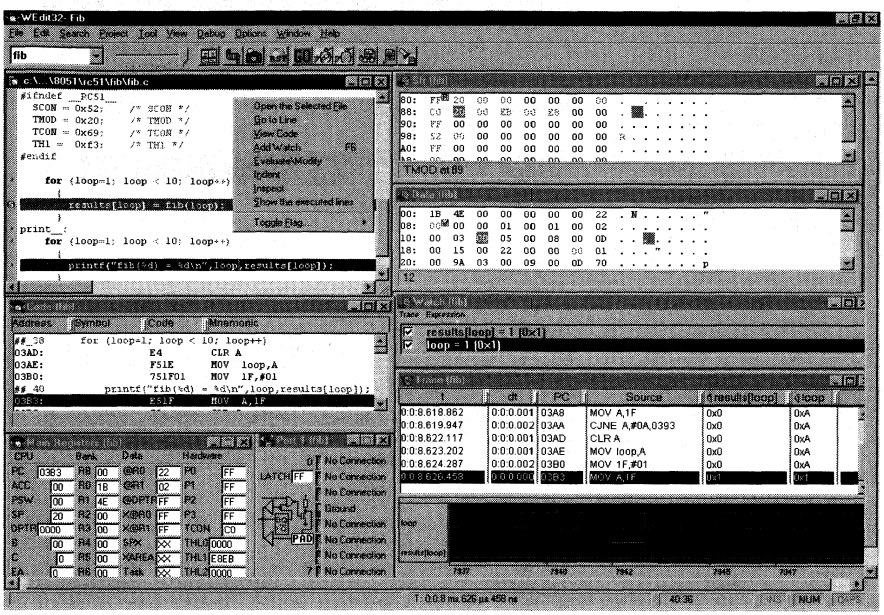
SmartXA Support

Rkit-XA has been adapted to provide full support of SmartXA specifications. SmartXA extensions are only supplied by Philips.

Local distributor :

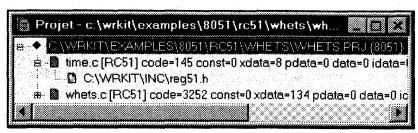
WEdit32

WEdit32 is a fully featured and Integrated Development Environment that provides smooth seamless access to all the tools in the professional developers arsenal. From editing to debugging, WEdit32 can manage all aspects of product development for any member of the 8051, 80C251, or 80C51XA families.



The Editor

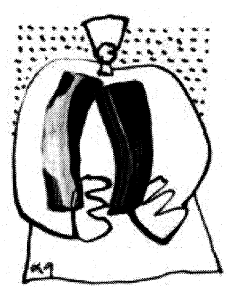
WEdit32 is based on a fast multi-document editor designed to meet the specific needs of the programmers art. The various methods, menus, commands, and shortcuts are all fully compliant with the Microsoft® specifications for Win95® and Win NT®. Classic commands, such as string search and block actions, have been added to provide specific and advanced features for the professional developer. A (customizable) color syntax highlighting editor is used to indicate specific syntax elements as they appear in the source file: key words, comments, identifiers, operators, and so on. Whether it's C or assembler, the syntax highlighting is keyed to the intrinsic file type. This permits the professional user to quickly and easily identify those parts of the code responsible for syntactic errors; for instance, an unclosed comment. Other helpful commands; such as "search for matching delimiter" (i.e. bracket, parenthesis, ...) make WEdit32 a powerful easily used tool specifically dedicated to providing your embedded solution.



Project Manager

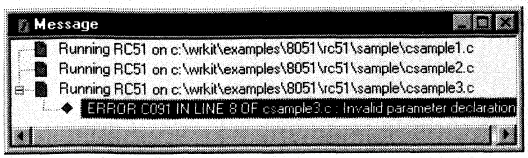
The project manager creates a link between the various files that comprise a project and the tools necessary to create that project. A project is dedicated to a target: 8051, 251 or XA; and each of the project's files are associated, by their file type, to the appropriate translating tool. C and assembler files are translated by the compiler and assembler, respectively. The linker manages object and library files, and output format conversion as necessary. In addition, the user may define that other custom tools be used too, even DOS applications. Options for all tools are globally defined for the course of the project. As specific needs dictate, it is a simple matter to alter any file's individual attributes as necessary to achieve your goals. The Project/Make command directs the integrated "make" utility to build or rebuild the target program for the current project. Each source file will be translated by its associated tool if any of its dependencies are found to be out of date. Dependency analysis, even of directly or indirectly included files, is automatic.

RAISONANCE



lessage Window

ıe message window displays all warning, error, and progress



ıessages generated during the execution of each project or file. licking on an error string in the message window automatically ɔsitions the cursor at the point of that error in the source code indow. On-line context sensitive help is available to determine ɔssible causes for that error.

)n-line Help

he help system provides information on nearly all aspects of /Edit32, and the various integrated tools driven by the IDE.

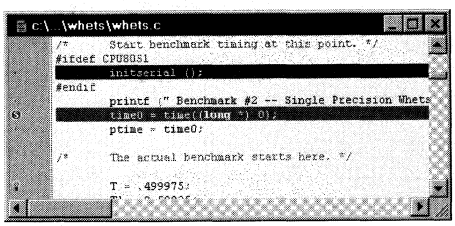
)n-line menu hints appear on the status line whenever you select menu command..

)esktop Context

Vhen a project is closed, editing and debugging contexts such as: ind, size and location of windows, breakpoints, "watch" ɔxpressions, and so on are saved. That same context is ıutomatically restored should the project be reloaded or the lebugging session be restarted.

ntegrated High-level Debug

VEdit32 provides a fully integrated source level debugging ınvironment. All information necessary is derived from the translators ısed to accomplish each step of the process. This includes mundane ıspects such as "path names", and source code specific information ıuch as complex type details.



Vith the simple click of a mouse button, the user can select ımong several powerful capabilities: simulate, monitor, or ɜmulate. The fast smooth integration afforded by WEdit32 ɔromotes a feeling of familiarity and ease of use, while providing ı level of comfort and efficiency that reduces the most difficult ınd complex applications to easily managed tasks. This seamless progression of the "code-translate-link-debug-test" cycle is the result of perfect communication between the programming tools and the debugger. This is the heart of WEdit32.

Raisonance S.A.

755, avenue Ambroise Croizat
38920 CROLLES FRANCE
Tel. : (+33) 4 76 08 18 16 Fax : (+33) 4 76 08 09 97
email: info@raisonance.fr
www: http://www.raisonance.fr

Debugging Modes

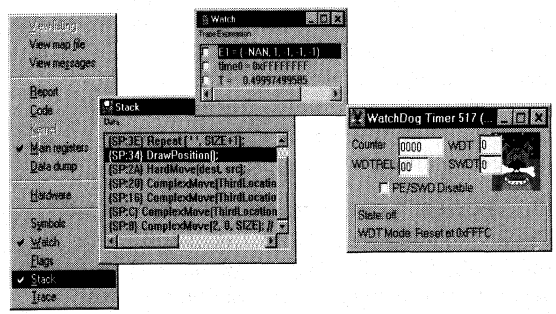
WEdit32 permits both integrated (fully interactive/multi-file "project" oriented), or standalone (interface only) debugging. In "project" mode, WEdit32 presents and manages the entire display from source to final results; permitting detailed analysis of each line of source code. In the interface only, "debugger" mode, WEdit32 provides a powerful and familiar interface for it's built-in debugger/simulator, or—via a fast and easy to use opto-isolated serial interface—connection to an external emulator (available separately) or target based ROM monitor. To round out these capabilities, WEdit32 provides the additional benefit of being able to do "multi processor" simulation between several processors at the same time.

Integral Simulation

WEdit32 includes simulation engines for most 8051, 80C251 and 80C51XA derivatives. The simulator/debugger is cleanly integrated into the presentation windows. A wide range of views can be selected to provide flexible direct examination of all memory spaces as well the all internal peripherals.
The simulation engines perform detailed and faithful simulations (including IDLE or Power down mode), of all peripherals—including interrupt and watchdog events—present on the selected component.

Advanced Features

WEdit32 provides a rich variety of views into an application:.

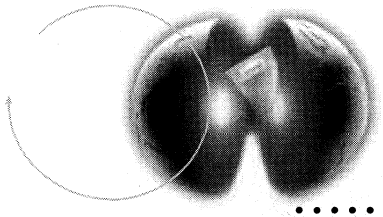


Hardware Requirements

WEdit32 requires an IBM-PC compatible, fitted with a 486 or above and at least 8Mb of RAM, Windows 95 or NT operating system (Windows 3.1 requires that the latest WIN32S be installed), and SVGA (or higher) video mode capability.

Local distributor :

The XA Development Solution



Now your application can really use the XA extended architecture

Take advantage of the extended architecture

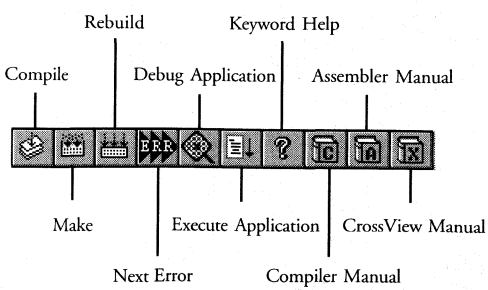
The Philips XA architecture is a family of true 16-bit microcontrollers that provides upward compatibility from the industry-standard 8-bit 80C51. With its experience in the 8051 market, TASKING has developed a toolset that eases the migration from 8051 and enables your application to take advantage of the XA extended architecture. The complete toolset includes a C++ compiler, ANSI C compiler, macro assembler, linker/locator, libraries, CrossView Pro debugger, and EDE our embedded development environment that provides a composite interface to the complete toolset.

Embedded Development Environment

The Embedded Development Environment (EDE), voted by EETimes "Best Technology", is a single, easy to use interface, that integrates the members of the XA toolset enabling you to build, edit, compile, and debug your embedded applications. EDE provides the following:

- Integrated tools delivering a rapid edit-compile-debug process
- Easy project setup with generation of Makefiles
- Push button control over a variety of development tasks
- Language sensitive editor (C & C++) with error parser
- Access to third party tools

EDE is project oriented and helps you organize your work by grouping together all the files associated with your project, and identifying the XA and its tools as the chosen architecture.



C Compiler

The C compiler for the XA architecture is ANSI compliant and uses the latest compiler technology to produce very efficient code. The state-of-the-art optimization techniques, each separately selectable, provide extensive control over the size and speed of the generated code. The standard ANSI language definition is enriched with a number of keywords making the compiler ideally suited to embedded program development, including writing exception handling in C and controlling the alignment and allocation of structures in memory.

Features

- **Total integrated development environment**
- **Tight and seamless navigation**
- **Tailorable to your working environment**
- **Visual point & click interface**
- **Highly optimized Compiler**
- **Kernel aware debugging**
- **Available on Windows 3.1, Windows 95, Windows NT, UNIX (SUN and HP9000)**
- **Proven and stable technology**

Data Types

All ANSI C data types are supported. In addition to these types, the *_sfrbit, _sfrbyte, _bit* and *_bitbyte* types are added. The keywords *_sfrbit* and *_sfrbyte* are available to access special functions registers which deal with I/O. Special function registers are treated like memory mapped variables declared with the *volatile* type qualifier.

Memory Models

The C compiler supports several reentrant memory models: tiny, small, medium and large. Each model uses a

TASKING

different default storage type for (non-register) automatic variables, (non-register) parameter passing areas, and declarations without explicit storage type. For more details see the memory model overview. Separate versions of the C and runtime libraries are supplied for all supported models, avoiding the need to recompile or rebuild these when using a particular model.

Compiler

- ANSI C ensures early error detection
- Complete XA support
- Intelligent configuration of system parameters
- Extensive user controlled mapping of memory, code and data
- In-line assembly
- IEEE-754 single and double precision floating point
- Complete ANSI C and run-time libraries
- IEEE-695 object format to ensure interoperability

Intrinsic Functions

The XA Compiler has a number of built-in (intrinsic) functions that enable you to access specific XA instructions directly in C instead of writing inline assembler.

In total, the XA compiler supports 26 intrinsic functions including:

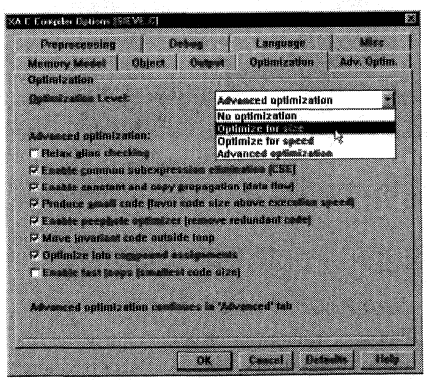
_rol	Use the `ROL` instruction to rotate left
_ror	Use the `ROR` instruction to rotate right
_nop	Generate `NOP` instruction
_testclear	Test and clear bit using the `JBC` instruction
_div32	32-bit by 16-bit signed divide using `div.d` instruction

Interrupt Functions

A C function can serve as an interrupt service routine by specifying the interrupt number via the *_interrupt* function qualifier. The *_using* function qualifier can be used to define the value of the PSW placed in the interrupt vector table. The compiler emits the corresponding interrupt vector and the appropriate entry and exit code.

Libraries

The compiler package includes ANSI C libraries, run time libraries including I/O calls (+ printf), memory management, arithmetic functions and floating point for all memory models. Floating point libraries are delivered in many different variants. You can choose between a double precision implementation (full ANSI-C) or the faster single precision, and between full IEEE-754 exception handling or a faster version without the trapping. Source code provided for most of the library routines allows you to tailor the libraries to your specific application.



C++ Compiler

The C++ compiler delivers the power of object-oriented design and coding techniques for the XA family. The C++ language is a superset of the C language allowing the intermixing of C and C++ within a single application. Thus, the benefits of C++ can be incorporated into an existing C application one module at a time, providing a graceful migration from C to C++. Inheritance reduces the number of places where software behavior is defined and thereby speeds up development. The C++ compiler automatically includes a pre-link phase when templates are used.

Linker/Locator

- ANSI-C intermodule type checking
- Incremental linking
- Generation of listfile including function call graph of whole application
- Automatic inclusion of corresponding C and floating point libraries
- Supports CPU and user mapping of memory, addressing modes, configuration bits, encryption array etc.
- Supports absolute IEEE-695, Intel Hex and Motorola S-record object format
- Extensive mapfiles

Assembler

The assembler is an optimizing assembler that translates XA assembly language into relocatable object code. An absolute or executable load image is then obtained by using the linker/locator. The assembler is supplied complete with linker, locator, librarian and object format utilities. Features include:

- Intel compatible macro preprocessor (MPL)
- extensive segment directives
- absolute listfile generation

Linker/Locator

An essential part of the software building process, the linker/locator enables you to configure the code to match your target environment. It brings together all the necessary relocatable objects, including library modules, resolves external references and then locates the modules in memory according to your specification.

Utilities

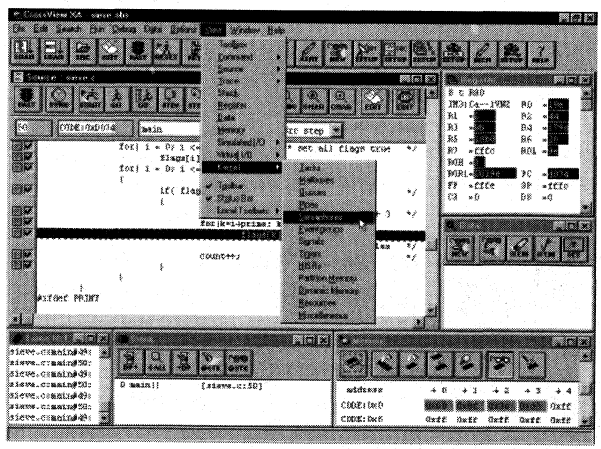
The assembler includes a number of programs to help you at various stages of development. The utilities programs include a librarian, Make, and an IEEE object reader.

The librarian creates a library, adds object modules to a library, removes object files from a library and lists the contents of a library. Make is a utility that automates the task of building or reconstructing your application. It prevents errors by ensuring that applications can be accurately rebuilt and saves time by recompiling only modules that changed since the last build. Make is invoked from the EDE by clicking on the *Make* button in the ribbon bar or from the command line. The IEEE reader enables you to view the contents of files including debug information.

CrossView Pro Debugger

An easty-to-use interface with powerful and extensive debugging features help you debug your applications faster. CrossView Pro is a true windows application complete with multiple, resizable, and independently controlled windows. It combines the flexibility of the C language with the control of code execution found in assembly language bringing functionality that reduces the time spent testing and debugging. Functionality includes:

- tracking scope and monitoring locals
- "intelligent" source window
- double click and right mouse button functions



You choose the windows you need to view the different aspects of your code during debugging.

Source Window

The working window is the source window. It lets you view source, set and clear breakpoints, assertions and code coverage markers, monitor and inspect variables, search for strings, functions, lines and addresses, call functions evaluate expressions, and view performance analysis data. The source window allows you to view your code at C level, assembly level or mixed level, where you have your C code intermixed with the corresponding assembly code.

CrossView Pro

- **Multi-window interface**
- **Code coverage**
- **Profiling**
- **Code and data breakpoints with optional macro execution**
- **Kernel awareness**
- **Single stepping**
- **Stack trace**
- **Simulated I/O**
- **C/C++ expression evaluation**

From the source window you can jump directly into the editor within EDE and you will find yourself positioned at the source line where you had your cursor in the debugger. This gives you immediate access to the source line where you have found a problem, that you want to correct.

Multiple Data Windows

Data windows enable you to watch or show data, browse for locals or globals, double-click to modify values or to expand and contract complex data structures. Within these windows you can reformat (change display radix and type) on an element-by-element basis. You can show or watch locals from any stack level, automatically track and display locals, and easily copy any variable to a new window as show or watch.

Register Window

The register window displays and modifies CPU register values. The window is fully configurable and is updated every time the program is stopped. Highlighted registers indicate what has changed since the last stop.

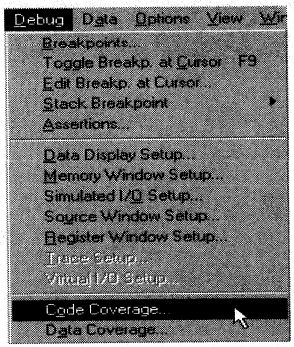
Stack Window

The stack window displays the state of the current stack frame. With simple point and click operations you can, setup

level breakpoints, display source for function calls, and display local variables for selected functions.

Memory Window

The memory window with ASCII display enables you to monitor any address change, double-click to modify, and have complete control over the size and format of data.



Software Assertions

Software assertions let you execute user specified command lists after running every line of source code. This is the software equivalent of data breakpoints, and can be used to set up sophisticated error checking mechanisms that uncover the most elusive bug.

C-Like Macro Language

With C-like macro language you can read and/or modify application variables and call application functions from macros. It has full C expression syntax.

Multiple Execution Environments

CrossView Pro supports multiple execution environments with a standard user interface. The execution environments available are Simulator and ROM Monitor.

ROM Monitor Environment

CrossView Pro ROM debugger can be used with any commercial off the shelf evaluation board or target application. CrossView Pro, running on a host computer system, communicates with the monitor on the target board via an RS232 interface, using a very efficient protocol. The resources used by the monitor program are kept to a minimum.

The monitor uses:

- approximately 3Kbytes code space
- less than 25 bytes of data space
- less than 24 bytes of system stack

Simulator

The simulator environment allows you to test, debug and monitor the performance of code in a known and repeatable environment independent of target hardware. It uses the same description file as the linker/locator when locating your application and therefore knows exactly where and how memory is mapped into. All CrossView Pro features, including C level trace, Code Coverage, and unlimited amount of code/data breakpoints are available to you, so you can test code before target hardware is available.

Cooperation with 3rd Parties

Working with other suppliers of products for the XA gives us the opportunity to improve the tools that we deliver. We cooperate with different hardware manufacturers for Emulators and Evaluation boards, and with software manufacturers for Real Time Kernels. Our extensive cooperation ensures you have access to the tools you need to be your most productive. For more information contact your local sales office or distributor.

The XA toolset is compatible with the realtime kernels from CMX (CMX) and Embedded System Products (RTXC). More kernel support is to be expected. Please contact us for availability or visit our web site for up-to-date information.

Customer Support

When you purchase a TASKING product, it is the beginning of a long term relationship. 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, fax or email.

A maintenance period is included with the purchase of TASKING products and entitles you to enhancements and improvements as well as individual response to problems. Annual maintenance agreements are available at the end of the maintenance period. These maintenance agreements provide you with all program enhancements released during the period of the contract, and assures a response to all problem reports submitted.

Availability

The XA development solution is available for PC (Windows 3.1, 95 and NT), Sun (SunOS and Solaris) and HP9000/700 (HPUX).

We have a policy of continued improvement for our products. For the latest information contact your local sales office.

Memory Models

Model	CPU Mode	Code Space	Data Space
tiny(default)	Page Zero System or User Mode	<= 64K	_near: <= 1K _far: <= 64K
small	Page Zero System or User Mode	<= 64K	_near: <= 1K _far: <= 64K
medium	Large Memory System or User Mode	> 64K	_near: <= 1K _far: <= 64K _huge: > 64K
large	Large Memory System Mode	> 64K	_near: <= 1K _far: <= 64K _huge: > 64K

Memory types

_near — Page Zero or Large Memory mode: Data is direct addressable, located in the default data segment.

_far — Page Zero mode: Data object is only indirectly addressable, located in the default data segment. The data object allocates less than 64K bytes of memory.
Large Memory mode: Data object can reside anywhere in memory. The data object allocates less than 64K bytes of memory.

_huge — Large Memory mode only: Data object can reside anywhere in memory and is not assumed to reside in the default data segment. The data object may allocate more than 64Kbytes of memory.

_bit/_bdat — bit-addressable on-chip RAM. Data object size is limited to 256bits/32 bytes.

_rom — Internal/external ROM. The compiler implies the type qualifier const.

Pointers and memory models

_near — Page Zero or Large Memory mode (all memory models): Pointers declared with _near are 16-bit values. Pointer arithmetic is 16-bit.

_far — Page Zero mode (tiny or small memory model): Pointers declared with _far are 16-bit values. Pointer arithmetic is 16-bit.
Large Memory mode (medium or large memory model): Pointers declared with _far are 32-bit values. Pointer arithmetic is 16-bit, the segment register is not affected, i.e., the objects pointed to must reside within one segment.

_huge — Large Memory mode only (medium or large memory model): Pointers declared with _huge are 32-bit values. Pointer arithmetic is 24-bit, the segment register is updated when required. The objects pointed to may reside in multiple segments.

Product Packaging and Ordering Codes

Each TASKING product comes with full documentation. The documentation is available on-line as well and provides full-text search capabilities for quick and easy lookup of topics.

Product Code	Package contents
TK012-002	EDE, Compiler, Assembler/Linker, CrossView Pro Simulator
TK012-012	EDE, C and C++ Compilers, Assembler/Linker, CrossView Pro Simulator
TK012-000	EDE, Assembler/Linker
TK012-041	CrossView Pro ROM Monitor
TK012-043	CrossView Pro Simulator

Contact TASKING for availability.

Demonstration versions of the XA tools are available on CD-ROM or downloadable from our web site at: http://www.tasking.com

Your Distributor

 XA p65-897-6

UNITED STATES (International Headquarters)

TASKING, Inc.
Norfolk Place, 333 Elm Street
Dedham, MA 02026-4530, USA
Phone: 800-458-8276 (within the USA),
781-320-9400 (outside the USA)
Fax: 781-320-9212
Email: sales_us@tasking.com

THE NETHERLANDS (European Headquarters)

TASKING Software BV
P.O. Box 899
3800 AW Amersfoort, The Netherlands
Phone: +31-33-4558584
Fax: +31-33-4550033
Email: sales_nl@tasking.com

GERMANY

TASKING GmbH
Brennerstraße 5
71229 Leonberg, Germany
Phone: +49-7152-97991-0
Fax: +49-7152-97991-20
Email: sales_de@tasking.com

ITALY

TASKING Srl
Via Napo Torriani 29
20124 Milano, Italy
Phone: +39-2-6698-2207
Fax: +39-2-6698-2189
Email: sales_it@tasking.com

JAPAN

Nihon TASKING K.K.
Shiba-ST Building, 6th floor
1-15-13 Shiba
Minato-ku
Tokyo 105, Japan
Phone: +81-3-3457-6831
Fax: +81-3-3457-6834
Email: sales@tasking.co.jp

UNITED KINGDOM

TASKING Ltd.
Enterprise House
Ocean Village
Southampton, Hampshire SO14 3XB
United Kingdom
Phone: +44-1703-334774
Fax: +44-1703-334772
Email: sales_uk@tasking.com

INTERNET

http://www.tasking.com

TASKING

Quality Development Tools Worldwide

Section 9
Package Information

CONTENTS

INTRODUCTION

There is no soldering method that is ideal for all IC packages. Wave soldering is often preferred when though-hole and surface mounted components are mixed on one printed-circuit board. However, wave soldering is not always suitable for surface mounted ICs, or for printed-circuits with high population densities. In these situations reflow soldering is often used.

This text gives a very brief insight to a complex technology. A more in-depth account of soldering ICs can be found in our *"IC Package Databook"* (order code 9398 652 90011).

THROUGH-HOLE MOUNTED PACKAGES

Table 1. Types of through-hole mounted packages

TYPE	DESCRIPTION
DIP	plastic dual in-line package
SDIP	plastic shrink dual in-line package
HDIP	plastic heat-dissipating dual in-line package
DBS	plastic dual in-line bent from a single in-line package
SIL	plastic single in-line package

Soldering by dipping or wave

The maximum permissible temperature of the solder is 260°C; solder at this temperature must not be in contact with the joint for more than 5 seconds. The total contact time of successive solder waves must not exceed 5 seconds.

The device may be mounted to the seating plane, but the temperature of the plastic body must not exceed the specified maximum storage temperature ($T_{stg\ max}$). If the printed-circuit board has been pre-heated, forced cooling may be necessary immediately after soldering to keep the temperature within the permissible limit.

Repairing soldered joints

Apply a low voltage soldering iron (less than 24V) to the lead(s) of the package, below the seating plane or not more than 2mm above it. If the temperature of the soldering iron bit is less than 300°C it may remain in contact for up to 10 seconds. If the bit temperature is between 300 and 400°C, contact may be up to 5 seconds.

SURFACE MOUNTED PACKAGES

Table 2. Types of surface mounted packages

TYPE	DESCRIPTION
SO	plastic small outline package
SSOP	plastic shrink small outline package
TSSOP	plastic thin shrink small outline package
VSO	plastic very small outline package
QFP	plastic quad flat package
LQFP	plastic low profile quad flat package
SQFP	plastic shrink quad flat package
TQFP	plastic thin quad flat package
PLCC	plastic leaded chip carrier

Reflow soldering

Reflow soldering techniques are suitable for all SMD packages, ease of soldering varies with the type of package as indicated in Table 3.

The choice of heating method may be indluenced by larger plastic packages (QFP or PLCC with 44 leads, or more). If infrared or vapor phase heating is used and the large packages are not absolutely dry (less than 0.1% moisture content by weight), vaporization of the small amount of moisture in them can cause cracking of the plastic body. For more information on moisture prevention, refer to the Drypack chapter in our *"Quality Reference Manual"* (order code 9398 510 63011).

Reflow soldering requires solder paste (a suspension of fine solder particles, flux and binding agent) to be applied to the printed-circuit board by screen printing, stenciling or pressure-syringe dispensing before package placement.

Several techniques exist for reflowing; for example, thermal conduction by heated belt. Dwell times vary between 50 and 300 seconds depending on heating method. Typical reflow temperatures range from 215 to 250°C.

Preheating is necessary to dry the paste and evaporate the binding agent. Preheating duration: 45 minutes at 45°C.

Table 3. Suitability of surface mounted packages for various soldering methods

Rating from 'a' to 'd': **'a' indicates most suitable** (soldering is not difficult); **'d' indicates least suitable** (soldering is achievable with difficulty).

TYPE	REFLOW METHOD					DOUBLE WAVE METHOD
	INFRARED	HOT BELT	HOT GAS	VAPOR PHASE	RESISTANCE	
SO	a	a	a	a	d	a
SSOP	a	a	a	c	d	c
TSSOP	b	b	b	c	d	d
VSO	b	b	a	b	a	b
QFP	b	b	a	c	a	c
LQFP	b	b	a	c	d	d
SQFP	b	b	a	c	d	d
TQFP	b	b	a	c	d	d
PLCC	c	b	b	d	d	b

Wave soldering

Wave soldering is **not** recommended for SSOP, TSSOP, QFP, LQFP, SQFP or TQFP packages. This is because of the likelihood of solder bridging due to closely-spaced leads and the possibility of incomplete solder penetration in multi-lead devices.

If wave soldering cannot be avoided, the following conditions must be observed:

- A double-wave (a turbulent wave with high upward pressure followed by a smooth laminar wave) soldering technique should be used.
- For SSOP, TSSOP and VSO packages, the longitudinal axis of the package footprint must be parallel to the solder flow **and** must incorporate solder theives at the downstream end.
- For QFP, LQFP and TQFP packages, the footprint must be at and angle of 45° to the board direction **and** must incorporate solder thieves downstream and at the side corners.

Even with these conditions, only consider wave soldering for the following package types:

- SO
- VSO
- PLCC
- SSOP **only with body width 4.4mm**, e.g., SSOP16 (SOT369-1) or SSOP20 (SOT266-1).
- QFP **except** QFP52 (SOT379-1), QFP100 (SOT317-1, SOT317-2 and SOT382-1) and QFP160 (SOT322-1); these are **not** suitable for wave soldering.
- LQFP **except** LQFP32 (SOT401-1), LQFP48 (SOT313-1, SOT313-2), LQFP64 (SOT314-2), LQFP80 (SOT315-1); these are **not** suitable for wave soldering.
- TQFP **except** TQFP64 (SOT357-1), TQFP80 (SOT375-1) and TQFP100 (SOT386-1); these are **not** suitable for wave soldering.

SQFP are **not** suitable for wave soldering.

During placement and before soldering, the package must be fixed with a droplet of adhesive. The adhesive can be applied by screen printing, pin transfer or syringe dispensing. The package can be soldered after the adhesive is cured.

Maximum permissible solder temperature is 260°C, and maximum duration of package immersion in solder is 10 seconds, if cooled to less than 150°C within 6 seconds. Typical dwell time is 4 seconds at 250°C.

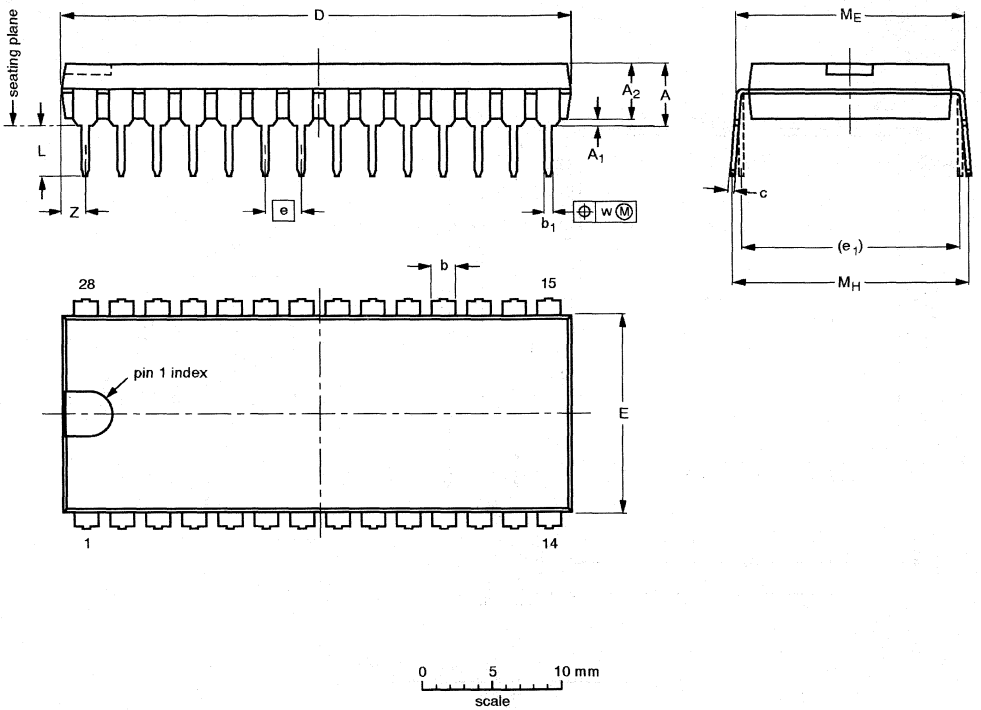
A mildly-activated flux will eliminate the need for removal of corrosive residues in most applications.

Repairing soldered joints

Fix the component by first soldering two diagonally-opposite end leads. Use only a low voltage soldering iron (less than 24V) applied to the flat part of the lead. Contact time must be limited to 10 seconds at up to 300°C. When using a dedicated tool, all other leads can be soldered in one operation within 2 to 5 seconds at between 270 and 320°C.

Package outlines

DIP28: plastic dual in-line package; 28 leads (600 mil) SOT117-1



DIMENSIONS (inch dimensions are derived from the original mm dimensions)

UNIT	A max.	A_1 min.	A_2 max.	b	b_1	c	D (1)	E (1)	e	e_1	L	M_E	M_H	w	Z (1) max.
mm	5.1	0.51	4.0	1.7 1.3	0.53 0.38	0.32 0.23	36.0 35.0	14.1 13.7	2.54	15.24	3.9 3.4	15.80 15.24	17.15 15.90	0.25	1.7
inches	0.20	0.020	0.16	0.066 0.051	0.020 0.014	0.013 0.009	1.41 1.34	0.56 0.54	0.10	0.60	0.15 0.13	0.62 0.60	0.68 0.63	0.01	0.067

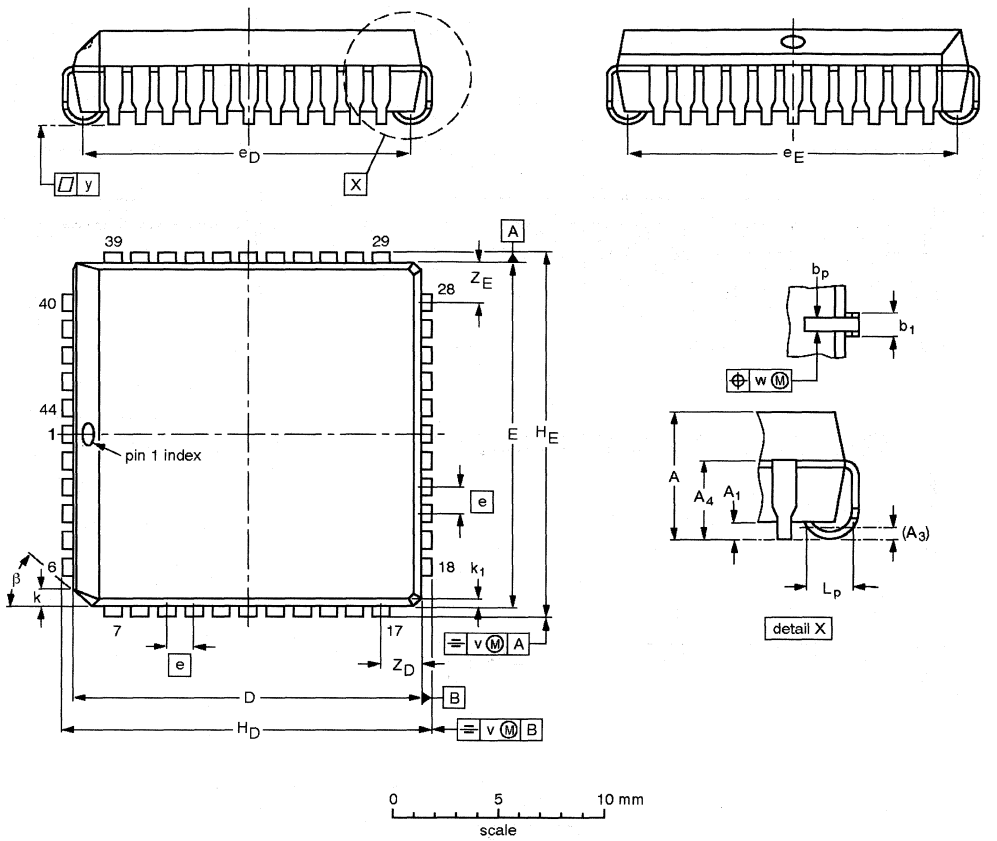
Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT117-1	051G05	MO-015AH				~~92-11-17~~ 95-01-14

Package outlines

PLCC44: plastic leaded chip carrier; 44 leads — SOT187-2



DIMENSIONS (millimetre dimensions are derived from the original inch dimensions)

UNIT	A	A_1 min.	A_3	A_4 max.	b_p	b_1	$D^{(1)}$	$E^{(1)}$	e	e_D	e_E	H_D	H_E	k	k_1 max.	L_p	v	w	y	$Z_D^{(1)}$ max.	$Z_E^{(1)}$ max.	β
mm	4.57 4.19	0.51	0.25	3.05	0.53 0.33	0.81 0.66	16.66 16.51	16.66 16.51	1.27	16.00 14.99	16.00 14.99	17.65 17.40	17.65 17.40	1.22 1.07	0.51	1.44 1.02	0.18	0.18	0.10	2.16	2.16	45°
inches	0.180 0.165	0.020	0.01	0.12	0.021 0.013	0.032 0.026	0.656 0.650	0.656 0.650	0.05	0.630 0.590	0.630 0.590	0.695 0.685	0.695 0.685	0.048 0.042	0.020	0.057 0.040	0.007	0.007	0.004	0.085	0.085	

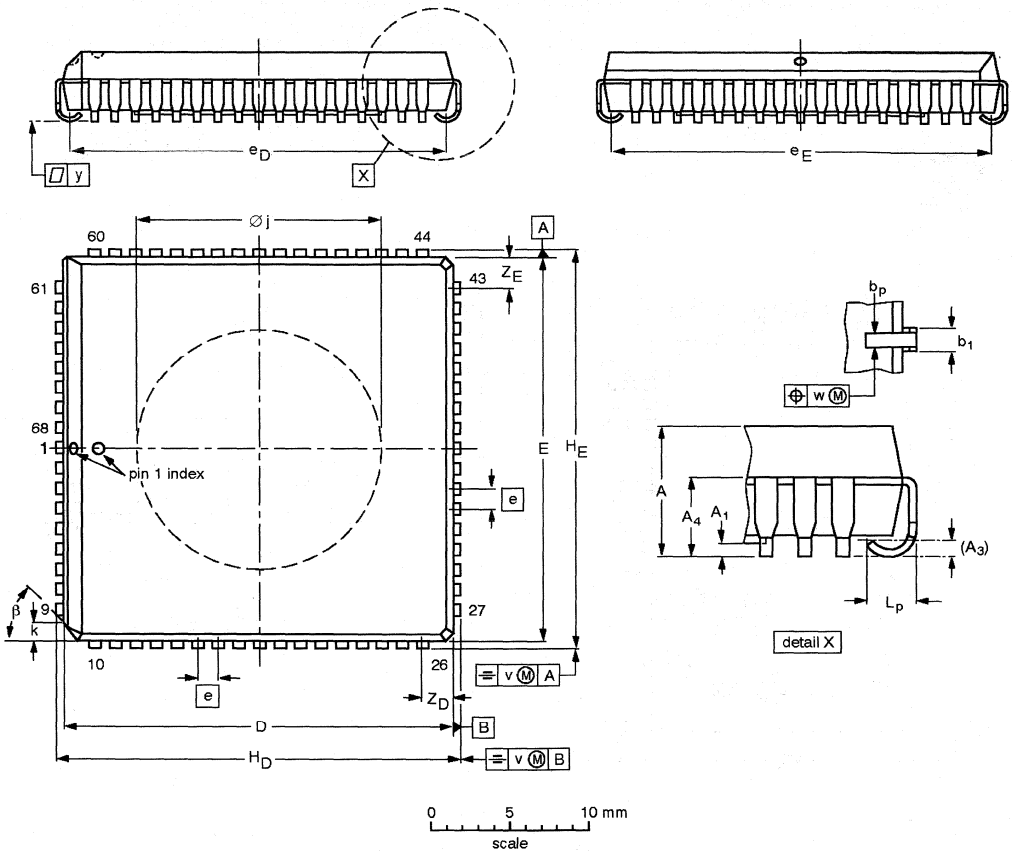
Note

1. Plastic or metal protrusions of 0.01 inches maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT187-2	112E10	MO-047AC				~~95-02-25~~ 97-12-16

Package outlines

PLCC68: plastic leaded chip carrier; 68 leads; pedestal — SOT188-3



DIMENSIONS (millimetre dimensions are derived from the original inch dimensions)

UNIT	A	A_1 min.	A_3	A_4 max.	b_p	b_1	$D^{(1)}$	$E^{(1)}$	e	e_D	e_E	H_D	H_E	k	$Ø_j$	L_p	v	w	y	$Z_D^{(1)}$ max.	$Z_E^{(1)}$ max.	β
mm	4.57 4.19	0.13	0.25	3.05	0.53 0.33	0.81 0.66	24.33 24.13	24.33 24.13	1.27	23.62 22.61	23.62 22.61	25.27 25.02	25.27 25.02	1.22 1.07	15.34 15.19	1.44 1.02	0.18	0.18	0.10	2.06	2.06	45°
inches	0.180 0.165	0.005	0.01	0.12	0.021 0.013	0.032 0.026	0.958 0.950	0.958 0.950	0.05	0.930 0.890	0.930 0.890	0.995 0.985	0.995 0.985	0.048 0.042	0.604 0.598	0.057 0.040	0.007	0.007	0.004	0.081	0.081	

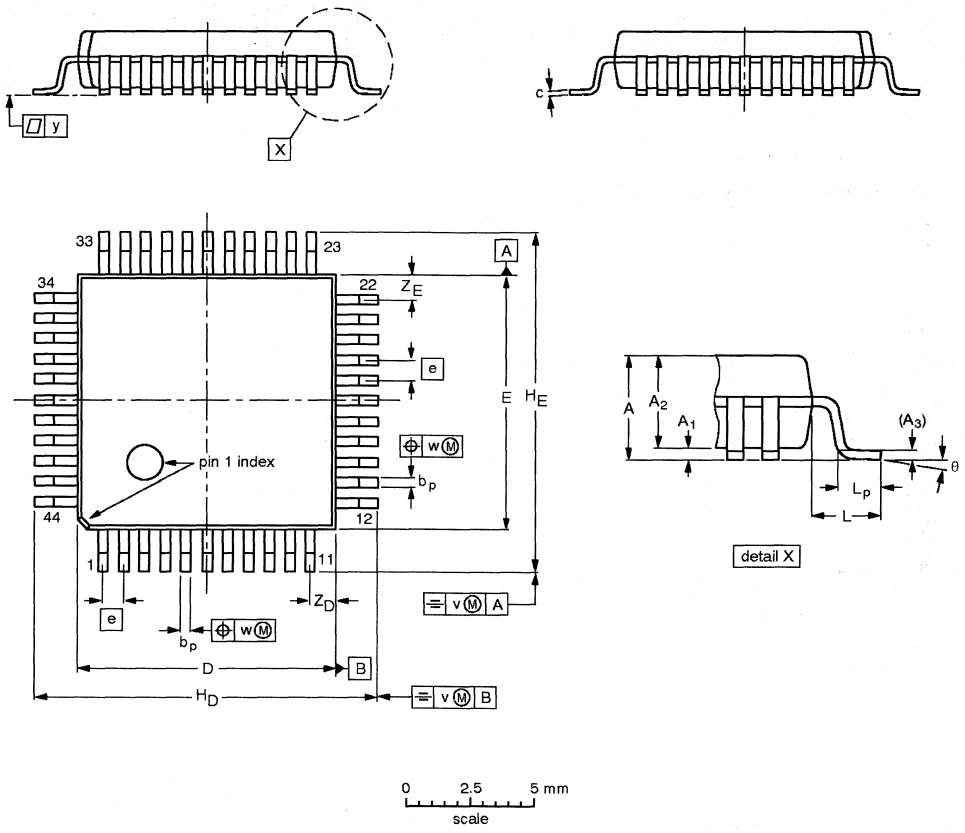
Note

1. Plastic or metal protrusions of 0.01 inches maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT188-3	112E10	MO-047AE				~~95-02-25~~ 97-12-16

Package outlines

LQFP44: plastic low profile quad flat package; 44 leads; body 10 x 10 x 1.4 mm — SOT389-1



DIMENSIONS (mm are the original dimensions)

UNIT	A max.	A_1	A_2	A_3	b_p	c	$D^{(1)}$	$E^{(1)}$	e	H_D	H_E	L	L_p	v	w	y	$Z_D^{(1)}$	$Z_E^{(1)}$	θ
mm	1.60	0.15 0.05	1.45 1.35	0.25	0.45 0.30	0.20 0.12	10.10 9.90	10.10 9.90	0.80	12.15 11.85	12.15 11.85	1.0	0.75 0.45	0.20	0.20	0.10	1.14 0.85	1.14 0.85	7° 0°

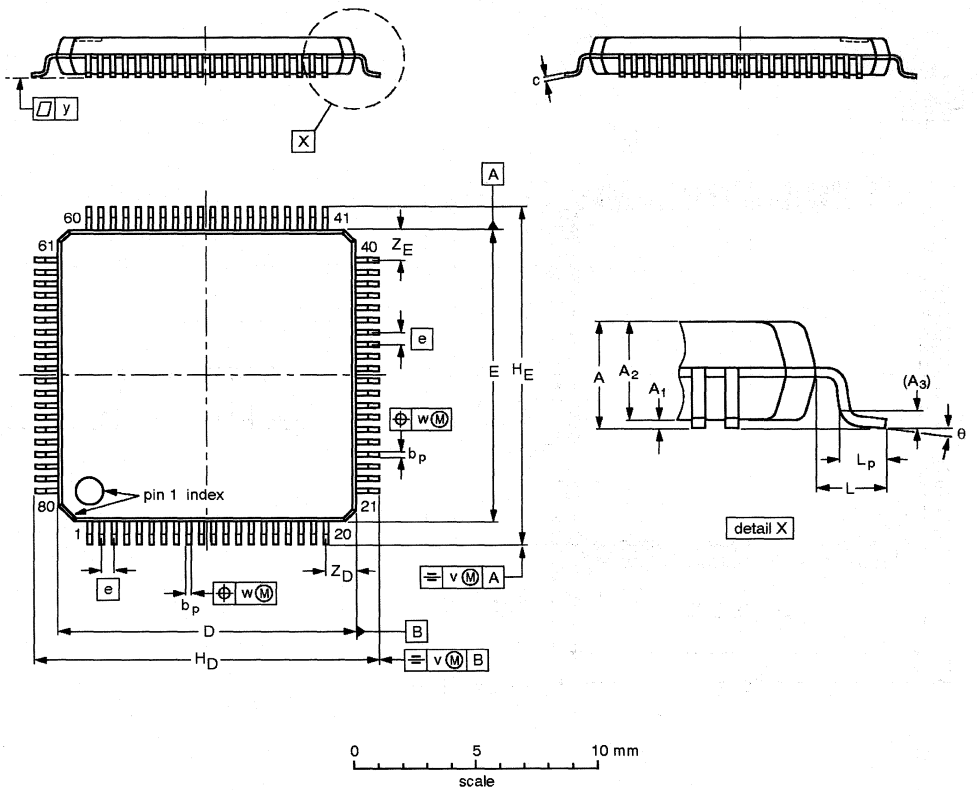
Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT389-1						~~95-12-19~~ 97-08-04

Package outlines

LQFP80: plastic low profile quad flat package; 80 leads; body 12 x 12 x 1.4 mm — SOT315-1



DIMENSIONS (mm are the original dimensions)

UNIT	A max.	A_1	A_2	A_3	b_p	c	$D^{(1)}$	$E^{(1)}$	e	H_D	H_E	L	L_p	v	w	y	$Z_D^{(1)}$	$Z_E^{(1)}$	θ
mm	1.6	0.16 0.04	1.5 1.3	0.25	0.27 0.13	0.18 0.12	12.1 11.9	12.1 11.9	0.5	14.15 13.85	14.15 13.85	1.0	0.75 0.30	0.2	0.15	0.1	1.45 1.05	1.45 1.05	7° 0°

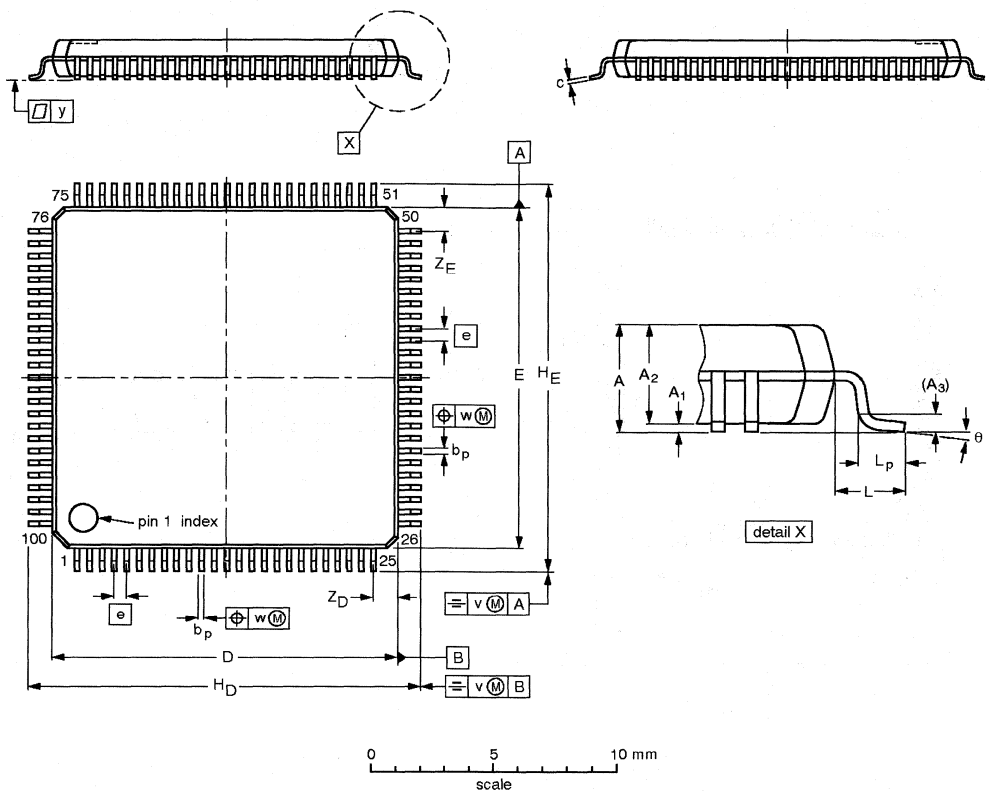
Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT315-1						~~95-12-19~~ 97-07-15

Package outlines

LQFP100: plastic low profile quad flat package; 100 leads; body 14 x 14 x 1.4 mm — SOT407-1



DIMENSIONS (mm are the original dimensions)

UNIT	A max.	A_1	A_2	A_3	b_p	c	$D^{(1)}$	$E^{(1)}$	e	H_D	H_E	L	L_p	v	w	y	$Z_D^{(1)}$	$Z_E^{(1)}$	θ
mm	1.6	0.20 0.05	1.5 1.3	0.25	0.28 0.16	0.18 0.12	14.1 13.9	14.1 13.9	0.5	16.25 15.75	16.25 15.75	1.0	0.75 0.45	0.2	0.12	0.1	1.15 0.85	1.15 0.85	7° 0°

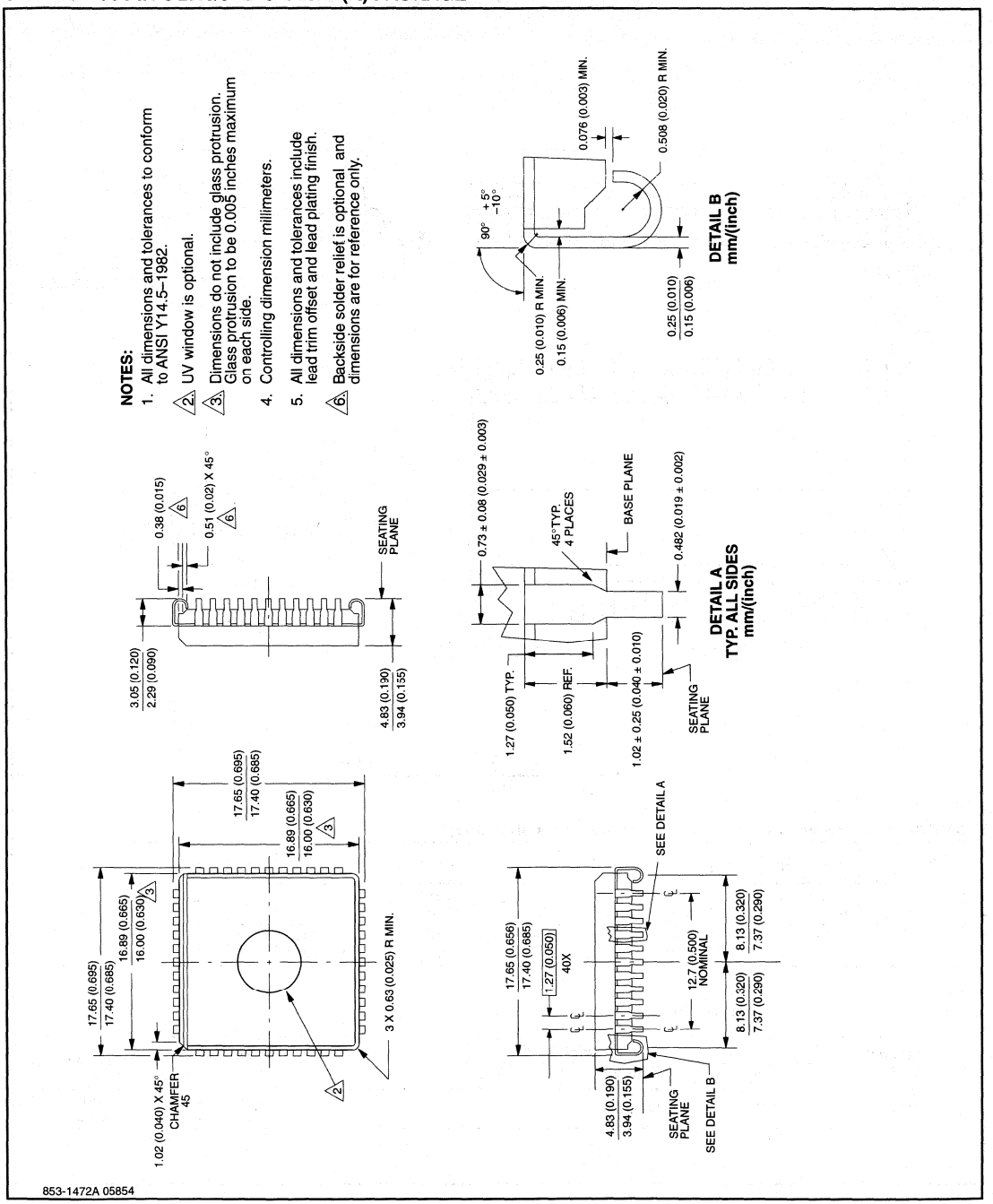
Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT407-1						~~95-12-19~~ 97-08-04

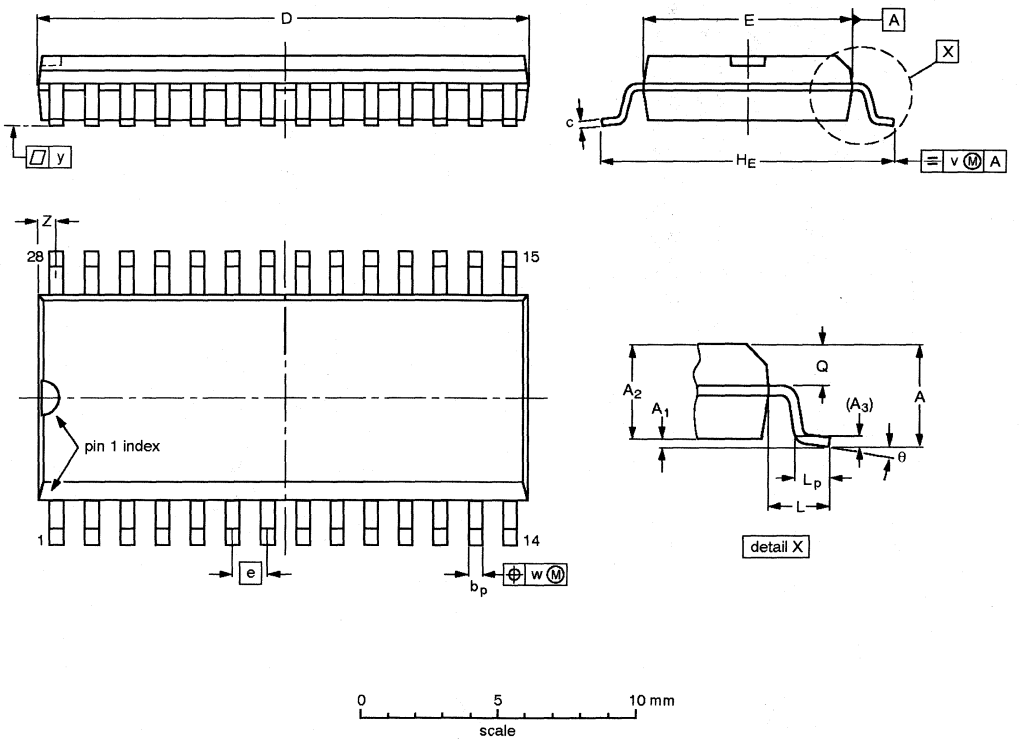
Package outlines

1472A 44-PIN CERQUAD J-BEND (K) PACKAGE



Package outlines

SO28: plastic small outline package; 28 leads; body width 7.5mm — SOT136-1



DIMENSIONS (inch dimensions are derived from the original mm dimensions)

UNIT	A max.	A_1	A_2	A_3	b_p	c	$D^{(1)}$	$E^{(1)}$	e	H_E	L	L_p	Q	v	w	y	$Z^{(1)}$	θ
mm	2.65	0.30 0.10	2.45 2.25	0.25	0.49 0.36	0.32 0.23	18.1 17.7	7.6 7.4	1.27	10.65 10.00	1.4	1.1 0.4	1.1 1.0	0.25	0.25	0.1	0.9 0.4	8^o 0^o
inches	0.10	0.012 0.004	0.096 0.089	0.01	0.019 0.014	0.013 0.009	0.71 0.69	0.30 0.29	0.050	0.419 0.394	0.055	0.043 0.016	0.043 0.039	0.01	0.01	0.004	0.035 0.016	

Note

1. Plastic or metal protrusions of 0.15 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT136-1	075E06	MS-013AE				~~95-01-24~~ 97-05-22

Appendix A
Philips Microcontroller Support Files and Software

available on CD-ROM only

Included on the CD-ROM are a selection of development software, example code, and other useful files from the Philips Microcontroller Support web site on the Internet. These files provide helpful tools and examples for many different tasks and apply to both the XA and 80C51 microcontroller families. The web file site is updated as new items and newer versions of existing items become available. The path to the support file site is: http://www.philipsmcu.com

Files on the CD-ROM include development support software and examples from Philips Semiconductors, third party vendors, and customers. Files are organized into various subjects to make finding the correct item(s) easier. The file categories include:

- Assemblers, Disassemblers, and Simulators Basic Utilities and Interpreters
- Monitors and Debuggers
- Code Examples
- Forth Programming Tools
- I^2C Related Files
- Miscellaneous Information and Utilities
- XA Microcontroller Examples and Development Tools

Most of these categories apply primarily to the 80C51 family, although the upward compatibility of the XA architecture allows extending 80C51 concepts to the XA. Tools and examples that are specific to the XA architecture are included in the XA category.

Many of the files included on the CD-ROM are compressed using the ZIP file compression format. This method also allows several related files to be collected together in a single file that can be copied and uncompressed. ZIP files can be uncompressed using many readily available archiving utilities. To use these files, it is recommended that they first be copied to a temporary directory on a hard disk drive and uncompressed there.

In order to copy support files to the user's system, they must first be found on the CD-ROM drive. The path to the main directory for these files is d:\... , where "d:" represents the drive letter of the drive where the CD-ROM has been loaded. It is typically easier to locate and copy files using a system utility with a graphical interface such as Windows Explorer or File Manager.

Some files on the CD-ROM are self-extracting archives in various formats. These files have the ".EXE" file name extension. Again, these should first be copied to a temporary directory on a hard disk drive, then executed in order to extract the archive contents.

The CD-ROM also includes some files that are simple text files. These are generally source code files in assembly, C, or some other language. Typically, they have file name extensions like ".ASM", ".A51", or ".C". These files may be read using any plain text editor such as WordPad in Microsoft Windows.

Development tool files are included on the CD-ROM that contain executable programs. Some of these may require an installation process prior to use, while others can be run as-is.

File Categories:

Assemblers, Disassemblers, and Simulators
Basic Utilities and Interpreters
Monitors and Debuggers
Code Examples
Forth Programming Tools
I²C Related Files
Miscellaneous Information and Utilities
XA Microcontroller Examples and Development Tools

Assemblers, Disassemblers, and Simulators

a51.zip	PseudoSam 8051 Cross Assembler, V1.4.09
as31.zip	C source for an 8051 assembler, and a simple monitor from Ken Stauffer.
d51v22.zip	8051 disassembler version 2.2.
dis8051f.zip	DataSync 8031/51 disassembler.
ml–asm51.zip	MetaLink's 8051 family macro assembler. (used in most of our app notes)
models2.zip	New and updated derivative model files for the MetaLink 80C51 assembler.
sim51_04.zip	8051 shareware simulator. Note: documentation is in German!
tasm30.zip	Table driven assembler for various Micros/CPUs.

Basic Utilities and Interpreters

bas051.zip	Converts IBM BASIC to '51 assembly.
basic–52.zip	Source files for BASIC-52 interpreter.
basic31a.zip	Improved BASIC-52 for 8031/8051 in external EPROM.
tb–51.zip	TinyBASIC for 8031, w/ source files.
tb51ml23.zip	MetaLink ASM compatible tiny BASIC.

Monitors and Debuggers

bm51.zip	Small background monitor (614 bytes) for 8051
db51ks.exe	Combined RS751/DEBUG51 for RT apps.
debug51.zip	80C51 code debugging tool from Axxon.
mon31–11.zip	Simple monitor routines for the 8031 with PseudoSam assembly source.
monplus.zip	A re-written and expanded 8031 monitor based on Ron Stubbers' original one.
pds225a.zip	Demo of Integrated Development Environment of the Philips PDS-51 emulator for the 80C51 family. Version 2.25.

Code Examples

51serial.zip	Serial port software examples for the 8051.
ad1.asm	A/D code for the 'C552.
an429.zip	Source for app note on '752 air flow measurement (AN429).
autobaud.zip	Example of automatic baud rate detection from AN447.
battchrg.c	Source code for a fast battery charger using the 8xC751. From app note AN439.
bootstrp.zip	Hex file Load-and-Go using 8051 UART from AN440.
canfiles.exe	Demo code and documentation for the 82C200 CAN bus controller and 8xC592/8xCE598 micros with integrated CAN controller.
cci6.zip	MTV demo code for on-screen display. Goes with Circuit Cellar Ink article fm '92.
clock.zip	Example of real time clock fm Sytronics.
coffey.asm	Displays the contents of the S87C752 A/D SFRs.
demo752.asm	Demonstration program for the A/D and PWM features of the 8xC752 from AN428.
dialer.zip	8031 BASED TELEPHONE # PULSE DIALER
dtmf.zip	80C31 code to generate DTMF and signalling tones BUSY, RING-BACK, etc.
dupuart.zip	Duplex software UART code for 751/752 from AN446.
eeprm851.zip	EEPROM driver routines for the 8xC851. From app note EIE/AN91009.
float51.zip	Floating point math for the 8051, written by one of Dave Dunfield's customers.
intrupts.asm	Demo of extra external interrupts on C51 from AN420.
ircon.zip	Interface to a Sharp infrared sensor that can receive Phillips RC5 IR control codes, and toggle relays.
keyer.asm	Ham Radio Keyer Using the 87C752.
keys.asm	8xC751 code to scan a keyboard and output to a PC/AT.
lcdriver.zip	Optrex LCD driver for 87C751.
math51.zip	Multi-byte math routines for the 8051
mazemous.zip	Source code for an IEEE maze navigating mouse using the 8xC751. From AN443.
midi8751.asm	Midi sample code.
morse.asm	Morse code sending routine.
mtv.zip	Demo program with a sample font and asm definitions for 8xC054 (MTV).
music750.zip	"Music box" program for 87C750. Contains reusable code to generate audio tones and do timing.
prn256k.zip	8xC451 code (from AN417) for a 256K printer buffer. Schematic in data book.
rs751.asm	Simplex UART routines for the 751 & 752 from AN423.
samples.zip	Sample 80C552 subroutines fm Sytronics.
serial.zip	Circular buffer code for standard UART.
strngout.zip	String output routine.
timeri.zip	Examples of Timer I used without I²C on the 8xC751/752. From AN427.
warmboot.zip	How to distinguish warm & cold startup on 80C51 based parts. From AN424.
water.zip	Code for an 8xC750 watering controller, which supports 8 independent zones, has a simple user interface, and battery backup. From AN459. Versions with the display in English and French are included.

Forth Programming Tools

eforth51.zip	eFORTH environment for the 8051.
forth51.zip	FORTH for 8051 family.
xd8051.zip	F-PC Forth environment for the 8051.

I²C Related Files

abmouse.zip	ACCESS.bus mouse code from AN445.
an435A.exe	Updated IIC_OS multimaster drivers for microcontrollers with byte I²C interfaces (552-type). From application note AN435.
i2c552–c.zip	I²C drivers for the 8xC552 with a C language interface.
i2c8584.zip	Code from app note AN425 using the 8584 I²C to parallel bus i/f with the 80C31.
i2c_528.exe	Code for 8xC528 I²C interface. From app note EIE/AN90015.
i2c_552.exe	I²C drivers for 8xC552 with PLM and C, from app note EIE/AN89004.
i2capp.zip	Source code for the app note AN422 on single master I²C with the 8xC751/752.
i2cbits.zip	I²C single master code for ANY 8051 type controller. 'Bit bangs' I²C on port pins
i2cbitst.zip	I²C bit banged routines for I²C peripherals including the 8591 A/D.
i2cdemo.zip	I²C Eval. Board (part#S87C00KSD) source code. This is an update to match the manual.
i2cinit.zip	Lets 8xC751 do system init of I²C and other devices (via reset pulse).
i2cpckb.zip	Interfaces a standard PC/AT keyboard to the I²C bus. From AN434.
mm751.zip	Multimaster I²C code for the 8xC751/752. From app note AN430.
mm751b.zip	I²C drivers for the 8xC751 and 752. From app note EIE/AN91007.
pci2c.zip	Software V3.2 for I²C PC printer port adapter (needs board in order to use).
pci2cbd.zip	Schematic of I²C printer port adapter.
pcx8584.exe	C routines for PCF8584 with application note AN95068.
slv751.zip	Slave I²C functions for 8xC751/752 from AN433.
tv400.exe	Software V4.00 for I²C PC printer port adapter (needs board in order to use).

Miscellaneous Information and Utilities

51to550.exe	Self extracting files containing artwork for adapter to allow programming the 87C550 in place of the 87C51.
8051net.zip	8051 Resource FAQ; Lists Internet ftp sites, 8051 support vendors
80c451	Orcad library element for 80C451 LCC.
80c552	Schematic symbol for use with Orcad.
demo_pwm.zip	Converts music to 8052 BASIC PWM program.
hexbin.zip	Intel HEX to Binary, w/ new features.
hexutil.zip	Hex file load and program utilities for 8052 BASIC.
hexutils.zip	Hex to bin, bin to hex, and hex to hex conversion, for object file fixes.
midiloop.gif	GIF of schematic showing example hardware to interface 8051 to MIDI.
plm752.zip	Modified PL/M-51 library for use with 87C752. The standard library won't work! Source code included. Must have Intel ASM51 and PLM51.
reg552.inc	80C552 declaration for Franklin asm.
regc552.h	80C552 C declarations for Franklin C.
sim.zip	Robot simulation and machine learning utilities.
tutor51.zip	TSR help screens with most of the common 8051 device info – handy

XA Microcontroller Examples and Development Tools

baudrate.txt	Tables of standard baud rates and crystal frequencies for the XA.
bin2bcd.xa	A simple 16-bit binary to BCD conversion routine.
ex0–int.xa	Demo setup of an external interrupt.
reverse.xa	Demonstration code for the four byte reversal routines from app note AN709: "Reversing bits within a data byte on the XA"
skel–g3.xa	This is a "skeleton" ASM file for the XA-G3. It can be used as a starting point for new code development, saving time by providing all of the interrupt vector definitions and standard startup code.
skel–s3.xa	This is a "skeleton" ASM file for the XA-S3. Use as a starting point for code development, providing standard interrupt vector definitions and startup code.
uart–int.xa	Sample code to drive an XA-G3 UART using interrupts.
ucos.zip	Source code for XA real-time multi-tasking kernel from Jean Labrosse (uC/OS).
xa–g3.equ	Philips generated assembler definitions for the XA-G3.
xa–s3.equ	Philips generated assembler definitions for the XA-S3.
xa_tools.zip	Integrated Development Tool for the Philips XA 16-bit microcontroller. Includes an assembler, simulator, and 8051 to XA source translator running under windows. This file is a self-extracting archive. Run xa-tools.exe and the run setup.exe.
xaflash.zip	This file contains all the files that compliment application note AN97019, "Using Flash Memory."
an96119.zip	Application note on using I²C with the XA-G3. Shows two ways to add single master I²C to the XA, with C source code.

Appendix B
Data Handbook System

CONTENTS

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

Book	*Title*
IC01	Semiconductors for Radio, Audio and CD/DVD Systems
IC02	Semiconductors for Television and Video Systems
IC03	Semiconductors for Wired Telecom Systems
IC04	HE4000B Logic Family CMOS
IC05	Advanced Low-power Schottky (ALS) Logic
IC06	High-speed CMOS Logic Family
IC11	General-purpose/Linear ICs
IC12	I^2C Peripherals
IC13	Programmable Logic Devices (PLD)
IC14	8048-based 8-bit Microcontrollers
IC15	FAST TTL Logic Series
IC16	CMOS ICs for Clocks, Watches and Real Time Clocks
IC17	Semiconductors for Wireless Communications
IC18	Semiconductors for In-Car Electronics
IC19	ICs for Data Communications
IC20	80C51-based 8-bit Microcontrollers
IC22	Multimedia ICs
IC23	BiCMOS Bus Interface Logic
IC24	Low Voltage CMOS & BiCMOS Logic
IC25	16-bit 80C51XA Microcontrollers (eXtended Architecture)
IC26	Integrated Circuit Packages
IC27	Complex Programmable Logic Devices

Discrete Semiconductors

Book	*Title*
SC01	Small-signal and Medium-power Diodes
SC02	Power Diodes
SC03	Power Thyristors and Triacs
SC04	Small-signal Transistors
SC05	Video Transistors and Modules for Monitors
SC06	High-voltage and Switching NPN Power Transistors
SC07	Small-signal Field-effect Transistors
SC13	Power MOS Transistors
SC14	RF Wideband Transistors
SC16	Wideband Hybrid Amplifier Modules for CATV
SC17	Semiconductor Sensors
SC18	Discrete Semiconductor Packages
SC19	RF & Microwave Power Transistors, RF Power Modules and Circulators/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 Television Tubes
DC02	Monochrome Monitor Tubes and Deflection Units
DC03	Television Tuners, Coaxial Aerial Input Assemblies
DC04	Colour Monitor and Multimedia Tubes
DC05	Wire Wound Components

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
PA06a	Surface Mounted Ceramic Multilayer Capacitors
PA06b	Leaded Ceramic Capacitors
PA08	Fixed Resistors
PA10	Quartz Crystals
PA11	Quartz Oscillators

MORE INFORMATION FROM PHILIPS COMPONENTS?

For more information contact your nearest Philips Components national organizaiton shown in the following list.

Australia: North Ryde, Tel. +61 2 9805 4455, Fax. +61 2 9805 4466
Austria: Wien, Tel. +43 1 60 101 12 41, Fax. +43 1 60 101 12 11
Belarus: Minsk, Tel. +375 172 200 924/733, Fax. +375 172 200 773
Benelux: Eindhoven, Tel. +31 40 2783 749, Fax. +31 40 2788 399
Brazil: São Paulo, Tel. +55 11 821 2333, Fax. +55 11 829 1849
Canada: Scarborough, Tel. 1 416 292 5161, Fax. 1 416 754 6248
China: Shanghai, Tel. +86 21 6354 1088, Fax. +86 21 6354 1060
Denmark: Copenhagen, Tel. +45 32 883 333, Fax. +45 31 571 949
Finland: Espoo, Tel. 358 9 615 800, Fax. 358 9 615 80510
France: Suresnes, Tel. +33 1 4099 6161, Fax. +33 1 4099 6493
Germany: Hamburg, Tel. +49 40 2489-0, Fax. +49 40 2489 1400
Greece: Tavros, Tel. +30 1 4894 339/+30 1 4894 239, Fax. +30 1 4814 240
Hong Kong: Kowloon, Tel. +852 2784 3000, Fax. +852 2784 3003
India: Mumbai, Tel. +91 22 4930 311, Fax. +91 22 4930 966/4950 304
Indonesia: Jakarta, Tel. +62 21 794 0040, Fax. +62 21 794 0080
Ireland: Dublin, Tel. +353 1 7640 203, Fax. +353 1 7640 210
Israel: Tel Aviv, Tel. +972 3 6450 444, Fax. +972 3 6491 007
Italy: Milano, Tel. +39 2 6752 2531, Fax. +39 2 6752 2557
Japan: Tokyo, Tel. +81 3 3740 5135, Fax. +81 3 3740 5035
Korea (Republic of)**:** Seoul, Tel. +82 2 709 1472, Fax. +82 2 709 1480
Malaysia: Pulau Pinang, Tel. +60 3 750 5213, Fax. +60 3 757 4880
Mexico: El Paso, Tel. +52 915 772 4020, Fax. +52 915 772 4332
New Zealand: Aukland, Tel. +64 9 815 4000, Fax. +64 9 849 7811
Norway: Oslo, Tel. +47 22 74 8000, Fax +47 22 74 8341
Pakistan: Karachi, Tel. +92 21 587 4641-49,
Fax. +92 21 577 035/+92 21 587 4546
Philippines: Manila, Tel. +63 2 816 6345, Fax. +63 2 817 3474
Poland: Warszawa, Tel. +48 22 612 2594, Fax. +48 22 612 2327
Portugal: Linda-A-Velha, Tel. +351 1 416 3160/416 3333,
Fax. +351 1 416 3174/416 3366
Russia: Moscow, Tel. +7 95 755 6918, Fax. +7 95 755 6919
Singapore: Singapore, Tel. +65 350 2000, Fax. +65 355 1758
South Africa: Johannesburg, Tel. +27 11 470 5911, Fax. +27 11 470 5494
Spain: Barcelona, Tel. +34 3 301 63 12, Fax. +34 3 301 42 43
Sweden: Stockholm, Tel. +46 8 5985 2000, Fax. +46 8 5985 2745
Switzerland: Zürich, Tel. +41 1 488 22 11, Fax. +41 1 481 7730
Taiwan: Taipei, Tel. +886 2 2134 2900, Fax. +886 2 2134 2929
Thailand: Bangkok, Tel. +66 2 745 4090, Fax. +66 2 398 0793
Turkey: Istanbul, Tel. +90 212 279 2770, Fax. +90 212 282 6707
United Kingdom: Dorking Tel. +44 1306 512 000, Fax. +44 1306 512 345
United States:
- Ann Arbor, MI, Tel. +1 734 996 9400, Fax. +1 734 761 2776
- Saugerties, NY, Tel. +1 914 246 2811, Fax. +1 914 246 0487
- San Jose, CA, Tel. +1 408 570 5600, Fax. +1 408 570 5700

Yugoslavia (Federal Republic of)**:** Belgrade, Tel. +381 11 625 344/373,
Fax. +381 11 635 777
Internet:
- Passive Components: www.passives.comp.philips.com

For all other countries apply to:
Philips Components, Marketing Communications, Building BF-1,
P.O. Box 218, 5600 MD EINDHOVEN, The Netherlands
Fax. +31–40–2724547.

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: (256) 464-9101
(256) 464-0111

Elcom, Inc.
Phone: (256) 830-4001

ARIZONA
Scottsdale
Thom Luke Sales, Inc.
Phone: (602) 451-5400

Tempe
Philips Semiconductors
Phone: (602) 820-2225

CALIFORNIA
Calabasas
Philips Semiconductors
Phone: (818) 880-6304

Centaur Corporation
Phone: (818) 878-5800

Granite Bay
B.A.E. Sales, Inc.
Phone: (916) 652-6777

Irvine
Philips Semiconductors
Phone: (714) 453-0770

Centaur Corporation
Phone: (714) 261-2123

San Diego
Philips Semiconductors
Phone: (619) 560-0242

Centaur Corporation
Phone: (619) 278-4950

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

Thom Luke Sales, Inc.
Phone: (303) 649-9717

CONNECTICUT
Wallingford
JEBCO, Inc.
Phone: (203) 265-1318

FLORIDA
(Norcross, Georgia)
Elcom, Inc.
Phone: (770) 447-8200

GEORGIA
Norcross
Elcom, Inc.
Phone: (770) 447-8200

IDAHO
(Englewood, Colorado)
Thom Luke Sales, Inc.
Phone: (303) 649-9717

ILLINOIS
Itasca
Philips Semiconductors
Phone: (630) 250-0050

INDIANA
Indianapolis
Mohrfield Marketing, Inc.
Phone: (317) 546-6969

Kokomo
Philips Semiconductors
Phone: (765) 459-5355

Leo
Mohrfield Marketing, Inc.
Phone: (219) 627-5355

KANSAS
(Bloomington, Minnesota)
High Technology Sales, Inc.
Phone: (612) 844-9933

KENTUCKY
(Indianapolis, Indiana)
Mohrfield Marketing, Inc.
Phone: (317) 546-6969

MARYLAND
(Rockville Centre, New York)
S-J Associates, Inc.
Phone: (516) 536-4242

MASSACHUSETTS
Chelmsford
JEBCO, Inc.
Phone: (978) 256-5800

Westford
Philips Semiconductors
Phone: (978) 692-6211

MICHIGAN
Farmington Hills
Philips Semiconductors
Phone: (248) 848-7600

Novi
Mohrfield Marketing, Inc.
Phone: (248) 380-8100

MINNESOTA
Bloomington
High Technology Sales, Inc.
Phone: (612) 844-9933

MISSOURI
(Bloomington, Minnesota)
High Technology Sales, Inc.
Phone: (612) 844-9933

NEBRASKA
(Bloomington, Minnesota)
High Technology Sales, Inc.
Phone: (612) 844-9933

NEW JERSEY
Toms River
Philips Semiconductors
Phone: (732) 505-1200
(732) 240-1479

NEW MEXICO
(Scottsdale, Arizona)
Thom Luke Sales, Inc.
Phone: (602) 451-5400

NEW YORK
Rockville Centre
S-J Associates, Inc.
Phone: (516) 536-4242

(Chelmsford, Massachusetts)
JEBCO, Inc.
Phone: (978) 256-5800

NORTH CAROLINA
Cary
Philips Semiconductors
Phone: (919) 462-1332
(919) 462-6361

Raleigh
Elcom, Inc.
Phone: (919) 743-5200

OHIO
(Indianapolis, Indiana)
Mohrfield Marketing, Inc.
Phone: (317) 546-6969

OKLAHOMA
(Richardson, Texas)
OM Associates, Inc.
Phone: (972) 690-96746

OREGON
Beaverton
Philips Semiconductors
Phone: (503) 627-0110

Cascade-Tech
Phone: (503) 645-9660

PENNSYLVANIA
(Indianapolis, Indiana)
Mohrfield Marketing, Inc.
Phone: (317) 546-6969

(Rockville Centre, New York)
S-J Associates, Inc.
Phone: (516) 536-4242

TENNESSEE
Dandridge
Philips Semiconductors
Phone: (423) 397-5557

TEXAS
Austin
OM Associates, Inc.
Phone: (512) 794-9971

Houston
Philips Semiconductors
Phone: (281) 999-1316

OM Associates, Inc.
Phone: (281) 376-6400

Richardson
Philips Semiconductors
Phone: (972) 644-1610

OM Associates, Inc.
Phone: (972) 690-6746

VIRGINIA
(Rockville Centre, New York)
S-J Associates, Inc.
Phone: (516) 536-4242

WISCONSIN
(Bloomington, Minnesota)
High Technology Sales, Inc.
Phone: (612) 844-9933

WASHINGTON
Kirkland
Cascade-Tech
Phone: (425) 822-7299

CANADA
PHILIPS SEMICONDUCTORS CANADA, LTD.

Calgary, Alberta
Tech-Trek, Ltd.
Phone: (403) 291-6866

Kanata, Ontario
Tech-Trek, Ltd.
Phone: (613) 599-8787

Mississauga, Ontario
Tech-Trek, Ltd.
Phone: (905) 238-0366

Richmond, B.C.
Tech-Trek, Ltd.
Phone: (604) 276-8735

Ville St. Laurent, Quebec
Tech-Trek, Ltd.
Phone: (514) 337-7540

MEXICO
Guadalajara
Mepco Centralab, Inc./Philips
Phone: 8-011-52-3-122-2325

Monterrey
Mepco Centralab, Inc./Philips
Phone: 8-011-52-8-399-0164

El Paso, TX
Philips Components
Phone: (915) 772-4020

PUERTO RICO
(Norcross, Georgia)
Elcom, Inc.
Phone: (770) 447-8200

DISTRIBUTORS

Contact one of our local distributors:
Allied Electronics
Arrow Electronics
Future Electronics
Hamilton Hallmark
Marshall Industries
Newark Electronics
Penstock
Richardson Electronics
Zeus Electronics

Philips Semiconductors – a worldwide company

Argentina: see South America

Australia: 34 Waterloo Road, NORTH RYDE, NSW 2113,
Tel. +61 2 9805 4455, Fax. +61 2 9805 4466

Austria: Computerstr. 6, A-1101 WIEN, P.O. Box 213, Tel. +43 160 1010,
Fax. +43 160 101 1210

Belarus: Hotel Minsk Business Center, Bld. 3, r. 1211, Volodarski Str. 6,
220050 MINSK, Tel. +375 172 200 733, Fax. +375 172 200 773

Belgium: see The Netherlands

Brazil: see South America

Bulgaria: Philips Bulgaria Ltd., Energoproject, 15th floor,
51 James Bourchier Blvd., 1407 SOFIA,
Tel. +359 2 689 211, Fax. +359 2 689 102

Canada: PHILIPS SEMICONDUCTORS/COMPONENTS, Tel. +1 800 234 7381

China/Hong Kong: 501 Hong Kong Industrial Technology Centre,
72 Tat Chee Avenue, Kowloon Tong, HONG KONG,
Tel. +852 2319 7888, Fax. +852 2319 7700

Colombia: see South America

Czech Republic: see Austria

Denmark: Prags Boulevard 80, PB 1919, DK-2300 COPENHAGEN S,
Tel. +45 32 88 2636, Fax. +45 31 57 0044

Finland: Sinikalliontie 3, FIN-02630 ESPOO,
Tel. +358 9 615800, Fax. +358 9 61580920

France: 51 Rue Carnot, BP317, 92156 SURESNES Cedex,
Tel. +33 1 40 99 6161, Fax. +33 1 40 99 6427

Germany: Hammerbrookstra§e 69, D-20097 HAMBURG,
Tel. +49 40 23 53 60, Fax. +49 40 23 536 300

Greece: No. 15, 25th March Street, GR 17778 TAVROS/ATHENS,
Tel. +30 1 4894 339/239, Fax. +30 1 4814 240

Hungary: see Austria

India: Philips INDIA Ltd, Band Box Building, 2nd floor,
254-D, Dr. Annie Besant Road, Worli, MUMBAI 400 025,
Tel. +91 22 493 8541, Fax. +91 22 493 0966

Indonesia: PT Philips Development Corporation, Semiconductors Division,
Gedung Philips, Jl. Buncit Raya Kav.99-100, JAKARTA 12510,
Tel. +62 21 794 0040 ext. 2501, Fax. +62 21 794 0080

Ireland: Newstead, Clonskeagh, DUBLIN 14,
Tel. +353 1 7640 000, Fax. +353 1 7640 200

Israel: RAPAC Electronics, 7 Kehilat Saloniki St, PO Box 18053,
TEL AVIV 61180, Tel. +972 3 645 0444, Fax. +972 3 649 1007

Italy: PHILIPS SEMICONDUCTORS, Piazza IV Novembre 3, 20124 MILANO,
Tel. +39 2 6752 2531, Fax. +39 2 6752 2557

Japan: Philips Bldg 13-37, Kohnan 2-chome, Minato-ku, TOKYO 108-8507,
Tel. +81 3 3740 5130, Fax. +81 3 3740 5077

Korea: Philips House, 260-199 Itaewon-dong, Yongsan-ku, SEOUL,
Tel. +82 2 709 1412, Fax. +82 2 709 1415

Malaysia: No. 76 Jalan Universiti, 46200 PETALING JAYA, SELANGOR,
Tel. +60 3 750 5214, Fax. +60 3 757 4880

Mexico: 5900 Gateway East, Suite 200, EL PASO, TEXAS 79905,
Tel. +9-5 800 234 7381

Middle East: see Italy

Netherlands: Postbus 90050, 5600 PB EINDHOVEN, Bldg. VB,
Tel. +31 40 27 82785, Fax. +31 40 27 88399

New Zealand: 2 Wagener Place, C.P.O. Box 1041, AUCKLAND,
Tel. +64 9 849 4160, Fax. +64 9 849 7811

Norway: Box 1, Manglerud 0612, OSLO,
Tel. +47 22 74 8000, Fax. +47 22 74 8341

Pakistan: see Singapore

Philippines: Philips Semiconductors Philippines Inc.,
106 Valero St. Salcedo Village, P.O. Box 2108 MCC, MAKATI, Metro MANILA,
Tel. +63 2 816 6380, Fax. +63 2 817 3474

Poland: Ul. Lukiska 10, PL 04-123 WARSZAWA,
Tel. +48 22 612 2831, Fax. +48 22 612 2327

Portugal: see Spain

Romania: see Italy

Russia: Philips Russia, Ul. Usatcheva 35A, 119048 MOSCOW,
Tel. +7 095 755 6918, Fax. +7 095 755 6919

Singapore: Lorong 1, Toa Payoh, SINGAPORE 319762,
Tel. +65 350 2538, Fax. +65 251 6500

Slovakia: see Austria

Slovenia: see Italy

South Africa: S.A. PHILIPS Pty Ltd., 195-215 Main Road Martindale,
2092 JOHANNESBURG, P.O. Box 7430 Johannesburg 2000,
Tel. +27 11 470 5911, Fax. +27 11 470 5494

South America: Al. Vicente Pinzon, 173, 6th floor, 04547-130 S O PAULO, SP,
Brazil, Tel. +55 11 821 2333, Fax. +55 11 821 2382

Spain: Balmes 22, 08007 BARCELONA,
Tel. +34 93 301 6312, Fax. +34 93 301 4107

Sweden: Kottbygatan 7, Akalla, S-16485 STOCKHOLM,
Tel. +46 8 5985 2000, Fax. +46 8 5985 2745

Switzerland: Allmendstrasse 140, CH-8027 Z RICH,
Tel. +41 1 488 2741 Fax. +41 1 488 3263

Taiwan: Philips Semiconductors, 6F, No. 96, Chien Kuo N. Rd., Sec. 1, TAIPEI,
Taiwan Tel. +886 2 2134 2865, Fax. +886 2 2134 2874

Thailand: PHILIPS ELECTRONICS (THAILAND) Ltd.,
209/2 Sanpavuth-Bangna Road Prakanong, BANGKOK 10260,
Tel. +66 2 745 4090, Fax. +66 2 398 0793

Turkey: Talatpasa Cad. No. 5, 80640 G LTEPE/ISTANBUL,
Tel. +90 212 279 2770, Fax. +90 212 282 6707

Ukraine: PHILIPS UKRAINE, 4 Patrice Lumumba str., Building B, Floor 7,
252042 KIEV, Tel. +380 44 264 2776, Fax. +380 44 268 0461

United Kingdom: Philips Semiconductors Ltd., 276 Bath Road, Hayes,
MIDDLESEX UB3 5BX, Tel. +44 181 730 5000, Fax. +44 181 754 8421

United States: 811 East Arques Avenue, SUNNYVALE, CA 94088-3409,
Tel. +1 800 234 7381

Uruguay: see South America

Vietnam: see Singapore

Yugoslavia: PHILIPS, Trg N. Pasica 5/v, 11000 BEOGRAD,
Tel. +381 11 625 344, Fax.+381 11 635 777

For all other countries apply to: Philips Semiconductors, International Marketing & Sales Communications, Building BE-p, P.O. Box 218, 5600 MD EINDHOVEN, The Netherlands, Fax. +31 40 27 24825

Internet: http://www.semiconductors.philips.com

SCH60

Printed in USA 455105/32.3M/04/pp908 Date of release: July 1998 Document order number: 9397 750 03975